Clp

1.16

Generated by Doxygen 1.8.9.1

Thu Oct 8 2015 22:39:25

Contents

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

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

- CoinSearchTree< class > [external]
- CoinSearchTreeCompareBest [external]
- CoinSearchTreeCompareBreadth [external]
- CoinSearchTreeCompareDepth [external]
- CoinSearchTreeComparePreferred [external]
- CoinSearchTreeManager [external]
- CoinSet [external]
  - CoinSosSet [external]
- CoinSnapshot [external]
- CoinThreadRandom [external]
- CoinTimer [external]
- CoinTreeNode [external]
- CoinTreeSiblings [external]
- CoinTriple< S, T, U > [external]
- CoinWarmStart [external]
  - CoinWarmStartBasis [external]
    - AbcWarmStart 73
  - CoinWarmStartDual [external]
  - CoinWarmStartPrimalDual [external]
  - CoinWarmStartVector< T > [external]
  - CoinWarmStartVector< double > [external]
  - CoinWarmStartVector< U > [external]
  - CoinWarmStartVectorPair< T, U > [external]
- CoinWarmStartDiff [external]
  - CoinWarmStartBasisDiff [external]
  - CoinWarmStartDualDiff [external]
  - CoinWarmStartPrimalDualDiff [external]
  - CoinWarmStartVectorDiff< T > [external]
  - CoinWarmStartVectorDiff< double > [external]
  - CoinWarmStartVectorDiff< U > [external]
  - CoinWarmStartVectorPairDiff< T, U > [external]
- CoinYacc [external]
- std::complex
- std::multimap< K, T >::const_iterator
- std::unordered_multimap< K, T >::const_iterator
- std::set< K >::const_iterator
- OsiCuts::const_iterator [external]
- std::unordered_set< K >::const_iterator
- std::multiset< K >::const_iterator
- std::unordered_multiset< K >::const_iterator
- std::vector< T >::const_iterator
- std::basic_string< Char >::const_iterator
- std::string::const_iterator
- std::wstring::const_iterator
- std::deque< T >::const_iterator
- std::list< T >::const_iterator
- std::forward_list< T >::const_iterator
- std::map< K, T >::const_iterator
- std::unordered_map< K, T >::const_iterator
- std::multimap< K, T >::const_reverse_iterator
- std::unordered_multimap< K, T >::const_reverse_iterator
- std::set< K >::const_reverse_iterator
- std::unordered_set< K >::const_reverse_iterator
- std::multiset< K >::const_reverse_iterator

std::basic_ifstream
std::ifstream
std::wifstream
std::basic_iostream
basic_fstream< char >
basic_fstream< wchar_t >
basic_stringstream< char >
basic_stringstream< wchar_t >
std::basic_fstream
std::fstream
std::wfstream
std::basic_stringstream
std::stringstream
std::wstringstream
std::basic_istringstream
std::istringstream
std::wistringstream
std::istream
std::wistream
std::basic_ostream
basic_iostream< char >
basic_iostream< wchar_t >
basic_ofstream< char >
basic_ofstream< wchar_t >
basic_ostringstream< char >
basic_ostringstream< wchar_t >
std::basic_iostream
std::basic_ofstream
std::ofstream
std::wofstream
std::basic_ostringstream
std::ostringstream
std::wostringstream
std::ostream
std::wostream
std::ios
std::wios
OsiCuts::iterator [external]
std::unordered_multimap< K, T >::iterator
std::set< K >::iterator
std::forward_list< T >::iterator
std::unordered_multiset< K >::iterator
std::list< T >::iterator
std::multiset< K >::iterator
std::deque< T >::iterator
std::basic_string< Char >::iterator
std::unordered_set< K >::iterator
std::string::iterator
std::vector< T >::iterator
std::multimap< K, T >::iterator
std::wstring::iterator
std::map< K, T >::iterator
std::unordered_map< K, T >::iterator
std::list< T >

std::wstring::reverse_iterator
std::vector< T >::reverse_iterator
std::map< K, T >::reverse_iterator
std::multimap< K, T >::reverse_iterator
std::string::reverse_iterator
std::unordered_set< K >::reverse_iterator
scatterStruct 341
std::set< K >
std::smart_ptr< T >
Coin::SmartPtr< T > [external]
std::stack< T >
symrec [external]
std::system_error
OsiUnitTest::TestOutcome [external]
OsiUnitTest::TestOutcomes [external]
std::thread
std::unique_ptr< T >
std::unordered_map< K, T >
std::unordered_multimap< K, T >
std::unordered_multiset< K >
std::unordered_set< K >
std::valarray< T >
std::vector< T >
std::vector< double >
std::vector< std::string >
std::weak_ptr< T >
K
S
T
U

Chapter 2

Class Index

2.1 Class List

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

Chapter 3

File Index

3.1 File List

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

Chapter 4

Class Documentation

4.1 AbcDualRowDantzig Class Reference

Dual Row Pivot Dantzig Algorithm Class.

`#include <AbcDualRowDantzig.hpp>`

Inheritance diagram for AbcDualRowDantzig:

4.2 AbcDualRowPivot Class Reference

Dual Row Pivot Abstract Base Class.

`#include <AbcDualRowPivot.hpp>`

Inheritance diagram for AbcDualRowPivot:

Collaboration diagram for AbcDualRowPivot:

Public Member Functions

Algorithmic methods

- virtual int pivotRow ()=0

  Returns pivot row, -1 if none.
- virtual double updateWeights1 (**CoinIndexedVector** &input, **CoinIndexedVector** &updateColumn)=0

  Does most of work for weights and returns pivot alpha.
- virtual void **updateWeightsOnly** (**CoinIndexedVector** &input)=0
- virtual double **updateWeights** (**CoinIndexedVector** &input, **CoinIndexedVector** &updateColumn)=0
- virtual void updateWeights2 (**CoinIndexedVector** &input, **CoinIndexedVector** &updateColumn)=0

  Actually updates weights.
- virtual void updatePrimalSolution (**CoinIndexedVector** &updateColumn, double theta)=0

  Updates primal solution (and maybe list of candidates) Uses input vector which it deletes Would be faster if we kept basic regions, but on other hand it means everything is always in sync.
- virtual void **updatePrimalSolutionAndWeights** (**CoinIndexedVector** &weightsVector, **CoinIndexedVector** &updateColumn, double theta)
- virtual void saveWeights (AbcSimplex ∗model, int mode)

Saves any weights round factorization as pivot rows may change Will be empty unless steepest edge (will save model) May also recompute infeasibility stuff 1) before factorization 2) after good factorization (if weights empty may initialize) 3) after something happened but no factorization (e.g.

- virtual void recomputeInfeasibilities ()

  Recompute infeasibilities.
- virtual void checkAccuracy ()

  checks accuracy and may re-initialize (may be empty)
- virtual void clearArrays ()

  Gets rid of all arrays (may be empty)
- virtual bool looksOptimal () const

  Returns true if would not find any row.

Constructors and destructors

- AbcDualRowPivot ()

  Default Constructor.
- AbcDualRowPivot (const AbcDualRowPivot &)

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

  Assignment operator.
- virtual ∼AbcDualRowPivot ()

  Destructor.
- virtual AbcDualRowPivot ∗ clone (bool copyData=true) const =0

  Clone.

Other

- AbcSimplex ∗ model ()

  Returns model.
- void setModel (AbcSimplex ∗newmodel)

  Sets model (normally to NULL)
- int type ()

  Returns type (above 63 is extra information)

Protected Attributes

Protected member data

- AbcSimplex ∗ model_

  Pointer to model.
- int type_

  Type of row pivot algorithm.

4.2.1 Detailed Description

Dual Row Pivot Abstract Base Class.

Abstract Base Class for describing an interface to an algorithm to choose row pivot in dual simplex algorithm. For some algorithms e.g. Dantzig choice then some functions may be null.

Definition at line 23 of file AbcDualRowPivot.hpp.

4.2.2 Member Function Documentation

4.2.2.1 virtual double AbcDualRowPivot::updateWeights1 (**CoinIndexedVector** & *input,* **CoinIndexedVector** & *updateColumn*) [pure virtual]

Does most of work for weights and returns pivot alpha.

Also does FT update

Implemented in AbcDualRowSteepest, and AbcDualRowDantzig.

4.2.2.2 virtual void AbcDualRowPivot::saveWeights (**AbcSimplex** * *model,* int *mode*) [virtual]

Saves any weights round factorization as pivot rows may change Will be empty unless steepest edge (will save model) May also recompute infeasibility stuff 1) before factorization 2) after good factorization (if weights empty may initialize) 3) after something happened but no factorization (e.g.

check for infeasible) 4) as 2 but restore weights from previous snapshot 5) for strong branching - initialize , infeasibilities

Reimplemented in AbcDualRowSteepest, and AbcDualRowDantzig.

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

- AbcDualRowPivot.hpp

4.3 AbcDualRowSteepest Class Reference

Dual Row Pivot Steepest Edge Algorithm Class.

`#include <AbcDualRowSteepest.hpp>`

Inheritance diagram for AbcDualRowSteepest:

Collaboration diagram for AbcDualRowSteepest:

Public Types

- enum Persistence

  enums for persistence

Public Member Functions

Algorithmic methods

- virtual int pivotRow ()

  Returns pivot row, -1 if none.
- virtual double updateWeights (**CoinIndexedVector** &input, **CoinIndexedVector** &updatedColumn)

  Updates weights and returns pivot alpha.
- virtual double updateWeights1 (**CoinIndexedVector** &input, **CoinIndexedVector** &updateColumn)

  Does most of work for weights and returns pivot alpha.
- virtual void **updateWeightsOnly** (**CoinIndexedVector** &input)
- virtual void updateWeights2 (**CoinIndexedVector** &input, **CoinIndexedVector** &updateColumn)

  Actually updates weights.
- virtual void updatePrimalSolution (**CoinIndexedVector** &input, double theta)

Updates primal solution (and maybe list of candidates) Uses input vector which it deletes.

- virtual void **updatePrimalSolutionAndWeights** (**CoinIndexedVector** &weightsVector, **CoinIndexedVector** &updateColumn, double theta)
- virtual void saveWeights (AbcSimplex *model, int mode)

  Saves any weights round factorization as pivot rows may change Save model May also recompute infeasibility stuff 1) before factorization 2) after good factorization (if weights empty may initialize) 3) after something happened but no factorization (e.g.
- virtual void recomputeInfeasibilities ()

  Recompute infeasibilities.
- virtual void clearArrays ()

  Gets rid of all arrays.
- virtual bool looksOptimal () const

  Returns true if would not find any row.

Constructors and destructors

- AbcDualRowSteepest (int mode=3)

  Default Constructor 0 is uninitialized, 1 full, 2 is partial uninitialized, 3 starts as 2 but may switch to 1.
- AbcDualRowSteepest (const AbcDualRowSteepest &)

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

  Assignment operator.
- void fill (const AbcDualRowSteepest &rhs)

  Fill most values.
- virtual ~AbcDualRowSteepest ()

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

  Clone.

gets and sets

- int mode () const

  Mode.
- void setPersistence (Persistence life)

  Set/ get persistence.
- Persistence **persistence** () const
- **CoinIndexedVector** * infeasible () const

  Infeasible vector.
- **CoinIndexedVector** * weights () const

  Weights vector.
- AbcSimplex * model () const

  Model.

Additional Inherited Members

4.3.1 Detailed Description

Dual Row Pivot Steepest Edge Algorithm Class.

See Forrest-Goldfarb paper for algorithm

Definition at line 21 of file AbcDualRowSteepest.hpp.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 AbcDualRowSteepest::AbcDualRowSteepest (int *mode* = 3)

Default Constructor 0 is uninitialized, 1 full, 2 is partial uninitialized, 3 starts as 2 but may switch to 1.

By partial is meant that the weights are updated as normal but only part of the infeasible basic variables are scanned. This can be faster on very easy problems.

4.3.3 Member Function Documentation

4.3.3.1 virtual double AbcDualRowSteepest::updateWeights (CoinIndexedVector & *input,* CoinIndexedVector & *updatedColumn*) [virtual]

Updates weights and returns pivot alpha.

Also does FT update

Implements AbcDualRowPivot.

4.3.3.2 virtual double AbcDualRowSteepest::updateWeights1 (CoinIndexedVector & *input,* CoinIndexedVector & *updateColumn*) [virtual]

Does most of work for weights and returns pivot alpha.

Also does FT update

Implements AbcDualRowPivot.

4.3.3.3 virtual void AbcDualRowSteepest::saveWeights (AbcSimplex * *model,* int *mode*) [virtual]

Saves any weights round factorization as pivot rows may change Save model May also recompute infeasibility stuff 1) before factorization 2) after good factorization (if weights empty may initialize) 3) after something happened but no factorization (e.g.

check for infeasible) 4) as 2 but restore weights from previous snapshot 5) for strong branching - initialize (uninitialized) , infeasibilities

Reimplemented from AbcDualRowPivot.

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

- AbcDualRowSteepest.hpp

4.4 AbcMatrix Class Reference

Collaboration diagram for AbcMatrix:

Public Member Functions

Useful methods

- **CoinPackedMatrix** * getPackedMatrix () const

Return a complete **CoinPackedMatrix***.*

- bool isColOrdered () const

  Whether the packed matrix is column major ordered or not.
- CoinBigIndex getNumElements () const

  Number of entries in the packed matrix.
- int getNumCols () const

  Number of columns.
- int getNumRows () const

  Number of rows.
- void setModel (AbcSimplex ∗model)

  Sets model.
- const double ∗ getElements () const

  A vector containing the elements in the packed matrix.
- double ∗ getMutableElements () const

  Mutable elements.
- const int ∗ getIndices () const

  A vector containing the minor indices of the elements in the packed matrix.
- int ∗ getMutableIndices () const

  A vector containing the minor indices of the elements in the packed matrix.
- const CoinBigIndex ∗ getVectorStarts () const

  Starts.
- CoinBigIndex ∗ **getMutableVectorStarts** () const
- const int ∗ getVectorLengths () const

  The lengths of the major-dimension vectors.
- int ∗ getMutableVectorLengths () const

  The lengths of the major-dimension vectors.
- CoinBigIndex ∗ rowStart () const

  Row starts.
- CoinBigIndex ∗ rowEnd () const

  Row ends.
- double ∗ rowElements () const

  Row elements.
- CoinSimplexInt ∗ rowColumns () const

  Row columns.
- **CoinPackedMatrix** ∗ reverseOrderedCopy () const

  Returns a new matrix in reverse order without gaps.
- CoinBigIndex countBasis (const int ∗whichColumn, int &numberColumnBasic)

  Returns number of elements in column part of basis.
- void fillBasis (const int ∗whichColumn, int &numberColumnBasic, int ∗row, int ∗start, int ∗rowCount, int ∗columnCount, CoinSimplexDouble ∗element)

  Fills in column part of basis.
- void fillBasis (const int ∗whichColumn, int &numberColumnBasic, int ∗row, int ∗start, int ∗rowCount, int ∗columnCount, long double ∗element)

  Fills in column part of basis.
- void scale (int numberRowsAlreadyScaled)

  Scales and creates row copy.
- void createRowCopy ()

  Creates row copy.
- void takeOutOfUseful (int sequence, **CoinIndexedVector** &spare)

  Take out of useful.
- void putIntofUseful (int sequence, **CoinIndexedVector** &spare)

  Put into useful.

- void inOutUseful (int sequenceIn, int sequenceOut)

  Put in and out for useful.
- void makeAllUseful (**CoinIndexedVector** &spare)

  Make all useful.
- void sortUseful (**CoinIndexedVector** &spare)

  Sort into useful.
- void moveLargestToStart ()

  Move largest in column to beginning (not used as doesn't help factorization)
- void unpack (**CoinIndexedVector** &rowArray, int column) const

  Unpacks a column into an **CoinIndexedVector**.
- void add (**CoinIndexedVector** &rowArray, int column, double multiplier) const

  Adds multiple of a column (or slack) into an CoinIndexedvector You can use quickAdd to add to vector.

Matrix times vector methods

- void timesModifyExcludingSlacks (double scalar, const double ∗x, double ∗y) const

  Return `y + A * scalar *x` *in* y.
- void timesModifyIncludingSlacks (double scalar, const double ∗x, double ∗y) const

  Return `y + A * scalar(+-1) *x` *in* y.
- void timesIncludingSlacks (double scalar, const double ∗x, double ∗y) const

  Return `A * scalar(+-1) *x` *in* y.
- void transposeTimesNonBasic (double scalar, const double ∗x, double ∗y) const

  Return A ∗ scalar(+-1) ∗x + y in y.
- void transposeTimesAll (const double ∗x, double ∗y) const

  Return y - A ∗ x in y.
- void transposeTimesBasic (double scalar, const double ∗x, double ∗y) const

  Return y + A ∗ scalar(+-1) ∗x in y.
- int transposeTimesNonBasic (double scalar, const **CoinIndexedVector** &x, **CoinIndexedVector** &z) const

  Return `x * scalar * A/code> in z.`
- double dualColumn1 (const **CoinIndexedVector** &update, **CoinPartitionedVector** &tableauRow, **CoinPartitionedVector** &candidateList) const

  gets sorted tableau row and a possible value of theta
- double dualColumn1Row (int iBlock, double upperThetaSlack, int &freeSequence, const **CoinIndexedVector** &update, **CoinPartitionedVector** &tableauRow, **CoinPartitionedVector** &candidateList) const

  gets sorted tableau row and a possible value of theta
- double dualColumn1RowFew (int iBlock, double upperThetaSlack, int &freeSequence, const **CoinIndexedVector** &update, **CoinPartitionedVector** &tableauRow, **CoinPartitionedVector** &candidateList) const

  gets sorted tableau row and a possible value of theta
- double dualColumn1Row2 (double upperThetaSlack, int &freeSequence, const **CoinIndexedVector** &update, **CoinPartitionedVector** &tableauRow, **CoinPartitionedVector** &candidateList) const

  gets sorted tableau row and a possible value of theta
- double dualColumn1Row1 (double upperThetaSlack, int &freeSequence, const **CoinIndexedVector** &update, **CoinPartitionedVector** &tableauRow, **CoinPartitionedVector** &candidateList) const

  gets sorted tableau row and a possible value of theta
- void dualColumn1Part (int iBlock, int &sequenceIn, double &upperTheta, const **CoinIndexedVector** &update, **CoinPartitionedVector** &tableauRow, **CoinPartitionedVector** &candidateList) const

  gets sorted tableau row and a possible value of theta On input first,last give what to scan On output is number in tableauRow and candidateList
- void rebalance () const

  rebalance for parallel
- int pivotColumnDantzig (const **CoinIndexedVector** &updates, **CoinPartitionedVector** &spare) const

  Get sequenceIn when Dantzig.

- int pivotColumnDantzig (int iBlock, bool doByRow, const **CoinIndexedVector** &updates, **CoinPartitionedVector** &spare, double &bestValue) const

    Get sequenceIn when Dantzig (One block)
- int primalColumnRow (int iBlock, bool doByRow, const **CoinIndexedVector** &update, **CoinPartitionedVector** &tableauRow) const

    gets tableau row - returns number of slacks in block
- int primalColumnRowAndDjs (int iBlock, const **CoinIndexedVector** &updateTableau, const **CoinIndexedVector** &updateDjs, **CoinPartitionedVector** &tableauRow) const

    gets tableau row and dj row - returns number of slacks in block
- int chooseBestDj (int iBlock, const **CoinIndexedVector** &infeasibilities, const double ∗weights) const

    Chooses best weighted dj.
- int primalColumnDouble (int iBlock, **CoinPartitionedVector** &updateForTableauRow, **CoinPartitionedVector** &updateForDjs, const **CoinIndexedVector** &updateForWeights, **CoinPartitionedVector** &spareColumn1, double ∗infeasibilities, double referenceIn, double devex, unsigned int ∗reference, double ∗weights, double scaleFactor) const

    does steepest edge double or triple update If scaleFactor!=0 then use with tableau row to update djs otherwise use updateForDjs Returns best sequence
- int primalColumnSparseDouble (int iBlock, **CoinPartitionedVector** &updateForTableauRow, **CoinPartitionedVector** &updateForDjs, const **CoinIndexedVector** &updateForWeights, **CoinPartitionedVector** &spareColumn1, double ∗infeasibilities, double referenceIn, double devex, unsigned int ∗reference, double ∗weights, double scaleFactor) const

    does steepest edge double or triple update If scaleFactor!=0 then use with tableau row to update djs otherwise use updateForDjs Returns best sequence
- int primalColumnDouble (**CoinPartitionedVector** &updateForTableauRow, **CoinPartitionedVector** &updateForDjs, const **CoinIndexedVector** &updateForWeights, **CoinPartitionedVector** &spareColumn1, **CoinIndexedVector** &infeasible, double referenceIn, double devex, unsigned int ∗reference, double ∗weights, double scaleFactor) const

    does steepest edge double or triple update If scaleFactor!=0 then use with tableau row to update djs otherwise use updateForDjs Returns best sequence
- void primalColumnSubset (int iBlock, const **CoinIndexedVector** &update, const **CoinPartitionedVector** &tableauRow, **CoinPartitionedVector** &weights) const

    gets subset updates
- void partialPricing (double startFraction, double endFraction, int &bestSequence, int &numberWanted)

    Partial pricing.
- void subsetTransposeTimes (const **CoinIndexedVector** &x, **CoinIndexedVector** &z) const

    Return $x * A$ in z but just for indices Already in z.
- void transposeTimes (const **CoinIndexedVector** &x, **CoinIndexedVector** &z) const

    Return $-x * A$ in z

Other

- **CoinPackedMatrix** ∗ matrix () const

    Returns **CoinPackedMatrix** *(non const)*
- int minimumObjectsScan () const

    Partial pricing tuning parameter - minimum number of "objects" to scan.
- void **setMinimumObjectsScan** (int value)
- int minimumGoodReducedCosts () const

    Partial pricing tuning parameter - minimum number of negative reduced costs to get.
- void **setMinimumGoodReducedCosts** (int value)
- double startFraction () const

    Current start of search space in matrix (as fraction)
- void **setStartFraction** (double value)
- double endFraction () const

Current end of search space in matrix (as fraction)
- void **setEndFraction** (double value)
- double savedBestDj () const

  Current best reduced cost.
- void **setSavedBestDj** (double value)
- int originalWanted () const

  Initial number of negative reduced costs wanted.
- void **setOriginalWanted** (int value)
- int currentWanted () const

  Current number of negative reduced costs which we still need.
- void **setCurrentWanted** (int value)
- int savedBestSequence () const

  Current best sequence.
- void **setSavedBestSequence** (int value)
- int ∗ startColumnBlock () const

  Start of each column block.
- const int ∗ blockStart () const

  Start of each block (in stored)
- bool **gotRowCopy** () const
- int blockStart (int block) const

  Start of each block (in stored)
- int numberColumnBlocks () const

  Number of actual column blocks.
- int numberRowBlocks () const

  Number of actual row blocks.

Constructors, destructor

- AbcMatrix ()

  Default constructor.
- ∼AbcMatrix ()

  Destructor.

Copy method

- AbcMatrix (const AbcMatrix &)

  The copy constructor.
- AbcMatrix (const **CoinPackedMatrix** &)

  The copy constructor from an **CoinPackedMatrix**.
- AbcMatrix (const AbcMatrix &wholeModel, int numberRows, const int ∗whichRows, int numberColumns, const int ∗whichColumns)

  Subset constructor (without gaps).
- **AbcMatrix** (const **CoinPackedMatrix** &wholeModel, int numberRows, const int ∗whichRows, int number←Columns, const int ∗whichColumns)
- AbcMatrix & **operator=** (const AbcMatrix &)
- void copy (const AbcMatrix ∗from)

  Copy contents - resizing if necessary - otherwise re-use memory.

Protected Attributes

Data members

The data members are protected to allow access for derived classes.

- **CoinPackedMatrix** * matrix_

  Data.
- AbcSimplex * model_

  Model.
- CoinBigIndex * rowStart_

  Start of each row (per block) - last lot are useless first all row starts for block 0, then for block2 so NUMBER_ROW_↩ BLOCKS+2 times number rows.
- double * element_

  Values by row.
- int * column_

  Columns.
- int startColumnBlock_ [NUMBER_COLUMN_BLOCKS+1]

  Start of each column block.
- int blockStart_ [NUMBER_ROW_BLOCKS+1]

  Start of each block (in stored)
- int numberColumnBlocks_

  Number of actual column blocks.
- int numberRowBlocks_

  Number of actual row blocks.
- double startFraction_

  Special row copy.
- double endFraction_

  Current end of search space in matrix (as fraction)
- double savedBestDj_

  Best reduced cost so far.
- int originalWanted_

  Initial number of negative reduced costs wanted.
- int currentWanted_

  Current number of negative reduced costs which we still need.
- int savedBestSequence_

  Saved best sequence in pricing.
- int minimumObjectsScan_

  Partial pricing tuning parameter - minimum number of "objects" to scan.
- int minimumGoodReducedCosts_

  Partial pricing tuning parameter - minimum number of negative reduced costs to get.

4.4.1 Detailed Description

Definition at line 22 of file AbcMatrix.hpp.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 AbcMatrix::AbcMatrix ( )

Default constructor.

4.4.2.2 AbcMatrix::AbcMatrix (const AbcMatrix &)

The copy constructor.

4.4.2.3 AbcMatrix::AbcMatrix (const CoinPackedMatrix &)

The copy constructor from an **CoinPackedMatrix**.

4.4.2.4 AbcMatrix::AbcMatrix (const AbcMatrix & *wholeModel*, int *numberRows*, const int * *whichRows*, int *numberColumns*, const int * *whichColumns*)

Subset constructor (without gaps).

Duplicates are allowed and order is as given

4.4.3 Member Function Documentation

4.4.3.1 bool AbcMatrix::isColOrdered ( ) const [inline]

Whether the packed matrix is column major ordered or not.

Definition at line 32 of file AbcMatrix.hpp.

4.4.3.2 CoinBigIndex AbcMatrix::getNumElements ( ) const [inline]

Number of entries in the packed matrix.

Definition at line 36 of file AbcMatrix.hpp.

4.4.3.3 int AbcMatrix::getNumCols ( ) const [inline]

Number of columns.

Definition at line 40 of file AbcMatrix.hpp.

4.4.3.4 int AbcMatrix::getNumRows ( ) const [inline]

Number of rows.

Definition at line 44 of file AbcMatrix.hpp.

4.4.3.5 const int* AbcMatrix::getVectorLengths ( ) const [inline]

The lengths of the major-dimension vectors.

Definition at line 73 of file AbcMatrix.hpp.

4.4.3.6 int* AbcMatrix::getMutableVectorLengths ( ) const [inline]

The lengths of the major-dimension vectors.

Definition at line 77 of file AbcMatrix.hpp.

4.4.3.7 void AbcMatrix::timesModifyExcludingSlacks (double *scalar*, const double * *x*, double * *y*) const

Return `y + A * scalar *x` in `y`.

Precondition

`x` must be of size `numColumns()`
`y` must be of size `numRows()`

4.4.3.8 void AbcMatrix::timesModifyIncludingSlacks (double *scalar*, const double * *x*, double * *y*) const

Return `y + A * scalar(+-1) *x` in `y`.

Precondition

`x` must be of size `numColumns()+numRows()`
`y` must be of size `numRows()`

4.4.3.9 void AbcMatrix::timesIncludingSlacks (double *scalar*, const double * *x*, double * *y*) const

Return `A * scalar(+-1) *x` in `y`.

Precondition

`x` must be of size `numColumns()+numRows()`
`y` must be of size `numRows()`

4.4.3.10 void AbcMatrix::transposeTimesNonBasic (double *scalar*, const double * *x*, double * *y*) const

Return A * scalar(+-1) *x + y in `y`.

Precondition

`x` must be of size `numRows()`
`y` must be of size `numRows()+numColumns()`

4.4.3.11 void AbcMatrix::transposeTimesAll (const double * *x*, double * *y*) const

Return y - A * x in `y`.

Precondition

`x` must be of size `numRows()`
`y` must be of size `numRows()+numColumns()`

4.4.3.12 void AbcMatrix::transposeTimesBasic (double *scalar*, const double * *x*, double * *y*) const

Return y + A * scalar(+-1) *x in `y`.

Precondition

`x` must be of size `numRows()`
`y` must be of size `numRows()`

4.4.3.13 int AbcMatrix::transposeTimesNonBasic (double *scalar*, const CoinIndexedVector & *x*, CoinIndexedVector & *z*) const

Return `x * scalar * A/code> in z.`

Note - x unpacked mode - z packed mode including slacks All these return atLo/atUp first then free/superbasic number of first set returned pivotVariable is extended to have that order reversePivotVariable used to update that list free/superbasic only stored in normal format can use spare array to get this effect may put djs alongside atLo/atUp Squashes small elements and knows about AbcSimplex

4.4.3.14 void AbcMatrix::subsetTransposeTimes (const CoinIndexedVector & *x*, CoinIndexedVector & *z*) const

Return `x *A` in `z` but just for indices Already in z.

Note - z always packed mode

4.4.3.15 int AbcMatrix::minimumObjectsScan ( ) const [inline]

Partial pricing tuning parameter - minimum number of "objects" to scan.

e.g. number of Gub sets but could be number of variables

Definition at line 294 of file AbcMatrix.hpp.

4.4.4 Member Data Documentation

4.4.4.1 double AbcMatrix::startFraction_ [protected]

Special row copy.

Special column copy Current start of search space in matrix (as fraction)

Definition at line 453 of file AbcMatrix.hpp.

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

- AbcMatrix.hpp

4.5 AbcMatrix2 Class Reference

Public Member Functions

Useful methods

- void transposeTimes (const AbcSimplex *model, const **CoinPackedMatrix** *rowCopy, const **CoinIndexedVector** &x, **CoinIndexedVector** &spareArray, **CoinIndexedVector** &z) const

    Return `x * -1 * A in z.`
- bool usefulInfo () const

    Returns true if copy has useful information.

Constructors, destructor

- AbcMatrix2 ()

Default constructor.
- AbcMatrix2 (AbcSimplex ∗model, const **CoinPackedMatrix** ∗rowCopy)

  Constructor from copy.
- ∼AbcMatrix2 ()

  Destructor.

Copy method

- AbcMatrix2 (const AbcMatrix2 &)

  The copy constructor.
- AbcMatrix2 & **operator=** (const AbcMatrix2 &)

Protected Attributes

Data members

The data members are protected to allow access for derived classes.

- int numberBlocks_

  Number of blocks.
- int numberRows_

  Number of rows.
- int ∗ offset_

  Column offset for each block (plus one at end)
- unsigned short ∗ count_

  Counts of elements in each part of row.
- CoinBigIndex ∗ rowStart_

  Row starts.
- unsigned short ∗ column_

  columns within block
- double ∗ work_

  work arrays

4.5.1 Detailed Description

Definition at line 495 of file AbcMatrix.hpp.

4.5.2 Constructor & Destructor Documentation

4.5.2.1 AbcMatrix2::AbcMatrix2 ( )

Default constructor.

4.5.2.2 AbcMatrix2::AbcMatrix2 (AbcSimplex ∗ *model*, const CoinPackedMatrix ∗ *rowCopy*)

Constructor from copy.

4.5.2.3 AbcMatrix2::AbcMatrix2 (const AbcMatrix2 &)

The copy constructor.

4.5.3 Member Function Documentation

4.5.3.1 void AbcMatrix2::transposeTimes (const AbcSimplex * *model*, const CoinPackedMatrix * *rowCopy*, const CoinIndexedVector & *x*, CoinIndexedVector & *spareArray*, CoinIndexedVector & *z*) const

Return `x * -1 * A in z.`

Note - x packed and z will be packed mode Squashes small elements and knows about AbcSimplex

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

- AbcMatrix.hpp

4.6 AbcMatrix3 Class Reference

Collaboration diagram for AbcMatrix3:

Public Member Functions

Useful methods

- void transposeTimes (const AbcSimplex *model, const double *pi, **CoinIndexedVector** &output) const
  Return `x * -1 * A in z.`
- void transposeTimes2 (const AbcSimplex *model, const double *pi, **CoinIndexedVector** &dj1, const double *piWeight, double referenceIn, double devex, unsigned int *reference, double *weights, double scaleFactor)
  Updates two arrays for steepest.

Constructors, destructor

- AbcMatrix3 ()
  Default constructor.
- AbcMatrix3 (AbcSimplex *model, const **CoinPackedMatrix** *columnCopy)
  Constructor from copy.
- ∼AbcMatrix3 ()
  Destructor.

Copy method

- AbcMatrix3 (const AbcMatrix3 &)
  The copy constructor.
- AbcMatrix3 & **operator=** (const AbcMatrix3 &)

Sort methods

- void sortBlocks (const AbcSimplex *model)
  Sort blocks.
- void swapOne (const AbcSimplex *model, const AbcMatrix *matrix, int iColumn)
  Swap one variable.

Protected Attributes

Data members

The data members are protected to allow access for derived classes.

- int numberBlocks_

  Number of blocks.
- int numberColumns_

  Number of columns.
- int * column_

  Column indices and reverse lookup (within block)
- CoinBigIndex * start_

  Starts for odd/long vectors.
- int * row_

  Rows.
- double * element_

  Elements.
- blockStruct * block_

  Blocks (ordinary start at 0 and go to first block)

4.6.1 Detailed Description

Definition at line 564 of file AbcMatrix.hpp.

4.6.2 Constructor & Destructor Documentation

4.6.2.1 AbcMatrix3::AbcMatrix3 ( )

Default constructor.

4.6.2.2 AbcMatrix3::AbcMatrix3 (**AbcSimplex** * *model*, const **CoinPackedMatrix** * *columnCopy*)

Constructor from copy.

4.6.2.3 AbcMatrix3::AbcMatrix3 (const **AbcMatrix3** &)

The copy constructor.

4.6.3 Member Function Documentation

4.6.3.1 void AbcMatrix3::transposeTimes (const **AbcSimplex** * *model*, const double * *pi*, **CoinIndexedVector** & *output*) const

Return `x * -1 * A in z.`

Note - x packed and z will be packed mode Squashes small elements and knows about AbcSimplex

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

- AbcMatrix.hpp

4.7 AbcNonLinearCost Class Reference

Public Member Functions

Constructors, destructor

- AbcNonLinearCost ()
  Default constructor.
- AbcNonLinearCost (AbcSimplex ∗model)
  Constructor from simplex.
- ∼AbcNonLinearCost ()
  Destructor.
- **AbcNonLinearCost** (const AbcNonLinearCost &)
- AbcNonLinearCost & **operator=** (const AbcNonLinearCost &)

Actual work in primal

- void checkInfeasibilities (double oldTolerance=0.0)
  Changes infeasible costs and computes number and cost of infeas Puts all non-basic (non free) variables to bounds and all free variables to zero if oldTolerance is non-zero.
- void checkInfeasibilities (int numberInArray, const int ∗index)
  Changes infeasible costs for each variable The indices are row indices and need converting to sequences.
- void checkChanged (int numberInArray, **CoinIndexedVector** ∗update)
  Puts back correct infeasible costs for each variable The input indices are row indices and need converting to sequences for costs.
- void goThru (int numberInArray, double multiplier, const int ∗index, const double ∗work, double ∗rhs)
  Goes through one bound for each variable.
- void goBack (int numberInArray, const int ∗index, double ∗rhs)
  Takes off last iteration (i.e.
- void goBackAll (const **CoinIndexedVector** ∗update)
  Puts back correct infeasible costs for each variable The input indices are row indices and need converting to sequences for costs.
- void zapCosts ()
  Temporary zeroing of feasible costs.
- void refreshCosts (const double ∗columnCosts)
  Refreshes costs always makes row costs zero.
- void feasibleBounds ()
  Puts feasible bounds into lower and upper.
- void refresh ()
  Refresh - assuming regions OK.
- void refreshFromPerturbed (double tolerance)
  Refresh - from original.
- double setOne (int sequence, double solutionValue)
  Sets bounds and cost for one variable Returns change in cost May need to be inline for speed.
- double setOneBasic (int iRow, double solutionValue)
  Sets bounds and cost for one variable Returns change in cost May need to be inline for speed.
- int setOneOutgoing (int sequence, double &solutionValue)
  Sets bounds and cost for outgoing variable may change value Returns direction.
- double nearest (int iRow, double solutionValue)
  Returns nearest bound.
- double changeInCost (int, double alpha) const
  Returns change in cost - one down if alpha >0.0, up if <0.0 Value is current - new.
- double **changeUpInCost** (int) const

- double **changeDownInCost** (int) const
- double changeInCost (int iRow, double alpha, double &rhs)

    This also updates next bound.

Gets and sets

- int numberInfeasibilities () const

    Number of infeasibilities.
- double changeInCost () const

    Change in cost.
- double feasibleCost () const

    Feasible cost.
- double feasibleReportCost () const

    Feasible cost with offset and direction (i.e. for reporting)
- double sumInfeasibilities () const

    Sum of infeasibilities.
- double largestInfeasibility () const

    Largest infeasibility.
- double averageTheta () const

    Average theta.
- void **setAverageTheta** (double value)
- void **setChangeInCost** (double value)

Private functions to deal with infeasible regions

- unsigned char * **statusArray** () const
- int **getCurrentStatus** (int sequence)
- void validate ()

    For debug.

4.7.1 Detailed Description

Definition at line 72 of file AbcNonLinearCost.hpp.

4.7.2 Constructor & Destructor Documentation

4.7.2.1 AbcNonLinearCost::AbcNonLinearCost (**AbcSimplex** * *model*)

Constructor from simplex.

This will just set up wasteful arrays for linear, but later may do dual analysis and even finding duplicate columns .

4.7.3 Member Function Documentation

4.7.3.1 void AbcNonLinearCost::checkInfeasibilities (double *oldTolerance* = `0.0`)

Changes infeasible costs and computes number and cost of infeas Puts all non-basic (non free) variables to bounds and all free variables to zero if oldTolerance is non-zero.

- but does not move those <= oldTolerance away

4.7.3.2 void AbcNonLinearCost::checkChanged (int *numberInArray,* CoinIndexedVector ∗ *update*)

Puts back correct infeasible costs for each variable The input indices are row indices and need converting to sequences for costs.

On input array is empty (but indices exist). On exit just changed costs will be stored as normal **CoinIndexedVector**

4.7.3.3 void AbcNonLinearCost::goThru (int *numberInArray,* double *multiplier,* const int ∗ *index,* const double ∗ *work,* double ∗ *rhs*)

Goes through one bound for each variable.

If multiplier∗work[iRow]>0 goes down, otherwise up. The indices are row indices and need converting to sequences Temporary offsets may be set Rhs entries are increased

4.7.3.4 void AbcNonLinearCost::goBack (int *numberInArray,* const int ∗ *index,* double ∗ *rhs*)

Takes off last iteration (i.e.

offsets closer to 0)

4.7.3.5 void AbcNonLinearCost::goBackAll (const CoinIndexedVector ∗ *update*)

Puts back correct infeasible costs for each variable The input indices are row indices and need converting to sequences for costs.

At the end of this all temporary offsets are zero

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

- AbcNonLinearCost.hpp

4.8 AbcPrimalColumnDantzig Class Reference

Primal Column Pivot Dantzig Algorithm Class.

`#include <AbcPrimalColumnDantzig.hpp>`

Inheritance diagram for AbcPrimalColumnDantzig:

Collaboration diagram for AbcPrimalColumnDantzig:

Public Member Functions

Algorithmic methods

- virtual int pivotColumn (**CoinPartitionedVector** ∗updates, **CoinPartitionedVector** ∗spareRow2, **Coin**←**PartitionedVector** ∗spareColumn1)

  Returns pivot column, -1 if none.
- virtual void saveWeights (AbcSimplex ∗model, int)

  Just sets model.

Constructors and destructors

- AbcPrimalColumnDantzig ()

  Default Constructor.
- AbcPrimalColumnDantzig (const AbcPrimalColumnDantzig &)

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

  Assignment operator.
- virtual ∼AbcPrimalColumnDantzig ()

  Destructor.
- virtual AbcPrimalColumnPivot ∗ clone (bool copyData=true) const

  Clone.

Additional Inherited Members

4.8.1 Detailed Description

Primal Column Pivot Dantzig Algorithm Class.

This is simplest choice - choose largest infeasibility

Definition at line 19 of file AbcPrimalColumnDantzig.hpp.

4.8.2 Member Function Documentation

4.8.2.1 virtual int AbcPrimalColumnDantzig::pivotColumn (**CoinPartitionedVector** ∗ *updates,* **CoinPartitionedVector** ∗ *spareRow2,* **CoinPartitionedVector** ∗ *spareColumn1*) `[virtual]`

Returns pivot column, -1 if none.

Lumbers over all columns - slow The Packed **CoinIndexedVector** updates has cost updates - for normal LP that is just +-weight where a feasibility changed. It also has reduced cost from last iteration in pivot row Can just do full price if you really want to be slow

Implements AbcPrimalColumnPivot.

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

- AbcPrimalColumnDantzig.hpp

4.9 AbcPrimalColumnPivot Class Reference

Primal Column Pivot Abstract Base Class.

`#include <AbcPrimalColumnPivot.hpp>`

Inheritance diagram for AbcPrimalColumnPivot:

Collaboration diagram for AbcPrimalColumnPivot:

Public Member Functions

Algorithmic methods

- virtual int pivotColumn (**CoinPartitionedVector** ∗updates, **CoinPartitionedVector** ∗spareRow2, **Coin←PartitionedVector** ∗spareColumn1)=0

Returns pivot column, -1 if none.
- virtual void updateWeights (**CoinIndexedVector** ∗input)
  Updates weights - part 1 (may be empty)
- virtual void saveWeights (AbcSimplex ∗model, int mode)=0
  Saves any weights round factorization as pivot rows may change Will be empty unless steepest edge (will save model) May also recompute infeasibility stuff 1) before factorization 2) after good factorization (if weights empty may initialize) 3) after something happened but no factorization (e.g.
- virtual int pivotRow (double &way)
  Signals pivot row choice: -2 (default) - use normal pivot row choice -1 to numberRows-1 - use this (will be checked) way should be -1 to go to lower bound, +1 to upper bound.
- virtual void clearArrays ()
  Gets rid of all arrays (may be empty)
- virtual bool looksOptimal () const
  Returns true if would not find any column.
- virtual void setLooksOptimal (bool flag)
  Sets optimality flag (for advanced use)

Constructors and destructors

- AbcPrimalColumnPivot ()
  Default Constructor.
- AbcPrimalColumnPivot (const AbcPrimalColumnPivot &)
  Copy constructor.
- AbcPrimalColumnPivot & operator= (const AbcPrimalColumnPivot &rhs)
  Assignment operator.
- virtual ∼AbcPrimalColumnPivot ()
  Destructor.
- virtual AbcPrimalColumnPivot ∗ clone (bool copyData=true) const =0
  Clone.

Other

- AbcSimplex ∗ model ()
  Returns model.
- void setModel (AbcSimplex ∗newmodel)
  Sets model.
- int type ()
  Returns type (above 63 is extra information)
- virtual int numberSprintColumns (int &numberIterations) const
  Returns number of extra columns for sprint algorithm - 0 means off.
- virtual void switchOffSprint ()
  Switch off sprint idea.
- virtual void maximumPivotsChanged ()
  Called when maximum pivots changes.

Protected Attributes

Protected member data

- AbcSimplex ∗ model_
  Pointer to model.
- int type_
  Type of column pivot algorithm.
- bool looksOptimal_
  Says if looks optimal (normally computed)

4.9.1 Detailed Description

Primal Column Pivot Abstract Base Class.

Abstract Base Class for describing an interface to an algorithm to choose column pivot in primal simplex algorithm. For some algorithms e.g. Dantzig choice then some functions may be null. For Dantzig the only one of any importance is pivotColumn.

If you wish to inherit from this look at AbcPrimalColumnDantzig.cpp as that is simplest version.

Definition at line 26 of file AbcPrimalColumnPivot.hpp.

4.9.2 Member Function Documentation

4.9.2.1 virtual int AbcPrimalColumnPivot::pivotColumn (CoinPartitionedVector ∗ *updates*, CoinPartitionedVector ∗ *spareRow2*, CoinPartitionedVector ∗ *spareColumn1*) [pure virtual]

Returns pivot column, -1 if none.

Normally updates reduced costs using result of last iteration before selecting incoming column.

The Packed **CoinIndexedVector** updates has cost updates - for normal LP that is just +-weight where a feasibility changed. It also has reduced cost from last iteration in pivot row

Inside pivotColumn the pivotRow_ and reduced cost from last iteration are also used.

So in the simplest case i.e. feasible we compute the row of the tableau corresponding to last pivot and add a multiple of this to current reduced costs.

We can use other arrays to help updates

Implemented in AbcPrimalColumnSteepest, and AbcPrimalColumnDantzig.

4.9.2.2 virtual void AbcPrimalColumnPivot::saveWeights (AbcSimplex ∗ *model*, int *mode*) [pure virtual]

Saves any weights round factorization as pivot rows may change Will be empty unless steepest edge (will save model) May also recompute infeasibility stuff 1) before factorization 2) after good factorization (if weights empty may initialize) 3) after something happened but no factorization (e.g.

check for infeasible) 4) as 2 but restore weights from previous snapshot 5) forces some initialization e.g. weights Also sets model

Implemented in AbcPrimalColumnSteepest, and AbcPrimalColumnDantzig.

4.9.2.3 virtual int AbcPrimalColumnPivot::numberSprintColumns (int & *numberIterations*) const [virtual]

Returns number of extra columns for sprint algorithm - 0 means off.

Also number of iterations before recompute

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

- AbcPrimalColumnPivot.hpp

4.10 AbcPrimalColumnSteepest Class Reference

Primal Column Pivot Steepest Edge Algorithm Class.

`#include <AbcPrimalColumnSteepest.hpp>`

Inheritance diagram for AbcPrimalColumnSteepest:

Collaboration diagram for AbcPrimalColumnSteepest:

Public Types

- enum Persistence

  enums for persistence

Public Member Functions

Algorithmic methods

- virtual int pivotColumn (**CoinPartitionedVector** ∗updates, **CoinPartitionedVector** ∗spareRow2, **CoinPartitionedVector** ∗spareColumn1)

  Returns pivot column, -1 if none.
- void justDjs (**CoinIndexedVector** ∗updates, **CoinIndexedVector** ∗spareColumn1)

  Just update djs.
- int partialPricing (**CoinIndexedVector** ∗updates, int numberWanted, int numberLook)

  Update djs doing partial pricing (dantzig)
- void djsAndDevex (**CoinIndexedVector** ∗updates, **CoinIndexedVector** ∗spareRow2, **CoinIndexedVector** ∗spareColumn1)

  Update djs, weights for Devex using djs.
- void djsAndDevex2 (**CoinIndexedVector** ∗updates, **CoinIndexedVector** ∗spareColumn1)

  Update djs, weights for Devex using pivot row.
- void justDevex (**CoinIndexedVector** ∗updates, **CoinIndexedVector** ∗spareColumn1)

  Update weights for Devex.
- int doSteepestWork (**CoinPartitionedVector** ∗updates, **CoinPartitionedVector** ∗spareRow2, **CoinPartitionedVector** ∗spareColumn1, int type)

  Does steepest work type - 0 - just djs 1 - just steepest 2 - both using scaleFactor 3 - both using extra array.
- virtual void updateWeights (**CoinIndexedVector** ∗input)

  Updates weights - part 1 - also checks accuracy.
- void checkAccuracy (int sequence, double relativeTolerance, **CoinIndexedVector** ∗rowArray1)

  Checks accuracy - just for debug.
- void initializeWeights ()

  Initialize weights.
- virtual void saveWeights (AbcSimplex ∗model, int mode)

  Save weights - this may initialize weights as well mode is - 1) before factorization 2) after factorization 3) just redo infeasibilities 4) restore weights 5) at end of values pass (so need initialization)
- virtual void unrollWeights ()

  Gets rid of last update.
- virtual void clearArrays ()

  Gets rid of all arrays.
- virtual bool looksOptimal () const

  Returns true if would not find any column.
- virtual void maximumPivotsChanged ()

  Called when maximum pivots changes.

gets and sets

- int mode () const
  Mode.

Constructors and destructors

- AbcPrimalColumnSteepest (int mode=3)
  Default Constructor 0 is exact devex, 1 full steepest, 2 is partial exact devex 3 switches between 0 and 2 depending on factorization 4 starts as partial dantzig/devex but then may switch between 0 and 2.
- AbcPrimalColumnSteepest (const AbcPrimalColumnSteepest &rhs)
  Copy constructor.
- AbcPrimalColumnSteepest & operator= (const AbcPrimalColumnSteepest &rhs)
  Assignment operator.
- virtual ∼AbcPrimalColumnSteepest ()
  Destructor.
- virtual AbcPrimalColumnPivot ∗ clone (bool copyData=true) const
  Clone.

Private functions to deal with devex

- bool reference (int i) const
  reference would be faster using AbcSimplex's status_, but I prefer to keep modularity.
- void **setReference** (int i, bool trueFalse)
- void setPersistence (Persistence life)
  Set/ get persistence.
- Persistence **persistence** () const

Additional Inherited Members

4.10.1 Detailed Description

Primal Column Pivot Steepest Edge Algorithm Class.

See Forrest-Goldfarb paper for algorithm

Definition at line 23 of file AbcPrimalColumnSteepest.hpp.

4.10.2 Constructor & Destructor Documentation

4.10.2.1 AbcPrimalColumnSteepest::AbcPrimalColumnSteepest (int *mode =* 3)

Default Constructor 0 is exact devex, 1 full steepest, 2 is partial exact devex 3 switches between 0 and 2 depending on factorization 4 starts as partial dantzig/devex but then may switch between 0 and 2.

By partial exact devex is meant that the weights are updated as normal but only part of the nonbasic variables are scanned. This can be faster on very easy problems.

4.10.3 Member Function Documentation

4.10.3.1 virtual int AbcPrimalColumnSteepest::pivotColumn (CoinPartitionedVector ∗ *updates,* CoinPartitionedVector ∗ *spareRow2,* CoinPartitionedVector ∗ *spareColumn1*) [virtual]

Returns pivot column, -1 if none.

The Packed **CoinIndexedVector** updates has cost updates - for normal LP that is just +-weight where a feasibility changed. It also has reduced cost from last iteration in pivot row Parts of operation split out into separate functions for profiling and speed

Implements AbcPrimalColumnPivot.

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

- AbcPrimalColumnSteepest.hpp

4.11 AbcSimplex Class Reference

Inheritance diagram for AbcSimplex:

Collaboration diagram for AbcSimplex:

Public Types

- enum Status

  enums for status of various sorts.

Public Member Functions

- void defaultFactorizationFrequency ()

  If user left factorization frequency then compute.

Constructors and destructor and copy

- AbcSimplex (bool emptyMessages=false)

  Default constructor.
- AbcSimplex (const AbcSimplex &rhs)

  Copy constructor.
- AbcSimplex (const ClpSimplex &rhs)

  Copy constructor from model.
- AbcSimplex (const ClpSimplex ∗wholeModel, int numberRows, const int ∗whichRows, int numberColumns, const int ∗whichColumns, bool dropNames=true, bool dropIntegers=true, bool fixOthers=false)

  Subproblem constructor.
- AbcSimplex (const AbcSimplex ∗wholeModel, int numberRows, const int ∗whichRows, int numberColumns, const int ∗whichColumns, bool dropNames=true, bool dropIntegers=true, bool fixOthers=false)

  Subproblem constructor.
- AbcSimplex (AbcSimplex ∗wholeModel, int numberColumns, const int ∗whichColumns)

  This constructor modifies original AbcSimplex and stores original stuff in created AbcSimplex.
- void originalModel (AbcSimplex ∗miniModel)

  This copies back stuff from miniModel and then deletes miniModel.
- AbcSimplex (const ClpSimplex ∗clpSimplex)

  This constructor copies from ClpSimplex.
- void putBackSolution (ClpSimplex ∗simplex)

  Put back solution into ClpSimplex.
- void makeBaseModel ()

  Array persistence flag If 0 then as now (delete/new) 1 then only do arrays if bigger needed 2 as 1 but give a bit extra if bigger needed.

- void deleteBaseModel ()

  Switch off base model.
- AbcSimplex ∗ baseModel () const

  See if we have base model.
- void setToBaseModel (AbcSimplex ∗model=NULL)

  Reset to base model (just size and arrays needed) If model NULL use internal copy.
- AbcSimplex & operator= (const AbcSimplex &rhs)

  Assignment operator. This copies the data.
- ∼AbcSimplex ()

  Destructor.

Functions most useful to user

- int dual ()

  Dual algorithm - see AbcSimplexDual.hpp for method.
- int **doAbcDual** ()
- int primal (int ifValuesPass)

  Primal algorithm - see AbcSimplexPrimal.hpp for method.
- int **doAbcPrimal** (int ifValuesPass)
- **CoinWarmStartBasis** ∗ getBasis () const

  Returns a basis (to be deleted by user)
- void setFactorization (AbcSimplexFactorization &factorization)

  Passes in factorization.
- AbcSimplexFactorization ∗ swapFactorization (AbcSimplexFactorization ∗factorization)

  Swaps factorization.
- AbcSimplexFactorization ∗ getEmptyFactorization ()

  Gets clean and emptyish factorization.
- int tightenPrimalBounds ()

  Tightens primal bounds to make dual faster.
- void setDualRowPivotAlgorithm (AbcDualRowPivot &choice)

  Sets row pivot choice algorithm in dual.
- void setPrimalColumnPivotAlgorithm (AbcPrimalColumnPivot &choice)

  Sets column pivot choice algorithm in primal.

most useful gets and sets

- AbcSimplexFactorization ∗ factorization () const

  factorization
- int factorizationFrequency () const

  Factorization frequency.
- void **setFactorizationFrequency** (int value)
- int maximumAbcNumberRows () const

  Maximum rows.
- int maximumNumberTotal () const

  Maximum Total.
- int **maximumTotal** () const
- bool isObjectiveLimitTestValid () const

  Return true if the objective limit test can be relied upon.
- int numberTotal () const

  Number of variables (includes spare rows)
- int numberTotalWithoutFixed () const

  Number of variables without fixed to zero (includes spare rows)

- **CoinPartitionedVector** ∗ usefulArray (int index)
  Useful arrays (0,1,2,3,4,5,6,7)
- **CoinPartitionedVector** ∗ **usefulArray** (int index) const
- double clpObjectiveValue () const
  Objective value.
- int ∗ pivotVariable () const
  Basic variables pivoting on which rows may be same as toExternal but may be as at invert.
- int stateOfProblem () const
  State of problem.
- void setStateOfProblem (int value)
  State of problem.
- double ∗ scaleFromExternal () const
  Points from external to internal.
- double ∗ scaleToExternal () const
  Scale from primal internal to external (in external order) Or other way for dual.
- double ∗ rowScale2 () const
  corresponds to rowScale etc
- double ∗ **inverseRowScale2** () const
- double ∗ **inverseColumnScale2** () const
- double ∗ **columnScale2** () const
- int **arrayForDualColumn** () const
- double upperTheta () const
  upper theta from dual column
- int **arrayForReplaceColumn** () const
- int **arrayForFlipBounds** () const
- int **arrayForFlipRhs** () const
- int **arrayForBtran** () const
- int **arrayForFtran** () const
- int **arrayForTableauRow** () const
- double valueIncomingDual () const
  value of incoming variable (in Dual)
- const double ∗ getColSolution () const
  Get pointer to array[getNumCols()] of primal solution vector.
- const double ∗ getRowPrice () const
  Get pointer to array[getNumRows()] of dual prices.
- const double ∗ getReducedCost () const
  Get a pointer to array[getNumCols()] of reduced costs.
- const double ∗ getRowActivity () const
  Get pointer to array[getNumRows()] of row activity levels (constraint matrix times the solution vector.

Functions less likely to be useful to casual user

- int getSolution ()
  Given an existing factorization computes and checks primal and dual solutions.
- void setClpSimplexObjectiveValue ()
  Sets objectiveValue_ from rawObjectiveValue_.
- void setupDualValuesPass (const double ∗fakeDuals, const double ∗fakePrimals, int type)
  Sets dual values pass djs using unscaled duals type 1 - values pass type 2 - just use as infeasibility weights type 3 - as 2 but crash.
- double minimizationObjectiveValue () const
  Gets objective value with all offsets but as for minimization.
- double currentDualTolerance () const
  Current dualTolerance (will end up as dualTolerance_)

- void **setCurrentDualTolerance** (double value)
- AbcNonLinearCost ∗ abcNonLinearCost () const

    Return pointer to details of costs.
- double ∗ perturbationSaved () const

    Perturbation (fixed) - is just scaled random numbers.
- double acceptablePivot () const

    Acceptable pivot for this iteration.
- int ordinaryVariables () const

    Set to 1 if no free or super basic.
- int numberOrdinary () const

    Number of ordinary (lo/up) in tableau row.
- void setNumberOrdinary (int number)

    Set number of ordinary (lo/up) in tableau row.
- double currentDualBound () const

    Current dualBound (will end up as dualBound_)
- AbcDualRowPivot ∗ dualRowPivot () const

    dual row pivot choice
- AbcPrimalColumnPivot ∗ primalColumnPivot () const

    primal column pivot choice
- AbcMatrix ∗ abcMatrix () const

    Abc Matrix.
- int internalFactorize (int solveType)

    Factorizes using current basis.
- void permuteIn ()

    Permutes in from ClpModel data - assumes scale factors done and AbcMatrix exists but is in original order (including slacks)

For now just add basicArray at end

But could partition into normal (i.e.

- void permuteBasis ()

    *deals with new basis and puts in abcPivotVariable_*
- void permuteOut (int whatsWanted)

    Permutes out - bit settings same as stateOfProblem.
- ClpDataSave saveData ()

    Save data.
- void restoreData (ClpDataSave saved)

    Restore data.
- void cleanStatus (bool valuesPass=false)

    Clean up status - make sure no superbasic etc.
- int computeDuals (double ∗givenDjs, **CoinIndexedVector** ∗array1, **CoinIndexedVector** ∗array2)

    Computes duals from scratch.
- int computePrimals (**CoinIndexedVector** ∗array1, **CoinIndexedVector** ∗array2)

    Computes primals from scratch. Returns number of refinements.
- void computeObjective ()

    Computes nonbasic cost and total cost.
- void setMultipleSequenceIn (int sequenceIn[4])

    set multiple sequence in
- void unpack (**CoinIndexedVector** &rowArray) const

    Unpacks one column of the matrix into indexed array Uses sequenceIn_.
- void unpack (**CoinIndexedVector** &rowArray, int sequence) const

    Unpacks one column of the matrix into indexed array.

- int housekeeping ()
  This does basis housekeeping and does values for in/out variables.
- void checkPrimalSolution (bool justBasic)
  This sets largest infeasibility and most infeasible and sum and number of infeasibilities (Primal)
- void checkDualSolution ()
  This sets largest infeasibility and most infeasible and sum and number of infeasibilities (Dual)
- void checkDualSolutionPlusFake ()
  This sets largest infeasibility and most infeasible and sum and number of infeasibilities AND sumFakeInfeasibilites_ (Dual)
- void checkBothSolutions ()
  This sets sum and number of infeasibilities (Dual and Primal)
- int gutsOfSolution (int type)
  Computes solutions - 1 do duals, 2 do primals, 3 both (returns number of refinements)
- int gutsOfPrimalSolution (int type)
  Computes solutions - 1 do duals, 2 do primals, 3 both (returns number of refinements)
- void saveGoodStatus ()
  Saves good status etc.
- void restoreGoodStatus (int type)
  Restores previous good status and says trouble.
- void refreshCosts ()
  After modifying first copy refreshes second copy and marks as updated.
- void **refreshLower** (unsigned int type=∼(ROW_LOWER_SAME|COLUMN_UPPER_SAME))
- void **refreshUpper** (unsigned int type=∼(ROW_LOWER_SAME|COLUMN_LOWER_SAME))
- void setupPointers (int maxRows, int maxColumns)
  Sets up all extra pointers.
- void copyFromSaved (int type=31)
  Copies all saved versions to working versions and may do something for perturbation.
- void fillPerturbation (int start, int number)
  fills in perturbationSaved_ from start with 0.5+random
- void checkArrays (int ignoreEmpty=0) const
  For debug - prints summary of arrays which are out of kilter.
- void checkDjs (int type=1) const
  For debug - summarizes dj situation (1 recomputes duals first, 2 checks duals as well)
- void checkSolutionBasic () const
  For debug - checks solutionBasic.
- void checkMoveBack (bool checkDuals)
  For debug - moves solution back to external and computes stuff (always checks djs)
- void setValuesPassAction (double incomingInfeasibility, double allowedInfeasibility)
  For advanced use.
- int cleanFactorization (int ifValuesPass)
  Get a clean factorization - i.e.
- void moveStatusToClp (ClpSimplex ∗clpModel)
  Move status and solution to ClpSimplex.
- void moveStatusFromClp (ClpSimplex ∗clpModel)
  Move status and solution from ClpSimplex.

protected methods

- int gutsOfSolution (double ∗givenDuals, const double ∗givenPrimals, bool valuesPass=false)
  May change basis and then returns number changed.
- void gutsOfDelete (int type)
  Does most of deletion for arrays etc(0 just null arrays, 1 delete first)

- void gutsOfCopy (const AbcSimplex &rhs)

  Does most of copying.
- void gutsOfInitialize (int numberRows, int numberColumns, bool doMore)

  Initializes arrays.
- void gutsOfResize (int numberRows, int numberColumns)

  resizes arrays
- void translate (int type)

  Translates ClpModel to AbcSimplex See DO_ bits in stateOfProblem_ for type e.g.
- void moveToBasic (int which=15)

  Moves basic stuff to basic area.

public methods

- double ∗ solutionRegion () const

  Return region.
- double ∗ **djRegion** () const
- double ∗ **lowerRegion** () const
- double ∗ **upperRegion** () const
- double ∗ **costRegion** () const
- double ∗ solutionRegion (int which) const

  Return region.
- double ∗ **djRegion** (int which) const
- double ∗ **lowerRegion** (int which) const
- double ∗ **upperRegion** (int which) const
- double ∗ **costRegion** (int which) const
- double ∗ solutionBasic () const

  Return region.
- double ∗ **djBasic** () const
- double ∗ **lowerBasic** () const
- double ∗ **upperBasic** () const
- double ∗ **costBasic** () const
- double ∗ abcPerturbation () const

  Perturbation.
- double ∗ fakeDjs () const

  Fake djs.
- unsigned char ∗ **internalStatus** () const
- AbcSimplex::Status **getInternalStatus** (int sequence) const
- AbcSimplex::Status **getInternalColumnStatus** (int sequence) const
- void **setInternalStatus** (int sequence, AbcSimplex::Status newstatus)
- void **setInternalColumnStatus** (int sequence, AbcSimplex::Status newstatus)
- void setInitialDenseFactorization (bool onOff)

  Normally the first factorization does sparse coding because the factorization could be singular.
- bool **initialDenseFactorization** () const
- int sequenceIn () const

  Return sequence In or Out.
- int **sequenceOut** () const
- void setSequenceIn (int sequence)

  Set sequenceIn or Out.
- void **setSequenceOut** (int sequence)
- int isColumn (int sequence) const

  Returns 1 if sequence indicates column.
- int sequenceWithin (int sequence) const

  Returns sequence number within section.
- int lastPivotRow () const

  Current/last pivot row (set after END of choosing pivot row in dual)

- int firstFree () const

  First Free_.
- int lastFirstFree () const

  Last firstFree_.
- int freeSequenceIn () const

  Free chosen vector.
- double currentAcceptablePivot () const

  Acceptable pivot for this iteration.
- int fakeSuperBasic (int iSequence)

  Returns 1 if fake superbasic 0 if free or true superbasic -1 if was fake but has cleaned itself up (sets status) -2 if wasn't fake.
- double solution (int sequence)

  Return row or column values.
- double & solutionAddress (int sequence)

  Return address of row or column values.
- double **reducedCost** (int sequence)
- double & **reducedCostAddress** (int sequence)
- double **lower** (int sequence)
- double & lowerAddress (int sequence)

  Return address of row or column lower bound.
- double **upper** (int sequence)
- double & upperAddress (int sequence)

  Return address of row or column upper bound.
- double **cost** (int sequence)
- double & costAddress (int sequence)

  Return address of row or column cost.
- double originalLower (int iSequence) const

  Return original lower bound.
- double originalUpper (int iSequence) const

  Return original lower bound.
- AbcSimplexProgress ∗ abcProgress ()

  For dealing with all issues of cycling etc.
- void clearArraysPublic (int which)

  Clears an array and says available (-1 does all) when no possibility of going parallel.
- int getAvailableArrayPublic () const

  Returns first available empty array (and sets flag) when no possibility of going parallel.
- void clearArrays (int which)

  Clears an array and says available (-1 does all)
- void clearArrays (**CoinPartitionedVector** ∗which)

  Clears an array and says available.
- int getAvailableArray () const

  Returns first available empty array (and sets flag)
- void setUsedArray (int which) const

  Say array going to be used.
- void setAvailableArray (int which) const

  Say array going available.
- void swapPrimalStuff ()

  Swaps primal stuff.
- void swapDualStuff (int lastSequenceOut, int lastDirectionOut)

  Swaps dual stuff.

Changing bounds on variables and constraints

- void setObjectiveCoefficient (int elementIndex, double elementValue)

  Set an objective function coefficient.
- void setObjCoeff (int elementIndex, double elementValue)

  Set an objective function coefficient.
- void setColumnLower (int elementIndex, double elementValue)

  Set a single column lower bound
  Use -DBL_MAX for -infinity.
- void setColumnUpper (int elementIndex, double elementValue)

  Set a single column upper bound
  Use DBL_MAX for infinity.
- void setColumnBounds (int elementIndex, double lower, double upper)

  Set a single column lower and upper bound.
- void setColumnSetBounds (const int ∗indexFirst, const int ∗indexLast, const double ∗boundList)

  Set the bounds on a number of columns simultaneously
  The default implementation just invokes setColLower() and setColUpper() over and over again.
- void setColLower (int elementIndex, double elementValue)

  Set a single column lower bound
  Use -DBL_MAX for -infinity.
- void setColUpper (int elementIndex, double elementValue)

  Set a single column upper bound
  Use DBL_MAX for infinity.
- void setColBounds (int elementIndex, double newlower, double newupper)

  Set a single column lower and upper bound.
- void setColSetBounds (const int ∗indexFirst, const int ∗indexLast, const double ∗boundList)

  Set the bounds on a number of columns simultaneously

- void setRowLower (int elementIndex, double elementValue)

  Set a single row lower bound
  Use -DBL_MAX for -infinity.
- void setRowUpper (int elementIndex, double elementValue)

  Set a single row upper bound
  Use DBL_MAX for infinity.
- void setRowBounds (int elementIndex, double lower, double upper)

  Set a single row lower and upper bound.
- void setRowSetBounds (const int ∗indexFirst, const int ∗indexLast, const double ∗boundList)

  Set the bounds on a number of rows simultaneously

- void resize (int newNumberRows, int newNumberColumns)

  Resizes rim part of model.

Friends

- void AbcSimplexUnitTest (const std::string &mpsDir)

  A function that tests the methods in the AbcSimplex class.

status methods

- void swap (int pivotRow, int nonBasicPosition)

  Swaps two variables.
- void setFlagged (int sequence)

  To flag a variable.
- void **clearFlagged** (int sequence)

- bool **flagged** (int sequence) const
- void createStatus ()

  Set up status array (can be used by OsiAbc).
- void crash (int type)

  Does sort of crash.
- void putStuffInBasis (int type)

  Puts more stuff in basis 1 bit set - do even if basis exists 2 bit set - don't bother staying triangular.
- void allSlackBasis ()

  Sets up all slack basis and resets solution to as it was after initial load or readMps.
- void checkConsistentPivots () const

  For debug - check pivotVariable consistent.
- void printStuff () const

  Print stuff.
- int startup (int ifValuesPass)

  Common bits of coding for dual and primal.
- double rawObjectiveValue () const

  Raw objective value (so always minimize in primal)
- void computeObjectiveValue (bool useWorkingSolution=false)

  Compute objective value from solution and put in objectiveValue_.
- double computeInternalObjectiveValue ()

  Compute minimization objective value from internal solution without perturbation.
- void moveInfo (const AbcSimplex &rhs, bool justStatus=false)

  Move status and solution across.
- void swap (int pivotRow, int nonBasicPosition, Status newStatus)

  Swaps two variables and does status.
- void **setFakeBound** (int sequence, FakeBound fakeBound)
- FakeBound **getFakeBound** (int sequence) const
- bool **atFakeBound** (int sequence) const
- void **setPivoted** (int sequence)
- void **clearPivoted** (int sequence)
- bool **pivoted** (int sequence) const
- void setActive (int iRow)

  To say row active in primal pivot row choice.
- void **clearActive** (int iRow)
- bool **active** (int iRow) const

data. Many arrays have a row part and a column part.

There is a single array with both - columns then rows and then normally two arrays pointing to rows and columns.

The single array is the owner of memory

- double sumNonBasicCosts_

  Sum of nonbasic costs.
- double rawObjectiveValue_

  Sum of costs (raw objective value)
- double objectiveOffset_

  Objective offset (from offset_)

- double perturbationFactor_

    Perturbation factor If <0.0 then virtual if 0.0 none if >0.0 use this as factor.
- double currentDualTolerance_

    Current dualTolerance (will end up as dualTolerance_)
- double currentDualBound_

    Current dualBound (will end up as dualBound_)
- double largestGap_

    Largest gap.
- double lastDualBound_

    Last dual bound.
- double sumFakeInfeasibilities_

    Sum of infeasibilities when using fake perturbation tolerance.
- double lastPrimalError_

    Last primal error.
- double lastDualError_

    Last dual error.
- double currentAcceptablePivot_

    Acceptable pivot for this iteration.
- double movement_

    Movement of variable.
- double objectiveChange_

    Objective change.
- double btranAlpha_

    Btran alpha.
- double ftAlpha_

    FT alpha.
- double minimumThetaMovement_

    Minimum theta movement.
- double initialSumInfeasibilities_

    Initial sum of infeasibilities.
- int lastFirstFree_

    Last firstFree_.
- int freeSequenceIn_

    Free chosen vector.
- int maximumAbcNumberRows_

    Maximum number rows.
- int maximumAbcNumberColumns_

    Maximum number columns.
- int maximumNumberTotal_

    Maximum numberTotal.
- int numberFlagged_

    Current number of variables flagged.
- int normalDualColumnIteration_

    Iteration at which to do relaxed dualColumn.
- int stateDualColumn_

    State of dual waffle -2 - in initial large tolerance phase -1 - in medium tolerance phase n - in correct tolerance phase and thought optimal n times.

- int numberTotal_

  Number of variables (includes spare rows)
- int numberTotalWithoutFixed_

  Number of variables without fixed to zero (includes spare rows)
- int startAtLowerOther_

  Start of variables at lower bound with upper.
- int startAtUpperNoOther_

  Start of variables at upper bound with no lower.
- int startAtUpperOther_

  Start of variables at upper bound with lower.
- int startOther_

  Start of superBasic, free or awkward bounds variables.
- int startFixed_

  Start of fixed variables.
- int **stateOfProblem_**
- int numberOrdinary_

  Number of ordinary (lo/up) in tableau row.
- int ordinaryVariables_

  Set to 1 if no free or super basic.
- int numberFreeNonBasic_

  Number of free nonbasic variables.
- int lastCleaned_

  Last time cleaned up.
- int lastPivotRow_

  Current/last pivot row (set after END of choosing pivot row in dual)
- int swappedAlgorithm_

  Nonzero (probably 10) if swapped algorithms.
- int initialNumberInfeasibilities_

  Initial number of infeasibilities.
- double ∗ scaleFromExternal_

  Points from external to internal.
- double ∗ scaleToExternal_

  Scale from primal internal to external (in external order) Or other way for dual.
- double ∗ columnUseScale_

  use this instead of columnScale
- double ∗ inverseColumnUseScale_

  use this instead of inverseColumnScale
- double ∗ offset_

  Primal offset (in external order) So internal value is (external-offset)∗scaleFromExternal.
- double ∗ offsetRhs_

  Offset for accumulated offsets∗matrix.
- double ∗ tempArray_

  Useful array of numberTotal length.
- unsigned char ∗ internalStatus_

  Working status ? may be signed ? link pi_ to an indexed array? may have saved from last factorization at end.
- unsigned char ∗ internalStatusSaved_

  Saved status.

- double ∗ abcPerturbation_

    Perturbation (fixed) - is just scaled random numbers If perturbationFactor_<0 then virtual perturbation.
- double ∗ perturbationSaved_

    saved perturbation
- double ∗ perturbationBasic_

    basic perturbation
- AbcMatrix ∗ abcMatrix_

    Working matrix.
- double ∗ abcLower_

    Working scaled copy of lower bounds has original scaled copy at end.
- double ∗ abcUpper_

    Working scaled copy of upper bounds has original scaled copy at end.
- double ∗ abcCost_

    Working scaled copy of objective ? where perturbed copy or can we always work with perturbed copy (in B&B) if we adjust increments/cutoffs ? should we save a fixed perturbation offset array has original scaled copy at end.
- double ∗ abcSolution_

    Working scaled primal solution may have saved from last factorization at end.
- double ∗ abcDj_

    Working scaled dual solution may have saved from last factorization at end.
- double ∗ lowerSaved_

    Saved scaled copy of lower bounds.
- double ∗ upperSaved_

    Saved scaled copy of upper bounds.
- double ∗ costSaved_

    Saved scaled copy of objective.
- double ∗ solutionSaved_

    Saved scaled primal solution.
- double ∗ djSaved_

    Saved scaled dual solution.
- double ∗ lowerBasic_

    Working scaled copy of basic lower bounds.
- double ∗ upperBasic_

    Working scaled copy of basic upper bounds.
- double ∗ costBasic_

    Working scaled copy of basic objective.
- double ∗ solutionBasic_

    Working scaled basic primal solution.
- double ∗ djBasic_

    Working scaled basic dual solution (want it to be zero)
- AbcDualRowPivot ∗ abcDualRowPivot_

    dual row pivot choice
- AbcPrimalColumnPivot ∗ abcPrimalColumnPivot_

    primal column pivot choice
- int ∗ abcPivotVariable_

    Basic variables pivoting on which rows followed by atLo/atUp then free/superbasic then fixed.
- int ∗ reversePivotVariable_

    Reverse abcPivotVariable_ for moving around.

- AbcSimplexFactorization ∗ abcFactorization_

    factorization
- AbcSimplex ∗ abcBaseModel_

    Saved version of solution.
- ClpSimplex ∗ clpModel_

    A copy of model as ClpSimplex with certain state.
- AbcNonLinearCost ∗ abcNonLinearCost_

    Very wasteful way of dealing with infeasibilities in primal.
- **CoinPartitionedVector usefulArray_** [ABC_NUMBER_USEFUL]
- AbcSimplexProgress abcProgress_

    For dealing with all issues of cycling etc.
- ClpDataSave saveData_

    For saving stuff at beginning.
- double upperTheta_

    upper theta from dual column
- int multipleSequenceIn_ [4]

    Multiple sequence in.
- int **numberFlipped_**
- int **numberDisasters_**
- int stateOfIteration_

    Where we are in iteration.
- int **arrayForDualColumn_**
- int **arrayForReplaceColumn_**
- int **arrayForFlipBounds_**
- int **arrayForFlipRhs_**
- int **arrayForBtran_**
- int **arrayForFtran_**
- int **arrayForTableauRow_**

Additional Inherited Members

4.11.1 Detailed Description

Definition at line 62 of file AbcSimplex.hpp.

4.11.2 Member Enumeration Documentation

4.11.2.1 enum AbcSimplex::Status

enums for status of various sorts.

ClpModel order (and warmstart) is isFree = 0x00, basic = 0x01, atUpperBound = 0x02, atLowerBound = 0x03, isFixed means fixed at lower bound and out of basis

Definition at line 74 of file AbcSimplex.hpp.

4.11.3 Constructor & Destructor Documentation

4.11.3.1 AbcSimplex::AbcSimplex (const ClpSimplex * *wholeModel*, int *numberRows*, const int * *whichRows*, int *numberColumns*, const int * *whichColumns*, bool *dropNames* = `true`, bool *dropIntegers* = `true`, bool *fixOthers* = `false`)

Subproblem constructor.

A subset of whole model is created from the row and column lists given. The new order is given by list order and duplicates are allowed. Name and integer information can be dropped Can optionally modify rhs to take into account variables NOT in list in this case duplicates are not allowed (also see getbackSolution)

4.11.3.2 AbcSimplex::AbcSimplex (const AbcSimplex * *wholeModel*, int *numberRows*, const int * *whichRows*, int *numberColumns*, const int * *whichColumns*, bool *dropNames* = `true`, bool *dropIntegers* = `true`, bool *fixOthers* = `false`)

Subproblem constructor.

A subset of whole model is created from the row and column lists given. The new order is given by list order and duplicates are allowed. Name and integer information can be dropped Can optionally modify rhs to take into account variables NOT in list in this case duplicates are not allowed (also see getbackSolution)

4.11.3.3 AbcSimplex::AbcSimplex (AbcSimplex * *wholeModel*, int *numberColumns*, const int * *whichColumns*)

This constructor modifies original AbcSimplex and stores original stuff in created AbcSimplex.

It is only to be used in conjunction with originalModel

4.11.4 Member Function Documentation

4.11.4.1 void AbcSimplex::originalModel (AbcSimplex * *miniModel*)

This copies back stuff from miniModel and then deletes miniModel.

Only to be used with mini constructor

4.11.4.2 void AbcSimplex::makeBaseModel ( )

Array persistence flag If 0 then as now (delete/new) 1 then only do arrays if bigger needed 2 as 1 but give a bit extra if bigger needed.

Save a copy of model with certain state - normally without cuts

4.11.4.3 int AbcSimplex::tightenPrimalBounds ( )

Tightens primal bounds to make dual faster.

Unless fixed or doTight>10, bounds are slightly looser than they could be. This is to make dual go faster and is probably not needed with a presolve. Returns non-zero if problem infeasible.

Fudge for branch and bound - put bounds on columns of factor * largest value (at continuous) - should improve stability in branch and bound on infeasible branches (0.0 is off)

4.11.4.4 int AbcSimplex::getSolution ( )

Given an existing factorization computes and checks primal and dual solutions.

Uses current problem arrays for bounds. Returns feasibility states

4.11.4.5 int AbcSimplex::internalFactorize (int *solveType*)

Factorizes using current basis.

solveType - 1 iterating, 0 initial, -1 external If 10 added then in primal values pass Return codes are as from Abc↩ SimplexFactorization unless initial factorization when total number of singularities is returned. Special case is number↩ Rows_+1 -> all slack basis. if initial should be before permute in pivotVariable may be same as toExternal

4.11.4.6 void AbcSimplex::permuteIn ( )

Permutes in from ClpModel data - assumes scale factors done and AbcMatrix exists but is in original order (including slacks)

For now just add basicArray at end

But could partition into normal (i.e.

reasonable lower/upper) abnormal - free, odd bounds

fixed

sets a valid pivotVariable Slacks always shifted by offset Fixed variables always shifted by offset Recode to allow row objective so can use pi from idiot etc

4.11.4.7 int AbcSimplex::computeDuals (double ∗ *givenDjs,* CoinIndexedVector ∗ *array1,* CoinIndexedVector ∗ *array2*)

Computes duals from scratch.

If givenDjs then allows for nonzero basic djs. Returns number of refinements

4.11.4.8 int AbcSimplex::housekeeping ( )

This does basis housekeeping and does values for in/out variables.

Can also decide to re-factorize

4.11.4.9 void AbcSimplex::setValuesPassAction (double *incomingInfeasibility,* double *allowedInfeasibility*)

For advanced use.

When doing iterative solves things can get nasty so on values pass if incoming solution has largest infeasibility < incomingInfeasibility throw out variables from basis until largest infeasibility < allowedInfeasibility or incoming largest infeasibility. If allowedInfeasibility>= incomingInfeasibility this is always possible altough you may end up with an all slack basis.

Defaults are 1.0,10.0

4.11.4.10 int AbcSimplex::cleanFactorization (int *ifValuesPass*)

Get a clean factorization - i.e.

throw out singularities may do more later

4.11.4.11 double* AbcSimplex::scaleFromExternal ( ) const [inline]

Points from external to internal.

Points from internal to external Scale from primal external to internal (in external order) Or other way for dual

Definition at line 443 of file AbcSimplex.hpp.

4.11.4.12 int AbcSimplex::gutsOfSolution (double * *givenDuals*, const double * *givenPrimals*, bool *valuesPass* = false)

May change basis and then returns number changed.

Computation of solutions may be overriden by given pi and solution

4.11.4.13 void AbcSimplex::translate (int *type*)

Translates ClpModel to AbcSimplex See DO_ bits in stateOfProblem_ for type e.g.

DO_BASIS_AND_ORDER

4.11.4.14 void AbcSimplex::setInitialDenseFactorization (bool *onOff*)

Normally the first factorization does sparse coding because the factorization could be singular.

This allows initial dense factorization when it is known to be safe

4.11.4.15 void AbcSimplex::createStatus ( )

Set up status array (can be used by OsiAbc).

Also can be used to set up all slack basis

4.11.4.16 void AbcSimplex::setColumnLower (int *elementIndex*, double *elementValue*)

Set a single column lower bound
Use -DBL_MAX for -infinity.

4.11.4.17 void AbcSimplex::setColumnUpper (int *elementIndex*, double *elementValue*)

Set a single column upper bound
Use DBL_MAX for infinity.

4.11.4.18 void AbcSimplex::setColumnSetBounds (const int * *indexFirst*, const int * *indexLast*, const double * *boundList*)

Set the bounds on a number of columns simultaneously
The default implementation just invokes setColLower() and setColUpper() over and over again.

Parameters

index↩ First,indexLast	pointers to the beginning and after the end of the array of the indices of the variables whose *either* bound changes
boundList	the new lower/upper bound pairs for the variables

4.11.4.19 void AbcSimplex::setColLower (int *elementIndex,* double *elementValue*) [inline]

Set a single column lower bound
Use -DBL_MAX for -infinity.

Definition at line 921 of file AbcSimplex.hpp.

4.11.4.20 void AbcSimplex::setColUpper (int *elementIndex,* double *elementValue*) [inline]

Set a single column upper bound
Use DBL_MAX for infinity.

Definition at line 926 of file AbcSimplex.hpp.

4.11.4.21 void AbcSimplex::setColSetBounds (const int ∗ *indexFirst,* const int ∗ *indexLast,* const double ∗ *boundList*) [inline]

Set the bounds on a number of columns simultaneously

Parameters

index↩ First,indexLast	pointers to the beginning and after the end of the array of the indices of the variables whose *either* bound changes
boundList	the new lower/upper bound pairs for the variables

Definition at line 942 of file AbcSimplex.hpp.

4.11.4.22 void AbcSimplex::setRowLower (int *elementIndex,* double *elementValue*)

Set a single row lower bound
Use -DBL_MAX for -infinity.

4.11.4.23 void AbcSimplex::setRowUpper (int *elementIndex,* double *elementValue*)

Set a single row upper bound
Use DBL_MAX for infinity.

4.11.4.24 void AbcSimplex::setRowSetBounds (const int ∗ *indexFirst,* const int ∗ *indexLast,* const double ∗ *boundList*)

Set the bounds on a number of rows simultaneously

Parameters

index←First,indexLast	pointers to the beginning and after the end of the array of the indices of the constraints whose *either* bound changes
boundList	the new lower/upper bound pairs for the constraints

4.11.5 Friends And Related Function Documentation

4.11.5.1 void AbcSimplexUnitTest (const std::string & *mpsDir*) [friend]

A function that tests the methods in the AbcSimplex class.

The only reason for it not to be a member method is that this way it doesn't have to be compiled into the library. And that's a gain, because the library should be compiled with optimization on, but this method should be compiled with debugging.

It also does some testing of AbcSimplexFactorization class

4.11.6 Member Data Documentation

4.11.6.1 double∗ AbcSimplex::scaleFromExternal_ [protected]

Points from external to internal.

Points from internal to external Scale from primal external to internal (in external order) Or other way for dual

Definition at line 1134 of file AbcSimplex.hpp.

4.11.6.2 AbcSimplex∗ AbcSimplex::abcBaseModel_ [protected]

Saved version of solution.

A copy of model with certain state - normally without cuts

Definition at line 1227 of file AbcSimplex.hpp.

4.11.6.3 AbcNonLinearCost∗ AbcSimplex::abcNonLinearCost_ [protected]

Very wasteful way of dealing with infeasibilities in primal.

However it will allow non-linearities and use of dual analysis. If it doesn't work it can easily be replaced.

Definition at line 1234 of file AbcSimplex.hpp.

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

- AbcSimplex.hpp

4.12 AbcSimplexDual Class Reference

This solves LPs using the dual simplex method.

`#include <AbcSimplexDual.hpp>`

Inheritance diagram for AbcSimplexDual:

Collaboration diagram for AbcSimplexDual:

Public Member Functions

Description of algorithm

- int dual ()

  Dual algorithm.
- int strongBranching (int numberVariables, const int ∗variables, double ∗newLower, double ∗newUpper, double ∗∗outputSolution, int ∗outputStatus, int ∗outputIterations, bool stopOnFirstInfeasible=true, bool always↩Finish=false, int startFinishOptions=0)

  For strong branching.
- AbcSimplexFactorization ∗ setupForStrongBranching (char ∗arrays, int numberRows, int numberColumns, bool solveLp=false)

  This does first part of StrongBranching.
- void cleanupAfterStrongBranching (AbcSimplexFactorization ∗factorization)

  This cleans up after strong branching.

Functions used in dual

- int whileIteratingSerial ()

  This has the flow between re-factorizations Broken out for clarity and will be used by strong branching.
- void **whileIterating2** ()
- int **whileIteratingParallel** (int numberIterations)
- int **whileIterating3** ()
- void **updatePrimalSolution** ()
- int **noPivotRow** ()
- int **noPivotColumn** ()
- void **dualPivotColumn** ()
- void createDualPricingVectorSerial ()

  Create dual pricing vector.
- int **getTableauColumnFlipAndStartReplaceSerial** ()
- void **getTableauColumnPart1Serial** ()
- void **getTableauColumnPart2** ()
- int **checkReplace** ()
- void **replaceColumnPart3** ()
- void **checkReplacePart1** ()
- void **checkReplacePart1a** ()
- void **checkReplacePart1b** ()
- void updateDualsInDual ()

  The duals are updated.
- int flipBounds ()

  The duals are updated by the given arrays.
- void flipBack (int number)

  Undo a flip.
- void dualColumn1 (bool doAll=false)

  Array has tableau row (row section) Puts candidates for rows in list Returns guess at upper theta (infinite if no pivot) and may set sequenceIn_ if free Can do all (if tableauRow created)
- double dualColumn1A ()

  Array has tableau row (row section) Just does slack part Returns guess at upper theta (infinite if no pivot) and may set sequenceIn_ if free.
- double dualColumn1B ()

  Do all given tableau row.
- void dualColumn2 ()

Chooses incoming Puts flipped ones in list If necessary will modify costs.
- void **dualColumn2Most** (dualColumnResult &result)
- void **dualColumn2First** (dualColumnResult &result)
- void dualColumn2 (dualColumnResult &result)

  Chooses part of incoming Puts flipped ones in list If necessary will modify costs.
- void checkPossibleCleanup (**CoinIndexedVector** *array)

  This sees what is best thing to do in branch and bound cleanup If sequenceIn_ $<$ 0 then can't do anything.
- void dualPivotRow ()

  Chooses dual pivot row Would be faster with separate region to scan and will have this (with square of infeasibility) when steepest For easy problems we can just choose one of the first rows we look at.
- int changeBounds (int initialize, double &changeCost)

  Checks if any fake bounds active - if so returns number and modifies updatedDualBound_ and everything.
- bool changeBound (int iSequence)

  As changeBounds but just changes new bounds for a single variable.
- void originalBound (int iSequence)

  Restores bound to original bound.
- int checkUnbounded (**CoinIndexedVector** &ray, double changeCost)

  Checks if tentative optimal actually means unbounded in dual Returns -3 if not, 2 if is unbounded.
- void statusOfProblemInDual (int type)

  Refactorizes if necessary Checks if finished.
- int whatNext ()

  Fast iterations.
- bool checkCutoff (bool computeObjective)

  see if cutoff reached
- int bounceTolerances (int type)

  Does something about fake tolerances.
- void perturb (double factor)

  Perturbs problem.
- void perturbB (double factor, int type)

  Perturbs problem B.
- int makeNonFreeVariablesDualFeasible (bool changeCosts=false)

  Make non free variables dual feasible by moving to a bound.
- int **fastDual** (bool alwaysFinish=false)
- int numberAtFakeBound ()

  Checks number of variables at fake bounds.
- int pivotResultPart1 ()

  Pivot in a variable and choose an outgoing one.
- int nextSuperBasic ()

  Get next free , -1 if none.
- void startupSolve ()

  Startup part of dual.
- void finishSolve ()

  Ending part of dual.
- void **gutsOfDual** ()
- int **resetFakeBounds** (int type)

Additional Inherited Members

4.12.1 Detailed Description

This solves LPs using the dual simplex method.

It inherits from AbcSimplex. It has no data of its own and is never created - only cast from a AbcSimplex object at algorithm time.

Definition at line 49 of file AbcSimplexDual.hpp.

4.12.2 Member Function Documentation

4.12.2.1 int AbcSimplexDual::dual ( )

Dual algorithm.

Method

It tries to be a single phase approach with a weight of 1.0 being given to getting optimal and a weight of updatedDual←
Bound_ being given to getting dual feasible. In this version I have used the idea that this weight can be thought of as a fake bound. If the distance between the lower and upper bounds on a variable is less than the feasibility weight then we are always better off flipping to other bound to make dual feasible. If the distance is greater then we make up a fake bound updatedDualBound_ away from one bound. If we end up optimal or primal infeasible, we check to see if bounds okay. If so we have finished, if not we increase updatedDualBound_ and continue (after checking if unbounded). I am undecided about free variables - there is coding but I am not sure about it. At present I put them in basis anyway.

The code is designed to take advantage of sparsity so arrays are seldom zeroed out from scratch or gone over in their entirety. The only exception is a full scan to find outgoing variable for Dantzig row choice. For steepest edge we keep an updated list of infeasibilities (actually squares). On easy problems we don't need full scan - just pick first reasonable.

One problem is how to tackle degeneracy and accuracy. At present I am using the modification of costs which I put in OSL and some of what I think is the dual analog of Gill et al. I am still not sure of the exact details.

The flow of dual is three while loops as follows:

while (not finished) {

while (not clean solution) {

Factorize and/or clean up solution by flipping variables so dual feasible. If looks finished check fake dual bounds. Repeat until status is iterating (-1) or finished (0,1,2)

}

while (status==-1) {

Iterate until no pivot in or out or time to re-factorize.

Flow is:

choose pivot row (outgoing variable). if none then we are primal feasible so looks as if done but we need to break and check bounds etc.

Get pivot row in tableau

Choose incoming column. If we don't find one then we look primal infeasible so break and check bounds etc. (Also the pivot tolerance is larger after any iterations so that may be reason)

If we do find incoming column, we may have to adjust costs to keep going forwards (anti-degeneracy). Check pivot will be stable and if unstable throw away iteration and break to re-factorize. If minor error re-factorize after iteration.

Update everything (this may involve flipping variables to stay dual feasible.

}

}

TODO's (or maybe not)

At present we never check we are going forwards. I overdid that in OSL so will try and make a last resort.

Needs partial scan pivot out option.

May need other anti-degeneracy measures, especially if we try and use loose tolerances as a way to solve in fewer iterations.

I like idea of dynamic scaling. This gives opportunity to decouple different implications of scaling for accuracy, iteration count and feasibility tolerance.

for use of exotic parameter startFinishoptions see Abcsimplex.hpp

4.12.2.2 int AbcSimplexDual::strongBranching (int *numberVariables*, const int ∗ *variables*, double ∗ *newLower*, double ∗ *newUpper*, double ∗∗ *outputSolution*, int ∗ *outputStatus*, int ∗ *outputIterations*, bool *stopOnFirstInfeasible* = `true`, bool *alwaysFinish* = `false`, int *startFinishOptions* = `0`)

For strong branching.

On input lower and upper are new bounds while on output they are change in objective function values (>1.0e50 infeasible). Return code is 0 if nothing interesting, -1 if infeasible both ways and +1 if infeasible one way (check values to see which one(s)) Solutions are filled in as well - even down, odd up - also status and number of iterations

4.12.2.3 int AbcSimplexDual::whileIteratingSerial ( )

This has the flow between re-factorizations Broken out for clarity and will be used by strong branching.

Reasons to come out: -1 iterations etc -2 inaccuracy -3 slight inaccuracy (and done iterations) +0 looks optimal (might be unbounded - but we will investigate) +1 looks infeasible +3 max iterations

If givenPi not NULL then in values pass (copy from ClpSimplexDual)

4.12.2.4 int AbcSimplexDual::flipBounds ( )

The duals are updated by the given arrays.

This is in values pass - so no changes to primal is madeWhile dualColumn gets flips this does actual flipping. returns number flipped

4.12.2.5 int AbcSimplexDual::changeBounds (int *initialize*, double & *changeCost*)

Checks if any fake bounds active - if so returns number and modifies updatedDualBound_ and everything.

Free variables will be left as free Returns number of bounds changed if >=0 Returns -1 if not initialize and no effect fills cost of change vector

4.12.2.6 bool AbcSimplexDual::changeBound (int *iSequence*)

As changeBounds but just changes new bounds for a single variable.

Returns true if change

4.12.2.7 void AbcSimplexDual::statusOfProblemInDual (int *type*)

Refactorizes if necessary Checks if finished.

Updates status. lastCleaned refers to iteration at which some objective/feasibility cleaning too place.

type - 0 initial so set up save arrays etc

- 1 normal -if good update save
- 2 restoring from saved

4.12.2.8 int AbcSimplexDual::whatNext ( )

Fast iterations.

Misses out a lot of initialization. Normally stops on maximum iterations, first re-factorization or tentative optimum. If looks interesting then continues as normal. Returns 0 if finished properly, 1 otherwise.Gets tableau column - does flips and checks what to do next Knows tableau column in 1, flips in 2 and gets an array for flips (as serial here)

4.12.2.9 int AbcSimplexDual::numberAtFakeBound ( )

Checks number of variables at fake bounds.

This is used by fastDual so can exit gracefully before end

4.12.2.10 int AbcSimplexDual::pivotResultPart1 ( )

Pivot in a variable and choose an outgoing one.

Assumes dual feasible - will not go through a reduced cost. Returns step length in theta Return codes as before but -1 means no acceptable pivot

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

- AbcSimplexDual.hpp

4.13 AbcSimplexFactorization Class Reference

This just implements AbcFactorization when an AbcMatrix object is passed.

`#include <AbcSimplexFactorization.hpp>`

Public Member Functions

factorization

- int factorize (AbcSimplex *model, int solveType, bool valuesPass)
  When part of LP - given by basic variables.

Constructors, destructor

- AbcSimplexFactorization (int numberRows=0)
  Default constructor.
- ∼AbcSimplexFactorization ()
  Destructor.

Copy method

- AbcSimplexFactorization (const AbcSimplexFactorization &, int denseIfSmaller=0)
  The copy constructor.
- AbcSimplexFactorization & **operator=** (const AbcSimplexFactorization &)
- void setFactorization (AbcSimplexFactorization &rhs)
  Sets factorization.

rank one updates which do exist

- double checkReplacePart1 (**CoinIndexedVector** ∗regionSparse, int pivotRow)

  Checks if can replace one Column to basis, returns update alpha Fills in region for use later partial update already in U.
- double checkReplacePart1 (**CoinIndexedVector** ∗regionSparse, **CoinIndexedVector** ∗partialUpdate, int pivotRow)

  Checks if can replace one Column to basis, returns update alpha Fills in region for use later partial update in vector.
- int checkReplacePart2 (int pivotRow, double btranAlpha, double ftranAlpha, double ftAlpha)

  Checks if can replace one Column to basis, returns 0=OK, 1=Probably OK, 2=singular, 3=no room, 5 max pivots.
- void replaceColumnPart3 (const AbcSimplex ∗model, **CoinIndexedVector** ∗regionSparse, **CoinIndexedVector** ∗tableauColumn, int pivotRow, double alpha)

  Replaces one Column to basis, partial update already in U.
- void replaceColumnPart3 (const AbcSimplex ∗model, **CoinIndexedVector** ∗regionSparse, **CoinIndexedVector** ∗tableauColumn, **CoinIndexedVector** ∗partialUpdate, int pivotRow, double alpha)

  Replaces one Column to basis, partial update in vector.

various uses of factorization (return code number elements)

which user may want to know about

- int updateColumnFT (**CoinIndexedVector** &regionSparseFT)

  Updates one column (FTRAN) Tries to do FT update number returned is negative if no room.
- int **updateColumnFTPart1** (**CoinIndexedVector** &regionSparseFT)
- void **updateColumnFTPart2** (**CoinIndexedVector** &regionSparseFT)
- void updateColumnFT (**CoinIndexedVector** &regionSparseFT, **CoinIndexedVector** &partialUpdate, int which)

  Updates one column (FTRAN) Tries to do FT update puts partial update in vector.
- int updateColumn (**CoinIndexedVector** &regionSparse) const

  Updates one column (FTRAN)
- int updateTwoColumnsFT (**CoinIndexedVector** &regionSparseFT, **CoinIndexedVector** &regionSparseOther)

  Updates one column (FTRAN) from regionFT Tries to do FT update number returned is negative if no room.
- int updateColumnTranspose (**CoinIndexedVector** &regionSparse) const

  Updates one column (BTRAN)
- void updateColumnCpu (**CoinIndexedVector** &regionSparse, int whichCpu) const

  Updates one column (FTRAN)
- void updateColumnTransposeCpu (**CoinIndexedVector** &regionSparse, int whichCpu) const

  Updates one column (BTRAN)
- void updateFullColumn (**CoinIndexedVector** &regionSparse) const

  Updates one full column (FTRAN)
- void updateFullColumnTranspose (**CoinIndexedVector** &regionSparse) const

  Updates one full column (BTRAN)
- void updateWeights (**CoinIndexedVector** &regionSparse) const

  Updates one column for dual steepest edge weights (FTRAN)

Lifted from CoinFactorization

- int numberElements () const

  Total number of elements in factorization.
- int maximumPivots () const

  Maximum number of pivots between factorizations.
- void maximumPivots (int value)

  Set maximum number of pivots between factorizations.
- bool usingFT () const

Returns true if doing FT.
- int pivots () const
  Returns number of pivots since factorization.
- void setModel (AbcSimplex ∗model)
  Sets model.
- void setPivots (int value) const
  Sets number of pivots since factorization.
- double areaFactor () const
  Whether larger areas needed.
- void areaFactor (double value)
  Set whether larger areas needed.
- double zeroTolerance () const
  Zero tolerance.
- void zeroTolerance (double value)
  Set zero tolerance.
- void saferTolerances (double zeroTolerance, double pivotTolerance)
  Set tolerances to safer of existing and given.
- int status () const
  Returns status.
- void setStatus (int value)
  Sets status.
- int numberDense () const
  Returns number of dense rows.
- bool **timeToRefactorize** () const
- void clearArrays ()
  Get rid of all memory.
- int numberRows () const
  Number of Rows after factorization.
- int numberSlacks () const
  Number of slacks at last factorization.
- double pivotTolerance () const
  Pivot tolerance.
- void pivotTolerance (double value)
  Set pivot tolerance.
- double minimumPivotTolerance () const
  Minimum pivot tolerance.
- void minimumPivotTolerance (double value)
  Set minimum pivot tolerance.
- double ∗ pivotRegion () const
  pivot region
- void almostDestructor ()
  Allows change of pivot accuracy check 1.0 == none >1.0 relaxed.
- void setDenseThreshold (int number)
  So we can temporarily switch off dense.
- int **getDenseThreshold** () const
- void forceOtherFactorization (int which)
  If nonzero force use of 1,dense 2,small 3,long.
- void goDenseOrSmall (int numberRows)
  Go over to dense code.
- int goDenseThreshold () const
  Get switch to dense if number rows <= this.
- void setGoDenseThreshold (int value)

Set switch to dense if number rows <= this.
- int goSmallThreshold () const

    Get switch to small if number rows <= this.
- void setGoSmallThreshold (int value)

    Set switch to small if number rows <= this.
- int goLongThreshold () const

    Get switch to long/ordered if number rows >= this.
- void setGoLongThreshold (int value)

    Set switch to long/ordered if number rows >= this.
- int typeOfFactorization () const

    Returns type.
- void synchronize (const ClpFactorization *otherFactorization, const AbcSimplex *model)

    Synchronize stuff.

other stuff

- void goSparse ()

    makes a row copy of L for speed and to allow very sparse problems
- void **checkMarkArrays** () const
- bool needToReorder () const

    Says whether to redo pivot order.
- CoinAbcAnyFactorization * factorization () const

    Pointer to factorization.

4.13.1 Detailed Description

This just implements AbcFactorization when an AbcMatrix object is passed.

Definition at line 27 of file AbcSimplexFactorization.hpp.

4.13.2 Constructor & Destructor Documentation

4.13.2.1 AbcSimplexFactorization::AbcSimplexFactorization (int *numberRows* = 0)

Default constructor.

4.13.2.2 AbcSimplexFactorization::AbcSimplexFactorization (const AbcSimplexFactorization & , int *denseIfSmaller* = 0)

The copy constructor.

4.13.3 Member Function Documentation

4.13.3.1 int AbcSimplexFactorization::factorize (AbcSimplex * *model,* int *solveType,* bool *valuesPass*)

When part of LP - given by basic variables.

Actually does factorization. Arrays passed in have non negative value to say basic. If status is okay, basic variables have pivot row - this is only needed if increasingRows_ >1. Allows scaling If status is singular, then basic variables have pivot row and ones thrown out have -1 returns 0 -okay, -1 singular, -2 too many in basis, -99 memory

4.13.3.2 int AbcSimplexFactorization::updateTwoColumnsFT (CoinIndexedVector & *regionSparseFT,* CoinIndexedVector & *regionSparseOther*) [inline]

Updates one column (FTRAN) from regionFT Tries to do FT update number returned is negative if no room.

Also updates regionOther

Definition at line 202 of file AbcSimplexFactorization.hpp.

4.13.3.3 void AbcSimplexFactorization::almostDestructor ( ) [inline]

Allows change of pivot accuracy check 1.0 == none >1.0 relaxed.

Delete all stuff (leaves as after **CoinFactorization()**)

Definition at line 338 of file AbcSimplexFactorization.hpp.

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

- AbcSimplexFactorization.hpp

4.14 AbcSimplexPrimal Class Reference

This solves LPs using the primal simplex method.

`#include <AbcSimplexPrimal.hpp>`

Inheritance diagram for AbcSimplexPrimal:

Collaboration diagram for AbcSimplexPrimal:

Classes

- struct pivotStruct

Public Member Functions

Description of algorithm

- int primal (int ifValuesPass=0, int startFinishOptions=0)
  Primal algorithm.

For advanced users

- void alwaysOptimal (bool onOff)
  Do not change infeasibility cost and always say optimal.
- bool **alwaysOptimal** () const
- void exactOutgoing (bool onOff)
  Normally outgoing variables can go out to slightly negative values (but within tolerance) - this is to help stability and and degeneracy.
- bool **exactOutgoing** () const

Functions used in primal

- int whileIterating (int valuesOption)

  This has the flow between re-factorizations.
- int pivotResult (int ifValuesPass=0)

  Do last half of an iteration.
- int **pivotResult4** (int ifValuesPass=0)
- int updatePrimalsInPrimal (**CoinIndexedVector** ∗rowArray, double theta, double &objectiveChange, int valuesPass)

  The primals are updated by the given array.
- void updatePrimalsInPrimal (**CoinIndexedVector** &rowArray, double theta, bool valuesPass)

  The primals are updated by the given array.
- void createUpdateDuals (**CoinIndexedVector** &rowArray, const double ∗originalCost, const double extra←Cost[4], double &objectiveChange, int valuesPass)

  After rowArray will have cost changes for use next iteration.
- double updateMinorCandidate (const **CoinIndexedVector** &updateBy, **CoinIndexedVector** &candidate, int sequenceIn)

  Update minor candidate vector - new reduced cost returned later try and get change in reduced cost (then may not need sequence in)
- void updatePartialUpdate (**CoinIndexedVector** &partialUpdate)

  Update partial Ftran by R update.
- int doFTUpdate (**CoinIndexedVector** ∗vector[4])

  Do FT update as separate function for minor iterations (nonzero return code on problems)
- void primalRow (**CoinIndexedVector** ∗rowArray, **CoinIndexedVector** ∗rhsArray, **CoinIndexedVector** ∗spareArray, int valuesPass)

  Row array has pivot column This chooses pivot row.
- void **primalRow** (**CoinIndexedVector** ∗rowArray, **CoinIndexedVector** ∗rhsArray, **CoinIndexedVector** ∗spareArray, pivotStruct &stuff)
- void primalColumn (**CoinPartitionedVector** ∗updateArray, **CoinPartitionedVector** ∗spareRow2, **Coin←PartitionedVector** ∗spareColumn1)

  Chooses primal pivot column updateArray has cost updates (also use pivotRow_ from last iteration) Would be faster with separate region to scan and will have this (with square of infeasibility) when steepest For easy problems we can just choose one of the first columns we look at.
- int checkUnbounded (**CoinIndexedVector** ∗ray, **CoinIndexedVector** ∗spare, double changeCost)

  Checks if tentative optimal actually means unbounded in primal Returns -3 if not, 2 if is unbounded.
- void statusOfProblemInPrimal (int type)

  Refactorizes if necessary Checks if finished.
- void perturb (int type)

  Perturbs problem (method depends on perturbation())
- bool unPerturb ()

  Take off effect of perturbation and say whether to try dual.
- int unflag ()

  Unflag all variables and return number unflagged.
- int nextSuperBasic (int superBasicType, **CoinIndexedVector** ∗columnArray)

  Get next superbasic -1 if none, Normal type is 1 If type is 3 then initializes sorted list if 2 uses list.
- void primalRay (**CoinIndexedVector** ∗rowArray)

  Create primal ray.
- void clearAll ()

  Clears all bits and clears rowArray[1] etc.
- int lexSolve ()

  Sort of lexicographic resolve.

Additional Inherited Members

4.14.1 Detailed Description

This solves LPs using the primal simplex method.

It inherits from AbcSimplex. It has no data of its own and is never created - only cast from a AbcSimplex object at algorithm time.

Definition at line 23 of file AbcSimplexPrimal.hpp.

4.14.2 Member Function Documentation

4.14.2.1 int AbcSimplexPrimal::primal (int *ifValuesPass* = 0, int *startFinishOptions* = 0)

Primal algorithm.

Method

It tries to be a single phase approach with a weight of 1.0 being given to getting optimal and a weight of infeasibility←Cost_ being given to getting primal feasible. In this version I have tried to be clever in a stupid way. The idea of fake bounds in dual seems to work so the primal analogue would be that of getting bounds on reduced costs (by a presolve approach) and using these for being above or below feasible region. I decided to waste memory and keep these explicitly. This allows for non-linear costs! I have not tested non-linear costs but will be glad to do something if a reasonable example is provided.

The code is designed to take advantage of sparsity so arrays are seldom zeroed out from scratch or gone over in their entirety. The only exception is a full scan to find incoming variable for Dantzig row choice. For steepest edge we keep an updated list of dual infeasibilities (actually squares). On easy problems we don't need full scan - just pick first reasonable. This method has not been coded.

One problem is how to tackle degeneracy and accuracy. At present I am using the modification of costs which I put in OSL and which was extended by Gill et al. I am still not sure whether we will also need explicit perturbation.

The flow of primal is three while loops as follows:

while (not finished) {

while (not clean solution) {

Factorize and/or clean up solution by changing bounds so primal feasible. If looks finished check fake primal bounds. Repeat until status is iterating (-1) or finished (0,1,2)

}

while (status==-1) {

Iterate until no pivot in or out or time to re-factorize.

Flow is:

choose pivot column (incoming variable). if none then we are primal feasible so looks as if done but we need to break and check bounds etc.

Get pivot column in tableau

```
Choose outgoing row.  If we don't find one then we look
```

primal unbounded so break and check bounds etc. (Also the pivot tolerance is larger after any iterations so that may be reason)

```
If we do find outgoing row, we may have to adjust costs to
```

keep going forwards (anti-degeneracy). Check pivot will be stable and if unstable throw away iteration and break to re-factorize. If minor error re-factorize after iteration.

Update everything (this may involve changing bounds on variables to stay primal feasible.

}

}

TODO's (or maybe not)

At present we never check we are going forwards. I overdid that in OSL so will try and make a last resort.

Needs partial scan pivot in option.

May need other anti-degeneracy measures, especially if we try and use loose tolerances as a way to solve in fewer iterations.

I like idea of dynamic scaling. This gives opportunity to decouple different implications of scaling for accuracy, iteration count and feasibility tolerance.

for use of exotic parameter startFinishoptions see Clpsimplex.hpp

4.14.2.2 void AbcSimplexPrimal::exactOutgoing (bool *onOff*)

Normally outgoing variables can go out to slightly negative values (but within tolerance) - this is to help stability and and degeneracy.

This can be switched off

4.14.2.3 int AbcSimplexPrimal::whileIterating (int *valuesOption*)

This has the flow between re-factorizations.

Returns a code to say where decision to exit was made Problem status set to:

-2 re-factorize -4 Looks optimal/infeasible -5 Looks unbounded +3 max iterations

valuesOption has original value of valuesPass

4.14.2.4 int AbcSimplexPrimal::pivotResult (int *ifValuesPass* = 0)

Do last half of an iteration.

This is split out so people can force incoming variable. If solveType_ is 2 then this may re-factorize while normally it would exit to re-factorize. Return codes Reasons to come out (normal mode/user mode): -1 normal -2 factorize now - good iteration/ NA -3 slight inaccuracy - refactorize - iteration done/ same but factor done -4 inaccuracy - refactorize - no iteration/ NA -5 something flagged - go round again/ pivot not possible +2 looks unbounded +3 max iterations (iteration done)

With solveType_ ==2 this should Pivot in a variable and choose an outgoing one. Assumes primal feasible - will not go through a bound. Returns step length in theta Returns ray in ray_

4.14.2.5 int AbcSimplexPrimal::updatePrimalsInPrimal (CoinIndexedVector * *rowArray*, double *theta*, double & *objectiveChange*, int *valuesPass*)

The primals are updated by the given array.

Returns number of infeasibilities. After rowArray will have cost changes for use next iteration

4.14.2.6 void AbcSimplexPrimal::updatePrimalsInPrimal (CoinIndexedVector & *rowArray,* double *theta,* bool *valuesPass*)

The primals are updated by the given array.

costs are changed

4.14.2.7 void AbcSimplexPrimal::primalRow (CoinIndexedVector ∗ *rowArray,* CoinIndexedVector ∗ *rhsArray,* CoinIndexedVector ∗ *spareArray,* int *valuesPass*)

Row array has pivot column This chooses pivot row.

Rhs array is used for distance to next bound (for speed) For speed, we may need to go to a bucket approach when many variables go through bounds If valuesPass non-zero then compute dj for direction

4.14.2.8 void AbcSimplexPrimal::statusOfProblemInPrimal (int *type*)

Refactorizes if necessary Checks if finished.

Updates status. lastCleaned refers to iteration at which some objective/feasibility cleaning too place.

type - 0 initial so set up save arrays etc

- 1 normal -if good update save

2 restoring from saved

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

- AbcSimplexPrimal.hpp

4.15 AbcTolerancesEtc Class Reference

Public Member Functions

Constructors and destructors

- AbcTolerancesEtc ()
  Default Constructor.
- AbcTolerancesEtc (const ClpSimplex ∗model)
  Useful Constructors.
- **AbcTolerancesEtc** (const AbcSimplex ∗model)
- AbcTolerancesEtc (const AbcTolerancesEtc &)
  Copy constructor.
- AbcTolerancesEtc & operator= (const AbcTolerancesEtc &rhs)
  Assignment operator.
- ∼AbcTolerancesEtc ()
  Destructor.

Public Attributes

Public member data

- double zeroTolerance_

Zero tolerance.
- double primalToleranceToGetOptimal_

  Primal tolerance needed to make dual feasible (<largeTolerance)
- double largeValue_

  Large bound value (for complementarity etc)
- double alphaAccuracy_

  For computing whether to re-factorize.
- double dualBound_

  Dual bound.
- double dualTolerance_

  Current dual tolerance for algorithm.
- double primalTolerance_

  Current primal tolerance for algorithm.
- double infeasibilityCost_

  Weight assigned to being infeasible in primal.
- double incomingInfeasibility_

  For advanced use.
- double **allowedInfeasibility_**
- int baseIteration_

  Iteration when we entered dual or primal.
- int numberRefinements_

  How many iterative refinements to do.
- int forceFactorization_

  Now for some reliability aids This forces re-factorization early.
- int perturbation_

  Perturbation: -50 to +50 - perturb by this power of ten (-6 sounds good) 100 - auto perturb if takes too long (1.0e-6 largest nonzero) 101 - we are perturbed 102 - don't try perturbing again default is 100.
- int dontFactorizePivots_

  If may skip final factorize then allow up to this pivots (default 20)
- int maximumPivots_

  For factorization Maximum number of pivots before factorization.

4.15.1 Detailed Description

Definition at line 256 of file CoinAbcCommon.hpp.

4.15.2 Member Data Documentation

4.15.2.1 double AbcTolerancesEtc::incomingInfeasibility_

For advanced use.

When doing iterative solves things can get nasty so on values pass if incoming solution has largest infeasibility < incomingInfeasibility throw out variables from basis until largest infeasibility < allowedInfeasibility. if allowedInfeasibility>= incomingInfeasibility this is always possible altough you may end up with an all slack basis.

Defaults are 1.0,10.0

Definition at line 312 of file CoinAbcCommon.hpp.

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

- CoinAbcCommon.hpp

4.16 AbcWarmStart Class Reference

As **CoinWarmStartBasis** but with alternatives (Also uses Clp status meaning for slacks)

`#include <AbcWarmStart.hpp>`

Inheritance diagram for AbcWarmStart:

Collaboration diagram for AbcWarmStart:

Public Member Functions

Methods to modify the warm start object

- virtual void setSize (int ns, int na)
  Set basis capacity; existing basis is discarded.
- virtual void resize (int newNumberRows, int newNumberColumns)
  Set basis capacity; existing basis is maintained.
- virtual void compressRows (int tgtCnt, const int ∗tgts)
  Delete a set of rows from the basis.
- virtual void deleteRows (int rawTgtCnt, const int ∗rawTgts)
  Delete a set of rows from the basis.
- virtual void deleteColumns (int number, const int ∗which)
  Delete a set of columns from the basis.
- void setModel (AbcSimplex ∗model)
  Set model.
- AbcSimplex ∗ model () const
  Get model.
- void createBasis0 (const AbcSimplex ∗model)
  Create Basis type 0.
- void createBasis12 (const AbcSimplex ∗model)
  Create Basis type 12.
- void createBasis34 (const AbcSimplex ∗model)
  Create Basis type 34.

Constructors, destructors, and related functions

- AbcWarmStart ()
  Default constructor.
- AbcWarmStart (AbcSimplex ∗model, int type)
  Constructs a warm start object with the specified status vectors.
- AbcWarmStart (const AbcWarmStart &ws)
  Copy constructor.
- virtual **CoinWarmStart** ∗ clone () const
  'Virtual constructor'
- virtual ∼AbcWarmStart ()
  Destructor.
- virtual AbcWarmStart & operator= (const AbcWarmStart &rhs)
  Assignment.
- virtual void assignBasisStatus (int ns, int na, char ∗&sStat, char ∗&aStat)
  Assign the status vectors to be the warm start information.

Protected Attributes

Protected data members

- int typeExtraInformation_

  Type of basis (always status arrays) 0 - as **CoinWarmStartBasis** *1,2 - plus factor order as shorts or ints (top bit set means column) 3,4 - plus compact saved factorization add 8 to say steepest edge weights stored (as floats) may want to change next,previous to tree info so can use a different basis for weights.*
- int lengthExtraInformation_

  Length of extra information in bytes.
- char ∗ extraInformation_

  The extra information.
- AbcSimplex ∗ model_

  Pointer back to AbcSimplex (can only be applied to that)
- AbcWarmStartOrganizer ∗ organizer_

  Pointer back to AbcWarmStartOrganizer for organization.
- AbcWarmStart ∗ previousBasis_

  Pointer to previous basis.
- AbcWarmStart ∗ nextBasis_

  Pointer to next basis.
- int stamp_

  Sequence stamp for deletion.
- int numberValidRows_

  Number of valid rows (rest should have slacks) Check to see if weights are OK for these rows and then just btran new ones for weights.

4.16.1 Detailed Description

As **CoinWarmStartBasis** but with alternatives (Also uses Clp status meaning for slacks)

Definition at line 75 of file AbcWarmStart.hpp.

4.16.2 Constructor & Destructor Documentation

4.16.2.1 AbcWarmStart::AbcWarmStart ( )

Default constructor.

Creates a warm start object representing an empty basis (0 rows, 0 columns).

4.16.2.2 AbcWarmStart::AbcWarmStart (AbcSimplex ∗ *model*, int *type*)

Constructs a warm start object with the specified status vectors.

The parameters are copied. Consider assignBasisStatus(int,int,char∗&,char∗&) if the object should assume ownership.

See also

AbcWarmStart::Status for a description of the packing used in the status arrays.

4.16.3 Member Function Documentation

4.16.3.1 virtual void AbcWarmStart::setSize (int *ns*, int *na*) [virtual]

Set basis capacity; existing basis is discarded.

After execution of this routine, the warm start object does not describe a valid basis: all structural and artificial variables have status isFree.

Reimplemented from **CoinWarmStartBasis**.

4.16.3.2 virtual void AbcWarmStart::resize (int *newNumberRows*, int *newNumberColumns*) [virtual]

Set basis capacity; existing basis is maintained.

After execution of this routine, the warm start object describes a valid basis: the status of new structural variables (added columns) is set to nonbasic at lower bound, and the status of new artificial variables (added rows) is set to basic. (The basis can be invalid if new structural variables do not have a finite lower bound.)

Reimplemented from **CoinWarmStartBasis**.

4.16.3.3 virtual void AbcWarmStart::compressRows (int *tgtCnt*, const int * *tgts*) [virtual]

Delete a set of rows from the basis.

Warning

This routine assumes that the set of indices to be deleted is sorted in ascending order and contains no duplicates. Use deleteRows() if this is not the case.
The resulting basis is guaranteed valid only if all deleted constraints are slack (hence the associated logicals are basic).

Removal of a tight constraint with a nonbasic logical implies that some basic variable must be made nonbasic. This correction is left to the client.

Reimplemented from **CoinWarmStartBasis**.

4.16.3.4 virtual void AbcWarmStart::deleteRows (int *rawTgtCnt*, const int * *rawTgts*) [virtual]

Delete a set of rows from the basis.

Warning

The resulting basis is guaranteed valid only if all deleted constraints are slack (hence the associated logicals are basic).

Removal of a tight constraint with a nonbasic logical implies that some basic variable must be made nonbasic. This correction is left to the client.

Reimplemented from **CoinWarmStartBasis**.

4.16.3.5 virtual void AbcWarmStart::deleteColumns (int *number*, const int * *which*) [virtual]

Delete a set of columns from the basis.

Warning

The resulting basis is guaranteed valid only if all deleted variables are nonbasic.

Removal of a basic variable implies that some nonbasic variable must be made basic. This correction is left to the client.

Reimplemented from **CoinWarmStartBasis**.

4.16.3.6 virtual void AbcWarmStart::assignBasisStatus (int *ns*, int *na*, char ∗& *sStat*, char ∗& *aStat*) [virtual]

Assign the status vectors to be the warm start information.

In this method the AbcWarmStart object assumes ownership of the pointers and upon return the argument pointers will be NULL. If copying is desirable, use the array constructor or the assignment operator .

Note

The pointers passed to this method will be freed using delete[], so they must be created using new[].

Reimplemented from **CoinWarmStartBasis**.

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

- AbcWarmStart.hpp

4.17 AbcWarmStartOrganizer Class Reference

Collaboration diagram for AbcWarmStartOrganizer:

Public Member Functions

- void createBasis0 ()

  Create Basis type 0.
- void createBasis12 ()

  Create Basis type 1,2.
- void createBasis34 ()

  Create Basis type 3,4.
- void deleteBasis (AbcWarmStart ∗basis)

  delete basis

Constructors, destructors, and related functions

- AbcWarmStartOrganizer (AbcSimplex ∗model=NULL)

  Default constructor.
- AbcWarmStartOrganizer (const AbcWarmStartOrganizer &ws)

  Copy constructor.
- virtual ∼AbcWarmStartOrganizer ()

  Destructor.
- virtual AbcWarmStartOrganizer & operator= (const AbcWarmStartOrganizer &rhs)

  Assignment.

Protected Attributes

Protected data members

- AbcSimplex * model_

  Pointer to AbcSimplex (can only be applied to that)
- AbcWarmStart * firstBasis_

  Pointer to first basis.
- AbcWarmStart * lastBasis_

  Pointer to last basis.
- int numberBases_

  Number of bases.
- int sizeBases_

  Size of bases (extra)

4.17.1 Detailed Description

Definition at line 23 of file AbcWarmStart.hpp.

4.17.2 Constructor & Destructor Documentation

4.17.2.1 AbcWarmStartOrganizer::AbcWarmStartOrganizer (AbcSimplex * *model* = NULL)

Default constructor.

Creates a warm start object organizer

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

- AbcWarmStart.hpp

4.18 ampl_info Struct Reference

4.18.1 Detailed Description

Definition at line 11 of file Clp_ampl.h.

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

- Clp_ampl.h

4.19 blockStruct Struct Reference

4.19.1 Detailed Description

Definition at line 562 of file ClpPackedMatrix.hpp.

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

- ClpPackedMatrix.hpp

4.20 blockStruct3 Struct Reference

4.20.1 Detailed Description

Definition at line 557 of file AbcMatrix.hpp.

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

- AbcMatrix.hpp

4.21 ClpNode::branchState Struct Reference

4.21.1 Detailed Description

Definition at line 121 of file ClpNode.hpp.

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

- ClpNode.hpp

4.22 CbcOrClpParam Class Reference

Very simple class for setting parameters.

`#include <CbcOrClpParam.hpp>`

Public Member Functions

Constructor and destructor

- CbcOrClpParam ()
  Constructors.
- **CbcOrClpParam** (std::string name, std::string help, double lower, double upper, CbcOrClpParameterType type, int display=2)
- **CbcOrClpParam** (std::string name, std::string help, int lower, int upper, CbcOrClpParameterType type, int display=2)
- **CbcOrClpParam** (std::string name, std::string help, std::string firstValue, CbcOrClpParameterType type, int whereUsed=7, int display=2)
- **CbcOrClpParam** (std::string name, std::string help, CbcOrClpParameterType type, int whereUsed=7, int display=2)
- CbcOrClpParam (const CbcOrClpParam &)
  Copy constructor.
- CbcOrClpParam & operator= (const CbcOrClpParam &rhs)
  Assignment operator. This copies the data.
- ∼CbcOrClpParam ()
  Destructor.

stuff

- void append (std::string keyWord)

Insert string (only valid for keywords)

- void addHelp (std::string keyWord)

  Adds one help line.
- std::string name () const

  Returns name.
- std::string shortHelp () const

  Returns short help.
- int setDoubleParameter (CbcModel &model, double value)

  Sets a double parameter (nonzero code if error)
- const char ∗ setDoubleParameterWithMessage (CbcModel &model, double value, int &returnCode)

  Sets double parameter and returns printable string and error code.
- double doubleParameter (CbcModel &model) const

  Gets a double parameter.
- int setIntParameter (CbcModel &model, int value)

  Sets a int parameter (nonzero code if error)
- const char ∗ setIntParameterWithMessage (CbcModel &model, int value, int &returnCode)

  Sets int parameter and returns printable string and error code.
- int intParameter (CbcModel &model) const

  Gets a int parameter.
- int setDoubleParameter (ClpSimplex ∗model, double value)

  Sets a double parameter (nonzero code if error)
- double doubleParameter (ClpSimplex ∗model) const

  Gets a double parameter.
- const char ∗ setDoubleParameterWithMessage (ClpSimplex ∗model, double value, int &returnCode)

  Sets double parameter and returns printable string and error code.
- int setIntParameter (ClpSimplex ∗model, int value)

  Sets a int parameter (nonzero code if error)
- const char ∗ setIntParameterWithMessage (ClpSimplex ∗model, int value, int &returnCode)

  Sets int parameter and returns printable string and error code.
- int intParameter (ClpSimplex ∗model) const

  Gets a int parameter.
- int setDoubleParameter (**OsiSolverInterface** ∗model, double value)

  Sets a double parameter (nonzero code if error)
- const char ∗ setDoubleParameterWithMessage (**OsiSolverInterface** ∗model, double value, int &returnCode)

  Sets double parameter and returns printable string and error code.
- double doubleParameter (**OsiSolverInterface** ∗model) const

  Gets a double parameter.
- int setIntParameter (**OsiSolverInterface** ∗model, int value)

  Sets a int parameter (nonzero code if error)
- const char ∗ setIntParameterWithMessage (**OsiSolverInterface** ∗model, int value, int &returnCode)

  Sets int parameter and returns printable string and error code.
- int intParameter (**OsiSolverInterface** ∗model) const

  Gets a int parameter.
- int checkDoubleParameter (double value) const

  Checks a double parameter (nonzero code if error)
- std::string matchName () const

  Returns name which could match.

- int lengthMatchName () const

  Returns length of name for ptinting.
- int parameterOption (std::string check) const

  Returns parameter option which matches (-1 if none)
- void printOptions () const

  Prints parameter options.
- std::string currentOption () const

  Returns current parameter option.
- void setCurrentOption (int value, bool **printIt**=false)

  Sets current parameter option.
- const char ∗ setCurrentOptionWithMessage (int value)

  Sets current parameter option and returns printable string.
- void setCurrentOption (const std::string value)

  Sets current parameter option using string.
- const char ∗ setCurrentOptionWithMessage (const std::string value)

  Sets current parameter option using string with message.
- int currentOptionAsInteger () const

  Returns current parameter option position.
- int currentOptionAsInteger (int &fakeInteger) const

  Returns current parameter option position but if fake keyword returns a fake value and sets fakeInteger to true value.
- void setIntValue (int value)

  Sets int value.
- const char ∗ setIntValueWithMessage (int value)

  Sets int value with message.
- int **intValue** () const
- void setDoubleValue (double value)

  Sets double value.
- const char ∗ setDoubleValueWithMessage (double value)

  Sets double value with message.
- double **doubleValue** () const
- void setStringValue (std::string value)

  Sets string value.
- std::string **stringValue** () const
- int matches (std::string input) const

  Returns 1 if matches minimum, 2 if matches less, 0 if not matched.
- CbcOrClpParameterType type () const

  type
- int displayThis () const

  whether to display
- void setLonghelp (const std::string help)

  Set Long help.
- void printLongHelp () const

  Print Long help.
- void printString () const

  Print action and string.
- int whereUsed () const

  7 if used everywhere, 1 - used by clp 2 - used by cbc 4 - used by ampl

- int fakeKeyWord () const

  Gets value of fake keyword.
- void setFakeKeyWord (int value, int fakeValue)

  Sets value of fake keyword.
- void setFakeKeyWord (int fakeValue)

  Sets value of fake keyword to current size of keywords.

4.22.1 Detailed Description

Very simple class for setting parameters.

Definition at line 295 of file CbcOrClpParam.hpp.

4.22.2 Member Function Documentation

4.22.2.1 int CbcOrClpParam::currentOptionAsInteger (int & *fakeInteger*) const

Returns current parameter option position but if fake keyword returns a fake value and sets fakeInteger to true value.

If not fake then fakeInteger is -COIN_INT_MAX

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

- CbcOrClpParam.hpp

4.23 ClpCholeskyBase Class Reference

Base class for Clp Cholesky factorization Will do better factorization.

`#include <ClpCholeskyBase.hpp>`

Inheritance diagram for ClpCholeskyBase:

Collaboration diagram for ClpCholeskyBase:

Public Member Functions

Gets

- int status () const

  status. Returns status
- int numberRowsDropped () const

  numberRowsDropped. Number of rows gone
- void resetRowsDropped ()

  reset numberRowsDropped and rowsDropped.
- char ∗ rowsDropped () const

  rowsDropped - which rows are gone
- double choleskyCondition () const

  choleskyCondition.
- double goDense () const

  goDense i.e. use dense factoriaztion if > this (default 0.7).

- void setGoDense (double value)
  goDense i.e. use dense factoriaztion if > this (default 0.7).
- int rank () const
  rank. Returns rank
- int numberRows () const
  Return number of rows.
- CoinBigIndex size () const
  Return size.
- longDouble * sparseFactor () const
  Return sparseFactor.
- longDouble * diagonal () const
  Return diagonal.
- longDouble * workDouble () const
  Return workDouble.
- bool kkt () const
  If KKT on.
- void setKKT (bool yesNo)
  Set KKT.
- void setIntegerParameter (int i, int value)
  Set integer parameter.
- int getIntegerParameter (int i)
  get integer parameter
- void setDoubleParameter (int i, double value)
  Set double parameter.
- double getDoubleParameter (int i)
  get double parameter

Constructors, destructor

- ClpCholeskyBase (int denseThreshold=-1)
  Constructor which has dense columns activated.
- virtual ~ClpCholeskyBase ()
  Destructor (has to be public)
- ClpCholeskyBase (const ClpCholeskyBase &)
  Copy.
- ClpCholeskyBase & operator= (const ClpCholeskyBase &)
  Assignment.

Protected Member Functions

Symbolic, factor and solve

- int symbolic1 (const CoinBigIndex *Astart, const int *Arow)
  Symbolic1 - works out size without clever stuff.
- void symbolic2 (const CoinBigIndex *Astart, const int *Arow)
  Symbolic2 - Fills in indices Uses lower triangular so can do cliques etc.
- void factorizePart2 (int *rowsDropped)
  Factorize - filling in rowsDropped and returning number dropped in integerParam.
- void solve (CoinWorkDouble *region, int type)
  solve - 1 just first half, 2 just second half - 3 both.
- int preOrder (bool lowerTriangular, bool includeDiagonal, bool doKKT)
  Forms ADAT - returns nonzero if not enough memory.
- void updateDense (longDouble *d, int *first)
  Updates dense part (broken out for profiling)

Protected Attributes

Data members

The data members are protected to allow access for derived classes.

- int type_
  type (may be useful) if > 20 do KKT
- bool doKKT_
  Doing full KKT (only used if default symbolic and factorization)
- double goDense_
  Go dense at this fraction.
- double choleskyCondition_
  choleskyCondition.
- ClpInterior * model_
  model.
- int numberTrials_
  numberTrials. Number of trials before rejection
- int numberRows_
  numberRows. Number of Rows in factorization
- int status_
  status. Status of factorization
- char * rowsDropped_
  rowsDropped
- int * permuteInverse_
  permute inverse.
- int * permute_
  main permute.
- int numberRowsDropped_
  numberRowsDropped. Number of rows gone
- longDouble * sparseFactor_
  sparseFactor.
- CoinBigIndex * choleskyStart_
  choleskyStart - element starts
- int * choleskyRow_
  choleskyRow (can be shorter than sparsefactor)
- CoinBigIndex * indexStart_
  Index starts.
- longDouble * diagonal_
  Diagonal.
- longDouble * workDouble_
  double work array
- int * link_
  link array
- CoinBigIndex * **workInteger_**
- int * **clique_**
- CoinBigIndex sizeFactor_
  sizeFactor.
- CoinBigIndex sizeIndex_
  Size of index array.
- int firstDense_
  First dense row.
- int integerParameters_ [64]

integerParameters
- double doubleParameters_ [64]

  doubleParameters;
- ClpMatrixBase ∗ rowCopy_

  Row copy of matrix.
- char ∗ whichDense_

  Dense indicators.
- longDouble ∗ denseColumn_

  Dense columns (updated)
- ClpCholeskyDense ∗ dense_

  Dense cholesky.
- int denseThreshold_

  Dense threshold (for taking out of Cholesky)

Virtual methods that the derived classes may provide

- virtual int order (ClpInterior ∗model)

  Orders rows and saves pointer to matrix.and model.
- virtual int symbolic ()

  Does Symbolic factorization given permutation.
- virtual int factorize (const CoinWorkDouble ∗diagonal, int ∗rowsDropped)

  Factorize - filling in rowsDropped and returning number dropped.
- virtual void solve (CoinWorkDouble ∗region)

  Uses factorization to solve.
- virtual void solveKKT (CoinWorkDouble ∗region1, CoinWorkDouble ∗region2, const CoinWorkDouble ∗diagonal, CoinWorkDouble diagonalScaleFactor)

  Uses factorization to solve.

Other

Clone

- virtual ClpCholeskyBase ∗ **clone** () const
- int type () const

  Returns type.
- void setType (int type)

  Sets type.
- void setModel (ClpInterior ∗model)

  model.

4.23.1 Detailed Description

Base class for Clp Cholesky factorization Will do better factorization.

very crude ordering

Derived classes may be using more sophisticated methods

Definition at line 53 of file ClpCholeskyBase.hpp.

4.23.2 Constructor & Destructor Documentation

4.23.2.1 ClpCholeskyBase::ClpCholeskyBase (int *denseThreshold* = -1)

Constructor which has dense columns activated.

Default is off.

4.23.3 Member Function Documentation

4.23.3.1 virtual int ClpCholeskyBase::order (ClpInterior * *model*) [virtual]

Orders rows and saves pointer to matrix.and model.

returns non-zero if not enough memory. You can use preOrder to set up ADAT If using default symbolic etc then must set sizeFactor_ to size of input matrix to order (and to symbolic). Also just permute_ and permuteInverse_ should be created

Reimplemented in ClpCholeskyTaucs, ClpCholeskyUfl, ClpCholeskyMumps, ClpCholeskyWssmp, ClpCholeskyWssmpKKT, and ClpCholeskyDense.

4.23.3.2 virtual int ClpCholeskyBase::symbolic ( ) [virtual]

Does Symbolic factorization given permutation.

This is called immediately after order. If user provides this then user must provide factorize and solve. Otherwise the default factorization is used returns non-zero if not enough memory

Reimplemented in ClpCholeskyTaucs, ClpCholeskyUfl, ClpCholeskyMumps, ClpCholeskyWssmp, ClpCholeskyWssmpKKT, and ClpCholeskyDense.

4.23.3.3 virtual int ClpCholeskyBase::factorize (const CoinWorkDouble * *diagonal*, int * *rowsDropped*) [virtual]

Factorize - filling in rowsDropped and returning number dropped.

If return code negative then out of memory

Reimplemented in ClpCholeskyDense.

4.23.3.4 virtual void ClpCholeskyBase::solve (CoinWorkDouble * *region*) [virtual]

Uses factorization to solve.

Reimplemented in ClpCholeskyDense.

4.23.3.5 virtual void ClpCholeskyBase::solveKKT (CoinWorkDouble * *region1*, CoinWorkDouble * *region2*, const CoinWorkDouble * *diagonal*, CoinWorkDouble *diagonalScaleFactor*) [virtual]

Uses factorization to solve.

- given as if KKT. region1 is rows+columns, region2 is rows

4.23.3.6 int ClpCholeskyBase::symbolic1 (const CoinBigIndex * *Astart*, const int * *Arow*) [protected]

Symbolic1 - works out size without clever stuff.

Uses upper triangular as much easier. Returns size

4.23.3.7 void ClpCholeskyBase::solve (CoinWorkDouble * *region*, int *type*) [protected]

solve - 1 just first half, 2 just second half - 3 both.

If 1 and 2 then diagonal has sqrt of inverse otherwise inverse

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

- ClpCholeskyBase.hpp

4.24 ClpCholeskyDense Class Reference

Inheritance diagram for ClpCholeskyDense:

Collaboration diagram for ClpCholeskyDense:

Public Member Functions

Virtual methods that the derived classes provides

- virtual int order (ClpInterior *model)
  Orders rows and saves pointer to matrix.and model.
- virtual int symbolic ()
  Does Symbolic factorization given permutation.
- virtual int factorize (const CoinWorkDouble *diagonal, int *rowsDropped)
  Factorize - filling in rowsDropped and returning number dropped.
- virtual void solve (CoinWorkDouble *region)
  Uses factorization to solve.

Non virtual methods for ClpCholeskyDense

- int reserveSpace (const ClpCholeskyBase *factor, int numberRows)
  Reserves space.
- CoinBigIndex space (int numberRows) const
  Returns space needed.
- void factorizePart2 (int *rowsDropped)
  part 2 of Factorize - filling in rowsDropped
- void factorizePart3 (int *rowsDropped)
  part 2 of Factorize - filling in rowsDropped - blocked
- void solveF1 (longDouble *a, int n, CoinWorkDouble *region)
  Forward part of solve.
- void **solveF2** (longDouble *a, int n, CoinWorkDouble *region, CoinWorkDouble *region2)
- void solveB1 (longDouble *a, int n, CoinWorkDouble *region)
  Backward part of solve.
- void **solveB2** (longDouble *a, int n, CoinWorkDouble *region, CoinWorkDouble *region2)
- int **bNumber** (const longDouble *array, int &, int &)
- longDouble * aMatrix () const

A.

- longDouble * diagonal () const

  Diagonal.

Constructors, destructor

- ClpCholeskyDense ()

  Default constructor.
- virtual ~ClpCholeskyDense ()

  Destructor.
- ClpCholeskyDense (const ClpCholeskyDense &)

  Copy.
- ClpCholeskyDense & operator= (const ClpCholeskyDense &)

  Assignment.
- virtual ClpCholeskyBase * clone () const

  Clone.

Additional Inherited Members

4.24.1 Detailed Description

Definition at line 14 of file ClpCholeskyDense.hpp.

4.24.2 Constructor & Destructor Documentation

4.24.2.1 ClpCholeskyDense::ClpCholeskyDense ( )

Default constructor.

4.24.3 Member Function Documentation

4.24.3.1 virtual int ClpCholeskyDense::order (ClpInterior * *model*) [virtual]

Orders rows and saves pointer to matrix.and model.

Returns non-zero if not enough memory

Reimplemented from ClpCholeskyBase.

4.24.3.2 virtual int ClpCholeskyDense::symbolic ( ) [virtual]

Does Symbolic factorization given permutation.

This is called immediately after order. If user provides this then user must provide factorize and solve. Otherwise the default factorization is used returns non-zero if not enough memory

Reimplemented from ClpCholeskyBase.

4.24.3.3 virtual int ClpCholeskyDense::factorize (const CoinWorkDouble * *diagonal*, int * *rowsDropped*) [virtual]

Factorize - filling in rowsDropped and returning number dropped.

If return code negative then out of memory

Reimplemented from ClpCholeskyBase.

4.24.3.4 virtual void ClpCholeskyDense::solve (CoinWorkDouble * *region*) [virtual]

Uses factorization to solve.

Reimplemented from ClpCholeskyBase.

4.24.3.5 int ClpCholeskyDense::reserveSpace (const ClpCholeskyBase * *factor*, int *numberRows*)

Reserves space.

If factor not NULL then just uses passed space Returns non-zero if not enough memory

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

- ClpCholeskyDense.hpp

4.25 ClpCholeskyDenseC Struct Reference

4.25.1 Detailed Description

Definition at line 88 of file ClpCholeskyDense.hpp.

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

- ClpCholeskyDense.hpp

4.26 ClpCholeskyMumps Class Reference

Mumps class for Clp Cholesky factorization.

`#include <ClpCholeskyMumps.hpp>`

Inheritance diagram for ClpCholeskyMumps:

Collaboration diagram for ClpCholeskyMumps:

Public Member Functions

Virtual methods that the derived classes provides

- virtual int order (ClpInterior *model)
  Orders rows and saves pointer to matrix.and model.
- virtual int symbolic ()
  Does Symbolic factorization given permutation.
- virtual int factorize (const double *diagonal, int *rowsDropped)

Factorize - filling in rowsDropped and returning number dropped.
- virtual void solve (double ∗region)

  Uses factorization to solve.

Constructors, destructor

- ClpCholeskyMumps (int denseThreshold=-1)

  Constructor which has dense columns activated.
- virtual ∼ClpCholeskyMumps ()

  Destructor.
- virtual ClpCholeskyBase ∗ clone () const

  Clone.

Additional Inherited Members

4.26.1 Detailed Description

Mumps class for Clp Cholesky factorization.

Definition at line 21 of file ClpCholeskyMumps.hpp.

4.26.2 Constructor & Destructor Documentation

4.26.2.1 ClpCholeskyMumps::ClpCholeskyMumps (int *denseThreshold* = -1)

Constructor which has dense columns activated.

Default is off.

4.26.3 Member Function Documentation

4.26.3.1 virtual int ClpCholeskyMumps::order (ClpInterior ∗ *model*) [virtual]

Orders rows and saves pointer to matrix.and model.

Returns non-zero if not enough memory

Reimplemented from ClpCholeskyBase.

4.26.3.2 virtual int ClpCholeskyMumps::symbolic ( ) [virtual]

Does Symbolic factorization given permutation.

This is called immediately after order. If user provides this then user must provide factorize and solve. Otherwise the default factorization is used returns non-zero if not enough memory

Reimplemented from ClpCholeskyBase.

4.26.3.3 virtual int ClpCholeskyMumps::factorize (const double ∗ *diagonal,* int ∗ *rowsDropped*) [virtual]

Factorize - filling in rowsDropped and returning number dropped.

If return code negative then out of memory

4.26.3.4 virtual void ClpCholeskyMumps::solve (double * *region*) [virtual]

Uses factorization to solve.

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

- ClpCholeskyMumps.hpp

4.27 ClpCholeskyTaucs Class Reference

Taucs class for Clp Cholesky factorization.

`#include <ClpCholeskyTaucs.hpp>`

Inheritance diagram for ClpCholeskyTaucs:

Collaboration diagram for ClpCholeskyTaucs:

Public Member Functions

Virtual methods that the derived classes provides

- virtual int order (ClpInterior *model)
  Orders rows and saves pointer to matrix.and model.
- virtual int symbolic ()
  Dummy.
- virtual int factorize (const double *diagonal, int *rowsDropped)
  Factorize - filling in rowsDropped and returning number dropped.
- virtual void solve (double *region)
  Uses factorization to solve.

Constructors, destructor

- ClpCholeskyTaucs ()
  Default constructor.
- virtual ~ClpCholeskyTaucs ()
  Destructor.
- **ClpCholeskyTaucs** (const ClpCholeskyTaucs &)
- ClpCholeskyTaucs & **operator=** (const ClpCholeskyTaucs &)
- virtual ClpCholeskyBase * clone () const
  Clone.

Additional Inherited Members

4.27.1 Detailed Description

Taucs class for Clp Cholesky factorization.

If you wish to use Sivan Toledo's TAUCS code see

`http://www.tau.ac.il/~stoledo/taucs/`

for terms of use

The taucs.h file was modified to put

#ifdef __cplusplus extern "C"{ #endif after line 440 (#endif) and #ifdef __cplusplus } #endif at end

I also modified LAPACK dpotf2.f (two places) to change the GO TO 30 on AJJ.Lt.0.0

to

```
IF( AJJ.LE.1.0e-20 ) THEN
   AJJ = 1.0e100;
ELSE
   AJJ = SQRT( AJJ )
END IF
```

Definition at line 43 of file ClpCholeskyTaucs.hpp.

4.27.2 Constructor & Destructor Documentation

4.27.2.1 ClpCholeskyTaucs::ClpCholeskyTaucs ( )

Default constructor.

4.27.3 Member Function Documentation

4.27.3.1 virtual int ClpCholeskyTaucs::order (ClpInterior * *model*) [virtual]

Orders rows and saves pointer to matrix.and model.

Returns non-zero if not enough memory

Reimplemented from ClpCholeskyBase.

4.27.3.2 virtual int ClpCholeskyTaucs::factorize (const double * *diagonal,* int * *rowsDropped*) [virtual]

Factorize - filling in rowsDropped and returning number dropped.

If return code negative then out of memory

4.27.3.3 virtual void ClpCholeskyTaucs::solve (double * *region*) [virtual]

Uses factorization to solve.

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

- ClpCholeskyTaucs.hpp

4.28 ClpCholeskyUfl Class Reference

Ufl class for Clp Cholesky factorization.

`#include <ClpCholeskyUfl.hpp>`

Inheritance diagram for ClpCholeskyUfl:

Collaboration diagram for ClpCholeskyUfl:

Public Member Functions

Virtual methods that the derived classes provides

- virtual int order (ClpInterior ∗model)
  Orders rows and saves pointer to matrix.and model.
- virtual int symbolic ()
  Does Symbolic factorization given permutation using CHOLMOD (if available).
- virtual int factorize (const double ∗diagonal, int ∗rowsDropped)
  Factorize - filling in rowsDropped and returning number dropped using CHOLMOD (if available).
- virtual void solve (double ∗region)
  Uses factorization to solve.

Constructors, destructor

- ClpCholeskyUfl (int denseThreshold=-1)
  Constructor which has dense columns activated.
- virtual ∼ClpCholeskyUfl ()
  Destructor.
- virtual ClpCholeskyBase ∗ clone () const
  Clone.

Additional Inherited Members

4.28.1 Detailed Description

Ufl class for Clp Cholesky factorization.

If you wish to use AMD code from University of Florida see

```
http://www.cise.ufl.edu/research/sparse/amd
```

for terms of use

If you wish to use CHOLMOD code from University of Florida see

```
http://www.cise.ufl.edu/research/sparse/cholmod
```

for terms of use

Definition at line 32 of file ClpCholeskyUfl.hpp.

4.28.2 Constructor & Destructor Documentation

4.28.2.1 ClpCholeskyUfl::ClpCholeskyUfl (int *denseThreshold* = -1)

Constructor which has dense columns activated.

Default is off.

4.28.3 Member Function Documentation

4.28.3.1 virtual int ClpCholeskyUfl::order (ClpInterior * *model*) [virtual]

Orders rows and saves pointer to matrix.and model.

Returns non-zero if not enough memory

Reimplemented from ClpCholeskyBase.

4.28.3.2 virtual int ClpCholeskyUfl::symbolic ( ) [virtual]

Does Symbolic factorization given permutation using CHOLMOD (if available).

This is called immediately after order. If user provides this then user must provide factorize and solve. Otherwise the default factorization is used returns non-zero if not enough memory.

Reimplemented from ClpCholeskyBase.

4.28.3.3 virtual int ClpCholeskyUfl::factorize (const double * *diagonal*, int * *rowsDropped*) [virtual]

Factorize - filling in rowsDropped and returning number dropped using CHOLMOD (if available).

If return code negative then out of memory

4.28.3.4 virtual void ClpCholeskyUfl::solve (double * *region*) [virtual]

Uses factorization to solve.

Uses CHOLMOD (if available).

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

- ClpCholeskyUfl.hpp

4.29 ClpCholeskyWssmp Class Reference

Wssmp class for Clp Cholesky factorization.

`#include <ClpCholeskyWssmp.hpp>`

Inheritance diagram for ClpCholeskyWssmp:

Collaboration diagram for ClpCholeskyWssmp:

Public Member Functions

Virtual methods that the derived classes provides

- virtual int order (ClpInterior *model)
  Orders rows and saves pointer to matrix.and model.
- virtual int symbolic ()
  Does Symbolic factorization given permutation.
- virtual int factorize (const double *diagonal, int *rowsDropped)
  Factorize - filling in rowsDropped and returning number dropped.

- virtual void solve (double *region)
  Uses factorization to solve.

Constructors, destructor

- ClpCholeskyWssmp (int denseThreshold=-1)
  Constructor which has dense columns activated.
- virtual ~ClpCholeskyWssmp ()
  Destructor.
- **ClpCholeskyWssmp** (const ClpCholeskyWssmp &)
- ClpCholeskyWssmp & **operator=** (const ClpCholeskyWssmp &)
- virtual ClpCholeskyBase * clone () const
  Clone.

Additional Inherited Members

4.29.1 Detailed Description

Wssmp class for Clp Cholesky factorization.

Definition at line 17 of file ClpCholeskyWssmp.hpp.

4.29.2 Constructor & Destructor Documentation

4.29.2.1 ClpCholeskyWssmp::ClpCholeskyWssmp (int *denseThreshold* = -1)

Constructor which has dense columns activated.

Default is off.

4.29.3 Member Function Documentation

4.29.3.1 virtual int ClpCholeskyWssmp::order (ClpInterior * *model*) [virtual]

Orders rows and saves pointer to matrix.and model.

Returns non-zero if not enough memory

Reimplemented from ClpCholeskyBase.

4.29.3.2 virtual int ClpCholeskyWssmp::symbolic ( ) [virtual]

Does Symbolic factorization given permutation.

This is called immediately after order. If user provides this then user must provide factorize and solve. Otherwise the default factorization is used returns non-zero if not enough memory

Reimplemented from ClpCholeskyBase.

4.29.3.3 virtual int ClpCholeskyWssmp::factorize (const double * *diagonal*, int * *rowsDropped*) [virtual]

Factorize - filling in rowsDropped and returning number dropped.

If return code negative then out of memory

4.29.3.4 virtual void ClpCholeskyWssmp::solve (double ∗ *region*) [virtual]

Uses factorization to solve.

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

- ClpCholeskyWssmp.hpp

4.30 ClpCholeskyWssmpKKT Class Reference

WssmpKKT class for Clp Cholesky factorization.

`#include <ClpCholeskyWssmpKKT.hpp>`

Inheritance diagram for ClpCholeskyWssmpKKT:

Collaboration diagram for ClpCholeskyWssmpKKT:

Public Member Functions

Virtual methods that the derived classes provides

- virtual int order (ClpInterior ∗model)
  Orders rows and saves pointer to matrix.and model.
- virtual int symbolic ()
  Does Symbolic factorization given permutation.
- virtual int factorize (const double ∗diagonal, int ∗rowsDropped)
  Factorize - filling in rowsDropped and returning number dropped.
- virtual void solve (double ∗region)
  Uses factorization to solve.
- virtual void solveKKT (double ∗region1, double ∗region2, const double ∗diagonal, double diagonalScaleFactor)
  Uses factorization to solve.

Constructors, destructor

- ClpCholeskyWssmpKKT (int denseThreshold=-1)
  Constructor which has dense columns activated.
- virtual ∼ClpCholeskyWssmpKKT ()
  Destructor.
- **ClpCholeskyWssmpKKT** (const ClpCholeskyWssmpKKT &)
- ClpCholeskyWssmpKKT & **operator=** (const ClpCholeskyWssmpKKT &)
- virtual ClpCholeskyBase ∗ clone () const
  Clone.

Additional Inherited Members

4.30.1 Detailed Description

WssmpKKT class for Clp Cholesky factorization.

Definition at line 17 of file ClpCholeskyWssmpKKT.hpp.

4.30.2 Constructor & Destructor Documentation

4.30.2.1 ClpCholeskyWssmpKKT::ClpCholeskyWssmpKKT (int *denseThreshold* = -1)

Constructor which has dense columns activated.

Default is off.

4.30.3 Member Function Documentation

4.30.3.1 virtual int ClpCholeskyWssmpKKT::order (ClpInterior * *model*) [virtual]

Orders rows and saves pointer to matrix.and model.

Returns non-zero if not enough memory

Reimplemented from ClpCholeskyBase.

4.30.3.2 virtual int ClpCholeskyWssmpKKT::symbolic ( ) [virtual]

Does Symbolic factorization given permutation.

This is called immediately after order. If user provides this then user must provide factorize and solve. Otherwise the default factorization is used returns non-zero if not enough memory

Reimplemented from ClpCholeskyBase.

4.30.3.3 virtual int ClpCholeskyWssmpKKT::factorize (const double * *diagonal*, int * *rowsDropped*) [virtual]

Factorize - filling in rowsDropped and returning number dropped.

If return code negative then out of memory

4.30.3.4 virtual void ClpCholeskyWssmpKKT::solve (double * *region*) [virtual]

Uses factorization to solve.

4.30.3.5 virtual void ClpCholeskyWssmpKKT::solveKKT (double * *region1*, double * *region2*, const double * *diagonal*, double *diagonalScaleFactor*) [virtual]

Uses factorization to solve.

- given as if KKT. region1 is rows+columns, region2 is rows

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

- ClpCholeskyWssmpKKT.hpp

4.31 ClpConstraint Class Reference

Constraint Abstract Base Class.

`#include <ClpConstraint.hpp>`

Inheritance diagram for ClpConstraint:

Public Member Functions

Stuff

- virtual int gradient (const ClpSimplex *model, const double *solution, double *gradient, double &functionValue, double &offset, bool useScaling=false, bool refresh=true) const =0

    Fills gradient.
- virtual double functionValue (const ClpSimplex *model, const double *solution, bool useScaling=false, bool refresh=true) const

    Constraint function value.
- virtual void resize (int newNumberColumns)=0

    Resize constraint.
- virtual void deleteSome (int numberToDelete, const int *which)=0

    Delete columns in constraint.
- virtual void reallyScale (const double *columnScale)=0

    Scale constraint.
- virtual int markNonlinear (char *which) const =0

    Given a zeroed array sets nonlinear columns to 1.
- virtual int markNonzero (char *which) const =0

    Given a zeroed array sets possible nonzero coefficients to 1.

Constructors and destructors

- ClpConstraint ()

    Default Constructor.
- ClpConstraint (const ClpConstraint &)

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

    Assignment operator.
- virtual ∼ClpConstraint ()

    Destructor.
- virtual ClpConstraint * clone () const =0

    Clone.

Other

- int type ()

    Returns type, 0 linear, 1 nonlinear.
- int rowNumber () const

    Row number (-1 is objective)
- virtual int numberCoefficients () const =0

    Number of possible coefficients in gradient.
- double functionValue () const

    Stored constraint function value.
- double offset () const

    Constraint offset.
- virtual void newXValues ()

    Say we have new primal solution - so may need to recompute.

Protected Attributes

Protected member data

- double * lastGradient_

  Gradient at last evaluation.
- double functionValue_

  Value of non-linear part of constraint.
- double offset_

  Value of offset for constraint.
- int type_

  Type of constraint - linear is 1.
- int rowNumber_

  Row number (-1 is objective)

4.31.1 Detailed Description

Constraint Abstract Base Class.

Abstract Base Class for describing a constraint or objective function

Definition at line 19 of file ClpConstraint.hpp.

4.31.2 Member Function Documentation

4.31.2.1 virtual int ClpConstraint::gradient (const ClpSimplex * *model,* const double * *solution,* double * *gradient,* double & *functionValue,* double & *offset,* bool *useScaling* = false*,* bool *refresh* = true) const [pure virtual]

Fills gradient.

If Linear then solution may be NULL, also returns true value of function and offset so we can use x not deltaX in constraint If refresh is false then uses last solution Uses model for scaling Returns non-zero if gradient undefined at current solution

Implemented in ClpConstraintLinear, and ClpConstraintQuadratic.

4.31.2.2 virtual int ClpConstraint::markNonlinear (char * *which*) const [pure virtual]

Given a zeroed array sets nonlinear columns to 1.

Returns number of nonlinear columns

Implemented in ClpConstraintLinear, and ClpConstraintQuadratic.

4.31.2.3 virtual int ClpConstraint::markNonzero (char * *which*) const [pure virtual]

Given a zeroed array sets possible nonzero coefficients to 1.

Returns number of nonzeros

Implemented in ClpConstraintLinear, and ClpConstraintQuadratic.

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

- ClpConstraint.hpp

4.32 ClpConstraintLinear Class Reference

Linear Constraint Class.

`#include <ClpConstraintLinear.hpp>`

Inheritance diagram for ClpConstraintLinear:

Collaboration diagram for ClpConstraintLinear:

Public Member Functions

Stuff

- virtual int gradient (const ClpSimplex *model, const double *solution, double *gradient, double &functionValue, double &offset, bool useScaling=false, bool refresh=true) const
  Fills gradient.
- virtual void resize (int newNumberColumns)
  Resize constraint.
- virtual void deleteSome (int numberToDelete, const int *which)
  Delete columns in constraint.
- virtual void reallyScale (const double *columnScale)
  Scale constraint.
- virtual int markNonlinear (char *which) const
  Given a zeroed array sets nonlinear columns to 1.
- virtual int markNonzero (char *which) const
  Given a zeroed array sets possible nonzero coefficients to 1.

Constructors and destructors

- ClpConstraintLinear ()
  Default Constructor.
- ClpConstraintLinear (int row, int numberCoefficients, int numberColumns, const int *column, const double *element)
  Constructor from constraint.
- ClpConstraintLinear (const ClpConstraintLinear &rhs)
  Copy constructor .
- ClpConstraintLinear & operator= (const ClpConstraintLinear &rhs)
  Assignment operator.
- virtual ~ClpConstraintLinear ()
  Destructor.
- virtual ClpConstraint * clone () const
  Clone.

Gets and sets

- virtual int numberCoefficients () const
  Number of coefficients.
- int numberColumns () const
  Number of columns in linear constraint.
- const int * column () const
  Columns.
- const double * coefficient () const
  Coefficients.

Additional Inherited Members

4.32.1 Detailed Description

Linear Constraint Class.

Definition at line 17 of file ClpConstraintLinear.hpp.

4.32.2 Member Function Documentation

4.32.2.1 virtual int ClpConstraintLinear::gradient (const ClpSimplex * *model*, const double * *solution*, double * *gradient*, double & *functionValue*, double & *offset*, bool *useScaling* = `false`, bool *refresh* = `true`) const [virtual]

Fills gradient.

If Linear then solution may be NULL, also returns true value of function and offset so we can use x not deltaX in constraint If refresh is false then uses last solution Uses model for scaling Returns non-zero if gradient udefined at current solution

Implements ClpConstraint.

4.32.2.2 virtual int ClpConstraintLinear::markNonlinear (char * *which*) const [virtual]

Given a zeroed array sets nonlinear columns to 1.

Returns number of nonlinear columns

Implements ClpConstraint.

4.32.2.3 virtual int ClpConstraintLinear::markNonzero (char * *which*) const [virtual]

Given a zeroed array sets possible nonzero coefficients to 1.

Returns number of nonzeros

Implements ClpConstraint.

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

- ClpConstraintLinear.hpp

4.33 ClpConstraintQuadratic Class Reference

Quadratic Constraint Class.

`#include <ClpConstraintQuadratic.hpp>`

Inheritance diagram for ClpConstraintQuadratic:

Collaboration diagram for ClpConstraintQuadratic:

Public Member Functions

Stuff

- virtual int gradient (const ClpSimplex ∗model, const double ∗solution, double ∗gradient, double &functionValue, double &offset, bool useScaling=false, bool refresh=true) const

  Fills gradient.
- virtual void resize (int newNumberColumns)

  Resize constraint.
- virtual void deleteSome (int numberToDelete, const int ∗which)

  Delete columns in constraint.
- virtual void reallyScale (const double ∗columnScale)

  Scale constraint.
- virtual int markNonlinear (char ∗which) const

  Given a zeroed array sets nonquadratic columns to 1.
- virtual int markNonzero (char ∗which) const

  Given a zeroed array sets possible nonzero coefficients to 1.

Constructors and destructors

- ClpConstraintQuadratic ()

  Default Constructor.
- ClpConstraintQuadratic (int row, int numberQuadraticColumns, int numberColumns, const CoinBigIndex ∗start, const int ∗column, const double ∗element)

  Constructor from quadratic.
- ClpConstraintQuadratic (const ClpConstraintQuadratic &rhs)

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

  Assignment operator.
- virtual ∼ClpConstraintQuadratic ()

  Destructor.
- virtual ClpConstraint ∗ clone () const

  Clone.

Gets and sets

- virtual int numberCoefficients () const

  Number of coefficients.
- int numberColumns () const

  Number of columns in constraint.
- CoinBigIndex ∗ start () const

  Column starts.
- const int ∗ column () const

  Columns.
- const double ∗ coefficient () const

  Coefficients.

Additional Inherited Members

4.33.1 Detailed Description

Quadratic Constraint Class.

Definition at line 17 of file ClpConstraintQuadratic.hpp.

4.33.2 Member Function Documentation

4.33.2.1 virtual int ClpConstraintQuadratic::gradient (const ClpSimplex * model, const double * solution, double * gradient, double & functionValue, double & offset, bool useScaling = false, bool refresh = true) const [virtual]

Fills gradient.

If Quadratic then solution may be NULL, also returns true value of function and offset so we can use x not deltaX in constraint If refresh is false then uses last solution Uses model for scaling Returns non-zero if gradient udefined at current solution

Implements ClpConstraint.

4.33.2.2 virtual int ClpConstraintQuadratic::markNonlinear (char * which) const [virtual]

Given a zeroed array sets nonquadratic columns to 1.

Returns number of nonquadratic columns

Implements ClpConstraint.

4.33.2.3 virtual int ClpConstraintQuadratic::markNonzero (char * which) const [virtual]

Given a zeroed array sets possible nonzero coefficients to 1.

Returns number of nonzeros

Implements ClpConstraint.

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

- ClpConstraintQuadratic.hpp

4.34 ClpDataSave Class Reference

This is a tiny class where data can be saved round calls.

```
#include <ClpModel.hpp>
```

Public Member Functions

Constructors and destructor

- ClpDataSave ()

  Default constructor.
- ClpDataSave (const ClpDataSave &)

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

  Assignment operator. This copies the data.
- ~ClpDataSave ()

  Destructor.

Public Attributes

data - with same names as in other classes

- double **dualBound_**
- double **infeasibilityCost_**
- double **pivotTolerance_**
- double **zeroFactorizationTolerance_**
- double **zeroSimplexTolerance_**
- double **acceptablePivot_**
- double **objectiveScale_**
- int **sparseThreshold_**
- int **perturbation_**
- int **forceFactorization_**
- int **scalingFlag_**
- unsigned int **specialOptions_**

4.34.1 Detailed Description

This is a tiny class where data can be saved round calls.

Definition at line 1269 of file ClpModel.hpp.

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

- ClpModel.hpp

4.35 ClpDisasterHandler Class Reference

Base class for Clp disaster handling.

`#include <ClpEventHandler.hpp>`

Inheritance diagram for ClpDisasterHandler:

Collaboration diagram for ClpDisasterHandler:

Public Member Functions

Virtual methods that the derived classe should provide.

- virtual void intoSimplex ()=0
  Into simplex.
- virtual bool check () const =0
  Checks if disaster.
- virtual void saveInfo ()=0
  saves information for next attempt
- virtual int typeOfDisaster ()
  Type of disaster 0 can fix, 1 abort.

Constructors, destructor

- ClpDisasterHandler (ClpSimplex *model=NULL)
  Default constructor.

- virtual ~ClpDisasterHandler ()
  Destructor.
- **ClpDisasterHandler** (const ClpDisasterHandler &)
- ClpDisasterHandler & **operator=** (const ClpDisasterHandler &)
- virtual ClpDisasterHandler * clone () const =0
  Clone.

Sets/gets

- void setSimplex (ClpSimplex *model)
  set model.
- ClpSimplex * simplex () const
  Get model.

Protected Attributes

Data members

The data members are protected to allow access for derived classes.

- ClpSimplex * model_
  Pointer to simplex.

4.35.1 Detailed Description

Base class for Clp disaster handling.

This is here to allow for disaster handling. By disaster I mean that Clp would otherwise give up

Definition at line 134 of file ClpEventHandler.hpp.

4.35.2 Constructor & Destructor Documentation

4.35.2.1 ClpDisasterHandler::ClpDisasterHandler (ClpSimplex * *model* = `NULL`)

Default constructor.

4.35.3 Member Function Documentation

4.35.3.1 void ClpDisasterHandler::setSimplex (ClpSimplex * *model*)

set model.

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

- ClpEventHandler.hpp

4.36 ClpDualRowDantzig Class Reference

Dual Row Pivot Dantzig Algorithm Class.

`#include <ClpDualRowDantzig.hpp>`

Inheritance diagram for ClpDualRowDantzig:

Collaboration diagram for ClpDualRowDantzig:

Public Member Functions

Algorithmic methods

- virtual int pivotRow ()
  Returns pivot row, -1 if none.
- virtual double updateWeights (**CoinIndexedVector** ∗input, **CoinIndexedVector** ∗spare, **CoinIndexedVector** ∗spare2, **CoinIndexedVector** ∗updatedColumn)
  Updates weights and returns pivot alpha.
- virtual void updatePrimalSolution (**CoinIndexedVector** ∗input, double theta, double &changeInObjective)
  Updates primal solution (and maybe list of candidates) Uses input vector which it deletes Computes change in objective function.

Constructors and destructors

- ClpDualRowDantzig ()
  Default Constructor.
- ClpDualRowDantzig (const ClpDualRowDantzig &)
  Copy constructor.
- ClpDualRowDantzig & operator= (const ClpDualRowDantzig &rhs)
  Assignment operator.
- virtual ∼ClpDualRowDantzig ()
  Destructor.
- virtual ClpDualRowPivot ∗ clone (bool copyData=true) const
  Clone.

Additional Inherited Members

4.36.1 Detailed Description

Dual Row Pivot Dantzig Algorithm Class.

This is simplest choice - choose largest infeasibility

Definition at line 19 of file ClpDualRowDantzig.hpp.

4.36.2 Member Function Documentation

4.36.2.1 virtual double ClpDualRowDantzig::updateWeights (CoinIndexedVector ∗ *input*, CoinIndexedVector ∗ *spare*, CoinIndexedVector ∗ *spare2*, CoinIndexedVector ∗ *updatedColumn*) [virtual]

Updates weights and returns pivot alpha.

Also does FT update

Implements ClpDualRowPivot.

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

- ClpDualRowDantzig.hpp

4.37 ClpDualRowPivot Class Reference

Dual Row Pivot Abstract Base Class.

`#include <ClpDualRowPivot.hpp>`

Inheritance diagram for ClpDualRowPivot:

Collaboration diagram for ClpDualRowPivot:

Public Member Functions

Algorithmic methods

- virtual int pivotRow ()=0
  Returns pivot row, -1 if none.
- virtual double updateWeights (**CoinIndexedVector** ∗input, **CoinIndexedVector** ∗spare, **CoinIndexedVector** ∗spare2, **CoinIndexedVector** ∗updatedColumn)=0
  Updates weights and returns pivot alpha.
- virtual void updatePrimalSolution (**CoinIndexedVector** ∗input, double theta, double &changeInObjective)=0
  Updates primal solution (and maybe list of candidates) Uses input vector which it deletes Computes change in objective function Would be faster if we kept basic regions, but on other hand it means everything is always in sync.
- virtual void saveWeights (ClpSimplex ∗model, int mode)
  Saves any weights round factorization as pivot rows may change Will be empty unless steepest edge (will save model) May also recompute infeasibility stuff 1) before factorization 2) after good factorization (if weights empty may initialize) 3) after something happened but no factorization (e.g.
- virtual void checkAccuracy ()
  checks accuracy and may re-initialize (may be empty)
- virtual void unrollWeights ()
  Gets rid of last update (may be empty)
- virtual void clearArrays ()
  Gets rid of all arrays (may be empty)
- virtual bool looksOptimal () const
  Returns true if would not find any row.
- virtual void maximumPivotsChanged ()
  Called when maximum pivots changes.

Constructors and destructors

- ClpDualRowPivot ()
  Default Constructor.
- ClpDualRowPivot (const ClpDualRowPivot &)
  Copy constructor.
- ClpDualRowPivot & operator= (const ClpDualRowPivot &rhs)
  Assignment operator.
- virtual ∼ClpDualRowPivot ()
  Destructor.
- virtual ClpDualRowPivot ∗ clone (bool copyData=true) const =0
  Clone.

Other

- ClpSimplex ∗ model ()
  Returns model.
- void setModel (ClpSimplex ∗newmodel)
  Sets model (normally to NULL)
- int type ()
  Returns type (above 63 is extra information)

Protected Attributes

Protected member data

- ClpSimplex * model_

  Pointer to model.
- int type_

  Type of row pivot algorithm.

4.37.1 Detailed Description

Dual Row Pivot Abstract Base Class.

Abstract Base Class for describing an interface to an algorithm to choose row pivot in dual simplex algorithm. For some algorithms e.g. Dantzig choice then some functions may be null.

Definition at line 22 of file ClpDualRowPivot.hpp.

4.37.2 Member Function Documentation

4.37.2.1 virtual double ClpDualRowPivot::updateWeights (**CoinIndexedVector** * *input,* **CoinIndexedVector** * *spare,* **CoinIndexedVector** * *spare2,* **CoinIndexedVector** * *updatedColumn*) `[pure virtual]`

Updates weights and returns pivot alpha.

Also does FT update

Implemented in ClpDualRowSteepest, and ClpDualRowDantzig.

4.37.2.2 virtual void ClpDualRowPivot::saveWeights (**ClpSimplex** * *model,* int *mode*) `[virtual]`

Saves any weights round factorization as pivot rows may change Will be empty unless steepest edge (will save model) May also recompute infeasibility stuff 1) before factorization 2) after good factorization (if weights empty may initialize) 3) after something happened but no factorization (e.g.

check for infeasible) 4) as 2 but restore weights from previous snapshot 5) for strong branching - initialize to 1 , infeasibilities 6) scale back 7) for strong branching - initialize full weights , infeasibilities

Reimplemented in ClpDualRowSteepest.

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

- ClpDualRowPivot.hpp

4.38 ClpDualRowSteepest Class Reference

Dual Row Pivot Steepest Edge Algorithm Class.

`#include <ClpDualRowSteepest.hpp>`

Inheritance diagram for ClpDualRowSteepest:

Collaboration diagram for ClpDualRowSteepest:

Public Types

- enum Persistence
  enums for persistence

Public Member Functions

Algorithmic methods

- virtual int pivotRow ()
  Returns pivot row, -1 if none.
- virtual double updateWeights (**CoinIndexedVector** ∗input, **CoinIndexedVector** ∗spare, **CoinIndexedVector** ∗spare2, **CoinIndexedVector** ∗updatedColumn)
  Updates weights and returns pivot alpha.
- virtual void updatePrimalSolution (**CoinIndexedVector** ∗input, double theta, double &changeInObjective)
  Updates primal solution (and maybe list of candidates) Uses input vector which it deletes Computes change in objective function.
- virtual void saveWeights (ClpSimplex ∗model, int mode)
  Saves any weights round factorization as pivot rows may change Save model May also recompute infeasibility stuff 1) before factorization 2) after good factorization (if weights empty may initialize) 3) after something happened but no factorization (e.g.
- void passInSavedWeights (const **CoinIndexedVector** ∗saved)
  Pass in saved weights.
- **CoinIndexedVector** ∗ savedWeights ()
  Get saved weights.
- virtual void unrollWeights ()
  Gets rid of last update.
- virtual void clearArrays ()
  Gets rid of all arrays.
- virtual bool looksOptimal () const
  Returns true if would not find any row.
- virtual void maximumPivotsChanged ()
  Called when maximum pivots changes.

Constructors and destructors

- ClpDualRowSteepest (int mode=3)
  Default Constructor 0 is uninitialized, 1 full, 2 is partial uninitialized, 3 starts as 2 but may switch to 1.
- ClpDualRowSteepest (const ClpDualRowSteepest &)
  Copy constructor.
- ClpDualRowSteepest & operator= (const ClpDualRowSteepest &rhs)
  Assignment operator.
- void fill (const ClpDualRowSteepest &rhs)
  Fill most values.
- virtual ∼ClpDualRowSteepest ()
  Destructor.
- virtual ClpDualRowPivot ∗ clone (bool copyData=true) const
  Clone.

gets and sets

- int mode () const
  Mode.
- void setMode (int mode)
  Set mode.
- void setPersistence (Persistence life)
  Set/ get persistence.
- Persistence **persistence** () const

Additional Inherited Members

4.38.1 Detailed Description

Dual Row Pivot Steepest Edge Algorithm Class.

See Forrest-Goldfarb paper for algorithm

Definition at line 21 of file ClpDualRowSteepest.hpp.

4.38.2 Constructor & Destructor Documentation

4.38.2.1 ClpDualRowSteepest::ClpDualRowSteepest (int *mode* = 3)

Default Constructor 0 is uninitialized, 1 full, 2 is partial uninitialized, 3 starts as 2 but may switch to 1.

By partial is meant that the weights are updated as normal but only part of the infeasible basic variables are scanned. This can be faster on very easy problems.

4.38.3 Member Function Documentation

4.38.3.1 virtual double ClpDualRowSteepest::updateWeights (CoinIndexedVector ∗ *input*, CoinIndexedVector ∗ *spare*, CoinIndexedVector ∗ *spare2*, CoinIndexedVector ∗ *updatedColumn*) [virtual]

Updates weights and returns pivot alpha.

Also does FT update

Implements ClpDualRowPivot.

4.38.3.2 virtual void ClpDualRowSteepest::saveWeights (ClpSimplex ∗ *model*, int *mode*) [virtual]

Saves any weights round factorization as pivot rows may change Save model May also recompute infeasibility stuff 1) before factorization 2) after good factorization (if weights empty may initialize) 3) after something happened but no factorization (e.g.

check for infeasible) 4) as 2 but restore weights from previous snapshot 5) for strong branching - initialize (uninitialized) , infeasibilities

Reimplemented from ClpDualRowPivot.

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

- ClpDualRowSteepest.hpp

4.39 ClpDummyMatrix Class Reference

This implements a dummy matrix as derived from ClpMatrixBase.

`#include <ClpDummyMatrix.hpp>`

Inheritance diagram for ClpDummyMatrix:

Collaboration diagram for ClpDummyMatrix:

Public Member Functions

Useful methods

- virtual **CoinPackedMatrix** ∗ getPackedMatrix () const

  Return a complete **CoinPackedMatrix**.
- virtual bool isColOrdered () const

  Whether the packed matrix is column major ordered or not.
- virtual CoinBigIndex getNumElements () const

  Number of entries in the packed matrix.
- virtual int getNumCols () const

  Number of columns.
- virtual int getNumRows () const

  Number of rows.
- virtual const double ∗ getElements () const

  A vector containing the elements in the packed matrix.
- virtual const int ∗ getIndices () const

  A vector containing the minor indices of the elements in the packed matrix.
- virtual const CoinBigIndex ∗ **getVectorStarts** () const
- virtual const int ∗ getVectorLengths () const

  The lengths of the major-dimension vectors.
- virtual void deleteCols (const int numDel, const int ∗indDel)

  Delete the columns whose indices are listed in `indDel`.
- virtual void deleteRows (const int numDel, const int ∗indDel)

  Delete the rows whose indices are listed in `indDel`.
- virtual ClpMatrixBase ∗ reverseOrderedCopy () const

  Returns a new matrix in reverse order without gaps.
- virtual CoinBigIndex countBasis (const int ∗whichColumn, int &numberColumnBasic)

  Returns number of elements in column part of basis.
- virtual void fillBasis (ClpSimplex ∗model, const int ∗whichColumn, int &numberColumnBasic, int ∗row, int ∗start, int ∗rowCount, int ∗columnCount, CoinFactorizationDouble ∗element)

  Fills in column part of basis.
- virtual void unpack (const ClpSimplex ∗model, **CoinIndexedVector** ∗rowArray, int column) const

  Unpacks a column into an CoinIndexedvector.
- virtual void unpackPacked (ClpSimplex ∗model, **CoinIndexedVector** ∗rowArray, int column) const

  Unpacks a column into an CoinIndexedvector in packed foramt Note that model is NOT const.
- virtual void add (const ClpSimplex ∗model, **CoinIndexedVector** ∗rowArray, int column, double multiplier) const

  Adds multiple of a column into an CoinIndexedvector You can use quickAdd to add to vector.
- virtual void add (const ClpSimplex ∗model, double ∗array, int column, double multiplier) const

  Adds multiple of a column into an array.
- virtual void releasePackedMatrix () const

  Allow any parts of a created CoinMatrix to be deleted Allow any parts of a created **CoinPackedMatrix** *to be deleted.*

Matrix times vector methods

- virtual void times (double scalar, const double ∗x, double ∗y) const

  Return `y + A * scalar *x` *in* `y`.
- virtual void times (double scalar, const double ∗x, double ∗y, const double ∗rowScale, const double ∗column←Scale) const

  And for scaling.
- virtual void transposeTimes (double scalar, const double ∗x, double ∗y) const

  Return `y + x * scalar * A` *in* `y`.

- virtual void transposeTimes (double scalar, const double ∗x, double ∗y, const double ∗rowScale, const double ∗columnScale) const

    And for scaling.
- virtual void transposeTimes (const ClpSimplex ∗model, double scalar, const **CoinIndexedVector** ∗x, **Coin**↩ **IndexedVector** ∗y, **CoinIndexedVector** ∗z) const

    Return `x * scalar * A + y` *in* `z`*.*
- virtual void subsetTransposeTimes (const ClpSimplex ∗model, const **CoinIndexedVector** ∗x, const **Coin**↩ **IndexedVector** ∗y, **CoinIndexedVector** ∗z) const

    Return `x *A` *in* `z` *but just for indices in y.*

Constructors, destructor

- ClpDummyMatrix ()

    Default constructor.
- ClpDummyMatrix (int numberColumns, int numberRows, int numberElements)

    Constructor with data.
- virtual ∼ClpDummyMatrix ()

    Destructor.

Copy method

- ClpDummyMatrix (const ClpDummyMatrix &)

    The copy constructor.
- ClpDummyMatrix (const **CoinPackedMatrix** &)

    The copy constructor from an CoinDummyMatrix.
- ClpDummyMatrix & **operator=** (const ClpDummyMatrix &)
- virtual ClpMatrixBase ∗ clone () const

    Clone.

Protected Attributes

Data members

The data members are protected to allow access for derived classes.

- int numberRows_

    Number of rows.
- int numberColumns_

    Number of columns.
- int numberElements_

    Number of elements.

Additional Inherited Members

4.39.1 Detailed Description

This implements a dummy matrix as derived from ClpMatrixBase.

This is so you can do ClpPdco but may come in useful elsewhere. It just has dimensions but no data

Definition at line 20 of file ClpDummyMatrix.hpp.

4.39.2 Constructor & Destructor Documentation

4.39.2.1 ClpDummyMatrix::ClpDummyMatrix ( )

Default constructor.

4.39.2.2 ClpDummyMatrix::ClpDummyMatrix (const ClpDummyMatrix &)

The copy constructor.

4.39.2.3 ClpDummyMatrix::ClpDummyMatrix (const CoinPackedMatrix &)

The copy constructor from an CoinDummyMatrix.

4.39.3 Member Function Documentation

4.39.3.1 virtual bool ClpDummyMatrix::isColOrdered ( ) const [inline], [virtual]

Whether the packed matrix is column major ordered or not.

Implements ClpMatrixBase.

Definition at line 28 of file ClpDummyMatrix.hpp.

4.39.3.2 virtual CoinBigIndex ClpDummyMatrix::getNumElements ( ) const [inline], [virtual]

Number of entries in the packed matrix.

Implements ClpMatrixBase.

Definition at line 32 of file ClpDummyMatrix.hpp.

4.39.3.3 virtual int ClpDummyMatrix::getNumCols ( ) const [inline], [virtual]

Number of columns.

Implements ClpMatrixBase.

Definition at line 36 of file ClpDummyMatrix.hpp.

4.39.3.4 virtual int ClpDummyMatrix::getNumRows ( ) const [inline], [virtual]

Number of rows.

Implements ClpMatrixBase.

Definition at line 40 of file ClpDummyMatrix.hpp.

4.39.3.5 virtual const double* ClpDummyMatrix::getElements ( ) const [virtual]

A vector containing the elements in the packed matrix.

Note that there might be gaps in this list, entries that do not belong to any major-dimension vector. To get the actual elements one should look at this vector together with vectorStarts and vectorLengths.

Implements ClpMatrixBase.

4.39.3.6 virtual const int* ClpDummyMatrix::getIndices ( ) const [virtual]

A vector containing the minor indices of the elements in the packed matrix.

Note that there might be gaps in this list, entries that do not belong to any major-dimension vector. To get the actual elements one should look at this vector together with vectorStarts and vectorLengths.

Implements ClpMatrixBase.

4.39.3.7 virtual const int* ClpDummyMatrix::getVectorLengths ( ) const [virtual]

The lengths of the major-dimension vectors.

Implements ClpMatrixBase.

4.39.3.8 virtual void ClpDummyMatrix::deleteCols (const int *numDel*, const int * *indDel*) [virtual]

Delete the columns whose indices are listed in `indDel`.

Implements ClpMatrixBase.

4.39.3.9 virtual void ClpDummyMatrix::deleteRows (const int *numDel*, const int * *indDel*) [virtual]

Delete the rows whose indices are listed in `indDel`.

Implements ClpMatrixBase.

4.39.3.10 virtual void ClpDummyMatrix::unpackPacked (ClpSimplex * *model*, CoinIndexedVector * *rowArray*, int *column*) const [virtual]

Unpacks a column into an CoinIndexedvector in packed foramt Note that model is NOT const.

Bounds and objective could be modified if doing column generation (just for this variable)

Implements ClpMatrixBase.

4.39.3.11 virtual void ClpDummyMatrix::times (double *scalar*, const double * *x*, double * *y*) const [virtual]

Return `y + A * scalar *x` in `y`.

Precondition

> `x` must be of size `numColumns()`
> `y` must be of size `numRows()`

4.39.3.12 virtual void ClpDummyMatrix::transposeTimes (double *scalar*, const double * *x*, double * *y*) const [virtual]

Return `y + x * scalar * A` in `y`.

Precondition

x must be of size numRows()
y must be of size numColumns()

4.39.3.13 virtual void ClpDummyMatrix::transposeTimes (const ClpSimplex ∗ *model*, double *scalar*, const CoinIndexedVector ∗ *x*, CoinIndexedVector ∗ *y*, CoinIndexedVector ∗ *z*) const [virtual]

Return `x * scalar * A + y` in `z`.

Can use y as temporary array (will be empty at end) Note - If x packed mode - then z packed mode

Implements ClpMatrixBase.

4.39.3.14 virtual void ClpDummyMatrix::subsetTransposeTimes (const ClpSimplex ∗ *model*, const CoinIndexedVector ∗ *x*, const CoinIndexedVector ∗ *y*, CoinIndexedVector ∗ *z*) const [virtual]

Return `x *A` in `z` but just for indices in y.

Note - If x packed mode - then z packed mode Squashes small elements and knows about ClpSimplex

Implements ClpMatrixBase.

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

- ClpDummyMatrix.hpp

4.40 ClpDynamicExampleMatrix Class Reference

This implements a dynamic matrix when we have a limit on the number of "interesting rows".

`#include <ClpDynamicExampleMatrix.hpp>`

Inheritance diagram for ClpDynamicExampleMatrix:

Collaboration diagram for ClpDynamicExampleMatrix:

Public Member Functions

Main functions provided

- virtual void partialPricing (ClpSimplex ∗model, double start, double end, int &bestSequence, int &number←Wanted)
  Partial pricing.
- virtual void createVariable (ClpSimplex ∗model, int &bestSequence)
  Creates a variable.
- virtual void packDown (const int ∗in, int numberToPack)
  If addColumn forces compression then this allows descendant to know what to do.

Constructors, destructor

- ClpDynamicExampleMatrix ()
  Default constructor.

- ClpDynamicExampleMatrix (ClpSimplex *model, int numberSets, int numberColumns, const int *starts, const double *lower, const double *upper, const int *startColumn, const int *row, const double *element, const double *cost, const double *columnLower=NULL, const double *columnUpper=NULL, const unsigned char *status=NULL, const unsigned char *dynamicStatus=NULL, int numberIds=0, const int *ids=NULL)

  This is the real constructor.
- virtual ~ClpDynamicExampleMatrix ()

  Destructor.

Copy method

- ClpDynamicExampleMatrix (const ClpDynamicExampleMatrix &)

  The copy constructor.
- ClpDynamicExampleMatrix & **operator=** (const ClpDynamicExampleMatrix &)
- virtual ClpMatrixBase * clone () const

  Clone.

gets and sets

- CoinBigIndex * startColumnGen () const

  Starts of each column.
- int * rowGen () const

  rows
- double * elementGen () const

  elements
- double * costGen () const

  costs
- int * fullStartGen () const

  full starts
- int * idGen () const

  ids in next level matrix
- double * columnLowerGen () const

  Optional lower bounds on columns.
- double * columnUpperGen () const

  Optional upper bounds on columns.
- int numberColumns () const

  size
- void **setDynamicStatusGen** (int sequence, DynamicStatus status)
- DynamicStatus **getDynamicStatusGen** (int sequence) const
- bool flaggedGen (int i) const

  Whether flagged.
- void **setFlaggedGen** (int i)
- void **unsetFlagged** (int i)

Protected Attributes

Data members

The data members are protected to allow access for derived classes.

- int numberColumns_

  size
- CoinBigIndex * startColumnGen_

  Starts of each column.

- int ∗ rowGen_

  rows
- double ∗ elementGen_

  elements
- double ∗ costGen_

  costs
- int ∗ fullStartGen_

  start of each set
- unsigned char ∗ dynamicStatusGen_

  for status and which bound
- int ∗ idGen_

  identifier for each variable up one level (startColumn_, etc).
- double ∗ columnLowerGen_

  Optional lower bounds on columns.
- double ∗ columnUpperGen_

  Optional upper bounds on columns.

Additional Inherited Members

4.40.1 Detailed Description

This implements a dynamic matrix when we have a limit on the number of "interesting rows".

This version inherits from ClpDynamicMatrix and knows that the real matrix is gub. This acts just like ClpDynamicMatrix but generates columns. This "generates" columns by choosing from stored set. It is maent as a starting point as to how you could use shortest path to generate columns.

So it has its own copy of all data needed. It populates ClpDynamicWatrix with enough to allow for gub keys and active variables. In turn ClpDynamicMatrix populates a **CoinPackedMatrix** with active columns and rows.

As there is one copy here and one in ClpDynamicmatrix these names end in Gen_

It is obviously more efficient to just use ClpDynamicMatrix but the ideas is to show how much code a user would have to write.

This does not work very well with bounds

Definition at line 33 of file ClpDynamicExampleMatrix.hpp.

4.40.2 Constructor & Destructor Documentation

4.40.2.1 ClpDynamicExampleMatrix::ClpDynamicExampleMatrix ( )

Default constructor.

4.40.2.2 ClpDynamicExampleMatrix::ClpDynamicExampleMatrix (ClpSimplex ∗ *model*, int *numberSets*, int *numberColumns*, const int ∗ *starts*, const double ∗ *lower*, const double ∗ *upper*, const int ∗ *startColumn*, const int ∗ *row*, const double ∗ *element*, const double ∗ *cost*, const double ∗ *columnLower* = NULL, const double ∗ *columnUpper* = NULL, const unsigned char ∗ *status* = NULL, const unsigned char ∗ *dynamicStatus* = NULL, int *numberIds* = 0, const int ∗ *ids* = NULL)

This is the real constructor.

It assumes factorization frequency will not be changed. This resizes model !!!! The contents of original matrix in model will be taken over and original matrix will be sanitized so can be deleted (to avoid a very small memory leak)

4.40.2.3 ClpDynamicExampleMatrix::ClpDynamicExampleMatrix (const ClpDynamicExampleMatrix &)

The copy constructor.

4.40.3 Member Function Documentation

4.40.3.1 virtual void ClpDynamicExampleMatrix::createVariable (ClpSimplex * *model,* int & *bestSequence*) [virtual]

Creates a variable.

This is called after partial pricing and will modify matrix. Will update bestSequence.

Reimplemented from ClpDynamicMatrix.

4.40.3.2 virtual void ClpDynamicExampleMatrix::packDown (const int * *in,* int *numberToPack*) [virtual]

If addColumn forces compression then this allows descendant to know what to do.

If >= then entry stayed in, if -1 then entry went out to lower bound.of zero. Entries at upper bound (really nonzero) never go out (at present).

Reimplemented from ClpDynamicMatrix.

4.40.4 Member Data Documentation

4.40.4.1 int* ClpDynamicExampleMatrix::idGen_ [protected]

identifier for each variable up one level (startColumn_, etc).

This is of length maximumGubColumns_. For this version it is just sequence number at this level

Definition at line 178 of file ClpDynamicExampleMatrix.hpp.

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

- ClpDynamicExampleMatrix.hpp

4.41 ClpDynamicMatrix Class Reference

This implements a dynamic matrix when we have a limit on the number of "interesting rows".

`#include <ClpDynamicMatrix.hpp>`

Inheritance diagram for ClpDynamicMatrix:

Collaboration diagram for ClpDynamicMatrix:

Public Types

- enum DynamicStatus

    enums for status of various sorts

Public Member Functions

Main functions provided

- virtual void partialPricing (ClpSimplex *model, double start, double end, int &bestSequence, int &number←Wanted)

  Partial pricing.
- virtual int updatePivot (ClpSimplex *model, double oldInValue, double oldOutValue)

  update information for a pivot (and effective rhs)
- virtual double * rhsOffset (ClpSimplex *model, bool forceRefresh=false, bool check=false)

  Returns effective RHS offset if it is being used.
- virtual void times (double scalar, const double *x, double *y) const

  Return $y + A * scalar *x$ *in* y.
- void modifyOffset (int sequence, double amount)

  Modifies rhs offset.
- double keyValue (int iSet) const

  Gets key value when none in small.
- virtual void dualExpanded (ClpSimplex *model, **CoinIndexedVector** *array, double *other, int mode)

  mode=0 - Set up before "updateTranspose" and "transposeTimes" for duals using extended updates array (and may use other if dual values pass) mode=1 - Update dual solution after "transposeTimes" using extended rows.
- virtual int generalExpanded (ClpSimplex *model, int mode, int &number)

  mode=0 - Create list of non-key basics in pivotVariable_ using number as numberBasic in and out mode=1 - Set all key variables as basic mode=2 - return number extra rows needed, number gives maximum number basic mode=3 - before replaceColumn mode=4 - return 1 if can do primal, 2 if dual, 3 if both mode=5 - save any status stuff (when in good state) mode=6 - restore status stuff mode=7 - flag given variable (normally sequenceIn) mode=8 - unflag all variables mode=9 - synchronize costs mode=10 - return 1 if there may be changing bounds on variable (column generation) mode=11 - make sure set is clean (used when a variable rejected - but not flagged) mode=12 - after factorize but before permute stuff mode=13 - at end of simplex to delete stuff
- virtual int refresh (ClpSimplex *model)

  Purely for column generation and similar ideas.
- virtual void createVariable (ClpSimplex *model, int &bestSequence)

  Creates a variable.
- virtual double reducedCost (ClpSimplex *model, int sequence) const

  Returns reduced cost of a variable.
- void gubCrash ()

  Does gub crash.
- void writeMps (const char *name)

  Writes out model (without names)
- void initialProblem ()

  Populates initial matrix from dynamic status.
- int addColumn (int numberEntries, const int *row, const double *element, double cost, double lower, double upper, int iSet, DynamicStatus status)

  Adds in a column to gub structure (called from descendant) and returns sequence.
- virtual void packDown (const int *, int)

  If addColumn forces compression then this allows descendant to know what to do.
- double columnLower (int sequence) const

  Gets lower bound (to simplify coding)
- double columnUpper (int sequence) const

  Gets upper bound (to simplify coding)

Constructors, destructor

- ClpDynamicMatrix ()

  Default constructor.

- ClpDynamicMatrix (ClpSimplex ∗model, int numberSets, int numberColumns, const int ∗starts, const double ∗lower, const double ∗upper, const CoinBigIndex ∗startColumn, const int ∗row, const double ∗element, const double ∗cost, const double ∗columnLower=NULL, const double ∗columnUpper=NULL, const unsigned char ∗status=NULL, const unsigned char ∗dynamicStatus=NULL)

  This is the real constructor.
- virtual ∼ClpDynamicMatrix ()

  Destructor.

Copy method

- ClpDynamicMatrix (const ClpDynamicMatrix &)

  The copy constructor.
- ClpDynamicMatrix (const **CoinPackedMatrix** &)

  The copy constructor from an **CoinPackedMatrix**.
- ClpDynamicMatrix & **operator=** (const ClpDynamicMatrix &)
- virtual ClpMatrixBase ∗ clone () const

  Clone.

gets and sets

- ClpSimplex::Status getStatus (int sequence) const

  Status of row slacks.
- void **setStatus** (int sequence, ClpSimplex::Status status)
- bool flaggedSlack (int i) const

  Whether flagged slack.
- void **setFlaggedSlack** (int i)
- void **unsetFlaggedSlack** (int i)
- int numberSets () const

  Number of sets (dynamic rows)
- int numberGubEntries () const

  Number of possible gub variables.
- int ∗ startSets () const

  Sets.
- bool flagged (int i) const

  Whether flagged.
- void **setFlagged** (int i)
- void **unsetFlagged** (int i)
- void **setDynamicStatus** (int sequence, DynamicStatus status)
- DynamicStatus **getDynamicStatus** (int sequence) const
- double objectiveOffset () const

  Saved value of objective offset.
- CoinBigIndex ∗ startColumn () const

  Starts of each column.
- int ∗ row () const

  rows
- double ∗ element () const

  elements
- double ∗ cost () const

  costs
- int ∗ id () const

  ids of active columns (just index here)
- double ∗ columnLower () const

  Optional lower bounds on columns.

- double ∗ columnUpper () const

  Optional upper bounds on columns.
- double ∗ lowerSet () const

  Lower bounds on sets.
- double ∗ upperSet () const

  Upper bounds on sets.
- int numberGubColumns () const

  size
- int firstAvailable () const

  first free
- int firstDynamic () const

  first dynamic
- int lastDynamic () const

  number of columns in dynamic model
- int numberStaticRows () const

  number of rows in original model
- int numberElements () const

  size of working matrix (max)
- int ∗ **keyVariable** () const
- void switchOffCheck ()

  Switches off dj checking each factorization (for BIG models)
- unsigned char ∗ gubRowStatus () const

  Status region for gub slacks.
- unsigned char ∗ dynamicStatus () const

  Status region for gub variables.
- int whichSet (int sequence) const

  Returns which set a variable is in.

Protected Attributes

Data members

The data members are protected to allow access for derived classes.

- double sumDualInfeasibilities_

  Sum of dual infeasibilities.
- double sumPrimalInfeasibilities_

  Sum of primal infeasibilities.
- double sumOfRelaxedDualInfeasibilities_

  Sum of Dual infeasibilities using tolerance based on error in duals.
- double sumOfRelaxedPrimalInfeasibilities_

  Sum of Primal infeasibilities using tolerance based on error in primals.
- double savedBestGubDual_

  Saved best dual on gub row in pricing.
- int savedBestSet_

  Saved best set in pricing.
- int ∗ backToPivotRow_

  Backward pointer to pivot row !!!
- int ∗ keyVariable_

  Key variable of set (only accurate if none in small problem)
- int ∗ toIndex_

  Backward pointer to extra row.

- int ∗ **fromIndex_**
- int numberSets_

  Number of sets (dynamic rows)
- int numberActiveSets_

  Number of active sets.
- double objectiveOffset_

  Saved value of objective offset.
- double ∗ lowerSet_

  Lower bounds on sets.
- double ∗ upperSet_

  Upper bounds on sets.
- unsigned char ∗ status_

  Status of slack on set.
- ClpSimplex ∗ model_

  Pointer back to model.
- int firstAvailable_

  first free
- int firstAvailableBefore_

  first free when iteration started
- int firstDynamic_

  first dynamic
- int lastDynamic_

  number of columns in dynamic model
- int numberStaticRows_

  number of rows in original model
- int numberElements_

  size of working matrix (max)
- int numberDualInfeasibilities_

  Number of dual infeasibilities.
- int numberPrimalInfeasibilities_

  Number of primal infeasibilities.
- int noCheck_

  If pricing will declare victory (i.e.
- double infeasibilityWeight_

  Infeasibility weight when last full pass done.
- int numberGubColumns_

  size
- int maximumGubColumns_

  current maximum number of columns (then compress)
- int maximumElements_

  current maximum number of elemnts (then compress)
- int ∗ startSet_

  Start of each set.
- int ∗ next_

  next in chain
- CoinBigIndex ∗ startColumn_

  Starts of each column.
- int ∗ row_

  rows
- double ∗ element_

  elements
- double ∗ cost_

costs
- int ∗ id_

  ids of active columns (just index here)
- unsigned char ∗ dynamicStatus_

  for status and which bound
- double ∗ columnLower_

  Optional lower bounds on columns.
- double ∗ columnUpper_

  Optional upper bounds on columns.

Additional Inherited Members

4.41.1 Detailed Description

This implements a dynamic matrix when we have a limit on the number of "interesting rows".

This version inherits from ClpPackedMatrix and knows that the real matrix is gub. A later version could use shortest path to generate columns.

Definition at line 20 of file ClpDynamicMatrix.hpp.

4.41.2 Constructor & Destructor Documentation

4.41.2.1 ClpDynamicMatrix::ClpDynamicMatrix ( )

Default constructor.

4.41.2.2 ClpDynamicMatrix::ClpDynamicMatrix (ClpSimplex ∗ *model*, int *numberSets*, int *numberColumns*, const int ∗ *starts*, const double ∗ *lower*, const double ∗ *upper*, const CoinBigIndex ∗ *startColumn*, const int ∗ *row*, const double ∗ *element*, const double ∗ *cost*, const double ∗ *columnLower* = NULL, const double ∗ *columnUpper* = NULL, const unsigned char ∗ *status* = NULL, const unsigned char ∗ *dynamicStatus* = NULL)

This is the real constructor.

It assumes factorization frequency will not be changed. This resizes model !!!! The contents of original matrix in model will be taken over and original matrix will be sanitized so can be deleted (to avoid a very small memory leak)

4.41.2.3 ClpDynamicMatrix::ClpDynamicMatrix (const ClpDynamicMatrix &)

The copy constructor.

4.41.2.4 ClpDynamicMatrix::ClpDynamicMatrix (const CoinPackedMatrix &)

The copy constructor from an **CoinPackedMatrix**.

4.41.3 Member Function Documentation

4.41.3.1 virtual double∗ ClpDynamicMatrix::rhsOffset (ClpSimplex ∗ *model*, bool *forceRefresh* = false, bool *check* = false) [virtual]

Returns effective RHS offset if it is being used.

This is used for long problems or big dynamic or anywhere where going through full columns is expensive. This may re-compute

Reimplemented from ClpMatrixBase.

4.41.3.2 virtual void ClpDynamicMatrix::times (double *scalar*, const double ∗ *x*, double ∗ *y*) const [virtual]

Return `y + A * scalar *x` in `y`.

Precondition

`x` must be of size `numColumns()`
`y` must be of size `numRows()`

Reimplemented from ClpPackedMatrix.

4.41.3.3 virtual void ClpDynamicMatrix::dualExpanded (ClpSimplex ∗ *model*, CoinIndexedVector ∗ *array*, double ∗ *other*, int *mode*) [virtual]

mode=0 - Set up before "updateTranspose" and "transposeTimes" for duals using extended updates array (and may use other if dual values pass) mode=1 - Update dual solution after "transposeTimes" using extended rows.

mode=2 - Compute all djs and compute key dual infeasibilities mode=3 - Report on key dual infeasibilities mode=4 - Modify before updateTranspose in partial pricing

Reimplemented from ClpMatrixBase.

4.41.3.4 virtual int ClpDynamicMatrix::refresh (ClpSimplex ∗ *model*) [virtual]

Purely for column generation and similar ideas.

Allows matrix and any bounds or costs to be updated (sensibly). Returns non-zero if any changes.

Reimplemented from ClpPackedMatrix.

4.41.3.5 virtual void ClpDynamicMatrix::createVariable (ClpSimplex ∗ *model*, int & *bestSequence*) [virtual]

Creates a variable.

This is called after partial pricing and will modify matrix. Will update bestSequence.

Reimplemented from ClpMatrixBase.

Reimplemented in ClpDynamicExampleMatrix.

4.41.3.6 virtual void ClpDynamicMatrix::packDown (const int ∗ , int) [inline], [virtual]

If addColumn forces compression then this allows descendant to know what to do.

If >=0 then entry stayed in, if -1 then entry went out to lower bound.of zero. Entries at upper bound (really nonzero) never go out (at present).

Reimplemented in ClpDynamicExampleMatrix.

Definition at line 109 of file ClpDynamicMatrix.hpp.

4.41.4 Member Data Documentation

4.41.4.1 int ClpDynamicMatrix::noCheck_ [protected]

If pricing will declare victory (i.e.

no check every factorization). -1 - always check 0 - don't check 1 - in don't check mode but looks optimal

Definition at line 349 of file ClpDynamicMatrix.hpp.

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

- ClpDynamicMatrix.hpp

4.42 ClpEventHandler Class Reference

Base class for Clp event handling.

`#include <ClpEventHandler.hpp>`

Inheritance diagram for ClpEventHandler:

Collaboration diagram for ClpEventHandler:

Public Types

- enum Event

    enums for what sort of event.

Public Member Functions

Virtual method that the derived classes should provide.

The base class instance does nothing and as event() is only useful method it would not be very useful NOT providing one!

- virtual int event (Event whichEvent)

    This can do whatever it likes.
- virtual int eventWithInfo (Event whichEvent, void ∗info)

    This can do whatever it likes.

Constructors, destructor

- ClpEventHandler (ClpSimplex ∗model=NULL)

    Default constructor.
- virtual ∼ClpEventHandler ()

    Destructor.
- **ClpEventHandler** (const ClpEventHandler &)
- ClpEventHandler & **operator=** (const ClpEventHandler &)
- virtual ClpEventHandler ∗ clone () const

    Clone.

Sets/gets

- void setSimplex (ClpSimplex ∗model)

    set model.
- ClpSimplex ∗ simplex () const

    Get model.

Protected Attributes

Data members

The data members are protected to allow access for derived classes.

- ClpSimplex * model_

  Pointer to simplex.

4.42.1 Detailed Description

Base class for Clp event handling.

This is just here to allow for event handling. By event I mean a Clp event e.g. end of values pass.

One use would be to let a user handle a system event e.g. Control-C. This could be done by deriving a class MyEventHandler which knows about such events. If one occurs MyEventHandler::event() could clear event status and return 3 (stopped).

Clp would then return to user code.

As it is called every iteration this should be fine grained enough.

User can derive and construct from CbcModel - not pretty

Definition at line 27 of file ClpEventHandler.hpp.

4.42.2 Member Enumeration Documentation

4.42.2.1 enum ClpEventHandler::Event

enums for what sort of event.

These will also be returned in ClpModel::secondaryStatus() as int

Definition at line 34 of file ClpEventHandler.hpp.

4.42.3 Constructor & Destructor Documentation

4.42.3.1 ClpEventHandler::ClpEventHandler (ClpSimplex * *model* = NULL)

Default constructor.

4.42.4 Member Function Documentation

4.42.4.1 virtual int ClpEventHandler::event (Event *whichEvent*) [virtual]

This can do whatever it likes.

If return code -1 then carries on if 0 sets ClpModel::status() to 5 (stopped by event) and will return to user. At present if <-1 carries on and if >0 acts as if 0 - this may change. For ClpSolve 2 -> too big return status of -2 and -> too small 3

Reimplemented in MyEventHandler.

4.42.4.2 virtual int ClpEventHandler::eventWithInfo (**Event** *whichEvent*, void * *info*) [virtual]

This can do whatever it likes.

Return code -1 means no action. This passes in something

4.42.4.3 void ClpEventHandler::setSimplex (**ClpSimplex** * *model*)

set model.

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

- ClpEventHandler.hpp

4.43 ClpFactorization Class Reference

This just implements **CoinFactorization** when an ClpMatrixBase object is passed.

`#include <ClpFactorization.hpp>`

Public Member Functions

factorization

- int factorize (ClpSimplex *model, int solveType, bool valuesPass)
  When part of LP - given by basic variables.

Constructors, destructor

- ClpFactorization ()
  Default constructor.
- ∼ClpFactorization ()
  Destructor.

Copy method

- ClpFactorization (const **CoinFactorization** &)
  The copy constructor from an **CoinFactorization**.
- ClpFactorization (const ClpFactorization &, int denseIfSmaller=0)
  The copy constructor.
- ClpFactorization (const **CoinOtherFactorization** &)
  The copy constructor from an **CoinOtherFactorization**.
- ClpFactorization & **operator=** (const ClpFactorization &)

rank one updates which do exist

- int replaceColumn (const ClpSimplex *model, **CoinIndexedVector** *regionSparse, **CoinIndexedVector** *tableauColumn, int pivotRow, double pivotCheck, bool checkBeforeModifying=false, double acceptable←Pivot=1.0e-8)
  Replaces one Column to basis, returns 0=OK, 1=Probably OK, 2=singular, 3=no room If checkBeforeModifying is true will do all accuracy checks before modifying factorization.

various uses of factorization (return code number elements)

which user may want to know about

- int updateColumnFT (**CoinIndexedVector** ∗regionSparse, **CoinIndexedVector** ∗regionSparse2)

  Updates one column (FTRAN) from region2 Tries to do FT update number returned is negative if no room region1 starts as zero and is zero at end.
- int updateColumn (**CoinIndexedVector** ∗regionSparse, **CoinIndexedVector** ∗regionSparse2, bool no←Permute=false) const

  Updates one column (FTRAN) from region2 region1 starts as zero and is zero at end.
- int updateTwoColumnsFT (**CoinIndexedVector** ∗regionSparse1, **CoinIndexedVector** ∗regionSparse2, **CoinIndexedVector** ∗regionSparse3, bool noPermuteRegion3=false)

  Updates one column (FTRAN) from region2 Tries to do FT update number returned is negative if no room.
- int updateColumnForDebug (**CoinIndexedVector** ∗regionSparse, **CoinIndexedVector** ∗regionSparse2, bool noPermute=false) const

  For debug (no statistics update)
- int updateColumnTranspose (**CoinIndexedVector** ∗regionSparse, **CoinIndexedVector** ∗regionSparse2) const

  Updates one column (BTRAN) from region2 region1 starts as zero and is zero at end.

Lifted from CoinFactorization

- int numberElements () const

  Total number of elements in factorization.
- int ∗ permute () const

  Returns address of permute region.
- int ∗ pivotColumn () const

  Returns address of pivotColumn region (also used for permuting)
- int maximumPivots () const

  Maximum number of pivots between factorizations.
- void maximumPivots (int value)

  Set maximum number of pivots between factorizations.
- int pivots () const

  Returns number of pivots since factorization.
- double areaFactor () const

  Whether larger areas needed.
- void areaFactor (double value)

  Set whether larger areas needed.
- double zeroTolerance () const

  Zero tolerance.
- void zeroTolerance (double value)

  Set zero tolerance.
- void saferTolerances (double zeroTolerance, double pivotTolerance)

  Set tolerances to safer of existing and given.
- int sparseThreshold () const

  get sparse threshold
- void sparseThreshold (int value)

  Set sparse threshold.
- int status () const

  Returns status.
- void setStatus (int value)

  Sets status.
- int numberDense () const

Returns number of dense rows.
- CoinBigIndex numberElementsU () const

  Returns number in U area.
- CoinBigIndex numberElementsL () const

  Returns number in L area.
- CoinBigIndex numberElementsR () const

  Returns number in R area.
- bool **timeToRefactorize** () const
- int messageLevel () const

  Level of detail of messages.
- void messageLevel (int value)

  Set level of detail of messages.
- void clearArrays ()

  Get rid of all memory.
- int numberRows () const

  Number of Rows after factorization.
- int denseThreshold () const

  Gets dense threshold.
- void setDenseThreshold (int value)

  Sets dense threshold.
- double pivotTolerance () const

  Pivot tolerance.
- void pivotTolerance (double value)

  Set pivot tolerance.
- void relaxAccuracyCheck (double value)

  Allows change of pivot accuracy check 1.0 == none >1.0 relaxed.
- int persistenceFlag () const

  Array persistence flag If 0 then as now (delete/new) 1 then only do arrays if bigger needed 2 as 1 but give a bit extra if bigger needed.
- void **setPersistenceFlag** (int value)
- void almostDestructor ()

  Delete all stuff (leaves as after **CoinFactorization()***)*
- double adjustedAreaFactor () const

  Returns areaFactor but adjusted for dense.
- void **setBiasLU** (int value)
- void setForrestTomlin (bool value)

  true if Forrest Tomlin update, false if PFI
- void setDefaultValues ()

  Sets default values.
- void forceOtherFactorization (int which)

  If nonzero force use of 1,dense 2,small 3,osl.
- int goOslThreshold () const

  Get switch to osl if number rows <= this.
- void setGoOslThreshold (int value)

  Set switch to osl if number rows <= this.
- int goDenseThreshold () const

  Get switch to dense if number rows <= this.
- void setGoDenseThreshold (int value)

  Set switch to dense if number rows <= this.
- int goSmallThreshold () const

  Get switch to small if number rows <= this.
- void setGoSmallThreshold (int value)

Set switch to small if number rows <= this.

- void goDenseOrSmall (int numberRows)

  Go over to dense or small code if small enough.
- void setFactorization (ClpFactorization &factorization)

  Sets factorization.
- int isDenseOrSmall () const

  Return 1 if dense code.

other stuff

- void goSparse ()

  makes a row copy of L for speed and to allow very sparse problems
- void cleanUp ()

  Cleans up i.e. gets rid of network basis.
- bool needToReorder () const

  Says whether to redo pivot order.
- bool networkBasis () const

  Says if a network basis.
- void getWeights (int *weights) const

  Fills weighted row list.

4.43.1 Detailed Description

This just implements **CoinFactorization** when an ClpMatrixBase object is passed.

If a network then has a dummy **CoinFactorization** and a genuine ClpNetworkBasis object

Definition at line 35 of file ClpFactorization.hpp.

4.43.2 Constructor & Destructor Documentation

4.43.2.1 ClpFactorization::ClpFactorization ( )

Default constructor.

4.43.2.2 ClpFactorization::ClpFactorization (const **CoinFactorization** &)

The copy constructor from an **CoinFactorization**.

4.43.2.3 ClpFactorization::ClpFactorization (const **ClpFactorization** & , int *denseIfSmaller* = 0)

The copy constructor.

4.43.2.4 ClpFactorization::ClpFactorization (const **CoinOtherFactorization** &)

The copy constructor from an **CoinOtherFactorization**.

4.43.3 Member Function Documentation

4.43.3.1 int ClpFactorization::factorize (ClpSimplex ∗ *model*, int *solveType*, bool *valuesPass*)

When part of LP - given by basic variables.

Actually does factorization. Arrays passed in have non negative value to say basic. If status is okay, basic variables have pivot row - this is only needed if increasingRows_ >1. Allows scaling If status is singular, then basic variables have pivot row and ones thrown out have -1 returns 0 -okay, -1 singular, -2 too many in basis, -99 memory

4.43.3.2 int ClpFactorization::replaceColumn (const ClpSimplex ∗ *model*, CoinIndexedVector ∗ *regionSparse*, CoinIndexedVector ∗ *tableauColumn*, int *pivotRow*, double *pivotCheck*, bool *checkBeforeModifying* = `false`, double *acceptablePivot* = `1.0e-8`)

Replaces one Column to basis, returns 0=OK, 1=Probably OK, 2=singular, 3=no room If checkBeforeModifying is true will do all accuracy checks before modifying factorization.

Whether to set this depends on speed considerations. You could just do this on first iteration after factorization and thereafter re-factorize partial update already in U

4.43.3.3 int ClpFactorization::updateTwoColumnsFT (CoinIndexedVector ∗ *regionSparse1*, CoinIndexedVector ∗ *regionSparse2*, CoinIndexedVector ∗ *regionSparse3*, bool *noPermuteRegion3* = `false`)

Updates one column (FTRAN) from region2 Tries to do FT update number returned is negative if no room.

Also updates region3 region1 starts as zero and is zero at end

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

- ClpFactorization.hpp

4.44 ClpGubDynamicMatrix Class Reference

This implements Gub rows plus a ClpPackedMatrix.

`#include <ClpGubDynamicMatrix.hpp>`

Inheritance diagram for ClpGubDynamicMatrix:

Collaboration diagram for ClpGubDynamicMatrix:

Public Member Functions

Main functions provided

- virtual void partialPricing (ClpSimplex ∗model, double start, double end, int &bestSequence, int &number←Wanted)

  Partial pricing.
- virtual int synchronize (ClpSimplex ∗model, int mode)

  This is local to Gub to allow synchronization: mode=0 when status of basis is good mode=1 when variable is flagged mode=2 when all variables unflagged (returns number flagged) mode=3 just reset costs (primal) mode=4 correct number of dual infeasibilities mode=5 return 4 if time to re-factorize mode=8 - make sure set is clean mode=9 - adjust lower, upper on set by incoming.
- virtual void useEffectiveRhs (ClpSimplex ∗model, bool cheapest=true)

Sets up an effective RHS and does gub crash if needed.

- virtual int updatePivot (ClpSimplex ∗model, double oldInValue, double oldOutValue)

  update information for a pivot (and effective rhs)
- void insertNonBasic (int sequence, int iSet)

  Add a new variable to a set.
- virtual double ∗ rhsOffset (ClpSimplex ∗model, bool forceRefresh=false, bool check=false)

  Returns effective RHS offset if it is being used.
- virtual void times (double scalar, const double ∗x, double ∗y) const

  Return `y + A * scalar *x` *in* `y`.
- virtual int checkFeasible (ClpSimplex ∗model, double &sum) const

  Just for debug Returns sum and number of primal infeasibilities.
- void cleanData (ClpSimplex ∗model)

  Cleans data after setWarmStart.

Constructors, destructor

- ClpGubDynamicMatrix ()

  Default constructor.
- virtual ∼ClpGubDynamicMatrix ()

  Destructor.

Copy method

- ClpGubDynamicMatrix (const ClpGubDynamicMatrix &)

  The copy constructor.
- ClpGubDynamicMatrix (ClpSimplex ∗model, int numberSets, int numberColumns, const int ∗starts, const double ∗lower, const double ∗upper, const int ∗startColumn, const int ∗row, const double ∗element, const double ∗cost, const double ∗lowerColumn=NULL, const double ∗upperColumn=NULL, const unsigned char ∗status=NULL)

  This is the real constructor.
- ClpGubDynamicMatrix & **operator=** (const ClpGubDynamicMatrix &)
- virtual ClpMatrixBase ∗ clone () const

  Clone.

Protected Attributes

Data members

The data members are protected to allow access for derived classes.

- double objectiveOffset_

  Saved value of objective offset.
- CoinBigIndex ∗ startColumn_

  Starts of each column.
- int ∗ row_

  rows
- double ∗ element_

  elements
- double ∗ cost_

  costs
- int ∗ fullStart_

  full starts
- int ∗ id_

ids of active columns (just index here)
- unsigned char * dynamicStatus_

  for status and which bound
- double * lowerColumn_

  Optional lower bounds on columns.
- double * upperColumn_

  Optional upper bounds on columns.
- double * lowerSet_

  Optional true lower bounds on sets.
- double * upperSet_

  Optional true upper bounds on sets.
- int numberGubColumns_

  size
- int firstAvailable_

  first free
- int savedFirstAvailable_

  saved first free
- int firstDynamic_

  first dynamic
- int lastDynamic_

  number of columns in dynamic model
- int numberElements_

  size of working matrix (max)

gets and sets

- enum DynamicStatus

  enums for status of various sorts
- bool flagged (int i) const

  Whether flagged.
- void **setFlagged** (int i)
- void **unsetFlagged** (int i)
- void **setDynamicStatus** (int sequence, DynamicStatus status)
- DynamicStatus **getDynamicStatus** (int sequence) const
- double objectiveOffset () const

  Saved value of objective offset.
- CoinBigIndex * startColumn () const

  Starts of each column.
- int * row () const

  rows
- double * element () const

  elements
- double * cost () const

  costs
- int * fullStart () const

  full starts
- int * id () const

  ids of active columns (just index here)
- double * lowerColumn () const

Optional lower bounds on columns.

- double ∗ upperColumn () const

  Optional upper bounds on columns.
- double ∗ lowerSet () const

  Optional true lower bounds on sets.
- double ∗ upperSet () const

  Optional true upper bounds on sets.
- int numberGubColumns () const

  size
- int firstAvailable () const

  first free
- void setFirstAvailable (int value)

  set first free
- int firstDynamic () const

  first dynamic
- int lastDynamic () const

  number of columns in dynamic model
- int numberElements () const

  size of working matrix (max)
- unsigned char ∗ gubRowStatus () const

  Status region for gub slacks.
- unsigned char ∗ dynamicStatus () const

  Status region for gub variables.
- int whichSet (int sequence) const

  Returns which set a variable is in.

Additional Inherited Members

4.44.1 Detailed Description

This implements Gub rows plus a ClpPackedMatrix.

This a dynamic version which stores the gub part and dynamically creates matrix. All bounds are assumed to be zero and infinity

This is just a simple example for real column generation

Definition at line 20 of file ClpGubDynamicMatrix.hpp.

4.44.2 Constructor & Destructor Documentation

4.44.2.1 ClpGubDynamicMatrix::ClpGubDynamicMatrix ( )

Default constructor.

4.44.2.2 ClpGubDynamicMatrix::ClpGubDynamicMatrix (const ClpGubDynamicMatrix &)

The copy constructor.

4.44.2.3 ClpGubDynamicMatrix::ClpGubDynamicMatrix (ClpSimplex * *model*, int *numberSets*, int *numberColumns*, const int * *starts*, const double * *lower*, const double * *upper*, const int * *startColumn*, const int * *row*, const double * *element*, const double * *cost*, const double * *lowerColumn* = NULL, const double * *upperColumn* = NULL, const unsigned char * *status* = NULL)

This is the real constructor.

It assumes factorization frequency will not be changed. This resizes model !!!!

4.44.3 Member Function Documentation

4.44.3.1 virtual double* ClpGubDynamicMatrix::rhsOffset (ClpSimplex * *model*, bool *forceRefresh* = false, bool *check* = false) [virtual]

Returns effective RHS offset if it is being used.

This is used for long problems or big gub or anywhere where going through full columns is expensive. This may re-compute

Reimplemented from ClpGubMatrix.

4.44.3.2 virtual void ClpGubDynamicMatrix::times (double *scalar*, const double * *x*, double * *y*) const [virtual]

Return `y + A * scalar *x` in `y`.

Precondition

`x` must be of size `numColumns()`
`y` must be of size `numRows()`

Reimplemented from ClpPackedMatrix.

4.44.3.3 virtual int ClpGubDynamicMatrix::checkFeasible (ClpSimplex * *model*, double & *sum*) const [virtual]

Just for debug Returns sum and number of primal infeasibilities.

Recomputes keys

Reimplemented from ClpMatrixBase.

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

- ClpGubDynamicMatrix.hpp

4.45 ClpGubMatrix Class Reference

This implements Gub rows plus a ClpPackedMatrix.

`#include <ClpGubMatrix.hpp>`

Inheritance diagram for ClpGubMatrix:

Collaboration diagram for ClpGubMatrix:

Public Member Functions

Main functions provided

- virtual ClpMatrixBase ∗ reverseOrderedCopy () const

  Returns a new matrix in reverse order without gaps (GUB wants NULL)
- virtual CoinBigIndex countBasis (const int ∗whichColumn, int &numberColumnBasic)

  Returns number of elements in column part of basis.
- virtual void fillBasis (ClpSimplex ∗model, const int ∗whichColumn, int &numberColumnBasic, int ∗row, int ∗start, int ∗rowCount, int ∗columnCount, CoinFactorizationDouble ∗element)

  Fills in column part of basis.
- virtual void unpack (const ClpSimplex ∗model, **CoinIndexedVector** ∗rowArray, int column) const

  Unpacks a column into an CoinIndexedvector.
- virtual void unpackPacked (ClpSimplex ∗model, **CoinIndexedVector** ∗rowArray, int column) const

  Unpacks a column into an CoinIndexedvector in packed foramt Note that model is NOT const.
- virtual void add (const ClpSimplex ∗model, **CoinIndexedVector** ∗rowArray, int column, double multiplier) const

  Adds multiple of a column into an CoinIndexedvector You can use quickAdd to add to vector.
- virtual void add (const ClpSimplex ∗model, double ∗array, int column, double multiplier) const

  Adds multiple of a column into an array.
- virtual void partialPricing (ClpSimplex ∗model, double start, double end, int &bestSequence, int &numberWanted)

  Partial pricing.
- virtual int hiddenRows () const

  Returns number of hidden rows e.g. gub.

Matrix times vector methods

- virtual void transposeTimes (const ClpSimplex ∗model, double scalar, const **CoinIndexedVector** ∗x, **CoinIndexedVector** ∗y, **CoinIndexedVector** ∗z) const

  Return $x * scalar * A + y$ *in* z.
- virtual void transposeTimesByRow (const ClpSimplex ∗model, double scalar, const **CoinIndexedVector** ∗x, **CoinIndexedVector** ∗y, **CoinIndexedVector** ∗z) const

  Return $x * scalar * A + y$ *in* z.
- virtual void subsetTransposeTimes (const ClpSimplex ∗model, const **CoinIndexedVector** ∗x, const **CoinIndexedVector** ∗y, **CoinIndexedVector** ∗z) const

  Return $x *A$ *in* z *but just for indices in y.*
- virtual int extendUpdated (ClpSimplex ∗model, **CoinIndexedVector** ∗update, int mode)

  expands an updated column to allow for extra rows which the main solver does not know about and returns number added if mode 0.
- virtual void primalExpanded (ClpSimplex ∗model, int mode)

  mode=0 - Set up before "update" and "times" for primal solution using extended rows mode=1 - Cleanup primal solution after "times" using extended rows.
- virtual void dualExpanded (ClpSimplex ∗model, **CoinIndexedVector** ∗array, double ∗other, int mode)

  mode=0 - Set up before "updateTranspose" and "transposeTimes" for duals using extended updates array (and may use other if dual values pass) mode=1 - Update dual solution after "transposeTimes" using extended rows.
- virtual int generalExpanded (ClpSimplex ∗model, int mode, int &number)

  mode=0 - Create list of non-key basics in pivotVariable_ using number as numberBasic in and out mode=1 - Set all key variables as basic mode=2 - return number extra rows needed, number gives maximum number basic mode=3 - before replaceColumn mode=4 - return 1 if can do primal, 2 if dual, 3 if both mode=5 - save any status stuff (when in good state) mode=6 - restore status stuff mode=7 - flag given variable (normally sequenceIn) mode=8 - unflag all variables mode=9 - synchronize costs mode=10 - return 1 if there may be changing bounds on variable (column generation) mode=11 - make sure set is clean (used when a variable rejected - but not flagged) mode=12 - after factorize but before permute stuff mode=13 - at end of simplex to delete stuff

- virtual int updatePivot (ClpSimplex *model, double oldInValue, double oldOutValue)

  update information for a pivot (and effective rhs)
- virtual void useEffectiveRhs (ClpSimplex *model, bool cheapest=true)

  Sets up an effective RHS and does gub crash if needed.
- virtual double * rhsOffset (ClpSimplex *model, bool forceRefresh=false, bool check=false)

  Returns effective RHS offset if it is being used.
- virtual int synchronize (ClpSimplex *model, int mode)

  This is local to Gub to allow synchronization: mode=0 when status of basis is good mode=1 when variable is flagged mode=2 when all variables unflagged (returns number flagged) mode=3 just reset costs (primal) mode=4 correct number of dual infeasibilities mode=5 return 4 if time to re-factorize mode=6 - return 1 if there may be changing bounds on variable (column generation) mode=7 - do extra restores for column generation mode=8 - make sure set is clean mode=9 - adjust lower, upper on set by incoming.
- virtual void correctSequence (const ClpSimplex *model, int &sequenceIn, int &sequenceOut)

  Correct sequence in and out to give true value.

Constructors, destructor

- ClpGubMatrix ()

  Default constructor.
- virtual ~ClpGubMatrix ()

  Destructor.

Copy method

- ClpGubMatrix (const ClpGubMatrix &)

  The copy constructor.
- ClpGubMatrix (const **CoinPackedMatrix** &)

  The copy constructor from an **CoinPackedMatrix**.
- ClpGubMatrix (const ClpGubMatrix &wholeModel, int numberRows, const int *whichRows, int numberColumns, const int *whichColumns)

  Subset constructor (without gaps).
- **ClpGubMatrix** (const **CoinPackedMatrix** &wholeModel, int numberRows, const int *whichRows, int numberColumns, const int *whichColumns)
- ClpGubMatrix (**CoinPackedMatrix** *matrix)

  This takes over ownership (for space reasons)
- ClpGubMatrix (ClpPackedMatrix *matrix, int numberSets, const int *start, const int *end, const double *lower, const double *upper, const unsigned char *status=NULL)

  This takes over ownership (for space reasons) and is the real constructor.
- ClpGubMatrix & **operator=** (const ClpGubMatrix &)
- virtual ClpMatrixBase * clone () const

  Clone.
- virtual ClpMatrixBase * subsetClone (int numberRows, const int *whichRows, int numberColumns, const int *whichColumns) const

  Subset clone (without gaps).
- void redoSet (ClpSimplex *model, int newKey, int oldKey, int iSet)

  redoes next_ for a set.

gets and sets

- ClpSimplex::Status getStatus (int sequence) const

  Status.
- void **setStatus** (int sequence, ClpSimplex::Status status)
- void setFlagged (int sequence)

  To flag a variable.

- void **clearFlagged** (int sequence)
- bool **flagged** (int sequence) const
- void setAbove (int sequence)

    To say key is above ub.
- void setFeasible (int sequence)

    To say key is feasible.
- void setBelow (int sequence)

    To say key is below lb.
- double **weight** (int sequence) const
- int ∗ start () const

    Starts.
- int ∗ end () const

    End.
- double ∗ lower () const

    Lower bounds on sets.
- double ∗ upper () const

    Upper bounds on sets.
- int ∗ keyVariable () const

    Key variable of set.
- int ∗ backward () const

    Backward pointer to set number.
- int numberSets () const

    Number of sets (gub rows)
- void switchOffCheck ()

    Switches off dj checking each factorization (for BIG models)

Protected Attributes

Data members

The data members are protected to allow access for derived classes.

- double sumDualInfeasibilities_

    Sum of dual infeasibilities.
- double sumPrimalInfeasibilities_

    Sum of primal infeasibilities.
- double sumOfRelaxedDualInfeasibilities_

    Sum of Dual infeasibilities using tolerance based on error in duals.
- double sumOfRelaxedPrimalInfeasibilities_

    Sum of Primal infeasibilities using tolerance based on error in primals.
- double infeasibilityWeight_

    Infeasibility weight when last full pass done.
- int ∗ start_

    Starts.
- int ∗ end_

    End.
- double ∗ lower_

    Lower bounds on sets.
- double ∗ upper_

    Upper bounds on sets.
- unsigned char ∗ status_

    Status of slacks.

- unsigned char ∗ saveStatus_

  Saved status of slacks.
- int ∗ savedKeyVariable_

  Saved key variables.
- int ∗ backward_

  Backward pointer to set number.
- int ∗ backToPivotRow_

  Backward pointer to pivot row !!!
- double ∗ changeCost_

  Change in costs for keys.
- int ∗ keyVariable_

  Key variable of set.
- int ∗ next_

  Next basic variable in set - starts at key and end with -(set+1).
- int ∗ toIndex_

  Backward pointer to index in **CoinIndexedVector**.
- int ∗ **fromIndex_**
- ClpSimplex ∗ model_

  Pointer back to model.
- int numberDualInfeasibilities_

  Number of dual infeasibilities.
- int numberPrimalInfeasibilities_

  Number of primal infeasibilities.
- int noCheck_

  If pricing will declare victory (i.e.
- int numberSets_

  Number of sets (gub rows)
- int saveNumber_

  Number in vector without gub extension.
- int possiblePivotKey_

  Pivot row of possible next key.
- int gubSlackIn_

  Gub slack in (set number or -1)
- int firstGub_

  First gub variables (same as start_[0] at present)
- int lastGub_

  last gub variable (same as end_[numberSets_-1] at present)
- int gubType_

  type of gub - 0 not contiguous, 1 contiguous add 8 bit to say no ubs on individual variables

Additional Inherited Members

4.45.1 Detailed Description

This implements Gub rows plus a ClpPackedMatrix.

There will be a version using ClpPlusMinusOne matrix but there is no point doing one with ClpNetworkMatrix (although an embedded network is attractive).

Definition at line 22 of file ClpGubMatrix.hpp.

4.45.2 Constructor & Destructor Documentation

4.45.2.1 ClpGubMatrix::ClpGubMatrix ( )

Default constructor.

4.45.2.2 ClpGubMatrix::ClpGubMatrix (const **ClpGubMatrix** &)

The copy constructor.

4.45.2.3 ClpGubMatrix::ClpGubMatrix (const **CoinPackedMatrix** &)

The copy constructor from an **CoinPackedMatrix**.

4.45.2.4 ClpGubMatrix::ClpGubMatrix (const **ClpGubMatrix** & *wholeModel*, int *numberRows*, const int * *whichRows*, int *numberColumns*, const int * *whichColumns*)

Subset constructor (without gaps).

Duplicates are allowed and order is as given

4.45.3 Member Function Documentation

4.45.3.1 virtual void ClpGubMatrix::unpackPacked (**ClpSimplex** * *model*, **CoinIndexedVector** * *rowArray*, int *column*) const [virtual]

Unpacks a column into an CoinIndexedvector in packed foramt Note that model is NOT const.

Bounds and objective could be modified if doing column generation (just for this variable)

Reimplemented from ClpPackedMatrix.

4.45.3.2 virtual void ClpGubMatrix::transposeTimes (const **ClpSimplex** * *model*, double *scalar*, const **CoinIndexedVector** * *x*, **CoinIndexedVector** * *y*, **CoinIndexedVector** * *z*) const [virtual]

Return `x * scalar * A + y` in `z`.

Can use y as temporary array (will be empty at end) Note - If x packed mode - then z packed mode Squashes small elements and knows about ClpSimplex

Reimplemented from ClpPackedMatrix.

4.45.3.3 virtual void ClpGubMatrix::transposeTimesByRow (const **ClpSimplex** * *model*, double *scalar*, const **CoinIndexedVector** * *x*, **CoinIndexedVector** * *y*, **CoinIndexedVector** * *z*) const [virtual]

Return `x * scalar * A + y` in `z`.

Can use y as temporary array (will be empty at end) Note - If x packed mode - then z packed mode Squashes small elements and knows about ClpSimplex. This version uses row copy

Reimplemented from ClpPackedMatrix.

4.45.3.4 virtual void ClpGubMatrix::subsetTransposeTimes (const ClpSimplex * *model*, const CoinIndexedVector * *x*, const CoinIndexedVector * *y*, CoinIndexedVector * *z*) const [virtual]

Return `x *A` in `z` but just for indices in y.

Note - z always packed mode

Reimplemented from ClpPackedMatrix.

4.45.3.5 virtual int ClpGubMatrix::extendUpdated (ClpSimplex * *model*, CoinIndexedVector * *update*, int *mode*) [virtual]

expands an updated column to allow for extra rows which the main solver does not know about and returns number added if mode 0.

If mode 1 deletes extra entries

This active in Gub

Reimplemented from ClpMatrixBase.

4.45.3.6 virtual void ClpGubMatrix::primalExpanded (ClpSimplex * *model*, int *mode*) [virtual]

mode=0 - Set up before "update" and "times" for primal solution using extended rows mode=1 - Cleanup primal solution after "times" using extended rows.

mode=2 - Check (or report on) primal infeasibilities

Reimplemented from ClpMatrixBase.

4.45.3.7 virtual void ClpGubMatrix::dualExpanded (ClpSimplex * *model*, CoinIndexedVector * *array*, double * *other*, int *mode*) [virtual]

mode=0 - Set up before "updateTranspose" and "transposeTimes" for duals using extended updates array (and may use other if dual values pass) mode=1 - Update dual solution after "transposeTimes" using extended rows.

mode=2 - Compute all djs and compute key dual infeasibilities mode=3 - Report on key dual infeasibilities mode=4 - Modify before updateTranspose in partial pricing

Reimplemented from ClpMatrixBase.

4.45.3.8 virtual double* ClpGubMatrix::rhsOffset (ClpSimplex * *model*, bool *forceRefresh* = `false`, bool *check* = `false`) [virtual]

Returns effective RHS offset if it is being used.

This is used for long problems or big gub or anywhere where going through full columns is expensive. This may re-compute

Reimplemented from ClpMatrixBase.

Reimplemented in ClpGubDynamicMatrix.

4.45.3.9 virtual ClpMatrixBase* ClpGubMatrix::subsetClone (int *numberRows*, const int * *whichRows*, int *numberColumns*, const int * *whichColumns*) const [virtual]

Subset clone (without gaps).

Duplicates are allowed and order is as given

Reimplemented from ClpPackedMatrix.

4.45.3.10 void ClpGubMatrix::redoSet (ClpSimplex * *model*, int *newKey*, int *oldKey*, int *iSet*)

redoes next_ for a set.

4.45.4 Member Data Documentation

4.45.4.1 int* ClpGubMatrix::next_ [mutable], [protected]

Next basic variable in set - starts at key and end with -(set+1).

Now changes to -(nonbasic+1). next_ has extra space for 2* longest set

Definition at line 323 of file ClpGubMatrix.hpp.

4.45.4.2 int ClpGubMatrix::noCheck_ [protected]

If pricing will declare victory (i.e.

no check every factorization). -1 - always check 0 - don't check 1 - in don't check mode but looks optimal

Definition at line 339 of file ClpGubMatrix.hpp.

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

- ClpGubMatrix.hpp

4.46 ClpHashValue Class Reference

Collaboration diagram for ClpHashValue:

Classes

- struct CoinHashLink
  Data.

Public Member Functions

Useful methods

- int index (double value) const
  Return index or -1 if not found.
- int addValue (double value)
  Add value to list and return index.
- int numberEntries () const
  Number of different entries.

Constructors, destructor

- ClpHashValue ()
  Default constructor.
- ClpHashValue (ClpSimplex ∗model)
  Useful constructor.
- virtual ∼ClpHashValue ()
  Destructor.

Copy method

- ClpHashValue (const ClpHashValue &)
  The copy constructor.
- ClpHashValue & operator= (const ClpHashValue &)
  =

Protected Attributes

Data members

The data members are protected to allow access for derived classes.

- CoinHashLink ∗ hash_
  Hash table.
- int numberHash_
  Number of entries in hash table.
- int maxHash_
  Maximum number of entries in hash table i.e. size.
- int lastUsed_
  Last used space.

4.46.1 Detailed Description

Definition at line 288 of file ClpNode.hpp.

4.46.2 Constructor & Destructor Documentation

4.46.2.1 ClpHashValue::ClpHashValue ( )

Default constructor.

4.46.2.2 ClpHashValue::ClpHashValue (**ClpSimplex** ∗ *model*)

Useful constructor.

4.46.2.3 ClpHashValue::ClpHashValue (const **ClpHashValue** &)

The copy constructor.

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

- ClpNode.hpp

4.47 ClpInterior Class Reference

This solves LPs using interior point methods.

`#include <ClpInterior.hpp>`

Inheritance diagram for ClpInterior:

Collaboration diagram for ClpInterior:

Public Member Functions

Constructors and destructor and copy

- ClpInterior ()
  Default constructor.
- ClpInterior (const ClpInterior &)
  Copy constructor.
- ClpInterior (const ClpModel &)
  Copy constructor from model.
- ClpInterior (const ClpModel ∗wholeModel, int numberRows, const int ∗whichRows, int numberColumns, const int ∗whichColumns, bool dropNames=true, bool dropIntegers=true)
  Subproblem constructor.
- ClpInterior & operator= (const ClpInterior &rhs)
  Assignment operator. This copies the data.
- ∼ClpInterior ()
  Destructor.
- void loadProblem (const ClpMatrixBase &matrix, const double ∗collb, const double ∗colub, const double ∗obj, const double ∗rowlb, const double ∗rowub, const double ∗rowObjective=NULL)
  Loads a problem (the constraints on the rows are given by lower and upper bounds).
- void **loadProblem** (const **CoinPackedMatrix** &matrix, const double ∗collb, const double ∗colub, const double ∗obj, const double ∗rowlb, const double ∗rowub, const double ∗rowObjective=NULL)
- void loadProblem (const int numcols, const int numrows, const CoinBigIndex ∗start, const int ∗index, const double ∗value, const double ∗collb, const double ∗colub, const double ∗obj, const double ∗rowlb, const double ∗rowub, const double ∗rowObjective=NULL)
  Just like the other loadProblem() method except that the matrix is given in a standard column major ordered format (without gaps).
- void loadProblem (const int numcols, const int numrows, const CoinBigIndex ∗start, const int ∗index, const double ∗value, const int ∗length, const double ∗collb, const double ∗colub, const double ∗obj, const double ∗rowlb, const double ∗rowub, const double ∗rowObjective=NULL)
  This one is for after presolve to save memory.
- int readMps (const char ∗filename, bool keepNames=false, bool ignoreErrors=false)
  Read an mps file from the given filename.
- void borrowModel (ClpModel &otherModel)
  Borrow model.
- void returnModel (ClpModel &otherModel)
  Return model - updates any scalars.

Functions most useful to user

- int pdco ()
  Pdco algorithm - see ClpPdco.hpp for method.
- int **pdco** (ClpPdcoBase ∗stuff, Options &options, Info &info, Outfo &outfo)
- int primalDual ()

Primal-Dual Predictor-Corrector barrier.

most useful gets and sets

- bool primalFeasible () const

    If problem is primal feasible.
- bool dualFeasible () const

    If problem is dual feasible.
- int algorithm () const

    Current (or last) algorithm.
- void setAlgorithm (int value)

    Set algorithm.
- CoinWorkDouble sumDualInfeasibilities () const

    Sum of dual infeasibilities.
- CoinWorkDouble sumPrimalInfeasibilities () const

    Sum of primal infeasibilities.
- CoinWorkDouble dualObjective () const

    dualObjective.
- CoinWorkDouble primalObjective () const

    primalObjective.
- CoinWorkDouble diagonalNorm () const

    diagonalNorm
- CoinWorkDouble linearPerturbation () const

    linearPerturbation
- void **setLinearPerturbation** (CoinWorkDouble value)
- CoinWorkDouble projectionTolerance () const

    projectionTolerance
- void **setProjectionTolerance** (CoinWorkDouble value)
- CoinWorkDouble diagonalPerturbation () const

    diagonalPerturbation
- void **setDiagonalPerturbation** (CoinWorkDouble value)
- CoinWorkDouble gamma () const

    gamma
- void **setGamma** (CoinWorkDouble value)
- CoinWorkDouble delta () const

    delta
- void **setDelta** (CoinWorkDouble value)
- CoinWorkDouble complementarityGap () const

    ComplementarityGap.
- CoinWorkDouble largestPrimalError () const

    Largest error on Ax-b.
- CoinWorkDouble largestDualError () const

    Largest error on basic duals.
- int maximumBarrierIterations () const

    Maximum iterations.
- void **setMaximumBarrierIterations** (int value)
- void setCholesky (ClpCholeskyBase *cholesky)

    Set cholesky (and delete present one)
- int numberFixed () const

    Return number fixed to see if worth presolving.
- void fixFixed (bool reallyFix=true)

    fix variables interior says should be.
- CoinWorkDouble * primalR () const

Primal erturbation vector.
- CoinWorkDouble ∗ dualR () const

  Dual erturbation vector.

public methods

- CoinWorkDouble rawObjectiveValue () const

  Raw objective value (so always minimize)
- int isColumn (int sequence) const

  Returns 1 if sequence indicates column.
- int sequenceWithin (int sequence) const

  Returns sequence number within section.
- void checkSolution ()

  Checks solution.
- CoinWorkDouble quadraticDjs (CoinWorkDouble ∗djRegion, const CoinWorkDouble ∗solution, CoinWork←Double scaleFactor)

  Modifies djs to allow for quadratic.
- void setFixed (int sequence)

  To say a variable is fixed.
- void **clearFixed** (int sequence)
- bool **fixed** (int sequence) const
- void setFlagged (int sequence)

  To flag a variable.
- void **clearFlagged** (int sequence)
- bool **flagged** (int sequence) const
- void setFixedOrFree (int sequence)

  To say a variable is fixed OR free.
- void **clearFixedOrFree** (int sequence)
- bool **fixedOrFree** (int sequence) const
- void setLowerBound (int sequence)

  To say a variable has lower bound.
- void **clearLowerBound** (int sequence)
- bool **lowerBound** (int sequence) const
- void setUpperBound (int sequence)

  To say a variable has upper bound.
- void **clearUpperBound** (int sequence)
- bool **upperBound** (int sequence) const
- void setFakeLower (int sequence)

  To say a variable has fake lower bound.
- void **clearFakeLower** (int sequence)
- bool **fakeLower** (int sequence) const
- void setFakeUpper (int sequence)

  To say a variable has fake upper bound.
- void **clearFakeUpper** (int sequence)
- bool **fakeUpper** (int sequence) const

Protected Member Functions

protected methods

- void gutsOfDelete ()

  Does most of deletion.
- void gutsOfCopy (const ClpInterior &rhs)

Does most of copying.
- bool createWorkingData ()

  Returns true if data looks okay, false if not.
- void **deleteWorkingData** ()
- bool sanityCheck ()

  Sanity check on input rim data.
- int housekeeping ()

  This does housekeeping.

Friends

- void ClpInteriorUnitTest (const std::string &mpsDir, const std::string &netlibDir)

  A function that tests the methods in the ClpInterior class.

data. Many arrays have a row part and a column part.

There is a single array with both - columns then rows and then normally two arrays pointing to rows and columns.

The single array is the owner of memory

- CoinWorkDouble largestPrimalError_

  Largest error on Ax-b.
- CoinWorkDouble largestDualError_

  Largest error on basic duals.
- CoinWorkDouble sumDualInfeasibilities_

  Sum of dual infeasibilities.
- CoinWorkDouble sumPrimalInfeasibilities_

  Sum of primal infeasibilities.
- CoinWorkDouble worstComplementarity_

  Worst complementarity.
- CoinWorkDouble ∗ lower_

  Working copy of lower bounds (Owner of arrays below)
- CoinWorkDouble ∗ rowLowerWork_

  Row lower bounds - working copy.
- CoinWorkDouble ∗ columnLowerWork_

  Column lower bounds - working copy.
- CoinWorkDouble ∗ upper_

  Working copy of upper bounds (Owner of arrays below)
- CoinWorkDouble ∗ rowUpperWork_

  Row upper bounds - working copy.
- CoinWorkDouble ∗ columnUpperWork_

  Column upper bounds - working copy.
- CoinWorkDouble ∗ cost_

  Working copy of objective.
- ClpLsqr ∗ lsqrObject_

  Pointer to Lsqr object.
- ClpPdcoBase ∗ pdcoStuff_

  Pointer to stuff.

- CoinWorkDouble mu_

    Below here is standard barrier stuff mu.
- CoinWorkDouble objectiveNorm_

    objectiveNorm.
- CoinWorkDouble rhsNorm_

    rhsNorm.
- CoinWorkDouble solutionNorm_

    solutionNorm.
- CoinWorkDouble dualObjective_

    dualObjective.
- CoinWorkDouble primalObjective_

    primalObjective.
- CoinWorkDouble diagonalNorm_

    diagonalNorm.
- CoinWorkDouble stepLength_

    stepLength
- CoinWorkDouble linearPerturbation_

    linearPerturbation
- CoinWorkDouble diagonalPerturbation_

    diagonalPerturbation
- CoinWorkDouble **gamma_**
- CoinWorkDouble **delta_**
- CoinWorkDouble targetGap_

    targetGap
- CoinWorkDouble projectionTolerance_

    projectionTolerance
- CoinWorkDouble maximumRHSError_

    maximumRHSError. maximum Ax
- CoinWorkDouble maximumBoundInfeasibility_

    maximumBoundInfeasibility.
- CoinWorkDouble maximumDualError_

    maximumDualError.
- CoinWorkDouble diagonalScaleFactor_

    diagonalScaleFactor.
- CoinWorkDouble scaleFactor_

    scaleFactor. For scaling objective
- CoinWorkDouble actualPrimalStep_

    actualPrimalStep
- CoinWorkDouble actualDualStep_

    actualDualStep
- CoinWorkDouble smallestInfeasibility_

    smallestInfeasibility
- CoinWorkDouble **historyInfeasibility_** [LENGTH_HISTORY]
- CoinWorkDouble complementarityGap_

    complementarityGap.
- CoinWorkDouble baseObjectiveNorm_

    baseObjectiveNorm

- CoinWorkDouble worstDirectionAccuracy_

  worstDirectionAccuracy
- CoinWorkDouble maximumRHSChange_

  maximumRHSChange
- CoinWorkDouble ∗ errorRegion_

  errorRegion. i.e. Ax
- CoinWorkDouble ∗ rhsFixRegion_

  rhsFixRegion.
- CoinWorkDouble ∗ upperSlack_

  upperSlack
- CoinWorkDouble ∗ lowerSlack_

  lowerSlack
- CoinWorkDouble ∗ diagonal_

  diagonal
- CoinWorkDouble ∗ solution_

  solution
- CoinWorkDouble ∗ workArray_

  work array
- CoinWorkDouble ∗ deltaX_

  delta X
- CoinWorkDouble ∗ deltaY_

  delta Y
- CoinWorkDouble ∗ deltaZ_

  deltaZ.
- CoinWorkDouble ∗ deltaW_

  deltaW.
- CoinWorkDouble ∗ deltaSU_

  deltaS.
- CoinWorkDouble ∗ **deltaSL_**
- CoinWorkDouble ∗ primalR_

  Primal regularization array.
- CoinWorkDouble ∗ dualR_

  Dual regularization array.
- CoinWorkDouble ∗ rhsB_

  rhs B
- CoinWorkDouble ∗ rhsU_

  rhsU.
- CoinWorkDouble ∗ rhsL_

  rhsL.
- CoinWorkDouble ∗ rhsZ_

  rhsZ.
- CoinWorkDouble ∗ rhsW_

  rhsW.
- CoinWorkDouble ∗ rhsC_

  rhs C
- CoinWorkDouble ∗ zVec_

  zVec

- CoinWorkDouble ∗ wVec_

  wVec
- ClpCholeskyBase ∗ cholesky_

  cholesky.
- int numberComplementarityPairs_

  numberComplementarityPairs i.e. ones with lower and/or upper bounds (not fixed)
- int numberComplementarityItems_

  numberComplementarityItems_ i.e. number of active bounds
- int maximumBarrierIterations_

  Maximum iterations.
- bool gonePrimalFeasible_

  gonePrimalFeasible.
- bool goneDualFeasible_

  goneDualFeasible.
- int algorithm_

  Which algorithm being used.
- CoinWorkDouble **xsize_**
- CoinWorkDouble **zsize_**
- CoinWorkDouble ∗ rhs_

  Rhs.
- CoinWorkDouble ∗ **x_**
- CoinWorkDouble ∗ **y_**
- CoinWorkDouble ∗ **dj_**

Additional Inherited Members

4.47.1 Detailed Description

This solves LPs using interior point methods.

It inherits from ClpModel and all its arrays are created at algorithm time.

Definition at line 72 of file ClpInterior.hpp.

4.47.2 Constructor & Destructor Documentation

4.47.2.1 ClpInterior::ClpInterior (const ClpModel ∗ *wholeModel*, int *numberRows*, const int ∗ *whichRows*, int *numberColumns*, const int ∗ *whichColumns*, bool *dropNames* = `true`, bool *dropIntegers* = `true`)

Subproblem constructor.

A subset of whole model is created from the row and column lists given. The new order is given by list order and duplicates are allowed. Name and integer information can be dropped

4.47.3 Member Function Documentation

4.47.3.1 void ClpInterior::loadProblem (const ClpMatrixBase & *matrix*, const double ∗ *collb*, const double ∗ *colub*, const double ∗ *obj*, const double ∗ *rowlb*, const double ∗ *rowub*, const double ∗ *rowObjective* = `NULL`)

Loads a problem (the constraints on the rows are given by lower and upper bounds).

If a pointer is 0 then the following values are the default:

- `colub`: all columns have upper bound infinity
- `collb`: all columns have lower bound 0
- `rowub`: all rows have upper bound infinity
- `rowlb`: all rows have lower bound -infinity
- `obj`: all variables have 0 objective coefficient

4.47.3.2 void ClpInterior::loadProblem (const int *numcols*, const int *numrows*, const CoinBigIndex * *start*, const int * *index*, const double * *value*, const double * *collb*, const double * *colub*, const double * *obj*, const double * *rowlb*, const double * *rowub*, const double * *rowObjective* = `NULL`)

Just like the other loadProblem() method except that the matrix is given in a standard column major ordered format (without gaps).

4.47.3.3 void ClpInterior::borrowModel (ClpModel & *otherModel*)

Borrow model.

This is so we dont have to copy large amounts of data around. It assumes a derived class wants to overwrite an empty model with a real one - while it does an algorithm. This is same as ClpModel one.

4.47.3.4 void ClpInterior::fixFixed (bool *reallyFix* = `true`)

fix variables interior says should be.

If reallyFix false then just set values to exact bounds

4.47.3.5 CoinWorkDouble ClpInterior::quadraticDjs (CoinWorkDouble * *djRegion*, const CoinWorkDouble * *solution*, CoinWorkDouble *scaleFactor*)

Modifies djs to allow for quadratic.

returns quadratic offset

4.47.4 Friends And Related Function Documentation

4.47.4.1 void ClpInteriorUnitTest (const std::string & *mpsDir*, const std::string & *netlibDir*) `[friend]`

A function that tests the methods in the ClpInterior class.

The only reason for it not to be a member method is that this way it doesn't have to be compiled into the library. And that's a gain, because the library should be compiled with optimization on, but this method should be compiled with debugging.

It also does some testing of ClpFactorization class

4.47.5 Member Data Documentation

4.47.5.1 CoinWorkDouble ClpInterior::mu_ [protected]

Below here is standard barrier stuff mu.

Definition at line 441 of file ClpInterior.hpp.

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

- ClpInterior.hpp

4.48 ClpLinearObjective Class Reference

Linear Objective Class.

`#include <ClpLinearObjective.hpp>`

Inheritance diagram for ClpLinearObjective:

Collaboration diagram for ClpLinearObjective:

Public Member Functions

Stuff

- virtual double ∗ gradient (const ClpSimplex ∗model, const double ∗solution, double &offset, bool refresh, int includeLinear=2)

  Returns objective coefficients.
- virtual double reducedGradient (ClpSimplex ∗model, double ∗region, bool useFeasibleCosts)

  Returns reduced gradient.Returns an offset (to be added to current one).
- virtual double stepLength (ClpSimplex ∗model, const double ∗solution, const double ∗change, double maximumTheta, double &currentObj, double &predictedObj, double &thetaObj)

  Returns step length which gives minimum of objective for solution + theta ∗ change vector up to maximum theta.
- virtual double objectiveValue (const ClpSimplex ∗model, const double ∗solution) const

  Return objective value (without any ClpModel offset) (model may be NULL)
- virtual void resize (int newNumberColumns)

  Resize objective.
- virtual void deleteSome (int numberToDelete, const int ∗which)

  Delete columns in objective.
- virtual void reallyScale (const double ∗columnScale)

  Scale objective.

Constructors and destructors

- ClpLinearObjective ()

  Default Constructor.
- ClpLinearObjective (const double ∗objective, int numberColumns)

  Constructor from objective.
- ClpLinearObjective (const ClpLinearObjective &)

  Copy constructor.
- ClpLinearObjective (const ClpLinearObjective &rhs, int numberColumns, const int ∗whichColumns)

  Subset constructor.

- ClpLinearObjective & operator= (const ClpLinearObjective &rhs)

  Assignment operator.
- virtual ∼ClpLinearObjective ()

  Destructor.
- virtual ClpObjective * clone () const

  Clone.
- virtual ClpObjective * subsetClone (int numberColumns, const int *whichColumns) const

  Subset clone.

Additional Inherited Members

4.48.1 Detailed Description

Linear Objective Class.

Definition at line 17 of file ClpLinearObjective.hpp.

4.48.2 Constructor & Destructor Documentation

4.48.2.1 ClpLinearObjective::ClpLinearObjective (const **ClpLinearObjective** & *rhs*, int *numberColumns*, const int * *whichColumns*)

Subset constructor.

Duplicates are allowed and order is as given.

4.48.3 Member Function Documentation

4.48.3.1 virtual double* ClpLinearObjective::gradient (const **ClpSimplex** * *model*, const double * *solution*, double & *offset*, bool *refresh*, int *includeLinear* = 2) [virtual]

Returns objective coefficients.

Offset is always set to 0.0. All other parameters unused.

Implements ClpObjective.

4.48.3.2 virtual double ClpLinearObjective::stepLength (**ClpSimplex** * *model*, const double * *solution*, const double * *change*, double *maximumTheta*, double & *currentObj*, double & *predictedObj*, double & *thetaObj*) [virtual]

Returns step length which gives minimum of objective for solution + theta * change vector up to maximum theta.

arrays are numberColumns+numberRows Also sets current objective, predicted and at maximumTheta

Implements ClpObjective.

4.48.3.3 virtual **ClpObjective*** ClpLinearObjective::subsetClone (int *numberColumns*, const int * *whichColumns*) const [virtual]

Subset clone.

Duplicates are allowed and order is as given.

Reimplemented from ClpObjective.

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

- ClpLinearObjective.hpp

4.49 ClpLsqr Class Reference

This class implements LSQR.

`#include <ClpLsqr.hpp>`

Collaboration diagram for ClpLsqr:

Public Member Functions

Constructors and destructors

- ClpLsqr ()
  Default constructor.
- ClpLsqr (ClpInterior ∗model)
  Constructor for use with Pdco model (note modified for pdco!!!!)
- ClpLsqr (const ClpLsqr &)
  Copy constructor.
- ClpLsqr & operator= (const ClpLsqr &rhs)
  Assignment operator. This copies the data.
- ∼ClpLsqr ()
  Destructor.

Methods

- bool setParam (char ∗parmName, int parmValue)
  Set an int parameter.
- void do_lsqr (**CoinDenseVector**< double > &b, double damp, double atol, double btol, double conlim, int itnlim, bool show, Info info, **CoinDenseVector**< double > &x, int ∗istop, int ∗itn, Outfo ∗outfo, bool precon, **CoinDenseVector**< double > &Pr)
  Call the Lsqr algorithm.
- void matVecMult (int, **CoinDenseVector**< double > ∗, **CoinDenseVector**< double > ∗)
  Matrix-vector multiply - implemented by user.
- void **matVecMult** (int, **CoinDenseVector**< double > &, **CoinDenseVector**< double > &)
- void borrowDiag1 (double ∗array)
  diag1 - we just borrow as it is part of a CoinDenseVector<double>

Public Attributes

Public member data

- int nrows_
  Row dimension of matrix.
- int ncols_
  Column dimension of matrix.
- ClpInterior ∗ model_
  Pointer to Model object for this instance.
- double ∗ diag1_
  Diagonal array 1.
- double diag2_
  Constant diagonal 2.

4.49.1 Detailed Description

This class implements LSQR.

```
LSQR solves  Ax = b  or  min ||b - Ax||_2  if damp = 0,
or   min || (b)  -  (  A    )x ||   otherwise.
         || (0)      (damp I)  ||2
A  is an m by n matrix defined by user provided routines
matVecMult(mode, y, x)
which performs the matrix-vector operations where y and x
are references or pointers to CoinDenseVector objects.
If mode = 1, matVecMult  must return  y = Ax   without altering x.
If mode = 2, matVecMult  must return  y = A'x  without altering x.

-----------------------------------------------------------------------
LSQR uses an iterative (conjugate-gradient-like) method.
For further information, see
1. C. C. Paige and M. A. Saunders (1982a).
   LSQR: An algorithm for sparse linear equations and sparse least squares,
   ACM TOMS 8(1), 43-71.
2. C. C. Paige and M. A. Saunders (1982b).
   Algorithm 583.  LSQR: Sparse linear equations and least squares problems,
   ACM TOMS 8(2), 195-209.
3. M. A. Saunders (1995).  Solution of sparse rectangular systems using
   LSQR and CRAIG, BIT 35, 588-604.

Input parameters:

atol, btol  are stopping tolerances.  If both are 1.0e-9 (say),
            the final residual norm should be accurate to about 9 digits.
            (The final x will usually have fewer correct digits,
            depending on cond(A) and the size of damp.)
conlim      is also a stopping tolerance.  lsqr terminates if an estimate
            of cond(A) exceeds conlim.  For compatible systems Ax = b,
            conlim could be as large as 1.0e+12 (say).  For least-squares
            problems, conlim should be less than 1.0e+8.
            Maximum precision can be obtained by setting
            atol = btol = conlim = zero, but the number of iterations
            may then be excessive.
itnlim      is an explicit limit on iterations (for safety).
show = 1    gives an iteration log,
show = 0    suppresses output.
info        is a structure special to pdco.m, used to test if
            was small enough, and continuing if necessary with smaller atol.


Output parameters:
x           is the final solution.
*istop       gives the reason for termination.
*istop       = 1 means x is an approximate solution to Ax = b.
            = 2 means x approximately solves the least-squares problem.
rnorm       = norm(r) if damp = 0, where r = b - Ax,
            = sqrt( norm(r)**2  +  damp**2 * norm(x)**2 ) otherwise.
xnorm       = norm(x).
var         estimates diag( inv(A'A) ).  Omitted in this special version.
outfo       is a structure special to pdco.m, returning information
            about whether atol had to be reduced.

Other potential output parameters:
anorm, acond, arnorm, xnorm
```

Definition at line 76 of file ClpLsqr.hpp.

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

- ClpLsqr.hpp

4.50 ClpMatrixBase Class Reference

Abstract base class for Clp Matrices.

`#include <ClpMatrixBase.hpp>`

Inheritance diagram for ClpMatrixBase:

Public Member Functions

Virtual methods that the derived classes must provide

- virtual **CoinPackedMatrix** ∗ getPackedMatrix () const =0

  Return a complete **CoinPackedMatrix***.*
- virtual bool isColOrdered () const =0

  Whether the packed matrix is column major ordered or not.
- virtual CoinBigIndex getNumElements () const =0

  Number of entries in the packed matrix.
- virtual int getNumCols () const =0

  Number of columns.
- virtual int getNumRows () const =0

  Number of rows.
- virtual const double ∗ getElements () const =0

  A vector containing the elements in the packed matrix.
- virtual const int ∗ getIndices () const =0

  A vector containing the minor indices of the elements in the packed matrix.
- virtual const CoinBigIndex ∗ **getVectorStarts** () const =0
- virtual const int ∗ getVectorLengths () const =0

  The lengths of the major-dimension vectors.
- virtual int getVectorLength (int index) const

  The length of a single major-dimension vector.
- virtual void deleteCols (const int numDel, const int ∗indDel)=0

  Delete the columns whose indices are listed in `indDel`*.*
- virtual void deleteRows (const int numDel, const int ∗indDel)=0

  Delete the rows whose indices are listed in `indDel`*.*
- virtual void appendCols (int number, const **CoinPackedVectorBase** ∗const ∗columns)

  Append Columns.
- virtual void appendRows (int number, const **CoinPackedVectorBase** ∗const ∗rows)

  Append Rows.
- virtual void modifyCoefficient (int row, int column, double newElement, bool keepZero=false)

  Modify one element of packed matrix.
- virtual int appendMatrix (int number, int type, const CoinBigIndex ∗starts, const int ∗index, const double ∗element, int numberOther=-1)

  Append a set of rows/columns to the end of the matrix.
- virtual ClpMatrixBase ∗ reverseOrderedCopy () const

  Returns a new matrix in reverse order without gaps Is allowed to return NULL if doesn't want to have row copy.
- virtual CoinBigIndex countBasis (const int ∗whichColumn, int &numberColumnBasic)=0

  Returns number of elements in column part of basis.
- virtual void fillBasis (ClpSimplex ∗model, const int ∗whichColumn, int &numberColumnBasic, int ∗row, int ∗start, int ∗rowCount, int ∗columnCount, CoinFactorizationDouble ∗element)=0

  Fills in column part of basis.
- virtual int scale (ClpModel ∗, const ClpSimplex ∗=NULL) const

  Creates scales for column copy (rowCopy in model may be modified) default does not allow scaling returns non-zero if no scaling done.

- virtual void scaleRowCopy (ClpModel ∗) const

  Scales rowCopy if column copy scaled Only called if scales already exist.
- virtual bool canGetRowCopy () const

  Returns true if can create row copy.
- virtual ClpMatrixBase ∗ scaledColumnCopy (ClpModel ∗) const

  Realy really scales column copy Only called if scales already exist.
- virtual bool allElementsInRange (ClpModel ∗, double, double, int=15)

  Checks if all elements are in valid range.
- virtual void setDimensions (int numrows, int numcols)

  Set the dimensions of the matrix.
- virtual void rangeOfElements (double &smallestNegative, double &largestNegative, double &smallestPositive, double &largestPositive)

  Returns largest and smallest elements of both signs.
- virtual void unpack (const ClpSimplex ∗model, **CoinIndexedVector** ∗rowArray, int column) const =0

  Unpacks a column into an CoinIndexedvector.
- virtual void unpackPacked (ClpSimplex ∗model, **CoinIndexedVector** ∗rowArray, int column) const =0

  Unpacks a column into an CoinIndexedvector in packed format Note that model is NOT const.
- virtual int refresh (ClpSimplex ∗)

  Purely for column generation and similar ideas.
- virtual void **reallyScale** (const double ∗rowScale, const double ∗columnScale)
- virtual CoinBigIndex ∗ dubiousWeights (const ClpSimplex ∗model, int ∗inputWeights) const

  Given positive integer weights for each row fills in sum of weights for each column (and slack).
- virtual void add (const ClpSimplex ∗model, **CoinIndexedVector** ∗rowArray, int column, double multiplier) const =0

  Adds multiple of a column into an CoinIndexedvector You can use quickAdd to add to vector.
- virtual void add (const ClpSimplex ∗model, double ∗array, int column, double multiplier) const =0

  Adds multiple of a column into an array.
- virtual void releasePackedMatrix () const =0

  Allow any parts of a created **CoinPackedMatrix** *to be deleted.*
- virtual bool canDoPartialPricing () const

  Says whether it can do partial pricing.
- virtual int hiddenRows () const

  Returns number of hidden rows e.g. gub.
- virtual void partialPricing (ClpSimplex ∗model, double start, double end, int &bestSequence, int &number←Wanted)

  Partial pricing.
- virtual int extendUpdated (ClpSimplex ∗model, **CoinIndexedVector** ∗update, int mode)

  expands an updated column to allow for extra rows which the main solver does not know about and returns number added.
- virtual void primalExpanded (ClpSimplex ∗model, int mode)

  utility primal function for dealing with dynamic constraints mode=0 - Set up before "update" and "times" for primal solution using extended rows mode=1 - Cleanup primal solution after "times" using extended rows.
- virtual void dualExpanded (ClpSimplex ∗model, **CoinIndexedVector** ∗array, double ∗other, int mode)

  utility dual function for dealing with dynamic constraints mode=0 - Set up before "updateTranspose" and "transpose←Times" for duals using extended updates array (and may use other if dual values pass) mode=1 - Update dual solution after "transposeTimes" using extended rows.
- virtual int generalExpanded (ClpSimplex ∗model, int mode, int &number)

  general utility function for dealing with dynamic constraints mode=0 - Create list of non-key basics in pivotVariable_ using number as numberBasic in and out mode=1 - Set all key variables as basic mode=2 - return number extra rows needed, number gives maximum number basic mode=3 - before replaceColumn mode=4 - return 1 if can do primal, 2 if dual, 3 if both mode=5 - save any status stuff (when in good state) mode=6 - restore status stuff mode=7 - flag given variable (normally sequenceIn) mode=8 - unflag all variables mode=9 - synchronize costs and bounds mode=10 - return 1 if there may be changing bounds on variable (column generation) mode=11 - make sure set is clean (used when a variable rejected - but not flagged) mode=12 - after factorize but before permute stuff mode=13 - at end of simplex to delete stuff

- virtual int updatePivot (ClpSimplex ∗model, double oldInValue, double oldOutValue)

  update information for a pivot (and effective rhs)
- virtual void createVariable (ClpSimplex ∗model, int &bestSequence)

  Creates a variable.
- virtual int checkFeasible (ClpSimplex ∗model, double &sum) const

  Just for debug if odd type matrix.
- double reducedCost (ClpSimplex ∗model, int sequence) const

  Returns reduced cost of a variable.
- virtual void correctSequence (const ClpSimplex ∗model, int &sequenceIn, int &sequenceOut)

  Correct sequence in and out to give true value (if both -1 maybe do whole matrix)

Matrix times vector methods

They can be faster if scalar is +- 1 Also for simplex I am not using basic/non-basic split

- virtual void times (double scalar, const double ∗COIN_RESTRICT x, double ∗COIN_RESTRICT y) const =0

  Return $y + A * x * scalar$ *in* y.
- virtual void times (double scalar, const double ∗COIN_RESTRICT x, double ∗COIN_RESTRICT y, const double ∗COIN_RESTRICT rowScale, const double ∗COIN_RESTRICT columnScale) const

  And for scaling - default aborts for when scaling not supported (unless pointers NULL when as normal)
- virtual void transposeTimes (double scalar, const double ∗COIN_RESTRICT x, double ∗COIN_RESTRICT y) const =0

  Return $y + x * scalar * A$ *in* y.
- virtual void transposeTimes (double scalar, const double ∗COIN_RESTRICT x, double ∗COIN_RESTRICT y, const double ∗COIN_RESTRICT rowScale, const double ∗COIN_RESTRICT columnScale, double ∗COIN_↩ RESTRICT spare=NULL) const

  And for scaling - default aborts for when scaling not supported (unless pointers NULL when as normal)
- virtual void transposeTimes (const ClpSimplex ∗model, double scalar, const **CoinIndexedVector** ∗x, **Coin↩ IndexedVector** ∗y, **CoinIndexedVector** ∗z) const =0

  Return $x * scalar *A + y$ *in* z.
- virtual void subsetTransposeTimes (const ClpSimplex ∗model, const **CoinIndexedVector** ∗x, const **Coin↩ IndexedVector** ∗y, **CoinIndexedVector** ∗z) const =0

  Return $x *A$ *in* z *but just for indices in y.*
- virtual bool canCombine (const ClpSimplex ∗, const **CoinIndexedVector** ∗) const

  Returns true if can combine transposeTimes and subsetTransposeTimes and if it would be faster.
- virtual void transposeTimes2 (const ClpSimplex ∗model, const **CoinIndexedVector** ∗pi1, **CoinIndexedVector** ∗dj1, const **CoinIndexedVector** ∗pi2, **CoinIndexedVector** ∗spare, double referenceIn, double devex, unsigned int ∗reference, double ∗weights, double scaleFactor)

  Updates two arrays for steepest and does devex weights (need not be coded)
- virtual void subsetTimes2 (const ClpSimplex ∗model, **CoinIndexedVector** ∗dj1, const **CoinIndexedVector** ∗pi2, **CoinIndexedVector** ∗dj2, double referenceIn, double devex, unsigned int ∗reference, double ∗weights, double scaleFactor)

  Updates second array for steepest and does devex weights (need not be coded)
- virtual void listTransposeTimes (const ClpSimplex ∗model, double ∗x, int ∗y, int number, double ∗z) const

  Return $x *A$ *in* z *but just for number indices in y.*

Other

Clone

- virtual ClpMatrixBase ∗ **clone** () const =0
- virtual ClpMatrixBase ∗ subsetClone (int numberRows, const int ∗whichRows, int numberColumns, const int ∗whichColumns) const

  Subset clone (without gaps).
- virtual void backToBasics ()

Gets rid of any mutable by products.
- int type () const
  Returns type.
- void setType (int newtype)
  Sets type.
- void useEffectiveRhs (ClpSimplex ∗model)
  Sets up an effective RHS.
- virtual double ∗ rhsOffset (ClpSimplex ∗model, bool forceRefresh=false, bool check=false)
  Returns effective RHS offset if it is being used.
- int lastRefresh () const
  If rhsOffset used this is iteration last refreshed.
- int refreshFrequency () const
  If rhsOffset used this is refresh frequency (0==off)
- void **setRefreshFrequency** (int value)
- bool skipDualCheck () const
  whether to skip dual checks most of time
- void **setSkipDualCheck** (bool yes)
- int minimumObjectsScan () const
  Partial pricing tuning parameter - minimum number of "objects" to scan.
- void **setMinimumObjectsScan** (int value)
- int minimumGoodReducedCosts () const
  Partial pricing tuning parameter - minimum number of negative reduced costs to get.
- void **setMinimumGoodReducedCosts** (int value)
- double startFraction () const
  Current start of search space in matrix (as fraction)
- void **setStartFraction** (double value)
- double endFraction () const
  Current end of search space in matrix (as fraction)
- void **setEndFraction** (double value)
- double savedBestDj () const
  Current best reduced cost.
- void **setSavedBestDj** (double value)
- int originalWanted () const
  Initial number of negative reduced costs wanted.
- void **setOriginalWanted** (int value)
- int currentWanted () const
  Current number of negative reduced costs which we still need.
- void **setCurrentWanted** (int value)
- int savedBestSequence () const
  Current best sequence.
- void **setSavedBestSequence** (int value)

Protected Attributes

Data members

The data members are protected to allow access for derived classes.

- double ∗ rhsOffset_
  Effective RHS offset if it is being used.
- double startFraction_
  Current start of search space in matrix (as fraction)
- double endFraction_
  Current end of search space in matrix (as fraction)

- double savedBestDj_

  Best reduced cost so far.
- int originalWanted_

  Initial number of negative reduced costs wanted.
- int currentWanted_

  Current number of negative reduced costs which we still need.
- int savedBestSequence_

  Saved best sequence in pricing.
- int type_

  type (may be useful)
- int lastRefresh_

  If rhsOffset used this is iteration last refreshed.
- int refreshFrequency_

  If rhsOffset used this is refresh frequency (0==off)
- int minimumObjectsScan_

  Partial pricing tuning parameter - minimum number of "objects" to scan.
- int minimumGoodReducedCosts_

  Partial pricing tuning parameter - minimum number of negative reduced costs to get.
- int trueSequenceIn_

  True sequence in (i.e. from larger problem)
- int trueSequenceOut_

  True sequence out (i.e. from larger problem)
- bool skipDualCheck_

  whether to skip dual checks most of time

Constructors, destructor

NOTE: All constructors are protected.

There's no need to expose them, after all, this is an abstract class.

- virtual ∼ClpMatrixBase ()

  Destructor (has to be public)
- ClpMatrixBase ()

  Default constructor.
- **ClpMatrixBase** (const ClpMatrixBase &)
- ClpMatrixBase & **operator=** (const ClpMatrixBase &)

4.50.1 Detailed Description

Abstract base class for Clp Matrices.

Since this class is abstract, no object of this type can be created.

If a derived class provides all methods then all Clp algorithms should work. Some can be very inefficient e.g. get↩ Elements etc is only used for tightening bounds for dual and the copies are deleted. Many methods can just be dummy i.e. abort(); if not all features are being used. So if column generation was being done then it makes no sense to do steepest edge so there would be no point providing subsetTransposeTimes.

Definition at line 38 of file ClpMatrixBase.hpp.

4.50.2 Constructor & Destructor Documentation

4.50.2.1 ClpMatrixBase::ClpMatrixBase ( ) [protected]

Default constructor.

4.50.3 Member Function Documentation

4.50.3.1 virtual bool ClpMatrixBase::isColOrdered ( ) const [pure virtual]

Whether the packed matrix is column major ordered or not.

Implemented in ClpPackedMatrix, ClpDummyMatrix, ClpNetworkMatrix, and ClpPlusMinusOneMatrix.

4.50.3.2 virtual CoinBigIndex ClpMatrixBase::getNumElements ( ) const [pure virtual]

Number of entries in the packed matrix.

Implemented in ClpPackedMatrix, ClpDummyMatrix, ClpNetworkMatrix, and ClpPlusMinusOneMatrix.

4.50.3.3 virtual int ClpMatrixBase::getNumCols ( ) const [pure virtual]

Number of columns.

Implemented in ClpPackedMatrix, ClpDummyMatrix, ClpNetworkMatrix, and ClpPlusMinusOneMatrix.

4.50.3.4 virtual int ClpMatrixBase::getNumRows ( ) const [pure virtual]

Number of rows.

Implemented in ClpPackedMatrix, ClpDummyMatrix, ClpNetworkMatrix, and ClpPlusMinusOneMatrix.

4.50.3.5 virtual const double* ClpMatrixBase::getElements ( ) const [pure virtual]

A vector containing the elements in the packed matrix.

Note that there might be gaps in this list, entries that do not belong to any major-dimension vector. To get the actual elements one should look at this vector together with vectorStarts and vectorLengths.

Implemented in ClpPackedMatrix, ClpDummyMatrix, ClpNetworkMatrix, and ClpPlusMinusOneMatrix.

4.50.3.6 virtual const int* ClpMatrixBase::getIndices ( ) const [pure virtual]

A vector containing the minor indices of the elements in the packed matrix.

Note that there might be gaps in this list, entries that do not belong to any major-dimension vector. To get the actual elements one should look at this vector together with vectorStarts and vectorLengths.

Implemented in ClpPackedMatrix, ClpDummyMatrix, ClpNetworkMatrix, and ClpPlusMinusOneMatrix.

4.50.3.7 virtual const int* ClpMatrixBase::getVectorLengths ( ) const [pure virtual]

The lengths of the major-dimension vectors.

Implemented in ClpPackedMatrix, ClpNetworkMatrix, ClpDummyMatrix, and ClpPlusMinusOneMatrix.

4.50.3.8 virtual int ClpMatrixBase::getVectorLength (int *index*) const [virtual]

The length of a single major-dimension vector.

Reimplemented in ClpPackedMatrix.

4.50.3.9 virtual void ClpMatrixBase::deleteCols (const int *numDel*, const int * *indDel*) [pure virtual]

Delete the columns whose indices are listed in `indDel`.

Implemented in ClpPackedMatrix, ClpNetworkMatrix, ClpDummyMatrix, and ClpPlusMinusOneMatrix.

4.50.3.10 virtual void ClpMatrixBase::deleteRows (const int *numDel*, const int * *indDel*) [pure virtual]

Delete the rows whose indices are listed in `indDel`.

Implemented in ClpPackedMatrix, ClpNetworkMatrix, ClpDummyMatrix, and ClpPlusMinusOneMatrix.

4.50.3.11 virtual void ClpMatrixBase::modifyCoefficient (int *row*, int *column*, double *newElement*, bool *keepZero* = false) [virtual]

Modify one element of packed matrix.

An element may be added. This works for either ordering If the new element is zero it will be deleted unless keepZero true

Reimplemented in ClpPackedMatrix.

4.50.3.12 virtual int ClpMatrixBase::appendMatrix (int *number*, int *type*, const CoinBigIndex * *starts*, const int * *index*, const double * *element*, int *numberOther* = -1) [virtual]

Append a set of rows/columns to the end of the matrix.

Returns number of errors i.e. if any of the new rows/columns contain an index that's larger than the number of columns-1/rows-1 (if numberOther>0) or duplicates If 0 then rows, 1 if columns

Reimplemented in ClpPackedMatrix, ClpNetworkMatrix, and ClpPlusMinusOneMatrix.

4.50.3.13 virtual ClpMatrixBase* ClpMatrixBase::scaledColumnCopy (ClpModel *) const [inline], [virtual]

Realy really scales column copy Only called if scales already exist.

Up to user to delete

Reimplemented in ClpPackedMatrix.

Definition at line 126 of file ClpMatrixBase.hpp.

4.50.3.14 virtual bool ClpMatrixBase::allElementsInRange (ClpModel * , double , double , int = 15) [inline], [virtual]

Checks if all elements are in valid range.

Can just return true if you are not paranoid. For Clp I will probably expect no zeros. Code can modify matrix to get rid of small elements. check bits (can be turned off to save time) : 1 - check if matrix has gaps 2 - check if zero elements 4 - check and compress duplicates 8 - report on large and small

Reimplemented in ClpPackedMatrix.

Definition at line 140 of file ClpMatrixBase.hpp.

4.50.3.15 virtual void ClpMatrixBase::setDimensions (int *numrows*, int *numcols*) [virtual]

Set the dimensions of the matrix.

In effect, append new empty columns/rows to the matrix. A negative number for either dimension means that that dimension doesn't change. Otherwise the new dimensions MUST be at least as large as the current ones otherwise an exception is thrown.

Reimplemented in ClpPackedMatrix, and ClpPlusMinusOneMatrix.

4.50.3.16 virtual void ClpMatrixBase::rangeOfElements (double & *smallestNegative*, double & *largestNegative*, double & *smallestPositive*, double & *largestPositive*) [virtual]

Returns largest and smallest elements of both signs.

Largest refers to largest absolute value. If returns zeros then can't tell anything

Reimplemented in ClpPackedMatrix, ClpNetworkMatrix, and ClpPlusMinusOneMatrix.

4.50.3.17 virtual void ClpMatrixBase::unpackPacked (ClpSimplex ∗ *model*, CoinIndexedVector ∗ *rowArray*, int *column*) const [pure virtual]

Unpacks a column into an CoinIndexedvector in packed format Note that model is NOT const.

Bounds and objective could be modified if doing column generation (just for this variable)

Implemented in ClpPackedMatrix, ClpNetworkMatrix, ClpPlusMinusOneMatrix, ClpDummyMatrix, and ClpGubMatrix.

4.50.3.18 virtual int ClpMatrixBase::refresh (ClpSimplex ∗) [inline], [virtual]

Purely for column generation and similar ideas.

Allows matrix and any bounds or costs to be updated (sensibly). Returns non-zero if any changes.

Reimplemented in ClpPackedMatrix, and ClpDynamicMatrix.

Definition at line 172 of file ClpMatrixBase.hpp.

4.50.3.19 virtual CoinBigIndex∗ ClpMatrixBase::dubiousWeights (const ClpSimplex ∗ *model*, int ∗ *inputWeights*) const [virtual]

Given positive integer weights for each row fills in sum of weights for each column (and slack).

Returns weights vector Default returns vector of ones

Reimplemented in ClpPackedMatrix, ClpNetworkMatrix, and ClpPlusMinusOneMatrix.

4.50.3.20 virtual int ClpMatrixBase::extendUpdated (ClpSimplex ∗ *model,* CoinIndexedVector ∗ *update,* int *mode*) [virtual]

expands an updated column to allow for extra rows which the main solver does not know about and returns number added.

This will normally be a no-op - it is in for GUB but may get extended to general non-overlapping and embedded networks.

mode 0 - extend mode 1 - delete etc

Reimplemented in ClpGubMatrix.

4.50.3.21 virtual void ClpMatrixBase::primalExpanded (ClpSimplex ∗ *model,* int *mode*) [virtual]

utility primal function for dealing with dynamic constraints mode=0 - Set up before "update" and "times" for primal solution using extended rows mode=1 - Cleanup primal solution after "times" using extended rows.

mode=2 - Check (or report on) primal infeasibilities

Reimplemented in ClpGubMatrix.

4.50.3.22 virtual void ClpMatrixBase::dualExpanded (ClpSimplex ∗ *model,* CoinIndexedVector ∗ *array,* double ∗ *other,* int *mode*) [virtual]

utility dual function for dealing with dynamic constraints mode=0 - Set up before "updateTranspose" and "transpose←Times" for duals using extended updates array (and may use other if dual values pass) mode=1 - Update dual solution after "transposeTimes" using extended rows.

mode=2 - Compute all djs and compute key dual infeasibilities mode=3 - Report on key dual infeasibilities mode=4 - Modify before updateTranspose in partial pricing

Reimplemented in ClpGubMatrix, and ClpDynamicMatrix.

4.50.3.23 virtual int ClpMatrixBase::generalExpanded (ClpSimplex ∗ *model,* int *mode,* int & *number*) [virtual]

general utility function for dealing with dynamic constraints mode=0 - Create list of non-key basics in pivotVariable_ using number as numberBasic in and out mode=1 - Set all key variables as basic mode=2 - return number extra rows needed, number gives maximum number basic mode=3 - before replaceColumn mode=4 - return 1 if can do primal, 2 if dual, 3 if both mode=5 - save any status stuff (when in good state) mode=6 - restore status stuff mode=7 - flag given variable (normally sequenceIn) mode=8 - unflag all variables mode=9 - synchronize costs and bounds mode=10 - return 1 if there may be changing bounds on variable (column generation) mode=11 - make sure set is clean (used when a variable rejected - but not flagged) mode=12 - after factorize but before permute stuff mode=13 - at end of simplex to delete stuff

Reimplemented in ClpGubMatrix, and ClpDynamicMatrix.

4.50.3.24 virtual void ClpMatrixBase::createVariable (ClpSimplex ∗ *model,* int & *bestSequence*) [virtual]

Creates a variable.

This is called after partial pricing and may modify matrix. May update bestSequence.

Reimplemented in ClpDynamicMatrix, and ClpDynamicExampleMatrix.

4.50.3.25 virtual int ClpMatrixBase::checkFeasible (**ClpSimplex** ∗ *model,* double & *sum*) const [virtual]

Just for debug if odd type matrix.

Returns number of primal infeasibilities.

Reimplemented in ClpGubDynamicMatrix.

4.50.3.26 virtual void ClpMatrixBase::times (double *scalar,* const double ∗COIN_RESTRICT *x,* double ∗COIN_RESTRICT *y*) const [pure virtual]

Return `y + A * x * scalar` in `y`.

Precondition

`x` must be of size `numColumns()`
`y` must be of size `numRows()`

4.50.3.27 virtual void ClpMatrixBase::transposeTimes (double *scalar,* const double ∗COIN_RESTRICT *x,* double ∗COIN_RESTRICT *y*) const [pure virtual]

Return `y + x * scalar * A` in `y`.

Precondition

`x` must be of size `numRows()`
`y` must be of size `numColumns()`

4.50.3.28 virtual void ClpMatrixBase::transposeTimes (const **ClpSimplex** ∗ *model,* double *scalar,* const **CoinIndexedVector** ∗ *x,* **CoinIndexedVector** ∗ *y,* **CoinIndexedVector** ∗ *z*) const [pure virtual]

Return `x * scalar *A + y` in `z`.

Can use y as temporary array (will be empty at end) Note - If x packed mode - then z packed mode Squashes small elements and knows about ClpSimplex

Implemented in ClpPackedMatrix, ClpPlusMinusOneMatrix, ClpNetworkMatrix, ClpDummyMatrix, and ClpGubMatrix.

4.50.3.29 virtual void ClpMatrixBase::subsetTransposeTimes (const **ClpSimplex** ∗ *model,* const **CoinIndexedVector** ∗ *x,* const **CoinIndexedVector** ∗ *y,* **CoinIndexedVector** ∗ *z*) const [pure virtual]

Return `x *A` in `z` but just for indices in y.

This is only needed for primal steepest edge. Note - z always packed mode

Implemented in ClpPackedMatrix, ClpPlusMinusOneMatrix, ClpNetworkMatrix, ClpDummyMatrix, and ClpGubMatrix.

4.50.3.30 virtual void ClpMatrixBase::listTransposeTimes (const **ClpSimplex** ∗ *model,* double ∗ *x,* int ∗ *y,* int *number,* double ∗ *z*) const [virtual]

Return `x *A` in `z` but just for number indices in y.

Default cheats with fake **CoinIndexedVector** and then calls subsetTransposeTimes

4.50.3.31 virtual ClpMatrixBase* ClpMatrixBase::subsetClone (int *numberRows*, const int * *whichRows*, int *numberColumns*, const int * *whichColumns*) const [virtual]

Subset clone (without gaps).

Duplicates are allowed and order is as given. Derived classes need not provide this as it may not always make sense

Reimplemented in ClpPackedMatrix, ClpPlusMinusOneMatrix, ClpNetworkMatrix, and ClpGubMatrix.

4.50.3.32 int ClpMatrixBase::type ( ) const [inline]

Returns type.

The types which code may need to know about are: 1 - ClpPackedMatrix 11 - ClpNetworkMatrix 12 - ClpPlusMinus↩OneMatrix

Definition at line 370 of file ClpMatrixBase.hpp.

4.50.3.33 virtual double* ClpMatrixBase::rhsOffset (ClpSimplex * *model*, bool *forceRefresh* = false, bool *check* = false) [virtual]

Returns effective RHS offset if it is being used.

This is used for long problems or big gub or anywhere where going through full columns is expensive. This may re-compute

Reimplemented in ClpGubMatrix, ClpGubDynamicMatrix, and ClpDynamicMatrix.

4.50.3.34 int ClpMatrixBase::minimumObjectsScan ( ) const [inline]

Partial pricing tuning parameter - minimum number of "objects" to scan.

e.g. number of Gub sets but could be number of variables

Definition at line 404 of file ClpMatrixBase.hpp.

4.50.4 Member Data Documentation

4.50.4.1 double* ClpMatrixBase::rhsOffset_ [protected]

Effective RHS offset if it is being used.

This is used for long problems or big gub or anywhere where going through full columns is expensive

Definition at line 488 of file ClpMatrixBase.hpp.

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

- ClpMatrixBase.hpp

4.51 ClpMessage Class Reference

This deals with Clp messages (as against Osi messages etc)

`#include <ClpMessage.hpp>`

Inheritance diagram for ClpMessage:

Collaboration diagram for ClpMessage:

Public Member Functions

Constructors etc

- ClpMessage (**Language language**=us_en)

  Constructor.

4.51.1 Detailed Description

This deals with Clp messages (as against Osi messages etc)

Definition at line 119 of file ClpMessage.hpp.

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

- ClpMessage.hpp

4.52 ClpModel Class Reference

Inheritance diagram for ClpModel:

Collaboration diagram for ClpModel:

Public Member Functions

- const double ∗ rowScale () const

  Scaling.
- double objectiveScale () const

  Scaling of objective.
- double rhsScale () const

  Scaling of rhs and bounds.
- void scaling (int mode=1)

  Sets or unsets scaling, 0 -off, 1 equilibrium, 2 geometric, 3 auto, 4 auto-but-as-initialSolve-in-bab.
- void unscale ()

  If we constructed a "really" scaled model then this reverses the operation.
- int scalingFlag () const

  Gets scalingFlag.
- double ∗ objective () const

  Objective.
- double ∗ rowObjective () const

  Row Objective.
- double ∗ columnLower () const

  Column Lower.
- double ∗ columnUpper () const

  Column Upper.

- **CoinPackedMatrix** ∗ matrix () const

  Matrix (if not ClpPackedmatrix be careful about memory leak.
- int getNumElements () const

  Number of elements in matrix.
- double getSmallElementValue () const

  Small element value - elements less than this set to zero, default is 1.0e-20.
- ClpMatrixBase ∗ rowCopy () const

  Row Matrix.
- void setNewRowCopy (ClpMatrixBase ∗newCopy)

  Set new row matrix.
- ClpMatrixBase ∗ clpMatrix () const

  Clp Matrix.
- ClpPackedMatrix ∗ clpScaledMatrix () const

  Scaled ClpPackedMatrix.
- void setClpScaledMatrix (ClpPackedMatrix ∗scaledMatrix)

  Sets pointer to scaled ClpPackedMatrix.
- ClpPackedMatrix ∗ swapScaledMatrix (ClpPackedMatrix ∗scaledMatrix)

  Swaps pointer to scaled ClpPackedMatrix.
- void replaceMatrix (ClpMatrixBase ∗matrix, bool deleteCurrent=false)

  Replace Clp Matrix (current is not deleted unless told to and new is used) So up to user to delete current.
- void replaceMatrix (**CoinPackedMatrix** ∗newmatrix, bool deleteCurrent=false)

  Replace Clp Matrix (current is not deleted unless told to and new is used) So up to user to delete current.
- double objectiveValue () const

  Objective value.
- char ∗ integerInformation () const

  Integer information.
- double ∗ infeasibilityRay (bool fullRay=false) const

  Infeasibility/unbounded ray (NULL returned if none/wrong) Up to user to use delete [] on these arrays.
- double ∗ ray () const

  For advanced users - no need to delete - sign not changed.
- bool rayExists () const

  just test if infeasibility or unbounded Ray exists
- void deleteRay ()

  just delete ray if exists
- const double ∗ internalRay () const

  Access internal ray storage. Users should call infeasibilityRay() or unboundedRay() instead.
- bool statusExists () const

  See if status (i.e. basis) array exists (partly for OsiClp)
- unsigned char ∗ statusArray () const

  Return address of status (i.e. basis) array (char[numberRows+numberColumns])
- unsigned char ∗ statusCopy () const

  Return copy of status (i.e.
- void copyinStatus (const unsigned char ∗statusArray)

  Copy in status (basis) vector.
- void setUserPointer (void ∗pointer)

  User pointer for whatever reason.
- void setTrustedUserPointer (ClpTrustedData ∗pointer)

Trusted user pointer.

- int whatsChanged () const

  What has changed in model (only for masochistic users)
- int numberThreads () const

  Number of threads (not really being used)

Constructors and destructor

Note - copy methods copy ALL data so can chew up memory until other copy is freed

- ClpModel (bool emptyMessages=false)

  Default constructor.
- ClpModel (const ClpModel &rhs, int scalingMode=-1)

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

  Assignment operator. This copies the data.
- ClpModel (const ClpModel ∗wholeModel, int numberRows, const int ∗whichRows, int numberColumns, const int ∗whichColumns, bool dropNames=true, bool dropIntegers=true)

  Subproblem constructor.
- ∼ClpModel ()

  Destructor.

Load model - loads some stuff and initializes others

- void loadProblem (const ClpMatrixBase &matrix, const double ∗collb, const double ∗colub, const double ∗obj, const double ∗rowlb, const double ∗rowub, const double ∗rowObjective=NULL)

  Loads a problem (the constraints on the rows are given by lower and upper bounds).
- void **loadProblem** (const **CoinPackedMatrix** &matrix, const double ∗collb, const double ∗colub, const double ∗obj, const double ∗rowlb, const double ∗rowub, const double ∗rowObjective=NULL)
- void loadProblem (const int numcols, const int numrows, const CoinBigIndex ∗start, const int ∗index, const double ∗value, const double ∗collb, const double ∗colub, const double ∗obj, const double ∗rowlb, const double ∗rowub, const double ∗rowObjective=NULL)

  Just like the other loadProblem() method except that the matrix is given in a standard column major ordered format (without gaps).
- int loadProblem (**CoinModel** &modelObject, bool tryPlusMinusOne=false)

  This loads a model from a coinModel object - returns number of errors.
- void loadProblem (const int numcols, const int numrows, const CoinBigIndex ∗start, const int ∗index, const double ∗value, const int ∗length, const double ∗collb, const double ∗colub, const double ∗obj, const double ∗rowlb, const double ∗rowub, const double ∗rowObjective=NULL)

  This one is for after presolve to save memory.
- void loadQuadraticObjective (const int numberColumns, const CoinBigIndex ∗start, const int ∗column, const double ∗element)

  Load up quadratic objective.
- void **loadQuadraticObjective** (const **CoinPackedMatrix** &matrix)
- void deleteQuadraticObjective ()

  Get rid of quadratic objective.
- void setRowObjective (const double ∗rowObjective)

  This just loads up a row objective.
- int readMps (const char ∗filename, bool keepNames=false, bool ignoreErrors=false)

  Read an mps file from the given filename.
- int readGMPL (const char ∗filename, const char ∗dataName, bool keepNames=false)

  Read GMPL files from the given filenames.
- void copyInIntegerInformation (const char ∗information)

  Copy in integer informations.

- void deleteIntegerInformation ()
  Drop integer informations.
- void setContinuous (int index)
  Set the index-th variable to be a continuous variable.
- void setInteger (int index)
  Set the index-th variable to be an integer variable.
- bool isInteger (int index) const
  Return true if the index-th variable is an integer variable.
- void resize (int newNumberRows, int newNumberColumns)
  Resizes rim part of model.
- void deleteRows (int number, const int ∗which)
  Deletes rows.
- void addRow (int numberInRow, const int ∗columns, const double ∗elements, double rowLower=-COIN_DB←L_MAX, double rowUpper=COIN_DBL_MAX)
  Add one row.
- void addRows (int number, const double ∗rowLower, const double ∗rowUpper, const CoinBigIndex ∗rowStarts, const int ∗columns, const double ∗elements)
  Add rows.
- void addRows (int number, const double ∗rowLower, const double ∗rowUpper, const CoinBigIndex ∗rowStarts, const int ∗rowLengths, const int ∗columns, const double ∗elements)
  Add rows.
- void **addRows** (int number, const double ∗rowLower, const double ∗rowUpper, const **CoinPackedVectorBase** ∗const ∗rows)
- int addRows (const **CoinBuild** &buildObject, bool tryPlusMinusOne=false, bool checkDuplicates=true)
  Add rows from a build object.
- int addRows (**CoinModel** &modelObject, bool tryPlusMinusOne=false, bool checkDuplicates=true)
  Add rows from a model object.
- void deleteColumns (int number, const int ∗which)
  Deletes columns.
- void deleteRowsAndColumns (int numberRows, const int ∗whichRows, int numberColumns, const int ∗which←Columns)
  Deletes rows AND columns (keeps old sizes)
- void addColumn (int numberInColumn, const int ∗rows, const double ∗elements, double columnLower=0.0, double columnUpper=COIN_DBL_MAX, double objective=0.0)
  Add one column.
- void addColumns (int number, const double ∗columnLower, const double ∗columnUpper, const double ∗objective, const CoinBigIndex ∗columnStarts, const int ∗rows, const double ∗elements)
  Add columns.
- void **addColumns** (int number, const double ∗columnLower, const double ∗columnUpper, const double ∗objective, const CoinBigIndex ∗columnStarts, const int ∗columnLengths, const int ∗rows, const double ∗elements)
- void **addColumns** (int number, const double ∗columnLower, const double ∗columnUpper, const double ∗objective, const **CoinPackedVectorBase** ∗const ∗columns)
- int addColumns (const **CoinBuild** &buildObject, bool tryPlusMinusOne=false, bool checkDuplicates=true)
  Add columns from a build object If tryPlusMinusOne then will try adding as +-1 matrix if no matrix exists.
- int addColumns (**CoinModel** &modelObject, bool tryPlusMinusOne=false, bool checkDuplicates=true)
  Add columns from a model object.
- void modifyCoefficient (int row, int column, double newElement, bool keepZero=false)
  Modify one element of a matrix.
- void chgRowLower (const double ∗rowLower)
  Change row lower bounds.
- void chgRowUpper (const double ∗rowUpper)
  Change row upper bounds.

- void chgColumnLower (const double ∗columnLower)

  Change column lower bounds.
- void chgColumnUpper (const double ∗columnUpper)

  Change column upper bounds.
- void chgObjCoefficients (const double ∗objIn)

  Change objective coefficients.
- void borrowModel (ClpModel &otherModel)

  Borrow model.
- void returnModel (ClpModel &otherModel)

  Return model - nulls all arrays so can be deleted safely also updates any scalars.
- void createEmptyMatrix ()

  Create empty ClpPackedMatrix.
- int cleanMatrix (double threshold=1.0e-20)

  Really clean up matrix (if ClpPackedMatrix).
- void copy (const ClpMatrixBase ∗from, ClpMatrixBase ∗&to)

  Copy contents - resizing if necessary - otherwise re-use memory.
- void dropNames ()

  Drops names - makes lengthnames 0 and names empty.
- void copyNames (const std::vector< std::string > &rowNames, const std::vector< std::string > &column←Names)

  Copies in names.
- void copyRowNames (const std::vector< std::string > &rowNames, int first, int last)

  Copies in Row names - modifies names first .. last-1.
- void copyColumnNames (const std::vector< std::string > &columnNames, int first, int last)

  Copies in Column names - modifies names first .. last-1.
- void copyRowNames (const char ∗const ∗rowNames, int first, int last)

  Copies in Row names - modifies names first .. last-1.
- void copyColumnNames (const char ∗const ∗columnNames, int first, int last)

  Copies in Column names - modifies names first .. last-1.
- void setRowName (int rowIndex, std::string &name)

  Set name of row.
- void setColumnName (int colIndex, std::string &name)

  Set name of col.
- int findNetwork (char ∗rotate, double fractionNeeded=0.75)

  Find a network subset.
- **CoinModel** ∗ createCoinModel () const

  This creates a coinModel object.
- int writeMps (const char ∗filename, int formatType=0, int numberAcross=2, double objSense=0.0) const

  Write the problem in MPS format to the specified file.

gets and sets

- int numberRows () const

  Number of rows.
- int **getNumRows** () const
- int getNumCols () const

  Number of columns.
- int **numberColumns** () const
- double primalTolerance () const

  Primal tolerance to use.
- void **setPrimalTolerance** (double value)
- double dualTolerance () const

Dual tolerance to use.
- void **setDualTolerance** (double value)
- double primalObjectiveLimit () const

  Primal objective limit.
- void **setPrimalObjectiveLimit** (double value)
- double dualObjectiveLimit () const

  Dual objective limit.
- void **setDualObjectiveLimit** (double value)
- double objectiveOffset () const

  Objective offset.
- void **setObjectiveOffset** (double value)
- double presolveTolerance () const

  Presolve tolerance to use.
- const std::string & **problemName** () const
- int numberIterations () const

  Number of iterations.
- int **getIterationCount** () const
- void **setNumberIterations** (int numberIterationsNew)
- int solveType () const

  Solve type - 1 simplex, 2 simplex interface, 3 Interior.
- void **setSolveType** (int type)
- int maximumIterations () const

  Maximum number of iterations.
- void **setMaximumIterations** (int value)
- double maximumSeconds () const

  Maximum time in seconds (from when set called)
- void **setMaximumSeconds** (double value)
- void **setMaximumWallSeconds** (double value)
- bool hitMaximumIterations () const

  Returns true if hit maximum iterations (or time)
- int status () const

  Status of problem: -1 - unknown e.g.
- int **problemStatus** () const
- void setProblemStatus (int problemStatusNew)

  Set problem status.
- int secondaryStatus () const

  Secondary status of problem - may get extended 0 - none 1 - primal infeasible because dual limit reached OR (probably primal infeasible but can't prove it - main status was 4) 2 - scaled problem optimal - unscaled problem has primal infeasibilities 3 - scaled problem optimal - unscaled problem has dual infeasibilities 4 - scaled problem optimal - unscaled problem has primal and dual infeasibilities 5 - giving up in primal with flagged variables 6 - failed due to empty problem check 7 - postSolve says not optimal 8 - failed due to bad element check 9 - status was 3 and stopped on time 10 - status was 3 but stopped as primal feasible 100 up - translation of enum from ClpEventHandler.
- void **setSecondaryStatus** (int newstatus)
- bool isAbandoned () const

  Are there a numerical difficulties?
- bool isProvenOptimal () const

  Is optimality proven?
- bool isProvenPrimalInfeasible () const

  Is primal infeasiblity proven?
- bool isProvenDualInfeasible () const

  Is dual infeasiblity proven?
- bool isPrimalObjectiveLimitReached () const

  Is the given primal objective limit reached?
- bool isDualObjectiveLimitReached () const

Is the given dual objective limit reached?

- bool isIterationLimitReached () const

  Iteration limit reached?

- double optimizationDirection () const

  Direction of optimization (1 - minimize, -1 - maximize, 0 - ignore.

- double **getObjSense** () const
- void **setOptimizationDirection** (double value)
- double ∗ primalRowSolution () const

  Primal row solution.

- const double ∗ **getRowActivity** () const
- double ∗ primalColumnSolution () const

  Primal column solution.

- const double ∗ **getColSolution** () const
- void **setColSolution** (const double ∗input)
- double ∗ dualRowSolution () const

  Dual row solution.

- const double ∗ **getRowPrice** () const
- double ∗ dualColumnSolution () const

  Reduced costs.

- const double ∗ **getReducedCost** () const
- double ∗ rowLower () const

  Row lower.

- const double ∗ **getRowLower** () const
- double ∗ rowUpper () const

  Row upper.

- const double ∗ **getRowUpper** () const

Changing bounds on variables and constraints

- void setObjectiveCoefficient (int elementIndex, double elementValue)

  Set an objective function coefficient.

- void setObjCoeff (int elementIndex, double elementValue)

  Set an objective function coefficient.

- void setColumnLower (int elementIndex, double elementValue)

  Set a single column lower bound
  Use -DBL_MAX for -infinity.

- void setColumnUpper (int elementIndex, double elementValue)

  Set a single column upper bound
  Use DBL_MAX for infinity.

- void setColumnBounds (int elementIndex, double lower, double upper)

  Set a single column lower and upper bound.

- void setColumnSetBounds (const int ∗indexFirst, const int ∗indexLast, const double ∗boundList)

  Set the bounds on a number of columns simultaneously
  The default implementation just invokes setColLower() and setColUpper() over and over again.

- void setColLower (int elementIndex, double elementValue)

  Set a single column lower bound
  Use -DBL_MAX for -infinity.

- void setColUpper (int elementIndex, double elementValue)

  Set a single column upper bound
  Use DBL_MAX for infinity.

- void setColBounds (int elementIndex, double lower, double upper)

  Set a single column lower and upper bound.

- void setColSetBounds (const int ∗indexFirst, const int ∗indexLast, const double ∗boundList)

Set the bounds on a number of columns simultaneously

- void setRowLower (int elementIndex, double elementValue)

  Set a single row lower bound
  Use -DBL_MAX for -infinity.
- void setRowUpper (int elementIndex, double elementValue)

  Set a single row upper bound
  Use DBL_MAX for infinity.
- void setRowBounds (int elementIndex, double lower, double upper)

  Set a single row lower and upper bound.
- void setRowSetBounds (const int ∗indexFirst, const int ∗indexLast, const double ∗boundList)

  Set the bounds on a number of rows simultaneously

Message handling

- void passInMessageHandler (**CoinMessageHandler** ∗handler)

  Pass in Message handler (not deleted at end)
- **CoinMessageHandler** ∗ pushMessageHandler (**CoinMessageHandler** ∗handler, bool &oldDefault)

  Pass in Message handler (not deleted at end) and return current.
- void popMessageHandler (**CoinMessageHandler** ∗oldHandler, bool oldDefault)

  back to previous message handler
- void newLanguage (**CoinMessages::Language** language)

  Set language.
- void **setLanguage** (**CoinMessages::Language** language)
- void setDefaultMessageHandler ()

  Overrides message handler with a default one.
- **CoinMessageHandler** ∗ messageHandler () const

  Return handler.
- **CoinMessages** messages () const

  Return messages.
- **CoinMessages** ∗ messagesPointer ()

  Return pointer to messages.
- **CoinMessages** coinMessages () const

  Return **Coin** *messages.*
- **CoinMessages** ∗ coinMessagesPointer ()

  Return pointer to **Coin** *messages.*
- void setLogLevel (int value)

  Amount of print out: 0 - none 1 - just final 2 - just factorizations 3 - as 2 plus a bit more 4 - verbose above that 8,16,32 etc just for selective debug.
- int **logLevel** () const
- bool defaultHandler () const

  Return true if default handler.
- void passInEventHandler (const ClpEventHandler ∗eventHandler)

  Pass in Event handler (cloned and deleted at end)
- ClpEventHandler ∗ eventHandler () const

  Event handler.
- **CoinThreadRandom** ∗ randomNumberGenerator ()

  Thread specific random number generator.
- **CoinThreadRandom** & mutableRandomNumberGenerator ()

  Thread specific random number generator.
- void setRandomSeed (int value)

  Set seed for thread specific random number generator.

- int lengthNames () const
  length of names (0 means no names0
- void setLengthNames (int value)
  length of names (0 means no names0
- const std::vector< std::string > * rowNames () const
  Row names.
- const std::string & **rowName** (int iRow) const
- std::string getRowName (int iRow) const
  Return name or Rnnnnnnn.
- const std::vector< std::string > * columnNames () const
  Column names.
- const std::string & **columnName** (int iColumn) const
- std::string getColumnName (int iColumn) const
  Return name or Cnnnnnnn.
- ClpObjective * objectiveAsObject () const
  Objective methods.
- void **setObjective** (ClpObjective *objective)
- void **setObjectivePointer** (ClpObjective *newobjective)
- int emptyProblem (int *infeasNumber=NULL, double *infeasSum=NULL, bool printMessage=true)
  Solve a problem with no elements - return status and dual and primal infeasibilites.

Matrix times vector methods

They can be faster if scalar is +- 1 These are covers so user need not worry about scaling Also for simplex I am not using basic/non-basic split

- void times (double scalar, const double *x, double *y) const
  Return $y + A * x * scalar$ *in* y*.*
- void transposeTimes (double scalar, const double *x, double *y) const
  Return $y + x * scalar * A$ *in* y*.*

Parameter set/get methods

The set methods return true if the parameter was set to the given value, false otherwise.

There can be various reasons for failure: the given parameter is not applicable for the solver (e.g., refactorization frequency for the volume algorithm), the parameter is not yet implemented for the solver or simply the value of the parameter is out of the range the solver accepts. If a parameter setting call returns false check the details of your solver.

The get methods return true if the given parameter is applicable for the solver and is implemented. In this case the value of the parameter is returned in the second argument. Otherwise they return false.

once it has been decided where solver sits this may be redone

- bool setIntParam (ClpIntParam key, int value)
  Set an integer parameter.
- bool setDblParam (ClpDblParam key, double value)
  Set an double parameter.
- bool setStrParam (ClpStrParam key, const std::string &value)
  Set an string parameter.
- bool **getIntParam** (ClpIntParam key, int &value) const
- bool **getDblParam** (ClpDblParam key, double &value) const
- bool **getStrParam** (ClpStrParam key, std::string &value) const
- void generateCpp (FILE *fp)
  Create C++ lines to get to current state.
- unsigned int specialOptions () const
  For advanced options 1 - Don't keep changing infeasibility weight 2 - Keep nonLinearCost round solves 4 - Force outgoing variables to exact bound (primal) 8 - Safe to use dense initial factorization 16 -Just use basic variables for operation if column generation 32 -Create ray even in BAB 64 -Treat problem as feasible until last minute (i.e.
- void **setSpecialOptions** (unsigned int value)
- bool **inCbcBranchAndBound** () const

Protected Member Functions

private or protected methods

- void gutsOfDelete (int type)

  Does most of deletion (0 = all, 1 = most)
- void gutsOfCopy (const ClpModel &rhs, int trueCopy=1)

  Does most of copying If trueCopy 0 then just points to arrays If -1 leaves as much as possible.
- void getRowBound (int iRow, double &lower, double &upper) const

  gets lower and upper bounds on rows
- void gutsOfLoadModel (int numberRows, int numberColumns, const double ∗collb, const double ∗colub, const double ∗obj, const double ∗rowlb, const double ∗rowub, const double ∗rowObjective=NULL)

  puts in format I like - 4 array matrix - may make row copy
- void gutsOfScaling ()

  Does much of scaling.
- double rawObjectiveValue () const

  Objective value - always minimize.
- bool permanentArrays () const

  If we are using maximumRows_ and Columns_.
- void startPermanentArrays ()

  Start using maximumRows_ and Columns_.
- void stopPermanentArrays ()

  Stop using maximumRows_ and Columns_.
- const char ∗const ∗ rowNamesAsChar () const

  Create row names as char ∗∗.
- const char ∗const ∗ columnNamesAsChar () const

  Create column names as char ∗∗.
- void deleteNamesAsChar (const char ∗const ∗names, int number) const

  Delete char ∗ version of names.
- void onStopped ()

  On stopped - sets secondary status.

Protected Attributes

data

- double optimizationDirection_

  Direction of optimization (1 - minimize, -1 - maximize, 0 - ignore.
- double dblParam_ [ClpLastDblParam]

  Array of double parameters.
- double objectiveValue_

  Objective value.
- double smallElement_

  Small element value.
- double objectiveScale_

  Scaling of objective.
- double rhsScale_

  Scaling of rhs and bounds.
- int numberRows_

  Number of rows.
- int numberColumns_

  Number of columns.

- double ∗ rowActivity_

  Row activities.
- double ∗ columnActivity_

  Column activities.
- double ∗ dual_

  Duals.
- double ∗ reducedCost_

  Reduced costs.
- double ∗ rowLower_

  Row lower.
- double ∗ rowUpper_

  Row upper.
- ClpObjective ∗ objective_

  Objective.
- double ∗ rowObjective_

  Row Objective (? sign) - may be NULL.
- double ∗ columnLower_

  Column Lower.
- double ∗ columnUpper_

  Column Upper.
- ClpMatrixBase ∗ matrix_

  Packed matrix.
- ClpMatrixBase ∗ rowCopy_

  Row copy if wanted.
- ClpPackedMatrix ∗ scaledMatrix_

  Scaled packed matrix.
- double ∗ ray_

  Infeasible/unbounded ray.
- double ∗ rowScale_

  Row scale factors for matrix.
- double ∗ columnScale_

  Column scale factors.
- double ∗ inverseRowScale_

  Inverse row scale factors for matrix (end of rowScale_)
- double ∗ inverseColumnScale_

  Inverse column scale factors for matrix (end of columnScale_)
- int scalingFlag_

  Scale flag, 0 none, 1 equilibrium, 2 geometric, 3, auto, 4 dynamic, 5 geometric on rows.
- unsigned char ∗ status_

  Status (i.e.
- char ∗ integerType_

  Integer information.
- void ∗ userPointer_

  User pointer for whatever reason.
- ClpTrustedData ∗ trustedUserPointer_

  Trusted user pointer e.g. for heuristics.
- int intParam_ [ClpLastIntParam]

  Array of integer parameters.
- int numberIterations_

  Number of iterations.
- int solveType_

Solve type - 1 simplex, 2 simplex interface, 3 Interior.
- unsigned int **whatsChanged_**
- int problemStatus_

  Status of problem.
- int secondaryStatus_

  Secondary status of problem.
- int lengthNames_

  length of names (0 means no names)
- int numberThreads_

  Number of threads (not very operational)
- unsigned int specialOptions_

  For advanced options See get and set for meaning.
- **CoinMessageHandler** $*$ handler_

  Message handler.
- bool defaultHandler_

  Flag to say if default handler (so delete)
- **CoinThreadRandom** randomNumberGenerator_

  Thread specific random number generator.
- ClpEventHandler $*$ eventHandler_

  Event handler.
- std::vector$<$ std::string $>$ rowNames_

  Row names.
- std::vector$<$ std::string $>$ columnNames_

  Column names.
- **CoinMessages** messages_

  Messages.
- **CoinMessages** coinMessages_

  Coin *messages.*
- int maximumColumns_

  Maximum number of columns in model.
- int maximumRows_

  Maximum number of rows in model.
- int maximumInternalColumns_

  Maximum number of columns (internal arrays) in model.
- int maximumInternalRows_

  Maximum number of rows (internal arrays) in model.
- **CoinPackedMatrix** baseMatrix_

  Base packed matrix.
- **CoinPackedMatrix** baseRowCopy_

  Base row copy.
- double $*$ savedRowScale_

  Saved row scale factors for matrix.
- double $*$ savedColumnScale_

  Saved column scale factors.
- std::string strParam_ [ClpLastStrParam]

  Array of string parameters.

4.52.1 Detailed Description

Definition at line 38 of file ClpModel.hpp.

4.52.2 Constructor & Destructor Documentation

4.52.2.1 ClpModel::ClpModel (const ClpModel & *rhs*, int *scalingMode* = `-1`)

Copy constructor.

May scale depending on mode -1 leave mode as is 0 -off, 1 equilibrium, 2 geometric, 3, auto, 4 auto-but-as-initialSolve-in-bab

4.52.2.2 ClpModel::ClpModel (const ClpModel * *wholeModel*, int *numberRows*, const int * *whichRows*, int *numberColumns*, const int * *whichColumns*, bool *dropNames* = `true`, bool *dropIntegers* = `true`)

Subproblem constructor.

A subset of whole model is created from the row and column lists given. The new order is given by list order and duplicates are allowed. Name and integer information can be dropped

4.52.3 Member Function Documentation

4.52.3.1 void ClpModel::loadProblem (const ClpMatrixBase & *matrix*, const double * *collb*, const double * *colub*, const double * *obj*, const double * *rowlb*, const double * *rowub*, const double * *rowObjective* = `NULL`)

Loads a problem (the constraints on the rows are given by lower and upper bounds).

If a pointer is 0 then the following values are the default:

- `colub`: all columns have upper bound infinity
- `collb`: all columns have lower bound 0
- `rowub`: all rows have upper bound infinity
- `rowlb`: all rows have lower bound -infinity
- `obj`: all variables have 0 objective coefficient

4.52.3.2 void ClpModel::loadProblem (const int *numcols*, const int *numrows*, const CoinBigIndex * *start*, const int * *index*, const double * *value*, const double * *collb*, const double * *colub*, const double * *obj*, const double * *rowlb*, const double * *rowub*, const double * *rowObjective* = `NULL`)

Just like the other loadProblem() method except that the matrix is given in a standard column major ordered format (without gaps).

4.52.3.3 int ClpModel::loadProblem (CoinModel & *modelObject*, bool *tryPlusMinusOne* = `false`)

This loads a model from a coinModel object - returns number of errors.

modelObject not const as may be changed as part of process If tryPlusMinusOne then will try adding as +-1 matrix

4.52.3.4 void ClpModel::loadQuadraticObjective (const int *numberColumns*, const CoinBigIndex * *start*, const int * *column*, const double * *element*)

Load up quadratic objective.

This is stored as a **CoinPackedMatrix**

4.52.3.5 int ClpModel::addRows (const CoinBuild & *buildObject*, bool *tryPlusMinusOne* = false, bool *checkDuplicates* = true)

Add rows from a build object.

If tryPlusMinusOne then will try adding as +-1 matrix if no matrix exists. Returns number of errors e.g. duplicates

4.52.3.6 int ClpModel::addRows (CoinModel & *modelObject*, bool *tryPlusMinusOne* = false, bool *checkDuplicates* = true)

Add rows from a model object.

returns -1 if object in bad state (i.e. has column information) otherwise number of errors.

modelObject non const as can be regularized as part of build If tryPlusMinusOne then will try adding as +-1 matrix if no matrix exists.

4.52.3.7 int ClpModel::addColumns (const CoinBuild & *buildObject*, bool *tryPlusMinusOne* = false, bool *checkDuplicates* = true)

Add columns from a build object If tryPlusMinusOne then will try adding as +-1 matrix if no matrix exists.

Returns number of errors e.g. duplicates

4.52.3.8 int ClpModel::addColumns (CoinModel & *modelObject*, bool *tryPlusMinusOne* = false, bool *checkDuplicates* = true)

Add columns from a model object.

returns -1 if object in bad state (i.e. has row information) otherwise number of errors modelObject non const as can be regularized as part of build If tryPlusMinusOne then will try adding as +-1 matrix if no matrix exists.

4.52.3.9 void ClpModel::borrowModel (ClpModel & *otherModel*)

Borrow model.

This is so we don't have to copy large amounts of data around. It assumes a derived class wants to overwrite an empty model with a real one - while it does an algorithm

4.52.3.10 int ClpModel::cleanMatrix (double *threshold* = 1.0e-20)

Really clean up matrix (if ClpPackedMatrix).

a) eliminate all duplicate AND small elements in matrix b) remove all gaps and set extraGap_ and extraMajor_ to 0.0 c) reallocate arrays and make max lengths equal to lengths d) orders elements returns number of elements eliminated or -1 if not ClpPackedMatrix

4.52.3.11 int ClpModel::findNetwork (char * *rotate*, double *fractionNeeded* = 0.75)

Find a network subset.

rotate array should be numberRows. On output -1 not in network 0 in network as is 1 in network with signs swapped Returns number of network rows

4.52.3.12 int ClpModel::writeMps (const char * *filename*, int *formatType* = 0, int *numberAcross* = 2, double *objSense* = 0.0) const

Write the problem in MPS format to the specified file.

Row and column names may be null. formatType is

- 0 - normal
- 1 - extra accuracy
- 2 - IEEE hex

Returns non-zero on I/O error

4.52.3.13 int ClpModel::solveType ( ) const [inline]

Solve type - 1 simplex, 2 simplex interface, 3 Interior.

Definition at line 373 of file ClpModel.hpp.

4.52.3.14 int ClpModel::status ( ) const [inline]

Status of problem: -1 - unknown e.g.

before solve or if postSolve says not optimal 0 - optimal 1 - primal infeasible 2 - dual infeasible 3 - stopped on iterations or time 4 - stopped due to errors 5 - stopped by event handler (virtual int ClpEventHandler::event())

Definition at line 401 of file ClpModel.hpp.

4.52.3.15 void ClpModel::setColumnLower (int *elementIndex*, double *elementValue*)

Set a single column lower bound
Use -DBL_MAX for -infinity.

4.52.3.16 void ClpModel::setColumnUpper (int *elementIndex*, double *elementValue*)

Set a single column upper bound
Use DBL_MAX for infinity.

4.52.3.17 void ClpModel::setColumnSetBounds (const int * *indexFirst*, const int * *indexLast*, const double * *boundList*)

Set the bounds on a number of columns simultaneously
The default implementation just invokes setColLower() and setColUpper() over and over again.

Parameters

index←First,indexLast	pointers to the beginning and after the end of the array of the indices of the variables whose *either* bound changes

boundList	the new lower/upper bound pairs for the variables

4.52.3.18 void ClpModel::setColLower (int *elementIndex*, double *elementValue*) [inline]

Set a single column lower bound
Use -DBL_MAX for -infinity.

Definition at line 545 of file ClpModel.hpp.

4.52.3.19 void ClpModel::setColUpper (int *elementIndex*, double *elementValue*) [inline]

Set a single column upper bound
Use DBL_MAX for infinity.

Definition at line 550 of file ClpModel.hpp.

4.52.3.20 void ClpModel::setColSetBounds (const int ∗ *indexFirst*, const int ∗ *indexLast*, const double ∗ *boundList*) [inline]

Set the bounds on a number of columns simultaneously

Parameters

index← First,indexLast	pointers to the beginning and after the end of the array of the indices of the variables whose *either* bound changes
boundList	the new lower/upper bound pairs for the variables

Definition at line 566 of file ClpModel.hpp.

4.52.3.21 void ClpModel::setRowLower (int *elementIndex*, double *elementValue*)

Set a single row lower bound
Use -DBL_MAX for -infinity.

4.52.3.22 void ClpModel::setRowUpper (int *elementIndex*, double *elementValue*)

Set a single row upper bound
Use DBL_MAX for infinity.

4.52.3.23 void ClpModel::setRowSetBounds (const int ∗ *indexFirst*, const int ∗ *indexLast*, const double ∗ *boundList*)

Set the bounds on a number of rows simultaneously

Parameters

index↩ First,indexLast	pointers to the beginning and after the end of the array of the indices of the constraints whose *either* bound changes
boundList	the new lower/upper bound pairs for the constraints

4.52.3.24 void ClpModel::unscale ( )

If we constructed a "really" scaled model then this reverses the operation.

Quantities may not be exactly as they were before due to rounding errors

4.52.3.25 void ClpModel::replaceMatrix (ClpMatrixBase * *matrix*, bool *deleteCurrent* = `false`)

Replace Clp Matrix (current is not deleted unless told to and new is used) So up to user to delete current.

This was used where matrices were being rotated. ClpModel takes ownership.

4.52.3.26 void ClpModel::replaceMatrix (CoinPackedMatrix * *newmatrix*, bool *deleteCurrent* = `false`) [inline]

Replace Clp Matrix (current is not deleted unless told to and new is used) So up to user to delete current.

This was used where matrices were being rotated. This version changes **CoinPackedMatrix** to ClpPackedMatrix. Clp↩ Model takes ownership.

Definition at line 749 of file ClpModel.hpp.

4.52.3.27 double* ClpModel::infeasibilityRay (bool *fullRay* = `false`) const

Infeasibility/unbounded ray (NULL returned if none/wrong) Up to user to use delete [] on these arrays.

4.52.3.28 unsigned char* ClpModel::statusCopy ( ) const

Return copy of status (i.e.

basis) array (char[numberRows+numberColumns]), use delete []

4.52.3.29 void ClpModel::times (double *scalar*, const double * *x*, double * *y*) const

Return `y + A * x * scalar` in `y`.

Precondition

`x` must be of size `numColumns()`
`y` must be of size `numRows()`

4.52.3.30 void ClpModel::transposeTimes (double *scalar*, const double * *x*, double * *y*) const

Return `y + x * scalar * A` in `y`.

Precondition

x must be of size numRows()
y must be of size numColumns()

4.52.3.31 unsigned int ClpModel::specialOptions ( ) const [inline]

For advanced options 1 - Don't keep changing infeasibility weight 2 - Keep nonLinearCost round solves 4 - Force outgoing variables to exact bound (primal) 8 - Safe to use dense initial factorization 16 -Just use basic variables for operation if column generation 32 -Create ray even in BAB 64 -Treat problem as feasible until last minute (i.e.

minimize infeasibilities) 128 - Switch off all matrix sanity checks 256 - No row copy 512 - If not in values pass, solution guaranteed, skip as much as possible 1024 - In branch and bound 2048 - Don't bother to re-factorize if $<$ 20 iterations 4096 - Skip some optimality checks 8192 - Do Primal when cleaning up primal 16384 - In fast dual (so we can switch off things) 32768 - called from Osi 65536 - keep arrays around as much as possible (also use maximumR/C) 131072 - transposeTimes is -1.0 and can skip basic and fixed 262144 - extra copy of scaled matrix 524288 - Clp fast dual 1048576 - don't need to finish dual (can return 3) 2097152 - zero costs! 4194304 - don't scale integer variables 8388608 - Idiot when not really sure about it NOTE - many applications can call Clp but there may be some short cuts which are taken which are not guaranteed safe from all applications. Vetted applications will have a bit set and the code may test this At present I expect a few such applications - if too many I will have to re-think. It is up to application owner to change the code if she/he needs these short cuts. I will not debug unless in **Coin** repository. See COIN_CLP_VETTED comments. 0x01000000 is Cbc (and in branch and bound) 0x02000000 is in a different branch and bound

Definition at line 1054 of file ClpModel.hpp.

4.52.4 Member Data Documentation

4.52.4.1 unsigned char* ClpModel::status_ [protected]

Status (i.e.

basis) Region. I know that not all algorithms need a status array, but it made sense for things like crossover and put all permanent stuff in one place. No assumption is made about what is in status array (although it might be good to reserve bottom 3 bits (i.e. 0-7 numeric) for classic status). This is number of columns + number of rows long (in that order).

Definition at line 1173 of file ClpModel.hpp.

4.52.4.2 int ClpModel::solveType_ [protected]

Solve type - 1 simplex, 2 simplex interface, 3 Interior.

Definition at line 1185 of file ClpModel.hpp.

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

- ClpModel.hpp

4.53 ClpNetworkBasis Class Reference

This deals with Factorization and Updates for network structures.

`#include <ClpNetworkBasis.hpp>`

Public Member Functions

Constructors and destructor and copy

- ClpNetworkBasis ()
  Default constructor.
- ClpNetworkBasis (const ClpSimplex ∗model, int numberRows, const CoinFactorizationDouble ∗pivotRegion, const int ∗permuteBack, const CoinBigIndex ∗startColumn, const int ∗numberInColumn, const int ∗indexRow, const CoinFactorizationDouble ∗element)
  Constructor from **CoinFactorization**.
- ClpNetworkBasis (const ClpNetworkBasis &other)
  Copy constructor.
- ∼ClpNetworkBasis ()
  Destructor.
- ClpNetworkBasis & operator= (const ClpNetworkBasis &other)
  = copy

Do factorization

- int factorize (const ClpMatrixBase ∗matrix, int rowIsBasic[ ], int columnIsBasic[ ])
  When part of LP - given by basic variables.

rank one updates which do exist

- int replaceColumn (**CoinIndexedVector** ∗column, int pivotRow)
  Replaces one Column to basis, returns 0=OK, 1=Probably OK, 2=singular!!

various uses of factorization (return code number elements)

which user may want to know about

- double updateColumn (**CoinIndexedVector** ∗regionSparse, **CoinIndexedVector** ∗regionSparse2, int pivot←Row)
  Updates one column (FTRAN) from region, Returns pivot value if "pivotRow" >=0.
- int updateColumn (**CoinIndexedVector** ∗regionSparse, double array[ ]) const
  Updates one column (FTRAN) to/from array For large problems you should ALWAYS know where the nonzeros are, so please try and migrate to previous method after you have got code working using this simple method - thank you! (the only exception is if you know input is dense e.g.
- int updateColumnTranspose (**CoinIndexedVector** ∗regionSparse, double array[ ]) const
  Updates one column transpose (BTRAN) For large problems you should ALWAYS know where the nonzeros are, so please try and migrate to previous method after you have got code working using this simple method - thank you! (the only exception is if you know input is dense e.g.
- int updateColumnTranspose (**CoinIndexedVector** ∗regionSparse, **CoinIndexedVector** ∗regionSparse2) const
  Updates one column (BTRAN) from region2.

4.53.1 Detailed Description

This deals with Factorization and Updates for network structures.

Definition at line 26 of file ClpNetworkBasis.hpp.

4.53.2 Member Function Documentation

4.53.2.1 int ClpNetworkBasis::factorize (const ClpMatrixBase * *matrix*, int *rowIsBasic[ ]*, int *columnIsBasic[ ]*)

When part of LP - given by basic variables.

Actually does factorization. Arrays passed in have non negative value to say basic. If status is okay, basic variables have pivot row - this is only needed if increasingRows_ >1. If status is singular, then basic variables have pivot row and ones thrown out have -1 returns 0 -okay, -1 singular, -2 too many in basis

4.53.2.2 int ClpNetworkBasis::updateColumn (CoinIndexedVector * *regionSparse*, double *array[ ]*) const

Updates one column (FTRAN) to/from array For large problems you should ALWAYS know where the nonzeros are, so please try and migrate to previous method after you have got code working using this simple method - thank you! (the only exception is if you know input is dense e.g.

rhs)

4.53.2.3 int ClpNetworkBasis::updateColumnTranspose (CoinIndexedVector * *regionSparse*, double *array[ ]*) const

Updates one column transpose (BTRAN) For large problems you should ALWAYS know where the nonzeros are, so please try and migrate to previous method after you have got code working using this simple method - thank you! (the only exception is if you know input is dense e.g.

dense objective) returns number of nonzeros

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

- ClpNetworkBasis.hpp

4.54 ClpNetworkMatrix Class Reference

This implements a simple network matrix as derived from ClpMatrixBase.

`#include <ClpNetworkMatrix.hpp>`

Inheritance diagram for ClpNetworkMatrix:

Collaboration diagram for ClpNetworkMatrix:

Public Member Functions

Useful methods

- virtual **CoinPackedMatrix** * getPackedMatrix () const
  Return a complete **CoinPackedMatrix***.*
- virtual bool isColOrdered () const
  Whether the packed matrix is column major ordered or not.
- virtual CoinBigIndex getNumElements () const
  Number of entries in the packed matrix.
- virtual int getNumCols () const
  Number of columns.
- virtual int getNumRows () const
  Number of rows.

- virtual const double ∗ getElements () const

  A vector containing the elements in the packed matrix.
- virtual const int ∗ getIndices () const

  A vector containing the minor indices of the elements in the packed matrix.
- virtual const CoinBigIndex ∗ **getVectorStarts** () const
- virtual const int ∗ getVectorLengths () const

  The lengths of the major-dimension vectors.
- virtual void deleteCols (const int numDel, const int ∗indDel)

  Delete the columns whose indices are listed in `indDel`*.*
- virtual void deleteRows (const int numDel, const int ∗indDel)

  Delete the rows whose indices are listed in `indDel`*.*
- virtual void appendCols (int number, const **CoinPackedVectorBase** ∗const ∗columns)

  Append Columns.
- virtual void appendRows (int number, const **CoinPackedVectorBase** ∗const ∗rows)

  Append Rows.
- virtual int appendMatrix (int number, int type, const CoinBigIndex ∗starts, const int ∗index, const double ∗element, int numberOther=-1)

  Append a set of rows/columns to the end of the matrix.
- virtual ClpMatrixBase ∗ reverseOrderedCopy () const

  Returns a new matrix in reverse order without gaps.
- virtual CoinBigIndex countBasis (const int ∗whichColumn, int &numberColumnBasic)

  Returns number of elements in column part of basis.
- virtual void fillBasis (ClpSimplex ∗model, const int ∗whichColumn, int &numberColumnBasic, int ∗row, int ∗start, int ∗rowCount, int ∗columnCount, CoinFactorizationDouble ∗element)

  Fills in column part of basis.
- virtual CoinBigIndex ∗ dubiousWeights (const ClpSimplex ∗model, int ∗inputWeights) const

  Given positive integer weights for each row fills in sum of weights for each column (and slack).
- virtual void rangeOfElements (double &smallestNegative, double &largestNegative, double &smallestPositive, double &largestPositive)

  Returns largest and smallest elements of both signs.
- virtual void unpack (const ClpSimplex ∗model, **CoinIndexedVector** ∗rowArray, int column) const

  Unpacks a column into an CoinIndexedvector.
- virtual void unpackPacked (ClpSimplex ∗model, **CoinIndexedVector** ∗rowArray, int column) const

  Unpacks a column into an CoinIndexedvector in packed format Note that model is NOT const.
- virtual void add (const ClpSimplex ∗model, **CoinIndexedVector** ∗rowArray, int column, double multiplier) const

  Adds multiple of a column into an CoinIndexedvector You can use quickAdd to add to vector.
- virtual void add (const ClpSimplex ∗model, double ∗array, int column, double multiplier) const

  Adds multiple of a column into an array.
- virtual void releasePackedMatrix () const

  Allow any parts of a created CoinMatrix to be deleted.
- virtual bool canDoPartialPricing () const

  Says whether it can do partial pricing.
- virtual void partialPricing (ClpSimplex ∗model, double start, double end, int &bestSequence, int &number←Wanted)

  Partial pricing.

Matrix times vector methods

- virtual void times (double scalar, const double ∗x, double ∗y) const

  Return `y + A * scalar *x` *in* `y`*.*
- virtual void times (double scalar, const double ∗x, double ∗y, const double ∗rowScale, const double ∗column←Scale) const

And for scaling.
- virtual void transposeTimes (double scalar, const double ∗x, double ∗y) const

  Return $y + x * scalar * A$ *in* y.
- virtual void transposeTimes (double scalar, const double ∗x, double ∗y, const double ∗rowScale, const double ∗columnScale, double ∗spare=NULL) const

  And for scaling.
- virtual void transposeTimes (const ClpSimplex ∗model, double scalar, const **CoinIndexedVector** ∗x, **CoinIndexedVector** ∗y, **CoinIndexedVector** ∗z) const

  Return $x * scalar * A + y$ *in* z.
- virtual void subsetTransposeTimes (const ClpSimplex ∗model, const **CoinIndexedVector** ∗x, const **CoinIndexedVector** ∗y, **CoinIndexedVector** ∗z) const

  Return $x *A$ *in* z *but just for indices in y.*

Other

- bool trueNetwork () const

  Return true if really network, false if has slacks.

Constructors, destructor

- ClpNetworkMatrix ()

  Default constructor.
- ClpNetworkMatrix (int numberColumns, const int ∗head, const int ∗tail)

  Constructor from two arrays.
- virtual ∼ClpNetworkMatrix ()

  Destructor.

Copy method

- ClpNetworkMatrix (const ClpNetworkMatrix &)

  The copy constructor.
- ClpNetworkMatrix (const **CoinPackedMatrix** &)

  The copy constructor from an CoinNetworkMatrix.
- ClpNetworkMatrix & **operator=** (const ClpNetworkMatrix &)
- virtual ClpMatrixBase ∗ clone () const

  Clone.
- ClpNetworkMatrix (const ClpNetworkMatrix &wholeModel, int numberRows, const int ∗whichRows, int numberColumns, const int ∗whichColumns)

  Subset constructor (without gaps).
- virtual ClpMatrixBase ∗ subsetClone (int numberRows, const int ∗whichRows, int numberColumns, const int ∗whichColumns) const

  Subset clone (without gaps).

Protected Attributes

Data members

The data members are protected to allow access for derived classes.

- **CoinPackedMatrix** ∗ matrix_

  For fake **CoinPackedMatrix**.
- int ∗ **lengths_**
- int ∗ indices_

  Data -1, then +1 rows in pairs (row==-1 if one entry)

- int numberRows_

  Number of rows.
- int numberColumns_

  Number of columns.
- bool trueNetwork_

  True if all entries have two elements.

Additional Inherited Members

4.54.1 Detailed Description

This implements a simple network matrix as derived from ClpMatrixBase.

If you want more sophisticated version then you could inherit from this. Also you might want to allow networks with gain

Definition at line 19 of file ClpNetworkMatrix.hpp.

4.54.2 Constructor & Destructor Documentation

4.54.2.1 ClpNetworkMatrix::ClpNetworkMatrix ( )

Default constructor.

4.54.2.2 ClpNetworkMatrix::ClpNetworkMatrix (const **ClpNetworkMatrix** &)

The copy constructor.

4.54.2.3 ClpNetworkMatrix::ClpNetworkMatrix (const **CoinPackedMatrix** &)

The copy constructor from an CoinNetworkMatrix.

4.54.2.4 ClpNetworkMatrix::ClpNetworkMatrix (const **ClpNetworkMatrix** & *wholeModel,* int *numberRows,* const int ∗ *whichRows,* int *numberColumns,* const int ∗ *whichColumns*)

Subset constructor (without gaps).

Duplicates are allowed and order is as given

4.54.3 Member Function Documentation

4.54.3.1 virtual bool ClpNetworkMatrix::isColOrdered ( ) const [inline], [virtual]

Whether the packed matrix is column major ordered or not.

Implements ClpMatrixBase.

Definition at line 27 of file ClpNetworkMatrix.hpp.

4.54.3.2 virtual CoinBigIndex ClpNetworkMatrix::getNumElements ( ) const [inline], [virtual]

Number of entries in the packed matrix.

Implements ClpMatrixBase.

Definition at line 31 of file ClpNetworkMatrix.hpp.

4.54.3.3 virtual int ClpNetworkMatrix::getNumCols ( ) const [inline], [virtual]

Number of columns.

Implements ClpMatrixBase.

Definition at line 35 of file ClpNetworkMatrix.hpp.

4.54.3.4 virtual int ClpNetworkMatrix::getNumRows ( ) const [inline], [virtual]

Number of rows.

Implements ClpMatrixBase.

Definition at line 39 of file ClpNetworkMatrix.hpp.

4.54.3.5 virtual const double* ClpNetworkMatrix::getElements ( ) const [virtual]

A vector containing the elements in the packed matrix.

Note that there might be gaps in this list, entries that do not belong to any major-dimension vector. To get the actual elements one should look at this vector together with vectorStarts and vectorLengths.

Implements ClpMatrixBase.

4.54.3.6 virtual const int* ClpNetworkMatrix::getIndices ( ) const [inline], [virtual]

A vector containing the minor indices of the elements in the packed matrix.

Note that there might be gaps in this list, entries that do not belong to any major-dimension vector. To get the actual elements one should look at this vector together with vectorStarts and vectorLengths.

Implements ClpMatrixBase.

Definition at line 53 of file ClpNetworkMatrix.hpp.

4.54.3.7 virtual const int* ClpNetworkMatrix::getVectorLengths ( ) const [virtual]

The lengths of the major-dimension vectors.

Implements ClpMatrixBase.

4.54.3.8 virtual void ClpNetworkMatrix::deleteCols (const int *numDel*, const int * *indDel*) [virtual]

Delete the columns whose indices are listed in `indDel`.

Implements ClpMatrixBase.

4.54.3.9 virtual void ClpNetworkMatrix::deleteRows (const int *numDel,* const int * *indDel*) [virtual]

Delete the rows whose indices are listed in `indDel`.

Implements ClpMatrixBase.

4.54.3.10 virtual int ClpNetworkMatrix::appendMatrix (int *number,* int *type,* const CoinBigIndex * *starts,* const int * *index,* const double * *element,* int *numberOther* = -1) [virtual]

Append a set of rows/columns to the end of the matrix.

Returns number of errors i.e. if any of the new rows/columns contain an index that's larger than the number of columns-1/rows-1 (if numberOther>0) or duplicates If 0 then rows, 1 if columns

Reimplemented from ClpMatrixBase.

4.54.3.11 virtual CoinBigIndex* ClpNetworkMatrix::dubiousWeights (const ClpSimplex * *model,* int * *inputWeights*) const [virtual]

Given positive integer weights for each row fills in sum of weights for each column (and slack).

Returns weights vector

Reimplemented from ClpMatrixBase.

4.54.3.12 virtual void ClpNetworkMatrix::rangeOfElements (double & *smallestNegative,* double & *largestNegative,* double & *smallestPositive,* double & *largestPositive*) [virtual]

Returns largest and smallest elements of both signs.

Largest refers to largest absolute value.

Reimplemented from ClpMatrixBase.

4.54.3.13 virtual void ClpNetworkMatrix::unpackPacked (ClpSimplex * *model,* CoinIndexedVector * *rowArray,* int *column*) const [virtual]

Unpacks a column into an CoinIndexedvector in packed format Note that model is NOT const.

Bounds and objective could be modified if doing column generation (just for this variable)

Implements ClpMatrixBase.

4.54.3.14 virtual void ClpNetworkMatrix::times (double *scalar,* const double * *x,* double * *y*) const [virtual]

Return `y + A * scalar *x` in `y`.

Precondition

`x` must be of size `numColumns()`
`y` must be of size `numRows()`

4.54.3.15 virtual void ClpNetworkMatrix::transposeTimes (double *scalar*, const double * *x*, double * *y*) const [virtual]

Return `y + x * scalar * A` in `y`.

Precondition

`x` must be of size `numRows()`
`y` must be of size `numColumns()`

4.54.3.16 virtual void ClpNetworkMatrix::transposeTimes (const ClpSimplex * *model*, double *scalar*, const CoinIndexedVector * *x*, CoinIndexedVector * *y*, CoinIndexedVector * *z*) const [virtual]

Return `x * scalar * A + y` in `z`.

Can use y as temporary array (will be empty at end) Note - If x packed mode - then z packed mode Squashes small elements and knows about ClpSimplex

Implements ClpMatrixBase.

4.54.3.17 virtual void ClpNetworkMatrix::subsetTransposeTimes (const ClpSimplex * *model*, const CoinIndexedVector * *x*, const CoinIndexedVector * *y*, CoinIndexedVector * *z*) const [virtual]

Return `x *A` in `z` but just for indices in y.

Note - z always packed mode

Implements ClpMatrixBase.

4.54.3.18 virtual ClpMatrixBase* ClpNetworkMatrix::subsetClone (int *numberRows*, const int * *whichRows*, int *numberColumns*, const int * *whichColumns*) const [virtual]

Subset clone (without gaps).

Duplicates are allowed and order is as given

Reimplemented from ClpMatrixBase.

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

- ClpNetworkMatrix.hpp

4.55 ClpNode Class Reference

Collaboration diagram for ClpNode:

Classes

- struct branchState

Public Member Functions

Useful methods

- void applyNode (ClpSimplex ∗model, int doBoundsEtc)

  Applies node to model 0 - just tree bounds 1 - tree bounds and basis etc 2 - saved bounds and basis etc.
- void chooseVariable (ClpSimplex ∗model, ClpNodeStuff ∗info)

  Choose a new variable.
- int fixOnReducedCosts (ClpSimplex ∗model)

  Fix on reduced costs.
- void createArrays (ClpSimplex ∗model)

  Create odd arrays.
- void cleanUpForCrunch ()

  Clean up as crunch is different model.

Gets and sets

- double objectiveValue () const

  Objective value.
- void setObjectiveValue (double value)

  Set objective value.
- const double ∗ primalSolution () const

  Primal solution.
- const double ∗ dualSolution () const

  Dual solution.
- double branchingValue () const

  Initial value of integer variable.
- double sumInfeasibilities () const

  Sum infeasibilities.
- int numberInfeasibilities () const

  Number infeasibilities.
- int depth () const

  Relative depth.
- double estimatedSolution () const

  Estimated solution value.
- int way () const

  Way for integer variable -1 down , +1 up.
- bool fathomed () const

  Return true if branch exhausted.
- void changeState ()

  Change state of variable i.e. go other way.
- int sequence () const

  Sequence number of integer variable (-1 if none)
- bool oddArraysExist () const

  If odd arrays exist.
- const unsigned char ∗ statusArray () const

  Status array.

Constructors, destructor

- ClpNode ()

  Default constructor.
- ClpNode (ClpSimplex ∗model, const ClpNodeStuff ∗stuff, int depth)

  Constructor from model.
- void gutsOfConstructor (ClpSimplex ∗model, const ClpNodeStuff ∗stuff, int arraysExist, int depth)

  Does work of constructor (partly so gdb will work)

- virtual ∼ClpNode ()

  Destructor.

Copy methods (at present illegal - will abort)

- ClpNode (const ClpNode &)

  The copy constructor.
- ClpNode & operator= (const ClpNode &)

  Operator =.

Protected Attributes

Data

- double branchingValue_

  Initial value of integer variable.
- double objectiveValue_

  Value of objective.
- double sumInfeasibilities_

  Sum of infeasibilities.
- double estimatedSolution_

  Estimated solution value.
- ClpFactorization ∗ factorization_

  Factorization.
- ClpDualRowSteepest ∗ weights_

  Steepest edge weights.
- unsigned char ∗ status_

  Status vector.
- double ∗ primalSolution_

  Primal solution.
- double ∗ dualSolution_

  Dual solution.
- int ∗ lower_

  Integer lower bounds (only used in fathomMany)
- int ∗ upper_

  Integer upper bounds (only used in fathomMany)
- int ∗ pivotVariables_

  Pivot variables for factorization.
- int ∗ fixed_

  Variables fixed by reduced costs (at end of branch) 0x10000000 added if fixed to UB.
- branchState branchState_

  State of branch.
- int sequence_

  Sequence number of integer variable (-1 if none)
- int numberInfeasibilities_

  Number of infeasibilities.
- int depth_

  Relative depth.
- int numberFixed_

  Number fixed by reduced cost.
- int flags_

Flags - 1 duals scaled.
- int maximumFixed_

  Maximum number fixed by reduced cost.
- int maximumRows_

  Maximum rows so far.
- int maximumColumns_

  Maximum columns so far.
- int maximumIntegers_

  Maximum Integers so far.

4.55.1 Detailed Description

Definition at line 19 of file ClpNode.hpp.

4.55.2 Constructor & Destructor Documentation

4.55.2.1 ClpNode::ClpNode ( )

Default constructor.

4.55.2.2 ClpNode::ClpNode (const ClpNode &)

The copy constructor.

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

- ClpNode.hpp

4.56 ClpNodeStuff Class Reference

Collaboration diagram for ClpNodeStuff:

Public Member Functions

Constructors, destructor

- ClpNodeStuff ()

  Default constructor.
- virtual ~ClpNodeStuff ()

  Destructor.

Copy methods (only copies ints etc, nulls arrays)

- ClpNodeStuff (const ClpNodeStuff &)

  The copy constructor.
- ClpNodeStuff & operator= (const ClpNodeStuff &)

  Operator =.
- void zap (int type)

  Zaps stuff 1 - arrays, 2 ints, 3 both.

Fill methods

- void fillPseudoCosts (const double *down, const double *up, const int *priority, const int *numberDown, const int *numberUp, const int *numberDownInfeasible, const int *numberUpInfeasible, int number)

  Fill with pseudocosts.
- void update (int way, int sequence, double change, bool feasible)

  Update pseudo costs.
- int maximumNodes () const

  Return maximum number of nodes.
- int maximumSpace () const

  Return maximum space for nodes.

Public Attributes

Data

- double integerTolerance_

  Integer tolerance.
- double integerIncrement_

  Integer increment.
- double smallChange_

  Small change in branch.
- double * downPseudo_

  Down pseudo costs.
- double * upPseudo_

  Up pseudo costs.
- int * priority_

  Priority.
- int * numberDown_

  Number of times down.
- int * numberUp_

  Number of times up.
- int * numberDownInfeasible_

  Number of times down infeasible.
- int * numberUpInfeasible_

  Number of times up infeasible.
- double * saveCosts_

  Copy of costs (local)
- ClpNode ** nodeInfo_

  Array of ClpNodes.
- ClpSimplex * large_

  Large model if crunched.
- int * whichRow_

  Which rows in large model.
- int * whichColumn_

  Which columns in large model.
- **CoinMessageHandler** * handler_

  Cbc's message handler.
- int nBound_

  Number bounds in large model.
- int saveOptions_

Save of specialOptions_ (local)

- int solverOptions_

  Options to pass to solver 1 - create external reduced costs for columns 2 - create external reduced costs for rows 4 - create external row activity (columns always done) Above only done if feasible 32 - just create up to nDepth_+1 nodes 65536 - set if activated.
- int maximumNodes_

  Maximum number of nodes to do.
- int numberBeforeTrust_

  Number before trust from CbcModel.
- int stateOfSearch_

  State of search from CbcModel.
- int nDepth_

  Number deep.
- int nNodes_

  Number nodes returned (-1 if fathom aborted)
- int numberNodesExplored_

  Number of nodes explored.
- int numberIterations_

  Number of iterations.
- int presolveType_

  Type of presolve - 0 none, 1 crunch.
- int startingDepth_

  Depth passed in.
- int nodeCalled_

  Node at which called.

4.56.1 Detailed Description

Definition at line 176 of file ClpNode.hpp.

4.56.2 Constructor & Destructor Documentation

4.56.2.1 ClpNodeStuff::ClpNodeStuff ( )

Default constructor.

4.56.2.2 ClpNodeStuff::ClpNodeStuff (const ClpNodeStuff &)

The copy constructor.

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

- ClpNode.hpp

4.57 ClpNonLinearCost Class Reference

Public Member Functions

Constructors, destructor

- ClpNonLinearCost ()

  Default constructor.
- ClpNonLinearCost (ClpSimplex ∗model, int method=1)

  Constructor from simplex.
- ClpNonLinearCost (ClpSimplex ∗model, const int ∗starts, const double ∗lower, const double ∗cost)

  Constructor from simplex and list of non-linearities (columns only) First lower of each column has to match real lower Last lower has to be <= upper (if == then cost ignored) This could obviously be changed to make more user friendly.
- ∼ClpNonLinearCost ()

  Destructor.
- **ClpNonLinearCost** (const ClpNonLinearCost &)
- ClpNonLinearCost & **operator=** (const ClpNonLinearCost &)

Actual work in primal

- void checkInfeasibilities (double oldTolerance=0.0)

  Changes infeasible costs and computes number and cost of infeas Puts all non-basic (non free) variables to bounds and all free variables to zero if oldTolerance is non-zero.
- void checkInfeasibilities (int numberInArray, const int ∗index)

  Changes infeasible costs for each variable The indices are row indices and need converting to sequences.
- void checkChanged (int numberInArray, **CoinIndexedVector** ∗update)

  Puts back correct infeasible costs for each variable The input indices are row indices and need converting to sequences for costs.
- void goThru (int numberInArray, double multiplier, const int ∗index, const double ∗work, double ∗rhs)

  Goes through one bound for each variable.
- void goBack (int numberInArray, const int ∗index, double ∗rhs)

  Takes off last iteration (i.e.
- void goBackAll (const **CoinIndexedVector** ∗update)

  Puts back correct infeasible costs for each variable The input indices are row indices and need converting to sequences for costs.
- void zapCosts ()

  Temporary zeroing of feasible costs.
- void refreshCosts (const double ∗columnCosts)

  Refreshes costs always makes row costs zero.
- void feasibleBounds ()

  Puts feasible bounds into lower and upper.
- void refresh ()

  Refresh - assuming regions OK.
- double setOne (int sequence, double solutionValue)

  Sets bounds and cost for one variable Returns change in cost May need to be inline for speed.
- void setOne (int sequence, double solutionValue, double lowerValue, double upperValue, double cost←Value=0.0)

  Sets bounds and infeasible cost and true cost for one variable This is for gub and column generation etc.
- int setOneOutgoing (int sequence, double &solutionValue)

  Sets bounds and cost for outgoing variable may change value Returns direction.
- double nearest (int sequence, double solutionValue)

  Returns nearest bound.
- double changeInCost (int sequence, double alpha) const

  Returns change in cost - one down if alpha >0.0, up if <0.0 Value is current - new.
- double **changeUpInCost** (int sequence) const
- double **changeDownInCost** (int sequence) const
- double changeInCost (int sequence, double alpha, double &rhs)

  This also updates next bound.
- double lower (int sequence) const

Returns current lower bound.
- double upper (int sequence) const

  Returns current upper bound.
- double cost (int sequence) const

  Returns current cost.

Gets and sets

- int numberInfeasibilities () const

  Number of infeasibilities.
- double changeInCost () const

  Change in cost.
- double feasibleCost () const

  Feasible cost.
- double feasibleReportCost () const

  Feasible cost with offset and direction (i.e. for reporting)
- double sumInfeasibilities () const

  Sum of infeasibilities.
- double largestInfeasibility () const

  Largest infeasibility.
- double averageTheta () const

  Average theta.
- void **setAverageTheta** (double value)
- void **setChangeInCost** (double value)
- void **setMethod** (int value)
- bool lookBothWays () const

  See if may want to look both ways.

Private functions to deal with infeasible regions

- bool **infeasible** (int i) const
- void **setInfeasible** (int i, bool trueFalse)
- unsigned char ∗ **statusArray** () const
- void validate ()

  For debug.

4.57.1 Detailed Description

Definition at line 78 of file ClpNonLinearCost.hpp.

4.57.2 Constructor & Destructor Documentation

4.57.2.1 ClpNonLinearCost::ClpNonLinearCost (**ClpSimplex** ∗ *model*, int *method* = `1`)

Constructor from simplex.

This will just set up wasteful arrays for linear, but later may do dual analysis and even finding duplicate columns .

4.57.3 Member Function Documentation

4.57.3.1 void ClpNonLinearCost::checkInfeasibilities (double *oldTolerance* = 0.0)

Changes infeasible costs and computes number and cost of infeas Puts all non-basic (non free) variables to bounds and all free variables to zero if oldTolerance is non-zero.

- but does not move those <= oldTolerance away

4.57.3.2 void ClpNonLinearCost::checkChanged (int *numberInArray,* CoinIndexedVector ∗ *update*)

Puts back correct infeasible costs for each variable The input indices are row indices and need converting to sequences for costs.

On input array is empty (but indices exist). On exit just changed costs will be stored as normal **CoinIndexedVector**

4.57.3.3 void ClpNonLinearCost::goThru (int *numberInArray,* double *multiplier,* const int ∗ *index,* const double ∗ *work,* double ∗ *rhs*)

Goes through one bound for each variable.

If multiplier∗work[iRow]>0 goes down, otherwise up. The indices are row indices and need converting to sequences Temporary offsets may be set Rhs entries are increased

4.57.3.4 void ClpNonLinearCost::goBack (int *numberInArray,* const int ∗ *index,* double ∗ *rhs*)

Takes off last iteration (i.e.

offsets closer to 0)

4.57.3.5 void ClpNonLinearCost::goBackAll (const CoinIndexedVector ∗ *update*)

Puts back correct infeasible costs for each variable The input indices are row indices and need converting to sequences for costs.

At the end of this all temporary offsets are zero

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

- ClpNonLinearCost.hpp

4.58 ClpObjective Class Reference

Objective Abstract Base Class.

`#include <ClpObjective.hpp>`

Inheritance diagram for ClpObjective:

Public Member Functions

Stuff

- virtual double ∗ gradient (const ClpSimplex ∗model, const double ∗solution, double &offset, bool refresh, int includeLinear=2)=0

    Returns gradient.
- virtual double reducedGradient (ClpSimplex ∗model, double ∗region, bool useFeasibleCosts)=0

    Returns reduced gradient.Returns an offset (to be added to current one).
- virtual double stepLength (ClpSimplex ∗model, const double ∗solution, const double ∗change, double maximumTheta, double &currentObj, double &predictedObj, double &thetaObj)=0

    Returns step length which gives minimum of objective for solution + theta ∗ change vector up to maximum theta.
- virtual double objectiveValue (const ClpSimplex ∗model, const double ∗solution) const =0

    Return objective value (without any ClpModel offset) (model may be NULL)
- virtual void resize (int newNumberColumns)=0

    Resize objective.
- virtual void deleteSome (int numberToDelete, const int ∗which)=0

    Delete columns in objective.
- virtual void reallyScale (const double ∗columnScale)=0

    Scale objective.
- virtual int markNonlinear (char ∗which)

    Given a zeroed array sets nonlinear columns to 1.
- virtual void newXValues ()

    Say we have new primal solution - so may need to recompute.

Constructors and destructors

- ClpObjective ()

    Default Constructor.
- ClpObjective (const ClpObjective &)

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

    Assignment operator.
- virtual ∼ClpObjective ()

    Destructor.
- virtual ClpObjective ∗ clone () const =0

    Clone.
- virtual ClpObjective ∗ subsetClone (int numberColumns, const int ∗whichColumns) const

    Subset clone.

Other

- int type () const

    Returns type (above 63 is extra information)
- void setType (int value)

    Sets type (above 63 is extra information)
- int activated () const

    Whether activated.
- void setActivated (int value)

    Set whether activated.
- double nonlinearOffset () const

    Objective offset.

Protected Attributes

Protected member data

- double offset_
  Value of non-linear part of objective.
- int type_
  Type of objective - linear is 1.
- int activated_
  Whether activated.

4.58.1 Detailed Description

Objective Abstract Base Class.

Abstract Base Class for describing an objective function

Definition at line 19 of file ClpObjective.hpp.

4.58.2 Member Function Documentation

4.58.2.1 virtual double∗ ClpObjective::gradient (const ClpSimplex ∗ *model,* const double ∗ *solution,* double & *offset,* bool *refresh,* int *includeLinear* = 2) [pure virtual]

Returns gradient.

If Linear then solution may be NULL, also returns an offset (to be added to current one) If refresh is false then uses last solution Uses model for scaling includeLinear 0 - no, 1 as is, 2 as feasible

Implemented in ClpQuadraticObjective, and ClpLinearObjective.

4.58.2.2 virtual double ClpObjective::stepLength (ClpSimplex ∗ *model,* const double ∗ *solution,* const double ∗ *change,* double *maximumTheta,* double & *currentObj,* double & *predictedObj,* double & *thetaObj*) [pure virtual]

Returns step length which gives minimum of objective for solution + theta ∗ change vector up to maximum theta.

arrays are numberColumns+numberRows Also sets current objective, predicted and at maximumTheta

Implemented in ClpQuadraticObjective, and ClpLinearObjective.

4.58.2.3 virtual int ClpObjective::markNonlinear (char ∗ *which*) [virtual]

Given a zeroed array sets nonlinear columns to 1.

Returns number of nonlinear columns

Reimplemented in ClpQuadraticObjective.

4.58.2.4 virtual ClpObjective∗ ClpObjective::subsetClone (int *numberColumns,* const int ∗ *whichColumns*) const [virtual]

Subset clone.

Duplicates are allowed and order is as given. Derived classes need not provide this as it may not always make sense

Reimplemented in ClpQuadraticObjective, and ClpLinearObjective.

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

- ClpObjective.hpp

4.59 ClpPackedMatrix Class Reference

Inheritance diagram for ClpPackedMatrix:

Collaboration diagram for ClpPackedMatrix:

Public Member Functions

Useful methods

- virtual **CoinPackedMatrix** ∗ getPackedMatrix () const
  Return a complete **CoinPackedMatrix**.
- virtual bool isColOrdered () const
  Whether the packed matrix is column major ordered or not.
- virtual CoinBigIndex getNumElements () const
  Number of entries in the packed matrix.
- virtual int getNumCols () const
  Number of columns.
- virtual int getNumRows () const
  Number of rows.
- virtual const double ∗ getElements () const
  A vector containing the elements in the packed matrix.
- double ∗ getMutableElements () const
  Mutable elements.
- virtual const int ∗ getIndices () const
  A vector containing the minor indices of the elements in the packed matrix.
- virtual const CoinBigIndex ∗ **getVectorStarts** () const
- virtual const int ∗ getVectorLengths () const
  The lengths of the major-dimension vectors.
- virtual int getVectorLength (int index) const
  The length of a single major-dimension vector.
- virtual void deleteCols (const int numDel, const int ∗indDel)
  Delete the columns whose indices are listed in `indDel`.
- virtual void deleteRows (const int numDel, const int ∗indDel)
  Delete the rows whose indices are listed in `indDel`.
- virtual void appendCols (int number, const **CoinPackedVectorBase** ∗const ∗columns)
  Append Columns.
- virtual void appendRows (int number, const **CoinPackedVectorBase** ∗const ∗rows)
  Append Rows.
- virtual int appendMatrix (int number, int type, const CoinBigIndex ∗starts, const int ∗index, const double ∗element, int numberOther=-1)
  Append a set of rows/columns to the end of the matrix.
- virtual void replaceVector (const int index, const int numReplace, const double ∗newElements)
  Replace the elements of a vector.
- virtual void modifyCoefficient (int row, int column, double newElement, bool keepZero=false)
  Modify one element of packed matrix.
- virtual ClpMatrixBase ∗ reverseOrderedCopy () const

Returns a new matrix in reverse order without gaps.

- virtual CoinBigIndex countBasis (const int ∗whichColumn, int &numberColumnBasic)

  Returns number of elements in column part of basis.
- virtual void fillBasis (ClpSimplex ∗model, const int ∗whichColumn, int &numberColumnBasic, int ∗row, int ∗start, int ∗rowCount, int ∗columnCount, CoinFactorizationDouble ∗element)

  Fills in column part of basis.
- virtual int scale (ClpModel ∗model, const ClpSimplex ∗baseModel=NULL) const

  Creates scales for column copy (rowCopy in model may be modified) returns non-zero if no scaling done.
- virtual void scaleRowCopy (ClpModel ∗model) const

  Scales rowCopy if column copy scaled Only called if scales already exist.
- void createScaledMatrix (ClpSimplex ∗model) const

  Creates scaled column copy if scales exist.
- virtual ClpMatrixBase ∗ scaledColumnCopy (ClpModel ∗model) const

  Realy really scales column copy Only called if scales already exist.
- virtual bool allElementsInRange (ClpModel ∗model, double smallest, double largest, int check=15)

  Checks if all elements are in valid range.
- virtual void rangeOfElements (double &smallestNegative, double &largestNegative, double &smallestPositive, double &largestPositive)

  Returns largest and smallest elements of both signs.
- virtual void unpack (const ClpSimplex ∗model, **CoinIndexedVector** ∗rowArray, int column) const

  Unpacks a column into an CoinIndexedvector.
- virtual void unpackPacked (ClpSimplex ∗model, **CoinIndexedVector** ∗rowArray, int column) const

  Unpacks a column into an CoinIndexedvector in packed foramt Note that model is NOT const.
- virtual void add (const ClpSimplex ∗model, **CoinIndexedVector** ∗rowArray, int column, double multiplier) const

  Adds multiple of a column into an CoinIndexedvector You can use quickAdd to add to vector.
- virtual void add (const ClpSimplex ∗model, double ∗array, int column, double multiplier) const

  Adds multiple of a column into an array.
- virtual void releasePackedMatrix () const

  Allow any parts of a created **CoinPackedMatrix** *to be deleted.*
- virtual CoinBigIndex ∗ dubiousWeights (const ClpSimplex ∗model, int ∗inputWeights) const

  Given positive integer weights for each row fills in sum of weights for each column (and slack).
- virtual bool canDoPartialPricing () const

  Says whether it can do partial pricing.
- virtual void partialPricing (ClpSimplex ∗model, double start, double end, int &bestSequence, int &number←Wanted)

  Partial pricing.
- virtual int refresh (ClpSimplex ∗model)

  makes sure active columns correct
- virtual void **reallyScale** (const double ∗rowScale, const double ∗columnScale)
- virtual void setDimensions (int numrows, int numcols)

  Set the dimensions of the matrix.

Matrix times vector methods

- virtual void times (double scalar, const double ∗x, double ∗y) const

  Return $y + A * scalar *x$ *in* y.
- virtual void times (double scalar, const double ∗x, double ∗y, const double ∗rowScale, const double ∗column←Scale) const

  And for scaling.
- virtual void transposeTimes (double scalar, const double ∗x, double ∗y) const

  Return $y + x * scalar * A$ *in* y.

- virtual void transposeTimes (double scalar, const double ∗x, double ∗y, const double ∗rowScale, const double ∗columnScale, double ∗spare=NULL) const

  And for scaling.
- void transposeTimesSubset (int number, const int ∗which, const double ∗pi, double ∗y, const double ∗rowScale, const double ∗columnScale, double ∗spare=NULL) const

  Return `y - pi * A` *in* `y`*.*
- virtual void transposeTimes (const ClpSimplex ∗model, double scalar, const **CoinIndexedVector** ∗x, **CoinIndexedVector** ∗y, **CoinIndexedVector** ∗z) const

  Return `x * scalar * A + y` *in* `z`*.*
- void transposeTimesByColumn (const ClpSimplex ∗model, double scalar, const **CoinIndexedVector** ∗x, **CoinIndexedVector** ∗y, **CoinIndexedVector** ∗z) const

  Return `x * scalar * A + y` *in* `z`*.*
- virtual void transposeTimesByRow (const ClpSimplex ∗model, double scalar, const **CoinIndexedVector** ∗x, **CoinIndexedVector** ∗y, **CoinIndexedVector** ∗z) const

  Return `x * scalar * A + y` *in* `z`*.*
- virtual void subsetTransposeTimes (const ClpSimplex ∗model, const **CoinIndexedVector** ∗x, const **CoinIndexedVector** ∗y, **CoinIndexedVector** ∗z) const

  Return `x *A` *in* `z` *but just for indices in y.*
- virtual bool canCombine (const ClpSimplex ∗model, const **CoinIndexedVector** ∗pi) const

  Returns true if can combine transposeTimes and subsetTransposeTimes and if it would be faster.
- virtual void transposeTimes2 (const ClpSimplex ∗model, const **CoinIndexedVector** ∗pi1, **CoinIndexedVector** ∗dj1, const **CoinIndexedVector** ∗pi2, **CoinIndexedVector** ∗spare, double referenceIn, double devex, unsigned int ∗reference, double ∗weights, double scaleFactor)

  Updates two arrays for steepest.
- virtual void subsetTimes2 (const ClpSimplex ∗model, **CoinIndexedVector** ∗dj1, const **CoinIndexedVector** ∗pi2, **CoinIndexedVector** ∗dj2, double referenceIn, double devex, unsigned int ∗reference, double ∗weights, double scaleFactor)

  Updates second array for steepest and does devex weights.
- void useEffectiveRhs (ClpSimplex ∗model)

  Sets up an effective RHS.

Other

- **CoinPackedMatrix** ∗ matrix () const

  Returns **CoinPackedMatrix** *(non const)*
- void setMatrixNull ()

  Just sets matrix_ to NULL so it can be used elsewhere.
- void makeSpecialColumnCopy ()

  Say we want special column copy.
- void releaseSpecialColumnCopy ()

  Say we don't want special column copy.
- bool zeros () const

  Are there zeros?
- bool wantsSpecialColumnCopy () const

  Do we want special column copy.
- int flags () const

  Flags.
- void checkGaps ()

  Sets flags_ correctly.
- int numberActiveColumns () const

  number of active columns (normally same as number of columns)
- void setNumberActiveColumns (int value)

  Set number of active columns (normally same as number of columns)

Constructors, destructor

- ClpPackedMatrix ()
  Default constructor.
- virtual ∼ClpPackedMatrix ()
  Destructor.

Copy method

- ClpPackedMatrix (const ClpPackedMatrix &)
  The copy constructor.
- ClpPackedMatrix (const **CoinPackedMatrix** &)
  The copy constructor from an **CoinPackedMatrix**.
- ClpPackedMatrix (const ClpPackedMatrix &wholeModel, int numberRows, const int ∗whichRows, int number←Columns, const int ∗whichColumns)
  Subset constructor (without gaps).
- **ClpPackedMatrix** (const **CoinPackedMatrix** &wholeModel, int numberRows, const int ∗whichRows, int numberColumns, const int ∗whichColumns)
- ClpPackedMatrix (**CoinPackedMatrix** ∗matrix)
  This takes over ownership (for space reasons)
- ClpPackedMatrix & **operator=** (const ClpPackedMatrix &)
- virtual ClpMatrixBase ∗ clone () const
  Clone.
- virtual void copy (const ClpPackedMatrix ∗from)
  Copy contents - resizing if necessary - otherwise re-use memory.
- virtual ClpMatrixBase ∗ subsetClone (int numberRows, const int ∗whichRows, int numberColumns, const int ∗whichColumns) const
  Subset clone (without gaps).
- void specialRowCopy (ClpSimplex ∗model, const ClpMatrixBase ∗rowCopy)
  make special row copy
- void specialColumnCopy (ClpSimplex ∗model)
  make special column copy
- virtual void correctSequence (const ClpSimplex ∗model, int &sequenceIn, int &sequenceOut)
  Correct sequence in and out to give true value.

Protected Member Functions

- void checkFlags (int type) const
  Check validity.

Protected Attributes

Data members

The data members are protected to allow access for derived classes.

- **CoinPackedMatrix** ∗ matrix_
  Data.
- int numberActiveColumns_
  number of active columns (normally same as number of columns)
- int flags_
  Flags - 1 - has zero elements 2 - has gaps 4 - has special row copy 8 - has special column copy 16 - wants special column copy.
- ClpPackedMatrix2 ∗ rowCopy_
  Special row copy.
- ClpPackedMatrix3 ∗ columnCopy_
  Special column copy.

4.59.1 Detailed Description

Definition at line 21 of file ClpPackedMatrix.hpp.

4.59.2 Constructor & Destructor Documentation

4.59.2.1 ClpPackedMatrix::ClpPackedMatrix ( )

Default constructor.

4.59.2.2 ClpPackedMatrix::ClpPackedMatrix (const ClpPackedMatrix &)

The copy constructor.

4.59.2.3 ClpPackedMatrix::ClpPackedMatrix (const CoinPackedMatrix &)

The copy constructor from an **CoinPackedMatrix**.

4.59.2.4 ClpPackedMatrix::ClpPackedMatrix (const ClpPackedMatrix & *wholeModel*, int *numberRows*, const int * *whichRows*, int *numberColumns*, const int * *whichColumns*)

Subset constructor (without gaps).

Duplicates are allowed and order is as given

4.59.3 Member Function Documentation

4.59.3.1 virtual bool ClpPackedMatrix::isColOrdered ( ) const [inline], [virtual]

Whether the packed matrix is column major ordered or not.

Implements ClpMatrixBase.

Definition at line 31 of file ClpPackedMatrix.hpp.

4.59.3.2 virtual CoinBigIndex ClpPackedMatrix::getNumElements ( ) const [inline], [virtual]

Number of entries in the packed matrix.

Implements ClpMatrixBase.

Definition at line 35 of file ClpPackedMatrix.hpp.

4.59.3.3 virtual int ClpPackedMatrix::getNumCols ( ) const [inline], [virtual]

Number of columns.

Implements ClpMatrixBase.

Definition at line 39 of file ClpPackedMatrix.hpp.

4.59.3.4 virtual int ClpPackedMatrix::getNumRows ( ) const [inline], [virtual]

Number of rows.

Implements ClpMatrixBase.

Definition at line 43 of file ClpPackedMatrix.hpp.

4.59.3.5 virtual const double* ClpPackedMatrix::getElements ( ) const [inline], [virtual]

A vector containing the elements in the packed matrix.

Note that there might be gaps in this list, entries that do not belong to any major-dimension vector. To get the actual elements one should look at this vector together with vectorStarts and vectorLengths.

Implements ClpMatrixBase.

Definition at line 51 of file ClpPackedMatrix.hpp.

4.59.3.6 virtual const int* ClpPackedMatrix::getIndices ( ) const [inline], [virtual]

A vector containing the minor indices of the elements in the packed matrix.

Note that there might be gaps in this list, entries that do not belong to any major-dimension vector. To get the actual elements one should look at this vector together with vectorStarts and vectorLengths.

Implements ClpMatrixBase.

Definition at line 63 of file ClpPackedMatrix.hpp.

4.59.3.7 virtual const int* ClpPackedMatrix::getVectorLengths ( ) const [inline], [virtual]

The lengths of the major-dimension vectors.

Implements ClpMatrixBase.

Definition at line 71 of file ClpPackedMatrix.hpp.

4.59.3.8 virtual int ClpPackedMatrix::getVectorLength (int *index*) const [inline], [virtual]

The length of a single major-dimension vector.

Reimplemented from ClpMatrixBase.

Definition at line 75 of file ClpPackedMatrix.hpp.

4.59.3.9 virtual void ClpPackedMatrix::deleteCols (const int *numDel*, const int * *indDel*) [virtual]

Delete the columns whose indices are listed in `indDel`.

Implements ClpMatrixBase.

4.59.3.10 virtual void ClpPackedMatrix::deleteRows (const int *numDel*, const int * *indDel*) [virtual]

Delete the rows whose indices are listed in `indDel`.

Implements ClpMatrixBase.

4.59.3.11 virtual int ClpPackedMatrix::appendMatrix (int *number*, int *type*, const CoinBigIndex * *starts*, const int * *index*, const double * *element*, int *numberOther* = -1) [virtual]

Append a set of rows/columns to the end of the matrix.

Returns number of errors i.e. if any of the new rows/columns contain an index that's larger than the number of columns-1/rows-1 (if numberOther>0) or duplicates If 0 then rows, 1 if columns

Reimplemented from ClpMatrixBase.

4.59.3.12 virtual void ClpPackedMatrix::replaceVector (const int *index*, const int *numReplace*, const double * *newElements*) [inline], [virtual]

Replace the elements of a vector.

The indices remain the same. This is only needed if scaling and a row copy is used. At most the number specified will be replaced. The index is between 0 and major dimension of matrix

Definition at line 100 of file ClpPackedMatrix.hpp.

4.59.3.13 virtual void ClpPackedMatrix::modifyCoefficient (int *row*, int *column*, double *newElement*, bool *keepZero* = false) [inline], [virtual]

Modify one element of packed matrix.

An element may be added. This works for either ordering If the new element is zero it will be deleted unless keepZero true

Reimplemented from ClpMatrixBase.

Definition at line 107 of file ClpPackedMatrix.hpp.

4.59.3.14 virtual ClpMatrixBase* ClpPackedMatrix::scaledColumnCopy (ClpModel * *model*) const [virtual]

Realy really scales column copy Only called if scales already exist.

Up to user ro delete

Reimplemented from ClpMatrixBase.

4.59.3.15 virtual bool ClpPackedMatrix::allElementsInRange (ClpModel * *model*, double *smallest*, double *largest*, int *check* = 15) [virtual]

Checks if all elements are in valid range.

Can just return true if you are not paranoid. For Clp I will probably expect no zeros. Code can modify matrix to get rid of small elements. check bits (can be turned off to save time) : 1 - check if matrix has gaps 2 - check if zero elements 4 - check and compress duplicates 8 - report on large and small

Reimplemented from ClpMatrixBase.

4.59.3.16 virtual void ClpPackedMatrix::rangeOfElements (double & *smallestNegative*, double & *largestNegative*, double & *smallestPositive*, double & *largestPositive*) [virtual]

Returns largest and smallest elements of both signs.

Largest refers to largest absolute value.

Reimplemented from ClpMatrixBase.

4.59.3.17 virtual void ClpPackedMatrix::unpackPacked (ClpSimplex * *model*, CoinIndexedVector * *rowArray*, int *column*) const [virtual]

Unpacks a column into an CoinIndexedvector in packed foramt Note that model is NOT const.

Bounds and objective could be modified if doing column generation (just for this variable)

Implements ClpMatrixBase.

Reimplemented in ClpGubMatrix.

4.59.3.18 virtual CoinBigIndex* ClpPackedMatrix::dubiousWeights (const ClpSimplex * *model*, int * *inputWeights*) const [virtual]

Given positive integer weights for each row fills in sum of weights for each column (and slack).

Returns weights vector

Reimplemented from ClpMatrixBase.

4.59.3.19 virtual void ClpPackedMatrix::setDimensions (int *numrows*, int *numcols*) [virtual]

Set the dimensions of the matrix.

In effect, append new empty columns/rows to the matrix. A negative number for either dimension means that that dimension doesn't change. Otherwise the new dimensions MUST be at least as large as the current ones otherwise an exception is thrown.

Reimplemented from ClpMatrixBase.

4.59.3.20 virtual void ClpPackedMatrix::times (double *scalar*, const double * *x*, double * *y*) const [virtual]

Return `y + A * scalar *x` in `y`.

Precondition

`x` must be of size `numColumns()`
`y` must be of size `numRows()`

Reimplemented in ClpGubDynamicMatrix, and ClpDynamicMatrix.

4.59.3.21 virtual void ClpPackedMatrix::transposeTimes (double *scalar*, const double * *x*, double * *y*) const [virtual]

Return `y + x * scalar * A` in `y`.

Precondition

`x` must be of size `numRows()`
`y` must be of size `numColumns()`

4.59.3.22 void ClpPackedMatrix::transposeTimesSubset (int *number*, const int * *which*, const double * *pi*, double * *y*, const double * *rowScale*, const double * *columnScale*, double * *spare* = NULL) const

Return `y - pi * A` in `y`.

Precondition

`pi` must be of size `numRows()`
`y` must be of size `numColumns()` This just does subset (but puts in correct place in y)

4.59.3.23 virtual void ClpPackedMatrix::transposeTimes (const ClpSimplex * *model*, double *scalar*, const CoinIndexedVector * *x*, CoinIndexedVector * *y*, CoinIndexedVector * *z*) const [virtual]

Return `x * scalar * A + y` in `z`.

Can use y as temporary array (will be empty at end) Note - If x packed mode - then z packed mode Squashes small elements and knows about ClpSimplex

Implements ClpMatrixBase.

Reimplemented in ClpGubMatrix.

4.59.3.24 void ClpPackedMatrix::transposeTimesByColumn (const ClpSimplex * *model*, double *scalar*, const CoinIndexedVector * *x*, CoinIndexedVector * *y*, CoinIndexedVector * *z*) const

Return `x * scalar * A + y` in `z`.

Note - If x packed mode - then z packed mode This does by column and knows no gaps Squashes small elements and knows about ClpSimplex

4.59.3.25 virtual void ClpPackedMatrix::transposeTimesByRow (const ClpSimplex * *model*, double *scalar*, const CoinIndexedVector * *x*, CoinIndexedVector * *y*, CoinIndexedVector * *z*) const [virtual]

Return `x * scalar * A + y` in `z`.

Can use y as temporary array (will be empty at end) Note - If x packed mode - then z packed mode Squashes small elements and knows about ClpSimplex. This version uses row copy

Reimplemented in ClpGubMatrix.

4.59.3.26 virtual void ClpPackedMatrix::subsetTransposeTimes (const ClpSimplex * *model*, const CoinIndexedVector * *x*, const CoinIndexedVector * *y*, CoinIndexedVector * *z*) const [virtual]

Return `x *A` in `z` but just for indices in y.

Note - z always packed mode

Implements ClpMatrixBase.

Reimplemented in ClpGubMatrix.

4.59.3.27 void ClpPackedMatrix::setMatrixNull ( ) [inline]

Just sets matrix_ to NULL so it can be used elsewhere.

used in GUB

Definition at line 302 of file ClpPackedMatrix.hpp.

4.59.3.28 virtual ClpMatrixBase∗ ClpPackedMatrix::subsetClone (int *numberRows*, const int ∗ *whichRows*, int *numberColumns*, const int ∗ *whichColumns*) const [virtual]

Subset clone (without gaps).

Duplicates are allowed and order is as given

Reimplemented from ClpMatrixBase.

Reimplemented in ClpGubMatrix.

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

- ClpPackedMatrix.hpp

4.60 ClpPackedMatrix2 Class Reference

Public Member Functions

Useful methods

- void transposeTimes (const ClpSimplex ∗model, const **CoinPackedMatrix** ∗rowCopy, const **CoinIndexedVector** ∗x, **CoinIndexedVector** ∗spareArray, **CoinIndexedVector** ∗z) const

    Return `x * -1 * A in z.`
- bool usefulInfo () const

    Returns true if copy has useful information.

Constructors, destructor

- ClpPackedMatrix2 ()

    Default constructor.
- ClpPackedMatrix2 (ClpSimplex ∗model, const **CoinPackedMatrix** ∗rowCopy)

    Constructor from copy.
- virtual ∼ClpPackedMatrix2 ()

    Destructor.

Copy method

- ClpPackedMatrix2 (const ClpPackedMatrix2 &)

    The copy constructor.
- ClpPackedMatrix2 & **operator=** (const ClpPackedMatrix2 &)

Protected Attributes

Data members

The data members are protected to allow access for derived classes.

- int numberBlocks_

    Number of blocks.

- int numberRows_
  Number of rows.
- int * offset_
  Column offset for each block (plus one at end)
- unsigned short * count_
  Counts of elements in each part of row.
- CoinBigIndex * rowStart_
  Row starts.
- unsigned short * column_
  columns within block
- double * work_
  work arrays

4.60.1 Detailed Description

Definition at line 500 of file ClpPackedMatrix.hpp.

4.60.2 Constructor & Destructor Documentation

4.60.2.1 ClpPackedMatrix2::ClpPackedMatrix2 ( )

Default constructor.

4.60.2.2 ClpPackedMatrix2::ClpPackedMatrix2 (**ClpSimplex** * *model*, const **CoinPackedMatrix** * *rowCopy*)

Constructor from copy.

4.60.2.3 ClpPackedMatrix2::ClpPackedMatrix2 (const **ClpPackedMatrix2** &)

The copy constructor.

4.60.3 Member Function Documentation

4.60.3.1 void ClpPackedMatrix2::transposeTimes (const **ClpSimplex** * *model*, const **CoinPackedMatrix** * *rowCopy*, const **CoinIndexedVector** * *x*, **CoinIndexedVector** * *spareArray*, **CoinIndexedVector** * *z*) const

Return `x * -1 * A in z.`

Note - x packed and z will be packed mode Squashes small elements and knows about ClpSimplex

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

- ClpPackedMatrix.hpp

4.61 ClpPackedMatrix3 Class Reference

Collaboration diagram for ClpPackedMatrix3:

Public Member Functions

Useful methods

- void transposeTimes (const ClpSimplex *model, const double *pi, **CoinIndexedVector** *output) const

  Return `x * -1 * A in z.`
- void transposeTimes2 (const ClpSimplex *model, const double *pi, **CoinIndexedVector** *dj1, const double *piWeight, double referenceIn, double devex, unsigned int *reference, double *weights, double scaleFactor)

  Updates two arrays for steepest.

Constructors, destructor

- ClpPackedMatrix3 ()

  Default constructor.
- ClpPackedMatrix3 (ClpSimplex *model, const **CoinPackedMatrix** *columnCopy)

  Constructor from copy.
- virtual ~ClpPackedMatrix3 ()

  Destructor.

Copy method

- ClpPackedMatrix3 (const ClpPackedMatrix3 &)

  The copy constructor.
- ClpPackedMatrix3 & **operator=** (const ClpPackedMatrix3 &)

Sort methods

- void sortBlocks (const ClpSimplex *model)

  Sort blocks.
- void swapOne (const ClpSimplex *model, const ClpPackedMatrix *matrix, int iColumn)

  Swap one variable.

Protected Attributes

Data members

The data members are protected to allow access for derived classes.

- int numberBlocks_

  Number of blocks.
- int numberColumns_

  Number of columns.
- int * column_

  Column indices and reverse lookup (within block)
- CoinBigIndex * start_

  Starts for odd/long vectors.
- int * row_

  Rows.
- double * element_

  Elements.
- blockStruct * block_

  Blocks (ordinary start at 0 and go to first block)

4.61.1 Detailed Description

Definition at line 569 of file ClpPackedMatrix.hpp.

4.61.2 Constructor & Destructor Documentation

4.61.2.1 ClpPackedMatrix3::ClpPackedMatrix3 ( )

Default constructor.

4.61.2.2 ClpPackedMatrix3::ClpPackedMatrix3 (**ClpSimplex** ∗ *model*, const **CoinPackedMatrix** ∗ *columnCopy*)

Constructor from copy.

4.61.2.3 ClpPackedMatrix3::ClpPackedMatrix3 (const **ClpPackedMatrix3** &)

The copy constructor.

4.61.3 Member Function Documentation

4.61.3.1 void ClpPackedMatrix3::transposeTimes (const **ClpSimplex** ∗ *model*, const double ∗ *pi*, **CoinIndexedVector** ∗ *output*) const

Return `x * -1 * A in z.`

Note - x packed and z will be packed mode Squashes small elements and knows about ClpSimplex

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

- ClpPackedMatrix.hpp

4.62 ClpPdco Class Reference

This solves problems in Primal Dual Convex Optimization.

`#include <ClpPdco.hpp>`

Inheritance diagram for ClpPdco:

Collaboration diagram for ClpPdco:

Public Member Functions

Description of algorithm

- int pdco ()
  Pdco algorithm.
- int **pdco** (ClpPdcoBase ∗stuff, Options &options, Info &info, Outfo &outfo)

Functions used in pdco

- void lsqr ()

  LSQR.
- void **matVecMult** (int, double ∗, double ∗)
- void **matVecMult** (int, **CoinDenseVector**< double > &, double ∗)
- void **matVecMult** (int, **CoinDenseVector**< double > &, **CoinDenseVector**< double > &)
- void **matVecMult** (int, **CoinDenseVector**< double > ∗, **CoinDenseVector**< double > ∗)
- void **getBoundTypes** (int ∗, int ∗, int ∗, int ∗∗)
- void **getGrad** (**CoinDenseVector**< double > &x, **CoinDenseVector**< double > &grad)
- void **getHessian** (**CoinDenseVector**< double > &x, **CoinDenseVector**< double > &H)
- double **getObj** (**CoinDenseVector**< double > &x)
- void **matPrecon** (double, double ∗, double ∗)
- void **matPrecon** (double, **CoinDenseVector**< double > &, double ∗)
- void **matPrecon** (double, **CoinDenseVector**< double > &, **CoinDenseVector**< double > &)
- void **matPrecon** (double, **CoinDenseVector**< double > ∗, **CoinDenseVector**< double > ∗)

Additional Inherited Members

4.62.1 Detailed Description

This solves problems in Primal Dual Convex Optimization.

It inherits from ClpInterior. It has no data of its own and is never created - only cast from a ClpInterior object at algorithm time.

Definition at line 22 of file ClpPdco.hpp.

4.62.2 Member Function Documentation

4.62.2.1 int ClpPdco::pdco ( )

Pdco algorithm.

Method

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

- ClpPdco.hpp

4.63 ClpPdcoBase Class Reference

Abstract base class for tailoring everything for Pcdo.

`#include <ClpPdcoBase.hpp>`

Public Member Functions

Virtual methods that the derived classes must provide

- virtual void **matVecMult** (ClpInterior ∗model, int mode, double ∗x, double ∗y) const =0
- virtual void **getGrad** (ClpInterior ∗model, **CoinDenseVector**< double > &x, **CoinDenseVector**< double > &grad) const =0
- virtual void **getHessian** (ClpInterior ∗model, **CoinDenseVector**< double > &x, **CoinDenseVector**< double > &H) const =0

- virtual double **getObj** (ClpInterior *model, **CoinDenseVector**< double > &x) const =0
- virtual void **matPrecon** (ClpInterior *model, double delta, double *x, double *y) const =0

Other

Clone

- virtual ClpPdcoBase * **clone** () const =0
- int type () const
  Returns type.
- void setType (int type)
  Sets type.
- int sizeD1 () const
  Returns size of d1.
- double getD1 () const
  Returns d1 as scalar.
- int sizeD2 () const
  Returns size of d2.
- double getD2 () const
  Returns d2 as scalar.

Protected Attributes

Data members

The data members are protected to allow access for derived classes.

- double d1_
  Should be dense vectors.
- double **d2_**
- int type_
  type (may be useful)

Constructors, destructor

NOTE: All constructors are protected.

There's no need to expose them, after all, this is an abstract class.

- virtual ~ClpPdcoBase ()
  Destructor (has to be public)
- ClpPdcoBase ()
  Default constructor.
- **ClpPdcoBase** (const ClpPdcoBase &)
- ClpPdcoBase & **operator=** (const ClpPdcoBase &)

4.63.1 Detailed Description

Abstract base class for tailoring everything for Pcdo.

Since this class is abstract, no object of this type can be created.

If a derived class provides all methods then all ClpPcdo algorithms should work.

Eventually we should be able to use ClpObjective and ClpMatrixBase.

Definition at line 25 of file ClpPdcoBase.hpp.

4.63.2 Constructor & Destructor Documentation

4.63.2.1 ClpPdcoBase::ClpPdcoBase ( ) [protected]

Default constructor.

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

- ClpPdcoBase.hpp

4.64 ClpPlusMinusOneMatrix Class Reference

This implements a simple +- one matrix as derived from ClpMatrixBase.

`#include <ClpPlusMinusOneMatrix.hpp>`

Inheritance diagram for ClpPlusMinusOneMatrix:

Collaboration diagram for ClpPlusMinusOneMatrix:

Public Member Functions

Useful methods

- virtual **CoinPackedMatrix** ∗ getPackedMatrix () const
  Return a complete **CoinPackedMatrix***.*
- virtual bool isColOrdered () const
  Whether the packed matrix is column major ordered or not.
- virtual CoinBigIndex getNumElements () const
  Number of entries in the packed matrix.
- virtual int getNumCols () const
  Number of columns.
- virtual int getNumRows () const
  Number of rows.
- virtual const double ∗ getElements () const
  A vector containing the elements in the packed matrix.
- virtual const int ∗ getIndices () const
  A vector containing the minor indices of the elements in the packed matrix.
- int ∗ **getMutableIndices** () const
- virtual const CoinBigIndex ∗ **getVectorStarts** () const
- virtual const int ∗ getVectorLengths () const
  The lengths of the major-dimension vectors.
- virtual void deleteCols (const int numDel, const int ∗indDel)
  Delete the columns whose indices are listed in `indDel`*.*
- virtual void deleteRows (const int numDel, const int ∗indDel)
  Delete the rows whose indices are listed in `indDel`*.*
- virtual void appendCols (int number, const **CoinPackedVectorBase** ∗const ∗columns)
  Append Columns.
- virtual void appendRows (int number, const **CoinPackedVectorBase** ∗const ∗rows)
  Append Rows.
- virtual int appendMatrix (int number, int type, const CoinBigIndex ∗starts, const int ∗index, const double ∗element, int numberOther=-1)
  Append a set of rows/columns to the end of the matrix.
- virtual ClpMatrixBase ∗ reverseOrderedCopy () const

Returns a new matrix in reverse order without gaps.
- virtual CoinBigIndex countBasis (const int ∗whichColumn, int &numberColumnBasic)

  Returns number of elements in column part of basis.
- virtual void fillBasis (ClpSimplex ∗model, const int ∗whichColumn, int &numberColumnBasic, int ∗row, int ∗start, int ∗rowCount, int ∗columnCount, CoinFactorizationDouble ∗element)

  Fills in column part of basis.
- virtual CoinBigIndex ∗ dubiousWeights (const ClpSimplex ∗model, int ∗inputWeights) const

  Given positive integer weights for each row fills in sum of weights for each column (and slack).
- virtual void rangeOfElements (double &smallestNegative, double &largestNegative, double &smallestPositive, double &largestPositive)

  Returns largest and smallest elements of both signs.
- virtual void unpack (const ClpSimplex ∗model, **CoinIndexedVector** ∗rowArray, int column) const

  Unpacks a column into an CoinIndexedvector.
- virtual void unpackPacked (ClpSimplex ∗model, **CoinIndexedVector** ∗rowArray, int column) const

  Unpacks a column into an CoinIndexedvector in packed foramt Note that model is NOT const.
- virtual void add (const ClpSimplex ∗model, **CoinIndexedVector** ∗rowArray, int column, double multiplier) const

  Adds multiple of a column into an CoinIndexedvector You can use quickAdd to add to vector.
- virtual void add (const ClpSimplex ∗model, double ∗array, int column, double multiplier) const

  Adds multiple of a column into an array.
- virtual void releasePackedMatrix () const

  Allow any parts of a created CoinMatrix to be deleted.
- virtual void setDimensions (int numrows, int numcols)

  Set the dimensions of the matrix.
- void checkValid (bool detail) const

  Just checks matrix valid - will say if dimensions not quite right if detail.

Matrix times vector methods

- virtual void times (double scalar, const double ∗x, double ∗y) const

  Return $y + A * scalar *x$ *in* y.
- virtual void times (double scalar, const double ∗x, double ∗y, const double ∗rowScale, const double ∗column↩ Scale) const

  And for scaling.
- virtual void transposeTimes (double scalar, const double ∗x, double ∗y) const

  Return $y + x * scalar * A$ *in* y.
- virtual void transposeTimes (double scalar, const double ∗x, double ∗y, const double ∗rowScale, const double ∗columnScale, double ∗spare=NULL) const

  And for scaling.
- virtual void transposeTimes (const ClpSimplex ∗model, double scalar, const **CoinIndexedVector** ∗x, **Coin↩ IndexedVector** ∗y, **CoinIndexedVector** ∗z) const

  Return $x * scalar * A + y$ *in* z.
- virtual void transposeTimesByRow (const ClpSimplex ∗model, double scalar, const **CoinIndexedVector** ∗x, **CoinIndexedVector** ∗y, **CoinIndexedVector** ∗z) const

  Return $x * scalar * A + y$ *in* z.
- virtual void subsetTransposeTimes (const ClpSimplex ∗model, const **CoinIndexedVector** ∗x, const **Coin↩ IndexedVector** ∗y, **CoinIndexedVector** ∗z) const

  Return $x *A$ *in* z *but just for indices in y.*
- virtual bool canCombine (const ClpSimplex ∗model, const **CoinIndexedVector** ∗pi) const

  Returns true if can combine transposeTimes and subsetTransposeTimes and if it would be faster.
- virtual void transposeTimes2 (const ClpSimplex ∗model, const **CoinIndexedVector** ∗pi1, **CoinIndexedVector** ∗dj1, const **CoinIndexedVector** ∗pi2, **CoinIndexedVector** ∗spare, double referenceIn, double devex, unsigned int ∗reference, double ∗weights, double scaleFactor)

Updates two arrays for steepest.

- virtual void subsetTimes2 (const ClpSimplex ∗model, **CoinIndexedVector** ∗dj1, const **CoinIndexedVector** ∗pi2, **CoinIndexedVector** ∗dj2, double referenceIn, double devex, unsigned int ∗reference, double ∗weights, double scaleFactor)

  Updates second array for steepest and does devex weights.

Other

- CoinBigIndex ∗ startPositive () const

  Return starts of +1s.
- CoinBigIndex ∗ startNegative () const

  Return starts of -1s.

Constructors, destructor

- ClpPlusMinusOneMatrix ()

  Default constructor.
- virtual ∼ClpPlusMinusOneMatrix ()

  Destructor.

Copy method

- ClpPlusMinusOneMatrix (const ClpPlusMinusOneMatrix &)

  The copy constructor.
- ClpPlusMinusOneMatrix (const **CoinPackedMatrix** &)

  The copy constructor from an CoinPlusMinusOneMatrix.
- ClpPlusMinusOneMatrix (int numberRows, int numberColumns, bool columnOrdered, const int ∗indices, const CoinBigIndex ∗startPositive, const CoinBigIndex ∗startNegative)

  Constructor from arrays.
- ClpPlusMinusOneMatrix (const ClpPlusMinusOneMatrix &wholeModel, int numberRows, const int ∗which←Rows, int numberColumns, const int ∗whichColumns)

  Subset constructor (without gaps).
- ClpPlusMinusOneMatrix & **operator=** (const ClpPlusMinusOneMatrix &)
- virtual ClpMatrixBase ∗ clone () const

  Clone.
- virtual ClpMatrixBase ∗ subsetClone (int numberRows, const int ∗whichRows, int numberColumns, const int ∗whichColumns) const

  Subset clone (without gaps).
- void passInCopy (int numberRows, int numberColumns, bool columnOrdered, int ∗indices, CoinBigIndex ∗startPositive, CoinBigIndex ∗startNegative)

  pass in copy (object takes ownership)
- virtual bool canDoPartialPricing () const

  Says whether it can do partial pricing.
- virtual void partialPricing (ClpSimplex ∗model, double start, double end, int &bestSequence, int &number←Wanted)

  Partial pricing.

Protected Attributes

Data members

The data members are protected to allow access for derived classes.

- **CoinPackedMatrix** ∗ matrix_

For fake **CoinPackedMatrix***.*

- int ∗ **lengths_**
- CoinBigIndex ∗COIN_RESTRICT startPositive_

  Start of +1's for each.
- CoinBigIndex ∗COIN_RESTRICT startNegative_

  Start of -1's for each.
- int ∗COIN_RESTRICT indices_

  Data -1, then +1 rows in pairs (row==-1 if one entry)
- int numberRows_

  Number of rows.
- int numberColumns_

  Number of columns.
- bool columnOrdered_

  True if column ordered.

Additional Inherited Members

4.64.1 Detailed Description

This implements a simple +- one matrix as derived from ClpMatrixBase.

Definition at line 18 of file ClpPlusMinusOneMatrix.hpp.

4.64.2 Constructor & Destructor Documentation

4.64.2.1 ClpPlusMinusOneMatrix::ClpPlusMinusOneMatrix ( )

Default constructor.

4.64.2.2 ClpPlusMinusOneMatrix::ClpPlusMinusOneMatrix (const ClpPlusMinusOneMatrix &)

The copy constructor.

4.64.2.3 ClpPlusMinusOneMatrix::ClpPlusMinusOneMatrix (const CoinPackedMatrix &)

The copy constructor from an CoinPlusMinusOneMatrix.

If not a valid matrix then getIndices will be NULL and startPositive[0] will have number of +1, startPositive[1] will have number of -1, startPositive[2] will have number of others,

4.64.2.4 ClpPlusMinusOneMatrix::ClpPlusMinusOneMatrix (const ClpPlusMinusOneMatrix & *wholeModel,* int *numberRows,* const int ∗ *whichRows,* int *numberColumns,* const int ∗ *whichColumns*)

Subset constructor (without gaps).

Duplicates are allowed and order is as given

4.64.3 Member Function Documentation

4.64.3.1 virtual bool ClpPlusMinusOneMatrix::isColOrdered ( ) const [virtual]

Whether the packed matrix is column major ordered or not.

Implements ClpMatrixBase.

4.64.3.2 virtual CoinBigIndex ClpPlusMinusOneMatrix::getNumElements ( ) const [virtual]

Number of entries in the packed matrix.

Implements ClpMatrixBase.

4.64.3.3 virtual int ClpPlusMinusOneMatrix::getNumCols ( ) const [inline], [virtual]

Number of columns.

Implements ClpMatrixBase.

Definition at line 30 of file ClpPlusMinusOneMatrix.hpp.

4.64.3.4 virtual int ClpPlusMinusOneMatrix::getNumRows ( ) const [inline], [virtual]

Number of rows.

Implements ClpMatrixBase.

Definition at line 34 of file ClpPlusMinusOneMatrix.hpp.

4.64.3.5 virtual const double* ClpPlusMinusOneMatrix::getElements ( ) const [virtual]

A vector containing the elements in the packed matrix.

Note that there might be gaps in this list, entries that do not belong to any major-dimension vector. To get the actual elements one should look at this vector together with vectorStarts and vectorLengths.

Implements ClpMatrixBase.

4.64.3.6 virtual const int* ClpPlusMinusOneMatrix::getIndices ( ) const [inline], [virtual]

A vector containing the minor indices of the elements in the packed matrix.

Note that there might be gaps in this list, entries that do not belong to any major-dimension vector. To get the actual elements one should look at this vector together with vectorStarts and vectorLengths.

Implements ClpMatrixBase.

Definition at line 48 of file ClpPlusMinusOneMatrix.hpp.

4.64.3.7 virtual const int* ClpPlusMinusOneMatrix::getVectorLengths ( ) const [virtual]

The lengths of the major-dimension vectors.

Implements ClpMatrixBase.

4.64.3.8 virtual void ClpPlusMinusOneMatrix::deleteCols (const int *numDel*, const int ∗ *indDel*) [virtual]

Delete the columns whose indices are listed in `indDel`.

Implements ClpMatrixBase.

4.64.3.9 virtual void ClpPlusMinusOneMatrix::deleteRows (const int *numDel*, const int ∗ *indDel*) [virtual]

Delete the rows whose indices are listed in `indDel`.

Implements ClpMatrixBase.

4.64.3.10 virtual int ClpPlusMinusOneMatrix::appendMatrix (int *number*, int *type*, const CoinBigIndex ∗ *starts*, const int ∗ *index*, const double ∗ *element*, int *numberOther* = -1) [virtual]

Append a set of rows/columns to the end of the matrix.

Returns number of errors i.e. if any of the new rows/columns contain an index that's larger than the number of columns-1/rows-1 (if numberOther>0) or duplicates If 0 then rows, 1 if columns

Reimplemented from ClpMatrixBase.

4.64.3.11 virtual CoinBigIndex∗ ClpPlusMinusOneMatrix::dubiousWeights (const ClpSimplex ∗ *model*, int ∗ *inputWeights*) const [virtual]

Given positive integer weights for each row fills in sum of weights for each column (and slack).

Returns weights vector

Reimplemented from ClpMatrixBase.

4.64.3.12 virtual void ClpPlusMinusOneMatrix::rangeOfElements (double & *smallestNegative*, double & *largestNegative*, double & *smallestPositive*, double & *largestPositive*) [virtual]

Returns largest and smallest elements of both signs.

Largest refers to largest absolute value.

Reimplemented from ClpMatrixBase.

4.64.3.13 virtual void ClpPlusMinusOneMatrix::unpackPacked (ClpSimplex ∗ *model*, CoinIndexedVector ∗ *rowArray*, int *column*) const [virtual]

Unpacks a column into an CoinIndexedvector in packed foramt Note that model is NOT const.

Bounds and objective could be modified if doing column generation (just for this variable)

Implements ClpMatrixBase.

4.64.3.14 virtual void ClpPlusMinusOneMatrix::setDimensions (int *numrows*, int *numcols*) [virtual]

Set the dimensions of the matrix.

In effect, append new empty columns/rows to the matrix. A negative number for either dimension means that that dimension doesn't change. Otherwise the new dimensions MUST be at least as large as the current ones otherwise an exception is thrown.

Reimplemented from ClpMatrixBase.

4.64.3.15 virtual void ClpPlusMinusOneMatrix::times (double *scalar*, const double ∗ *x*, double ∗ *y*) const [virtual]

Return `y + A * scalar *x` in `y`.

Precondition

`x` must be of size `numColumns()`
`y` must be of size `numRows()`

4.64.3.16 virtual void ClpPlusMinusOneMatrix::transposeTimes (double *scalar*, const double ∗ *x*, double ∗ *y*) const [virtual]

Return `y + x * scalar * A` in `y`.

Precondition

`x` must be of size `numRows()`
`y` must be of size `numColumns()`

4.64.3.17 virtual void ClpPlusMinusOneMatrix::transposeTimes (const ClpSimplex ∗ *model*, double *scalar*, const CoinIndexedVector ∗ *x*, CoinIndexedVector ∗ *y*, CoinIndexedVector ∗ *z*) const [virtual]

Return `x * scalar * A + y` in `z`.

Can use y as temporary array (will be empty at end) Note - If x packed mode - then z packed mode Squashes small elements and knows about ClpSimplex

Implements ClpMatrixBase.

4.64.3.18 virtual void ClpPlusMinusOneMatrix::transposeTimesByRow (const ClpSimplex ∗ *model*, double *scalar*, const CoinIndexedVector ∗ *x*, CoinIndexedVector ∗ *y*, CoinIndexedVector ∗ *z*) const [virtual]

Return `x * scalar * A + y` in `z`.

Can use y as temporary array (will be empty at end) Note - If x packed mode - then z packed mode Squashes small elements and knows about ClpSimplex. This version uses row copy

4.64.3.19 virtual void ClpPlusMinusOneMatrix::subsetTransposeTimes (const ClpSimplex ∗ *model*, const CoinIndexedVector ∗ *x*, const CoinIndexedVector ∗ *y*, CoinIndexedVector ∗ *z*) const [virtual]

Return `x *A` in `z` but just for indices in y.

Note - z always packed mode

Implements ClpMatrixBase.

4.64.3.20 virtual **ClpMatrixBase**∗ ClpPlusMinusOneMatrix::subsetClone (int *numberRows,* const int ∗ *whichRows,* int *numberColumns,* const int ∗ *whichColumns*) const [virtual]

Subset clone (without gaps).

Duplicates are allowed and order is as given

Reimplemented from ClpMatrixBase.

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

- ClpPlusMinusOneMatrix.hpp

4.65 ClpPredictorCorrector Class Reference

This solves LPs using the predictor-corrector method due to Mehrotra.

`#include <ClpPredictorCorrector.hpp>`

Inheritance diagram for ClpPredictorCorrector:

Collaboration diagram for ClpPredictorCorrector:

Public Member Functions

Description of algorithm

- int solve ()
  Primal Dual Predictor Corrector algorithm.

Functions used in algorithm

- CoinWorkDouble findStepLength (int phase)
  findStepLength.
- CoinWorkDouble findDirectionVector (const int phase)
  findDirectionVector.
- int createSolution ()
  createSolution. Creates solution from scratch (- code if no memory)
- CoinWorkDouble complementarityGap (int &numberComplementarityPairs, int &numberComplementarityItems, const int phase)
  complementarityGap. Computes gap
- void setupForSolve (const int phase)
  setupForSolve.
- void solveSystem (CoinWorkDouble ∗region1, CoinWorkDouble ∗region2, const CoinWorkDouble ∗region1In, const CoinWorkDouble ∗region2In, const CoinWorkDouble ∗saveRegion1, const CoinWorkDouble ∗saveRegion2, bool gentleRefine)
  Does solve.
- bool checkGoodMove (const bool doCorrector, CoinWorkDouble &bestNextGap, bool allowIncreasingGap)
  sees if looks plausible change in complementarity
- bool checkGoodMove2 (CoinWorkDouble move, CoinWorkDouble &bestNextGap, bool allowIncreasingGap)
  : checks for one step size
- int updateSolution (CoinWorkDouble nextGap)
  updateSolution. Updates solution at end of iteration
- CoinWorkDouble affineProduct ()
  Save info on products of affine deltaT∗deltaW and deltaS∗deltaZ.
- void debugMove (int phase, CoinWorkDouble primalStep, CoinWorkDouble dualStep)
  See exactly what would happen given current deltas.

Additional Inherited Members

4.65.1 Detailed Description

This solves LPs using the predictor-corrector method due to Mehrotra.

It also uses multiple centrality corrections as in Gondzio.

See; S. Mehrotra, "On the implementation of a primal-dual interior point method", SIAM Journal on optimization, 2 (1992) J. Gondzio, "Multiple centraility corrections in a primal-dual method for linear programming", Computational Optimization and Applications",6 (1996)

It is rather basic as Interior point is not my speciality

It inherits from ClpInterior. It has no data of its own and is never created - only cast from a ClpInterior object at algorithm time.

It can also solve QPs

Definition at line 37 of file ClpPredictorCorrector.hpp.

4.65.2 Member Function Documentation

4.65.2.1 int ClpPredictorCorrector::solve ( )

Primal Dual Predictor Corrector algorithm.

Method

Big TODO

4.65.2.2 void ClpPredictorCorrector::solveSystem (CoinWorkDouble ∗ *region1,* CoinWorkDouble ∗ *region2,* const CoinWorkDouble ∗ *region1In,* const CoinWorkDouble ∗ *region2In,* const CoinWorkDouble ∗ *saveRegion1,* const CoinWorkDouble ∗ *saveRegion2,* bool *gentleRefine*)

Does solve.

region1 is for deltaX (columns+rows), region2 for deltaPi (rows)

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

- ClpPredictorCorrector.hpp

4.66 ClpPresolve Class Reference

This is the Clp interface to CoinPresolve.

`#include <ClpPresolve.hpp>`

Public Member Functions

Main Constructor, destructor

- ClpPresolve ()
  Default constructor.

- virtual ∼ClpPresolve ()

  Virtual destructor.

presolve - presolves a model, transforming the model

and saving information in the ClpPresolve object needed for postsolving.

This underlying (protected) method is virtual; the idea is that in the future, one could override this method to customize how the various presolve techniques are applied.

This version of presolve returns a pointer to a new presolved model. NULL if infeasible or unbounded. This should be paired with postsolve below. The advantage of going back to original model is that it will be exactly as it was i.e. 0.0 will not become 1.0e-19. If keepIntegers is true then bounds may be tightened in original. Bounds will be moved by up to feasibilityTolerance to try and stay feasible. Names will be dropped in presolved model if asked

- ClpSimplex ∗ **presolvedModel** (ClpSimplex &si, double feasibilityTolerance=0.0, bool keepIntegers=true, int numberPasses=5, bool dropNames=false, bool doRowObjective=false, const char ∗prohibitedRows=NULL, const char ∗prohibitedColumns=NULL)
- int presolvedModelToFile (ClpSimplex &si, std::string fileName, double feasibilityTolerance=0.0, bool keepIntegers=true, int numberPasses=5, bool dropNames=false, bool doRowObjective=false)

  This version saves data in a file.
- ClpSimplex ∗ model () const

  Return pointer to presolved model, Up to user to destroy.
- ClpSimplex ∗ originalModel () const

  Return pointer to original model.
- void setOriginalModel (ClpSimplex ∗model)

  Set pointer to original model.
- const int ∗ originalColumns () const

  return pointer to original columns
- const int ∗ originalRows () const

  return pointer to original rows
- void setNonLinearValue (double value)

  "Magic" number.
- double **nonLinearValue** () const
- bool doDual () const

  Whether we want to do dual part of presolve.
- void **setDoDual** (bool doDual)
- bool doSingleton () const

  Whether we want to do singleton part of presolve.
- void **setDoSingleton** (bool doSingleton)
- bool doDoubleton () const

  Whether we want to do doubleton part of presolve.
- void **setDoDoubleton** (bool doDoubleton)
- bool doTripleton () const

  Whether we want to do tripleton part of presolve.
- void **setDoTripleton** (bool doTripleton)
- bool doTighten () const

  Whether we want to do tighten part of presolve.
- void **setDoTighten** (bool doTighten)
- bool doForcing () const

  Whether we want to do forcing part of presolve.
- void **setDoForcing** (bool doForcing)
- bool doImpliedFree () const

  Whether we want to do impliedfree part of presolve.
- void **setDoImpliedFree** (bool doImpliedfree)
- bool doDupcol () const

Whether we want to do dupcol part of presolve.
- void **setDoDupcol** (bool doDupcol)
- bool doDuprow () const

  Whether we want to do duprow part of presolve.
- void **setDoDuprow** (bool doDuprow)
- bool doDependency () const

  Whether we want to do dependency part of presolve.
- void **setDoDependency** (bool doDependency)
- bool doSingletonColumn () const

  Whether we want to do singleton column part of presolve.
- void **setDoSingletonColumn** (bool doSingleton)
- bool doGubrow () const

  Whether we want to do gubrow part of presolve.
- void **setDoGubrow** (bool doGubrow)
- bool doTwoxTwo () const

  Whether we want to do twoxtwo part of presolve.
- void **setDoTwoxtwo** (bool doTwoxTwo)
- bool doIntersection () const

  Whether we want to allow duplicate intersections.
- void **setDoIntersection** (bool doIntersection)
- int zeroSmall () const

  How much we want to zero small values from aggregation - ratio 0 - 1.0e-12, 1 1.0e-11, 2 1.0e-10, 3 1.0e-9.
- void **setZeroSmall** (int value)
- int presolveActions () const

  Set whole group.
- void **setPresolveActions** (int action)
- void setSubstitution (int value)

  Substitution level.
- void statistics ()

  Asks for statistics.
- int presolveStatus () const

  Return presolve status (0,1,2)

postsolve - postsolve the problem. If the problem

has not been solved to optimality, there are no guarantees.

If you are using an algorithm like simplex that has a concept of "basic" rows/cols, then set updateStatus

Note that if you modified the original problem after presolving, then you must "undo" these modifications before calling postsolve. This version updates original

- virtual void **postsolve** (bool updateStatus=true)
- void destroyPresolve ()

  Gets rid of presolve actions (e.g.when infeasible)

private or protected data

- virtual const **CoinPresolveAction** ∗ presolve (**CoinPresolveMatrix** ∗prob)

  If you want to apply the individual presolve routines differently, or perhaps add your own to the mix, define a derived class and override this method.
- virtual void postsolve (**CoinPostsolveMatrix** &prob)

  Postsolving is pretty generic; just apply the transformations in reverse order.
- virtual ClpSimplex ∗ gutsOfPresolvedModel (ClpSimplex ∗originalModel, double feasibilityTolerance, bool keep↩ Integers, int numberPasses, bool dropNames, bool doRowObjective, const char ∗prohibitedRows=NULL, const char ∗prohibitedColumns=NULL)

  This is main part of Presolve.

4.66.1 Detailed Description

This is the Clp interface to CoinPresolve.

Definition at line 15 of file ClpPresolve.hpp.

4.66.2 Member Function Documentation

4.66.2.1 int ClpPresolve::presolvedModelToFile (**ClpSimplex** & *si*, std::string *fileName*, double *feasibilityTolerance* = `0.0`, bool *keepIntegers* = `true`, int *numberPasses* = `5`, bool *dropNames* = `false`, bool *doRowObjective* = `false`)

This version saves data in a file.

The passed in model is updated to be presolved model. Returns non-zero if infeasible

4.66.2.2 void ClpPresolve::setNonLinearValue (double *value*) `[inline]`

"Magic" number.

If this is non-zero then any elements with this value may change and so presolve is very limited in what can be done to the row and column. This is for non-linear problems.

Definition at line 76 of file ClpPresolve.hpp.

4.66.2.3 virtual void ClpPresolve::postsolve (**CoinPostsolveMatrix** & *prob*) `[protected], [virtual]`

Postsolving is pretty generic; just apply the transformations in reverse order.

You will probably only be interested in overriding this method if you want to add code to test for consistency while debugging new presolve techniques.

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

- ClpPresolve.hpp

4.67 ClpPrimalColumnDantzig Class Reference

Primal Column Pivot Dantzig Algorithm Class.

`#include <ClpPrimalColumnDantzig.hpp>`

Inheritance diagram for ClpPrimalColumnDantzig:

Collaboration diagram for ClpPrimalColumnDantzig:

Public Member Functions

Algorithmic methods

- virtual int pivotColumn (**CoinIndexedVector** ∗updates, **CoinIndexedVector** ∗spareRow1, **CoinIndexedVector** ∗spareRow2, **CoinIndexedVector** ∗spareColumn1, **CoinIndexedVector** ∗spareColumn2)

  Returns pivot column, -1 if none.
- virtual void saveWeights (ClpSimplex ∗model, int)

  Just sets model.

Constructors and destructors

- ClpPrimalColumnDantzig ()
  Default Constructor.
- ClpPrimalColumnDantzig (const ClpPrimalColumnDantzig &)
  Copy constructor.
- ClpPrimalColumnDantzig & operator= (const ClpPrimalColumnDantzig &rhs)
  Assignment operator.
- virtual ∼ClpPrimalColumnDantzig ()
  Destructor.
- virtual ClpPrimalColumnPivot ∗ clone (bool copyData=true) const
  Clone.

Additional Inherited Members

4.67.1 Detailed Description

Primal Column Pivot Dantzig Algorithm Class.

This is simplest choice - choose largest infeasibility

Definition at line 19 of file ClpPrimalColumnDantzig.hpp.

4.67.2 Member Function Documentation

4.67.2.1 virtual int ClpPrimalColumnDantzig::pivotColumn (CoinIndexedVector ∗ *updates*, CoinIndexedVector ∗ *spareRow1*, CoinIndexedVector ∗ *spareRow2*, CoinIndexedVector ∗ *spareColumn1*, CoinIndexedVector ∗ *spareColumn2*) `[virtual]`

Returns pivot column, -1 if none.

Lumbers over all columns - slow The Packed **CoinIndexedVector** updates has cost updates - for normal LP that is just +-weight where a feasibility changed. It also has reduced cost from last iteration in pivot row Can just do full price if you really want to be slow

Implements ClpPrimalColumnPivot.

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

- ClpPrimalColumnDantzig.hpp

4.68 ClpPrimalColumnPivot Class Reference

Primal Column Pivot Abstract Base Class.

`#include <ClpPrimalColumnPivot.hpp>`

Inheritance diagram for ClpPrimalColumnPivot:

Collaboration diagram for ClpPrimalColumnPivot:

Public Member Functions

Algorithmic methods

- virtual int pivotColumn (**CoinIndexedVector** ∗updates, **CoinIndexedVector** ∗spareRow1, **CoinIndexedVector** ∗spareRow2, **CoinIndexedVector** ∗spareColumn1, **CoinIndexedVector** ∗spareColumn2)=0
  Returns pivot column, -1 if none.
- virtual void updateWeights (**CoinIndexedVector** ∗input)
  Updates weights - part 1 (may be empty)
- virtual void saveWeights (ClpSimplex ∗model, int mode)=0
  Saves any weights round factorization as pivot rows may change Will be empty unless steepest edge (will save model) May also recompute infeasibility stuff 1) before factorization 2) after good factorization (if weights empty may initialize) 3) after something happened but no factorization (e.g.
- virtual int pivotRow (double &way)
  Signals pivot row choice: -2 (default) - use normal pivot row choice -1 to numberRows-1 - use this (will be checked) way should be -1 to go to lower bound, +1 to upper bound.
- virtual void clearArrays ()
  Gets rid of all arrays (may be empty)
- virtual bool looksOptimal () const
  Returns true if would not find any column.
- virtual void setLooksOptimal (bool flag)
  Sets optimality flag (for advanced use)

Constructors and destructors

- ClpPrimalColumnPivot ()
  Default Constructor.
- ClpPrimalColumnPivot (const ClpPrimalColumnPivot &)
  Copy constructor.
- ClpPrimalColumnPivot & operator= (const ClpPrimalColumnPivot &rhs)
  Assignment operator.
- virtual ∼ClpPrimalColumnPivot ()
  Destructor.
- virtual ClpPrimalColumnPivot ∗ clone (bool copyData=true) const =0
  Clone.

Other

- ClpSimplex ∗ model ()
  Returns model.
- void setModel (ClpSimplex ∗newmodel)
  Sets model.
- int type ()
  Returns type (above 63 is extra information)
- virtual int numberSprintColumns (int &numberIterations) const
  Returns number of extra columns for sprint algorithm - 0 means off.
- virtual void switchOffSprint ()
  Switch off sprint idea.
- virtual void maximumPivotsChanged ()
  Called when maximum pivots changes.

Protected Attributes

Protected member data

- ClpSimplex ∗ model_
  Pointer to model.
- int type_
  Type of column pivot algorithm.
- bool looksOptimal_
  Says if looks optimal (normally computed)

4.68.1 Detailed Description

Primal Column Pivot Abstract Base Class.

Abstract Base Class for describing an interface to an algorithm to choose column pivot in primal simplex algorithm. For some algorithms e.g. Dantzig choice then some functions may be null. For Dantzig the only one of any importance is pivotColumn.

If you wish to inherit from this look at ClpPrimalColumnDantzig.cpp as that is simplest version.

Definition at line 25 of file ClpPrimalColumnPivot.hpp.

4.68.2 Member Function Documentation

4.68.2.1 virtual int ClpPrimalColumnPivot::pivotColumn (**CoinIndexedVector** ∗ *updates,* **CoinIndexedVector** ∗ *spareRow1,* **CoinIndexedVector** ∗ *spareRow2,* **CoinIndexedVector** ∗ *spareColumn1,* **CoinIndexedVector** ∗ *spareColumn2*) `[pure virtual]`

Returns pivot column, -1 if none.

Normally updates reduced costs using result of last iteration before selecting incoming column.

The Packed **CoinIndexedVector** updates has cost updates - for normal LP that is just +-weight where a feasibility changed. It also has reduced cost from last iteration in pivot row

Inside pivotColumn the pivotRow_ and reduced cost from last iteration are also used.

So in the simplest case i.e. feasible we compute the row of the tableau corresponding to last pivot and add a multiple of this to current reduced costs.

We can use other arrays to help updates

Implemented in ClpPrimalColumnSteepest, ClpPrimalColumnDantzig, and ClpPrimalQuadraticDantzig.

4.68.2.2 virtual void ClpPrimalColumnPivot::saveWeights (**ClpSimplex** ∗ *model,* int *mode*) `[pure virtual]`

Saves any weights round factorization as pivot rows may change Will be empty unless steepest edge (will save model) May also recompute infeasibility stuff 1) before factorization 2) after good factorization (if weights empty may initialize) 3) after something happened but no factorization (e.g.

check for infeasible) 4) as 2 but restore weights from previous snapshot 5) forces some initialization e.g. weights Also sets model

Implemented in ClpPrimalColumnSteepest, ClpPrimalColumnDantzig, and ClpPrimalQuadraticDantzig.

4.68.2.3 virtual int ClpPrimalColumnPivot::numberSprintColumns (int & *numberIterations*) const `[virtual]`

Returns number of extra columns for sprint algorithm - 0 means off.

Also number of iterations before recompute

Reimplemented in ClpPrimalColumnSteepest.

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

- ClpPrimalColumnPivot.hpp

4.69 ClpPrimalColumnSteepest Class Reference

Primal Column Pivot Steepest Edge Algorithm Class.

`#include <ClpPrimalColumnSteepest.hpp>`

Inheritance diagram for ClpPrimalColumnSteepest:

Collaboration diagram for ClpPrimalColumnSteepest:

Public Types

- enum Persistence

  enums for persistence

Public Member Functions

Algorithmic methods

- virtual int pivotColumn (**CoinIndexedVector** ∗updates, **CoinIndexedVector** ∗spareRow1, **CoinIndexedVector** ∗spareRow2, **CoinIndexedVector** ∗spareColumn1, **CoinIndexedVector** ∗spareColumn2)

  Returns pivot column, -1 if none.
- int pivotColumnOldMethod (**CoinIndexedVector** ∗updates, **CoinIndexedVector** ∗spareRow1, **CoinIndexedVector** ∗spareRow2, **CoinIndexedVector** ∗spareColumn1, **CoinIndexedVector** ∗spareColumn2)

  For quadratic or funny nonlinearities.
- void justDjs (**CoinIndexedVector** ∗updates, **CoinIndexedVector** ∗spareRow2, **CoinIndexedVector** ∗spareColumn1, **CoinIndexedVector** ∗spareColumn2)

  Just update djs.
- int partialPricing (**CoinIndexedVector** ∗updates, **CoinIndexedVector** ∗spareRow2, int numberWanted, int numberLook)

  Update djs doing partial pricing (dantzig)
- void djsAndDevex (**CoinIndexedVector** ∗updates, **CoinIndexedVector** ∗spareRow2, **CoinIndexedVector** ∗spareColumn1, **CoinIndexedVector** ∗spareColumn2)

  Update djs, weights for Devex using djs.
- void djsAndSteepest (**CoinIndexedVector** ∗updates, **CoinIndexedVector** ∗spareRow2, **CoinIndexedVector** ∗spareColumn1, **CoinIndexedVector** ∗spareColumn2)

  Update djs, weights for Steepest using djs.
- void djsAndDevex2 (**CoinIndexedVector** ∗updates, **CoinIndexedVector** ∗spareRow2, **CoinIndexedVector** ∗spareColumn1, **CoinIndexedVector** ∗spareColumn2)

  Update djs, weights for Devex using pivot row.
- void djsAndSteepest2 (**CoinIndexedVector** ∗updates, **CoinIndexedVector** ∗spareRow2, **CoinIndexedVector** ∗spareColumn1, **CoinIndexedVector** ∗spareColumn2)

  Update djs, weights for Steepest using pivot row.
- void justDevex (**CoinIndexedVector** ∗updates, **CoinIndexedVector** ∗spareRow2, **CoinIndexedVector** ∗spareColumn1, **CoinIndexedVector** ∗spareColumn2)

  Update weights for Devex.
- void justSteepest (**CoinIndexedVector** ∗updates, **CoinIndexedVector** ∗spareRow2, **CoinIndexedVector** ∗spareColumn1, **CoinIndexedVector** ∗spareColumn2)

  Update weights for Steepest.
- void transposeTimes2 (const **CoinIndexedVector** ∗pi1, **CoinIndexedVector** ∗dj1, const **CoinIndexedVector** ∗pi2, **CoinIndexedVector** ∗dj2, **CoinIndexedVector** ∗spare, double scaleFactor)

  Updates two arrays for steepest.
- virtual void updateWeights (**CoinIndexedVector** ∗input)

Updates weights - part 1 - also checks accuracy.
- void checkAccuracy (int sequence, double relativeTolerance, **CoinIndexedVector** ∗rowArray1, **CoinIndexedVector** ∗rowArray2)

  Checks accuracy - just for debug.
- void initializeWeights ()

  Initialize weights.
- virtual void saveWeights (ClpSimplex ∗model, int mode)

  Save weights - this may initialize weights as well mode is - 1) before factorization 2) after factorization 3) just redo infeasibilities 4) restore weights 5) at end of values pass (so need initialization)
- virtual void unrollWeights ()

  Gets rid of last update.
- virtual void clearArrays ()

  Gets rid of all arrays.
- virtual bool looksOptimal () const

  Returns true if would not find any column.
- virtual void maximumPivotsChanged ()

  Called when maximum pivots changes.

gets and sets

- int mode () const

  Mode.
- virtual int numberSprintColumns (int &numberIterations) const

  Returns number of extra columns for sprint algorithm - 0 means off.
- virtual void switchOffSprint ()

  Switch off sprint idea.

Constructors and destructors

- ClpPrimalColumnSteepest (int mode=3)

  Default Constructor 0 is exact devex, 1 full steepest, 2 is partial exact devex 3 switches between 0 and 2 depending on factorization 4 starts as partial dantzig/devex but then may switch between 0 and 2.
- ClpPrimalColumnSteepest (const ClpPrimalColumnSteepest &rhs)

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

  Assignment operator.
- virtual ∼ClpPrimalColumnSteepest ()

  Destructor.
- virtual ClpPrimalColumnPivot ∗ clone (bool copyData=true) const

  Clone.

Private functions to deal with devex

- bool reference (int i) const

  reference would be faster using ClpSimplex's status_, but I prefer to keep modularity.
- void **setReference** (int i, bool trueFalse)
- void setPersistence (Persistence life)

  Set/ get persistence.
- Persistence **persistence** () const

Additional Inherited Members

4.69.1 Detailed Description

Primal Column Pivot Steepest Edge Algorithm Class.

See Forrest-Goldfarb paper for algorithm

Definition at line 23 of file ClpPrimalColumnSteepest.hpp.

4.69.2 Constructor & Destructor Documentation

4.69.2.1 ClpPrimalColumnSteepest::ClpPrimalColumnSteepest (int *mode* = 3)

Default Constructor 0 is exact devex, 1 full steepest, 2 is partial exact devex 3 switches between 0 and 2 depending on factorization 4 starts as partial dantzig/devex but then may switch between 0 and 2.

By partial exact devex is meant that the weights are updated as normal but only part of the nonbasic variables are scanned. This can be faster on very easy problems.

4.69.3 Member Function Documentation

4.69.3.1 virtual int ClpPrimalColumnSteepest::pivotColumn (CoinIndexedVector ∗ *updates*, CoinIndexedVector ∗ *spareRow1*, CoinIndexedVector ∗ *spareRow2*, CoinIndexedVector ∗ *spareColumn1*, CoinIndexedVector ∗ *spareColumn2*) [virtual]

Returns pivot column, -1 if none.

The Packed **CoinIndexedVector** updates has cost updates - for normal LP that is just +-weight where a feasibility changed. It also has reduced cost from last iteration in pivot row Parts of operation split out into separate functions for profiling and speed

Implements ClpPrimalColumnPivot.

4.69.3.2 virtual int ClpPrimalColumnSteepest::numberSprintColumns (int & *numberIterations*) const [virtual]

Returns number of extra columns for sprint algorithm - 0 means off.

Also number of iterations before recompute

Reimplemented from ClpPrimalColumnPivot.

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

- ClpPrimalColumnSteepest.hpp

4.70 ClpPrimalQuadraticDantzig Class Reference

Primal Column Pivot Dantzig Algorithm Class.

`#include <ClpPrimalQuadraticDantzig.hpp>`

Inheritance diagram for ClpPrimalQuadraticDantzig:

Collaboration diagram for ClpPrimalQuadraticDantzig:

Public Member Functions

Algorithmic methods

- virtual int pivotColumn (**CoinIndexedVector** ∗updates, **CoinIndexedVector** ∗spareRow1, **CoinIndexedVector** ∗spareRow2, **CoinIndexedVector** ∗spareColumn1, **CoinIndexedVector** ∗spareColumn2)

  Returns pivot column, -1 if none.
- virtual void saveWeights (ClpSimplex ∗model, int mode)

  Just sets model.

Constructors and destructors

- ClpPrimalQuadraticDantzig ()

  Default Constructor.
- ClpPrimalQuadraticDantzig (const ClpPrimalQuadraticDantzig &)

  Copy constructor.
- ClpPrimalQuadraticDantzig (ClpSimplexPrimalQuadratic ∗model, ClpQuadraticInfo ∗info)

  Constructor from model.
- ClpPrimalQuadraticDantzig & operator= (const ClpPrimalQuadraticDantzig &rhs)

  Assignment operator.
- virtual ∼ClpPrimalQuadraticDantzig ()

  Destructor.
- virtual ClpPrimalColumnPivot ∗ clone (bool copyData=true) const

  Clone.

Additional Inherited Members

4.70.1 Detailed Description

Primal Column Pivot Dantzig Algorithm Class.

This is simplest choice - choose largest infeasibility

Definition at line 20 of file ClpPrimalQuadraticDantzig.hpp.

4.70.2 Member Function Documentation

4.70.2.1 virtual int ClpPrimalQuadraticDantzig::pivotColumn (CoinIndexedVector ∗ *updates,* CoinIndexedVector ∗ *spareRow1,* CoinIndexedVector ∗ *spareRow2,* CoinIndexedVector ∗ *spareColumn1,* CoinIndexedVector ∗ *spareColumn2*) [virtual]

Returns pivot column, -1 if none.

Lumbers over all columns - slow updateArray has cost updates (also use pivotRow_ from last iteration) Can just do full price if you really want to be slow

Implements ClpPrimalColumnPivot.

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

- ClpPrimalQuadraticDantzig.hpp

4.71 ClpQuadraticObjective Class Reference

Quadratic Objective Class.

`#include <ClpQuadraticObjective.hpp>`

Inheritance diagram for ClpQuadraticObjective:

Collaboration diagram for ClpQuadraticObjective:

Public Member Functions

Stuff

- virtual double * gradient (const ClpSimplex *model, const double *solution, double &offset, bool refresh, int includeLinear=2)

  Returns gradient.
- virtual double reducedGradient (ClpSimplex *model, double *region, bool useFeasibleCosts)

  Resize objective.
- virtual double stepLength (ClpSimplex *model, const double *solution, const double *change, double maximumTheta, double &currentObj, double &predictedObj, double &thetaObj)

  *Returns step length which gives minimum of objective for solution + theta * change vector up to maximum theta.*
- virtual double objectiveValue (const ClpSimplex *model, const double *solution) const

  Return objective value (without any ClpModel offset) (model may be NULL)
- virtual void resize (int newNumberColumns)

  Resize objective.
- virtual void deleteSome (int numberToDelete, const int *which)

  Delete columns in objective.
- virtual void reallyScale (const double *columnScale)

  Scale objective.
- virtual int markNonlinear (char *which)

  Given a zeroed array sets nonlinear columns to 1.

Constructors and destructors

- ClpQuadraticObjective ()

  Default Constructor.
- ClpQuadraticObjective (const double *linearObjective, int numberColumns, const CoinBigIndex *start, const int *column, const double *element, int numberExtendedColumns_=-1)

  Constructor from objective.
- ClpQuadraticObjective (const ClpQuadraticObjective &rhs, int type=0)

  Copy constructor .
- ClpQuadraticObjective (const ClpQuadraticObjective &rhs, int numberColumns, const int *whichColumns)

  Subset constructor.
- ClpQuadraticObjective & operator= (const ClpQuadraticObjective &rhs)

  Assignment operator.
- virtual ~ClpQuadraticObjective ()

  Destructor.
- virtual ClpObjective * clone () const

  Clone.
- virtual ClpObjective * subsetClone (int numberColumns, const int *whichColumns) const

  Subset clone.
- void loadQuadraticObjective (const int numberColumns, const CoinBigIndex *start, const int *column, const double *element, int numberExtendedColumns=-1)

Load up quadratic objective.
- void **loadQuadraticObjective** (const **CoinPackedMatrix** &matrix)
- void deleteQuadraticObjective ()

  Get rid of quadratic objective.

Gets and sets

- **CoinPackedMatrix** ∗ quadraticObjective () const

  Quadratic objective.
- double ∗ linearObjective () const

  Linear objective.
- int numberExtendedColumns () const

  Length of linear objective which could be bigger.
- int numberColumns () const

  Number of columns in quadratic objective.
- bool fullMatrix () const

  If a full or half matrix.

Additional Inherited Members

4.71.1 Detailed Description

Quadratic Objective Class.

Definition at line 18 of file ClpQuadraticObjective.hpp.

4.71.2 Constructor & Destructor Documentation

4.71.2.1 ClpQuadraticObjective::ClpQuadraticObjective (const ClpQuadraticObjective & *rhs*, int *type* = 0)

Copy constructor .

If type is -1 then make sure half symmetric, if +1 then make sure full

4.71.2.2 ClpQuadraticObjective::ClpQuadraticObjective (const ClpQuadraticObjective & *rhs*, int *numberColumns*, const int ∗ *whichColumns*)

Subset constructor.

Duplicates are allowed and order is as given.

4.71.3 Member Function Documentation

4.71.3.1 virtual double∗ ClpQuadraticObjective::gradient (const ClpSimplex ∗ *model*, const double ∗ *solution*, double & *offset*, bool *refresh*, int *includeLinear* = 2) [virtual]

Returns gradient.

If Quadratic then solution may be NULL, also returns an offset (to be added to current one) If refresh is false then uses last solution Uses model for scaling includeLinear 0 - no, 1 as is, 2 as feasible

Implements ClpObjective.

4.71.3.2 virtual double ClpQuadraticObjective::reducedGradient (**ClpSimplex** ∗ *model,* double ∗ *region,* bool *useFeasibleCosts*) [virtual]

Resize objective.

Returns reduced gradient.Returns an offset (to be added to current one).

Implements ClpObjective.

4.71.3.3 virtual double ClpQuadraticObjective::stepLength (**ClpSimplex** ∗ *model,* const double ∗ *solution,* const double ∗ *change,* double *maximumTheta,* double & *currentObj,* double & *predictedObj,* double & *thetaObj*) [virtual]

Returns step length which gives minimum of objective for solution + theta ∗ change vector up to maximum theta.

arrays are numberColumns+numberRows Also sets current objective, predicted and at maximumTheta

Implements ClpObjective.

4.71.3.4 virtual int ClpQuadraticObjective::markNonlinear (char ∗ *which*) [virtual]

Given a zeroed array sets nonlinear columns to 1.

Returns number of nonlinear columns

Reimplemented from ClpObjective.

4.71.3.5 virtual **ClpObjective**∗ ClpQuadraticObjective::subsetClone (int *numberColumns,* const int ∗ *whichColumns*) const [virtual]

Subset clone.

Duplicates are allowed and order is as given.

Reimplemented from ClpObjective.

4.71.3.6 void ClpQuadraticObjective::loadQuadraticObjective (const int *numberColumns,* const CoinBigIndex ∗ *start,* const int ∗ *column,* const double ∗ *element,* int *numberExtendedColumns* = −1)

Load up quadratic objective.

This is stored as a **CoinPackedMatrix**

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

- ClpQuadraticObjective.hpp

4.72 ClpSimplex Class Reference

This solves LPs using the simplex method.

`#include <ClpSimplex.hpp>`

Inheritance diagram for ClpSimplex:

Collaboration diagram for ClpSimplex:

Public Types

- enum Status

  enums for status of various sorts.

Public Member Functions

Constructors and destructor and copy

- ClpSimplex (bool emptyMessages=false)

  Default constructor.
- ClpSimplex (const ClpSimplex &rhs, int scalingMode=-1)

  Copy constructor.
- ClpSimplex (const ClpModel &rhs, int scalingMode=-1)

  Copy constructor from model.
- ClpSimplex (const ClpModel ∗wholeModel, int numberRows, const int ∗whichRows, int numberColumns, const int ∗whichColumns, bool dropNames=true, bool dropIntegers=true, bool fixOthers=false)

  Subproblem constructor.
- ClpSimplex (const ClpSimplex ∗wholeModel, int numberRows, const int ∗whichRows, int numberColumns, const int ∗whichColumns, bool dropNames=true, bool dropIntegers=true, bool fixOthers=false)

  Subproblem constructor.
- ClpSimplex (ClpSimplex ∗wholeModel, int numberColumns, const int ∗whichColumns)

  This constructor modifies original ClpSimplex and stores original stuff in created ClpSimplex.
- void originalModel (ClpSimplex ∗miniModel)

  This copies back stuff from miniModel and then deletes miniModel.
- int **abcState** () const
- void **setAbcState** (int state)
- void setPersistenceFlag (int value)

  Array persistence flag If 0 then as now (delete/new) 1 then only do arrays if bigger needed 2 as 1 but give a bit extra if bigger needed.
- void makeBaseModel ()

  Save a copy of model with certain state - normally without cuts.
- void deleteBaseModel ()

  Switch off base model.
- ClpSimplex ∗ baseModel () const

  See if we have base model.
- void setToBaseModel (ClpSimplex ∗model=NULL)

  Reset to base model (just size and arrays needed) If model NULL use internal copy.
- ClpSimplex & operator= (const ClpSimplex &rhs)

  Assignment operator. This copies the data.
- ∼ClpSimplex ()

  Destructor.
- void loadProblem (const ClpMatrixBase &matrix, const double ∗collb, const double ∗colub, const double ∗obj, const double ∗rowlb, const double ∗rowub, const double ∗rowObjective=NULL)

  Loads a problem (the constraints on the rows are given by lower and upper bounds).
- void **loadProblem** (const **CoinPackedMatrix** &matrix, const double ∗collb, const double ∗colub, const double ∗obj, const double ∗rowlb, const double ∗rowub, const double ∗rowObjective=NULL)
- void loadProblem (const int numcols, const int numrows, const CoinBigIndex ∗start, const int ∗index, const double ∗value, const double ∗collb, const double ∗colub, const double ∗obj, const double ∗rowlb, const double ∗rowub, const double ∗rowObjective=NULL)

  Just like the other loadProblem() method except that the matrix is given in a standard column major ordered format (without gaps).

- void loadProblem (const int numcols, const int numrows, const CoinBigIndex *start, const int *index, const double *value, const int *length, const double *collb, const double *colub, const double *obj, const double *rowlb, const double *rowub, const double *rowObjective=NULL)

    This one is for after presolve to save memory.
- int loadProblem (**CoinModel** &modelObject, bool keepSolution=false)

    This loads a model from a coinModel object - returns number of errors.
- int readMps (const char *filename, bool keepNames=false, bool ignoreErrors=false)

    Read an mps file from the given filename.
- int readGMPL (const char *filename, const char *dataName, bool keepNames=false)

    Read GMPL files from the given filenames.
- int readLp (const char *filename, const double epsilon=1e-5)

    Read file in LP format from file with name filename.
- void borrowModel (ClpModel &otherModel)

    Borrow model.
- void **borrowModel** (ClpSimplex &otherModel)
- void passInEventHandler (const ClpEventHandler *eventHandler)

    Pass in Event handler (cloned and deleted at end)
- void getbackSolution (const ClpSimplex &smallModel, const int *whichRow, const int *whichColumn)

    Puts solution back into small model.
- int loadNonLinear (void *info, int &numberConstraints, ClpConstraint **&constraints)

    Load nonlinear part of problem from AMPL info Returns 0 if linear 1 if quadratic objective 2 if quadratic constraints 3 if nonlinear objective 4 if nonlinear constraints -1 on failure.

Functions most useful to user

- int initialSolve (ClpSolve &options)

    General solve algorithm which can do presolve.
- int initialSolve ()

    Default initial solve.
- int initialDualSolve ()

    Dual initial solve.
- int initialPrimalSolve ()

    Primal initial solve.
- int initialBarrierSolve ()

    Barrier initial solve.
- int initialBarrierNoCrossSolve ()

    Barrier initial solve, not to be followed by crossover.
- int dual (int ifValuesPass=0, int startFinishOptions=0)

    Dual algorithm - see ClpSimplexDual.hpp for method.
- int **dualDebug** (int ifValuesPass=0, int startFinishOptions=0)
- int primal (int ifValuesPass=0, int startFinishOptions=0)

    Primal algorithm - see ClpSimplexPrimal.hpp for method.
- int nonlinearSLP (int numberPasses, double deltaTolerance)

    Solves nonlinear problem using SLP - may be used as crash for other algorithms when number of iterations small.
- int nonlinearSLP (int numberConstraints, ClpConstraint **constraints, int numberPasses, double delta←Tolerance)

    Solves problem with nonlinear constraints using SLP - may be used as crash for other algorithms when number of iterations small.
- int barrier (bool crossover=true)

    Solves using barrier (assumes you have good cholesky factor code).
- int reducedGradient (int phase=0)

    Solves non-linear using reduced gradient.
- int solve (**CoinStructuredModel** *model)

Solve using structure of model and maybe in parallel.

- int loadProblem (**CoinStructuredModel** &modelObject, bool originalOrder=true, bool keepSolution=false)

  This loads a model from a **CoinStructuredModel** *object - returns number of errors.*
- int cleanup (int cleanupScaling)

  When scaling is on it is possible that the scaled problem is feasible but the unscaled is not.
- int dualRanging (int numberCheck, const int ∗which, double ∗costIncrease, int ∗sequenceIncrease, double ∗costDecrease, int ∗sequenceDecrease, double ∗valueIncrease=NULL, double ∗valueDecrease=NULL)

  Dual ranging.
- int primalRanging (int numberCheck, const int ∗which, double ∗valueIncrease, int ∗sequenceIncrease, double ∗valueDecrease, int ∗sequenceDecrease)

  Primal ranging.
- int modifyCoefficientsAndPivot (int number, const int ∗which, const CoinBigIndex ∗start, const int ∗row, const double ∗newCoefficient, const unsigned char ∗newStatus=NULL, const double ∗newLower=NULL, const double ∗newUpper=NULL, const double ∗newObjective=NULL)

  Modifies coefficients etc and if necessary pivots in and out.
- int outDuplicateRows (int numberLook, int ∗whichRows, bool noOverlaps=false, double tolerance=-1.0, double cleanUp=0.0)

  Take out duplicate rows (includes scaled rows and intersections).
- double moveTowardsPrimalFeasible ()

  Try simple crash like techniques to get closer to primal feasibility returns final sum of infeasibilities.
- void removeSuperBasicSlacks (int threshold=0)

  Try simple crash like techniques to remove super basic slacks but only if > threshold.
- ClpSimplex ∗ miniPresolve (char ∗rowType, char ∗columnType, void ∗∗info)

  Mini presolve (faster) Char arrays must be numberRows and numberColumns long on entry second part must be filled in as follows - 0 - possible >0 - take out and do something (depending on value - TBD) -1 row/column can't vanish but can have entries removed/changed -2 don't touch at all on exit <=0 ones will be in presolved problem struct will be created and will be long enough (information on length etc in first entry) user must delete struct.
- void miniPostsolve (const ClpSimplex ∗presolvedModel, void ∗info)

  After mini presolve.
- void miniSolve (char ∗rowType, char ∗columnType, int algorithm, int startUp)

  mini presolve and solve
- int writeBasis (const char ∗filename, bool writeValues=false, int formatType=0) const

  Write the basis in MPS format to the specified file.
- int readBasis (const char ∗filename)

  Read a basis from the given filename, returns -1 on file error, 0 if no values, 1 if values.
- **CoinWarmStartBasis** ∗ getBasis () const

  Returns a basis (to be deleted by user)
- void setFactorization (ClpFactorization &factorization)

  Passes in factorization.
- ClpFactorization ∗ **swapFactorization** (ClpFactorization ∗factorization)
- void copyFactorization (ClpFactorization &factorization)

  Copies in factorization to existing one.
- int tightenPrimalBounds (double factor=0.0, int doTight=0, bool tightIntegers=false)

  Tightens primal bounds to make dual faster.
- int crash (double gap, int pivot)

  Crash - at present just aimed at dual, returns -2 if dual preferred and crash basis created -1 if dual preferred and all slack basis preferred 0 if basis going in was not all slack 1 if primal preferred and all slack basis preferred 2 if primal preferred and crash basis created.
- void setDualRowPivotAlgorithm (ClpDualRowPivot &choice)

  Sets row pivot choice algorithm in dual.
- void setPrimalColumnPivotAlgorithm (ClpPrimalColumnPivot &choice)

  Sets column pivot choice algorithm in primal.
- void markHotStart (void ∗&saveStuff)

  Create a hotstart point of the optimization process.

- void solveFromHotStart (void ∗saveStuff)

  Optimize starting from the hotstart.
- void unmarkHotStart (void ∗saveStuff)

  Delete the snapshot.
- int strongBranching (int numberVariables, const int ∗variables, double ∗newLower, double ∗newUpper, double ∗∗outputSolution, int ∗outputStatus, int ∗outputIterations, bool stopOnFirstInfeasible=true, bool alwaysFinish=false, int startFinishOptions=0)

  For strong branching.
- int fathom (void ∗stuff)

  Fathom - 1 if solution.
- int fathomMany (void ∗stuff)

  Do up to N deep - returns -1 - no solution nNodes_ valid nodes >= if solution and that node gives solution ClpNode array is 2∗∗N long.
- double doubleCheck ()

  Double checks OK.
- int startFastDual2 (ClpNodeStuff ∗stuff)

  Starts Fast dual2.
- int fastDual2 (ClpNodeStuff ∗stuff)

  Like Fast dual.
- void stopFastDual2 (ClpNodeStuff ∗stuff)

  Stops Fast dual2.
- ClpSimplex ∗ fastCrunch (ClpNodeStuff ∗stuff, int mode)

  Deals with crunch aspects mode 0 - in 1 - out with solution 2 - out without solution returns small model or NULL.

Needed for functionality of OsiSimplexInterface

- int pivot ()

  Pivot in a variable and out a variable.
- int primalPivotResult ()

  Pivot in a variable and choose an outgoing one.
- int dualPivotResultPart1 ()

  Pivot out a variable and choose an incoing one.
- int pivotResultPart2 (int algorithm, int state)

  Do actual pivot state is 0 if need tableau column, 1 if in rowArray_[1].
- int startup (int ifValuesPass, int startFinishOptions=0)

  Common bits of coding for dual and primal.
- void **finish** (int startFinishOptions=0)
- bool statusOfProblem (bool initial=false)

  Factorizes and returns true if optimal.
- void defaultFactorizationFrequency ()

  If user left factorization frequency then compute.
- void copyEnabledStuff (const ClpSimplex ∗rhs)

  Copy across enabled stuff from one solver to another.

most useful gets and sets

- bool primalFeasible () const

  If problem is primal feasible.
- bool dualFeasible () const

  If problem is dual feasible.
- ClpFactorization ∗ factorization () const

  factorization
- bool sparseFactorization () const

Sparsity on or off.
- void **setSparseFactorization** (bool value)
- int factorizationFrequency () const

  Factorization frequency.
- void **setFactorizationFrequency** (int value)
- double dualBound () const

  Dual bound.
- void **setDualBound** (double value)
- double infeasibilityCost () const

  Infeasibility cost.
- void **setInfeasibilityCost** (double value)
- int perturbation () const

  Amount of print out: 0 - none 1 - just final 2 - just factorizations 3 - as 2 plus a bit more 4 - verbose above that 8,16,32 etc just for selective debug.
- void **setPerturbation** (int value)
- int algorithm () const

  Current (or last) algorithm.
- void setAlgorithm (int value)

  Set algorithm.
- bool isObjectiveLimitTestValid () const

  Return true if the objective limit test can be relied upon.
- double sumDualInfeasibilities () const

  Sum of dual infeasibilities.
- void **setSumDualInfeasibilities** (double value)
- double sumOfRelaxedDualInfeasibilities () const

  Sum of relaxed dual infeasibilities.
- void **setSumOfRelaxedDualInfeasibilities** (double value)
- int numberDualInfeasibilities () const

  Number of dual infeasibilities.
- void **setNumberDualInfeasibilities** (int value)
- int numberDualInfeasibilitiesWithoutFree () const

  Number of dual infeasibilities (without free)
- double sumPrimalInfeasibilities () const

  Sum of primal infeasibilities.
- void **setSumPrimalInfeasibilities** (double value)
- double sumOfRelaxedPrimalInfeasibilities () const

  Sum of relaxed primal infeasibilities.
- void **setSumOfRelaxedPrimalInfeasibilities** (double value)
- int numberPrimalInfeasibilities () const

  Number of primal infeasibilities.
- void **setNumberPrimalInfeasibilities** (int value)
- int saveModel (const char ∗fileName)

  Save model to file, returns 0 if success.
- int restoreModel (const char ∗fileName)

  Restore model from file, returns 0 if success, deletes current model.
- void checkSolution (int setToBounds=0)

  Just check solution (for external use) - sets sum of infeasibilities etc.
- void checkSolutionInternal ()

  Just check solution (for internal use) - sets sum of infeasibilities etc.
- void checkUnscaledSolution ()

  Check unscaled primal solution but allow for rounding error.
- **CoinIndexedVector** ∗ rowArray (int index) const

  Useful row length arrays (0,1,2,3,4,5)
- **CoinIndexedVector** ∗ columnArray (int index) const

Useful column length arrays (0,1,2,3,4,5)
- double alphaAccuracy () const

  Initial value for alpha accuracy calculation (-1.0 off)
- void **setAlphaAccuracy** (double value)
- void setDisasterHandler (ClpDisasterHandler ∗handler)

  Objective value.
- ClpDisasterHandler ∗ disasterHandler () const

  Get disaster handler.
- double largeValue () const

  Large bound value (for complementarity etc)
- void **setLargeValue** (double value)
- double largestPrimalError () const

  Largest error on Ax-b.
- double largestDualError () const

  Largest error on basic duals.
- void setLargestPrimalError (double value)

  Largest error on Ax-b.
- void setLargestDualError (double value)

  Largest error on basic duals.
- double zeroTolerance () const

  Get zero tolerance.
- void setZeroTolerance (double value)

  Set zero tolerance.
- int ∗ pivotVariable () const

  Basic variables pivoting on which rows.
- bool automaticScaling () const

  If automatic scaling on.
- void **setAutomaticScaling** (bool onOff)
- double currentDualTolerance () const

  Current dual tolerance.
- void **setCurrentDualTolerance** (double value)
- double currentPrimalTolerance () const

  Current primal tolerance.
- void **setCurrentPrimalTolerance** (double value)
- int numberRefinements () const

  How many iterative refinements to do.
- void **setNumberRefinements** (int value)
- double alpha () const

  Alpha (pivot element) for use by classes e.g. steepestedge.
- void **setAlpha** (double value)
- double dualIn () const

  Reduced cost of last incoming for use by classes e.g. steepestedge.
- void setDualIn (double value)

  Set reduced cost of last incoming to force error.
- int pivotRow () const

  Pivot Row for use by classes e.g. steepestedge.
- void **setPivotRow** (int value)
- double valueIncomingDual () const

  value of incoming variable (in Dual)

public methods

- double ∗ solutionRegion (int section) const

Return row or column sections - not as much needed as it once was.

- double ∗ **djRegion** (int section) const
- double ∗ **lowerRegion** (int section) const
- double ∗ **upperRegion** (int section) const
- double ∗ **costRegion** (int section) const
- double ∗ solutionRegion () const

  Return region as single array.
- double ∗ **djRegion** () const
- double ∗ **lowerRegion** () const
- double ∗ **upperRegion** () const
- double ∗ **costRegion** () const
- Status **getStatus** (int sequence) const
- void **setStatus** (int sequence, Status newstatus)
- bool startPermanentArrays ()

  Start or reset using maximumRows_ and Columns_ - true if change.
- void setInitialDenseFactorization (bool onOff)

  Normally the first factorization does sparse coding because the factorization could be singular.
- bool **initialDenseFactorization** () const
- int sequenceIn () const

  Return sequence In or Out.
- int **sequenceOut** () const
- void setSequenceIn (int sequence)

  Set sequenceIn or Out.
- void **setSequenceOut** (int sequence)
- int directionIn () const

  Return direction In or Out.
- int **directionOut** () const
- void setDirectionIn (int direction)

  Set directionIn or Out.
- void **setDirectionOut** (int direction)
- double valueOut () const

  Value of Out variable.
- void setValueOut (double value)

  Set value of out variable.
- double dualOut () const

  Dual value of Out variable.
- void setDualOut (double value)

  Set dual value of out variable.
- void setLowerOut (double value)

  Set lower of out variable.
- void setUpperOut (double value)

  Set upper of out variable.
- void setTheta (double value)

  Set theta of out variable.
- int isColumn (int sequence) const

  Returns 1 if sequence indicates column.
- int sequenceWithin (int sequence) const

  Returns sequence number within section.
- double solution (int sequence)

  Return row or column values.
- double & solutionAddress (int sequence)

  Return address of row or column values.
- double **reducedCost** (int sequence)
- double & **reducedCostAddress** (int sequence)

- double **lower** (int sequence)
- double & lowerAddress (int sequence)

  Return address of row or column lower bound.
- double **upper** (int sequence)
- double & upperAddress (int sequence)

  Return address of row or column upper bound.
- double **cost** (int sequence)
- double & costAddress (int sequence)

  Return address of row or column cost.
- double originalLower (int iSequence) const

  Return original lower bound.
- double originalUpper (int iSequence) const

  Return original lower bound.
- double theta () const

  Theta (pivot change)
- double bestPossibleImprovement () const

  Best possible improvement using djs (primal) or obj change by flipping bounds to make dual feasible (dual)
- ClpNonLinearCost ∗ nonLinearCost () const

  Return pointer to details of costs.
- int moreSpecialOptions () const

  Return more special options 1 bit - if presolve says infeasible in ClpSolve return 2 bit - if presolved problem infeasible return 4 bit - keep arrays like upper_ around 8 bit - if factorization kept can still declare optimal at once 16 bit - if checking replaceColumn accuracy before updating 32 bit - say optimal if primal feasible! 64 bit - give up easily in dual (and say infeasible) 128 bit - no objective, 0-1 and in B&B 256 bit - in primal from dual or vice versa 512 bit - alternative use of solveType_ 1024 bit - don't do row copy of factorization 2048 bit - perturb in complete fathoming 4096 bit - try more for complete fathoming 8192 bit - don't even think of using primal if user asks for dual (and vv) 16384 bit - in initialSolve so be more flexible 32768 bit - don't swap algorithms from dual if small infeasibility 65536 bit - perturb in postsolve cleanup (even if < 10000 rows) 131072 bit (∗3) initial stateDualColumn 524288 bit - stop when primal feasible.
- void setMoreSpecialOptions (int value)

  Set more special options 1 bit - if presolve says infeasible in ClpSolve return 2 bit - if presolved problem infeasible return 4 bit - keep arrays like upper_ around 8 bit - no free or superBasic variables 16 bit - if checking replaceColumn accuracy before updating 32 bit - say optimal if primal feasible! 64 bit - give up easily in dual (and say infeasible) 128 bit - no objective, 0-1 and in B&B 256 bit - in primal from dual or vice versa 512 bit - alternative use of solveType_ 1024 bit - don't do row copy of factorization 2048 bit - perturb in complete fathoming 4096 bit - try more for complete fathoming 8192 bit - don't even think of using primal if user asks for dual (and vv) 16384 bit - in initialSolve so be more flexible 32768 bit - don't swap algorithms from dual if small infeasibility 65536 bit - perturb in postsolve cleanup (even if < 10000 rows) 131072 bit (∗3) initial stateDualColumn 524288 bit - stop when primal feasible 1048576 bit - don't perturb even if long time 2097152 bit - no primal in fastDual2 if feasible 4194304 bit - tolerances have been changed by code 8388608 bit - tolerances are dynamic (at first)

status methods

- void **setFakeBound** (int sequence, FakeBound fakeBound)
- FakeBound **getFakeBound** (int sequence) const
- void **setRowStatus** (int sequence, Status newstatus)
- Status **getRowStatus** (int sequence) const
- void **setColumnStatus** (int sequence, Status newstatus)
- Status **getColumnStatus** (int sequence) const
- void **setPivoted** (int sequence)
- void **clearPivoted** (int sequence)
- bool **pivoted** (int sequence) const
- void setFlagged (int sequence)

  To flag a variable (not inline to allow for column generation)
- void **clearFlagged** (int sequence)
- bool **flagged** (int sequence) const
- void setActive (int iRow)

To say row active in primal pivot row choice.
- void **clearActive** (int iRow)
- bool **active** (int iRow) const
- void setPerturbed (int iSequence)

  To say perturbed.
- void **clearPerturbed** (int iSequence)
- bool **perturbed** (int iSequence) const
- void createStatus ()

  Set up status array (can be used by OsiClp).
- void allSlackBasis (bool resetSolution=false)

  Sets up all slack basis and resets solution to as it was after initial load or readMps.
- int lastBadIteration () const

  So we know when to be cautious.
- void setLastBadIteration (int value)

  Set so we know when to be cautious.
- int progressFlag () const

  Progress flag - at present 0 bit says artificials out.
- ClpSimplexProgress ∗ progress ()

  For dealing with all issues of cycling etc.
- int forceFactorization () const

  Force re-factorization early value.
- void forceFactorization (int value)

  Force re-factorization early.
- double rawObjectiveValue () const

  Raw objective value (so always minimize in primal)
- void computeObjectiveValue (bool useWorkingSolution=false)

  Compute objective value from solution and put in objectiveValue_.
- double computeInternalObjectiveValue ()

  Compute minimization objective value from internal solution without perturbation.
- double ∗ infeasibilityRay (bool fullRay=false) const

  Infeasibility/unbounded ray (NULL returned if none/wrong) Up to user to use delete [] on these arrays.
- int numberExtraRows () const

  Number of extra rows.
- int maximumBasic () const

  Maximum number of basic variables - can be more than number of rows if GUB.
- int baseIteration () const

  Iteration when we entered dual or primal.
- void generateCpp (FILE ∗fp, bool defaultFactor=false)

  Create C++ lines to get to current state.
- ClpFactorization ∗ getEmptyFactorization ()

  Gets clean and emptyish factorization.
- void setEmptyFactorization ()

  May delete or may make clean and emptyish factorization.
- void moveInfo (const ClpSimplex &rhs, bool justStatus=false)

  Move status and solution across.

Basis handling

- void getBInvARow (int row, double ∗z, double ∗slack=NULL)

  Get a row of the tableau (slack part in slack if not NULL)
- void getBInvRow (int row, double ∗z)

  Get a row of the basis inverse.
- void getBInvACol (int col, double ∗vec)

Get a column of the tableau.
- void getBInvCol (int col, double ∗vec)

  Get a column of the basis inverse.
- void getBasics (int ∗index)

  Get basic indices (order of indices corresponds to the order of elements in a vector retured by getBInvACol() and getBInvCol()).

Changing bounds on variables and constraints

- void setObjectiveCoefficient (int elementIndex, double elementValue)

  Set an objective function coefficient.
- void setObjCoeff (int elementIndex, double elementValue)

  Set an objective function coefficient.
- void setColumnLower (int elementIndex, double elementValue)

  Set a single column lower bound
  Use -DBL_MAX for -infinity.
- void setColumnUpper (int elementIndex, double elementValue)

  Set a single column upper bound
  Use DBL_MAX for infinity.
- void setColumnBounds (int elementIndex, double lower, double upper)

  Set a single column lower and upper bound.
- void setColumnSetBounds (const int ∗indexFirst, const int ∗indexLast, const double ∗boundList)

  Set the bounds on a number of columns simultaneously
  The default implementation just invokes setColLower() and setColUpper() over and over again.
- void setColLower (int elementIndex, double elementValue)

  Set a single column lower bound
  Use -DBL_MAX for -infinity.
- void setColUpper (int elementIndex, double elementValue)

  Set a single column upper bound
  Use DBL_MAX for infinity.
- void setColBounds (int elementIndex, double newlower, double newupper)

  Set a single column lower and upper bound.
- void setColSetBounds (const int ∗indexFirst, const int ∗indexLast, const double ∗boundList)

  Set the bounds on a number of columns simultaneously

- void setRowLower (int elementIndex, double elementValue)

  Set a single row lower bound
  Use -DBL_MAX for -infinity.
- void setRowUpper (int elementIndex, double elementValue)

  Set a single row upper bound
  Use DBL_MAX for infinity.
- void setRowBounds (int elementIndex, double lower, double upper)

  Set a single row lower and upper bound.
- void setRowSetBounds (const int ∗indexFirst, const int ∗indexLast, const double ∗boundList)

  Set the bounds on a number of rows simultaneously

- void resize (int newNumberRows, int newNumberColumns)

  Resizes rim part of model.

Protected Member Functions

protected methods

- int gutsOfSolution (double ∗givenDuals, const double ∗givenPrimals, bool valuesPass=false)

May change basis and then returns number changed.
- void gutsOfDelete (int type)

  Does most of deletion (0 = all, 1 = most, 2 most + factorization)
- void gutsOfCopy (const ClpSimplex &rhs)

  Does most of copying.
- bool createRim (int what, bool makeRowCopy=false, int startFinishOptions=0)

  puts in format I like (rowLower,rowUpper) also see StandardMatrix 1 bit does rows (now and columns), (2 bit does column bounds), 4 bit does objective(s).
- void createRim1 (bool initial)

  Does rows and columns.
- void createRim4 (bool initial)

  Does objective.
- void createRim5 (bool initial)

  Does rows and columns and objective.
- void deleteRim (int getRidOfFactorizationData=2)

  releases above arrays and does solution scaling out.
- bool sanityCheck ()

  Sanity check on input rim data (after scaling) - returns true if okay.

Friends

- void ClpSimplexUnitTest (const std::string &mpsDir)

  A function that tests the methods in the ClpSimplex class.

Functions less likely to be useful to casual user

- int getSolution (const double ∗rowActivities, const double ∗columnActivities)

  Given an existing factorization computes and checks primal and dual solutions.
- int getSolution ()

  Given an existing factorization computes and checks primal and dual solutions.
- int createPiecewiseLinearCosts (const int ∗starts, const double ∗lower, const double ∗gradient)

  Constructs a non linear cost from list of non-linearities (columns only) First lower of each column is taken as real lower Last lower is taken as real upper and cost ignored.
- ClpDualRowPivot ∗ dualRowPivot () const

  dual row pivot choice
- ClpPrimalColumnPivot ∗ primalColumnPivot () const

  primal column pivot choice
- bool goodAccuracy () const

  Returns true if model looks OK.
- void returnModel (ClpSimplex &otherModel)

  Return model - updates any scalars.
- int internalFactorize (int solveType)

  Factorizes using current basis.
- ClpDataSave saveData ()

  Save data.
- void restoreData (ClpDataSave saved)

  Restore data.
- void cleanStatus ()

  Clean up status.

- int factorize ()

  Factorizes using current basis. For external use.
- void computeDuals (double ∗givenDjs)

  Computes duals from scratch.
- void computePrimals (const double ∗rowActivities, const double ∗columnActivities)

  Computes primals from scratch.
- void add (double ∗array, int column, double multiplier) const

  Adds multiple of a column into an array.
- void unpack (**CoinIndexedVector** ∗rowArray) const

  Unpacks one column of the matrix into indexed array Uses sequenceIn_ Also applies scaling if needed.
- void unpack (**CoinIndexedVector** ∗rowArray, int sequence) const

  Unpacks one column of the matrix into indexed array Slack if sequence>= numberColumns Also applies scaling if needed.
- void unpackPacked (**CoinIndexedVector** ∗rowArray)

  Unpacks one column of the matrix into indexed array as packed vector Uses sequenceIn_ Also applies scaling if needed.
- void unpackPacked (**CoinIndexedVector** ∗rowArray, int sequence)

  Unpacks one column of the matrix into indexed array as packed vector Slack if sequence>= numberColumns Also applies scaling if needed.
- void setValuesPassAction (double incomingInfeasibility, double allowedInfeasibility)

  For advanced use.
- int cleanFactorization (int ifValuesPass)

  Get a clean factorization - i.e.
- int housekeeping (double objectiveChange)

  This does basis housekeeping and does values for in/out variables.
- void checkPrimalSolution (const double ∗rowActivities=NULL, const double ∗columnActivies=NULL)

  This sets largest infeasibility and most infeasible and sum and number of infeasibilities (Primal)
- void checkDualSolution ()

  This sets largest infeasibility and most infeasible and sum and number of infeasibilities (Dual)
- void checkBothSolutions ()

  This sets sum and number of infeasibilities (Dual and Primal)
- double scaleObjective (double value)

  If input negative scales objective so maximum <= -value and returns scale factor used.
- int solveDW (**CoinStructuredModel** ∗model, ClpSolve &options)

  Solve using Dantzig-Wolfe decomposition and maybe in parallel.
- int solveBenders (**CoinStructuredModel** ∗model, ClpSolve &options)

  Solve using Benders decomposition and maybe in parallel.

data. Many arrays have a row part and a column part.

There is a single array with both - columns then rows and then normally two arrays pointing to rows and columns.

The single array is the owner of memory

- double bestPossibleImprovement_

  Best possible improvement using djs (primal) or obj change by flipping bounds to make dual feasible (dual)
- double zeroTolerance_

  Zero tolerance.
- int columnPrimalSequence_

  Sequence of worst (-1 if feasible)

- int rowPrimalSequence_

  Sequence of worst (-1 if feasible)
- double bestObjectiveValue_

  "Best" objective value
- int moreSpecialOptions_

  More special options - see set for details.
- int baseIteration_

  Iteration when we entered dual or primal.
- double primalToleranceToGetOptimal_

  Primal tolerance needed to make dual feasible (<largeTolerance)
- double largeValue_

  Large bound value (for complementarity etc)
- double largestPrimalError_

  Largest error on Ax-b.
- double largestDualError_

  Largest error on basic duals.
- double alphaAccuracy_

  For computing whether to re-factorize.
- double dualBound_

  Dual bound.
- double alpha_

  Alpha (pivot element)
- double theta_

  Theta (pivot change)
- double lowerIn_

  Lower Bound on In variable.
- double valueIn_

  Value of In variable.
- double upperIn_

  Upper Bound on In variable.
- double dualIn_

  Reduced cost of In variable.
- double lowerOut_

  Lower Bound on Out variable.
- double valueOut_

  Value of Out variable.
- double upperOut_

  Upper Bound on Out variable.
- double dualOut_

  Infeasibility (dual) or ? (primal) of Out variable.
- double dualTolerance_

  Current dual tolerance for algorithm.
- double primalTolerance_

  Current primal tolerance for algorithm.
- double sumDualInfeasibilities_

  Sum of dual infeasibilities.
- double sumPrimalInfeasibilities_

Sum of primal infeasibilities.
- double infeasibilityCost_

  Weight assigned to being infeasible in primal.
- double sumOfRelaxedDualInfeasibilities_

  Sum of Dual infeasibilities using tolerance based on error in duals.
- double sumOfRelaxedPrimalInfeasibilities_

  Sum of Primal infeasibilities using tolerance based on error in primals.
- double acceptablePivot_

  Acceptable pivot value just after factorization.
- double minimumPrimalTolerance_

  Minimum primal tolerance.
- double **averageInfeasibility_** [CLP_INFEAS_SAVE]
- double ∗ lower_

  Working copy of lower bounds (Owner of arrays below)
- double ∗ rowLowerWork_

  Row lower bounds - working copy.
- double ∗ columnLowerWork_

  Column lower bounds - working copy.
- double ∗ upper_

  Working copy of upper bounds (Owner of arrays below)
- double ∗ rowUpperWork_

  Row upper bounds - working copy.
- double ∗ columnUpperWork_

  Column upper bounds - working copy.
- double ∗ cost_

  Working copy of objective (Owner of arrays below)
- double ∗ rowObjectiveWork_

  Row objective - working copy.
- double ∗ objectiveWork_

  Column objective - working copy.
- **CoinIndexedVector** ∗ rowArray_ [6]

  Useful row length arrays.
- **CoinIndexedVector** ∗ columnArray_ [6]

  Useful column length arrays.
- int sequenceIn_

  Sequence of In variable.
- int directionIn_

  Direction of In, 1 going up, -1 going down, 0 not a clude.
- int sequenceOut_

  Sequence of Out variable.
- int directionOut_

  Direction of Out, 1 to upper bound, -1 to lower bound, 0 - superbasic.
- int pivotRow_

  Pivot Row.
- int lastGoodIteration_

  Last good iteration (immediately after a re-factorization)
- double ∗ dj_

Working copy of reduced costs (Owner of arrays below)
- double ∗ rowReducedCost_

    Reduced costs of slacks not same as duals (or - duals)
- double ∗ reducedCostWork_

    Possible scaled reduced costs.
- double ∗ solution_

    Working copy of primal solution (Owner of arrays below)
- double ∗ rowActivityWork_

    Row activities - working copy.
- double ∗ columnActivityWork_

    Column activities - working copy.
- int numberDualInfeasibilities_

    Number of dual infeasibilities.
- int numberDualInfeasibilitiesWithoutFree_

    Number of dual infeasibilities (without free)
- int numberPrimalInfeasibilities_

    Number of primal infeasibilities.
- int numberRefinements_

    How many iterative refinements to do.
- ClpDualRowPivot ∗ dualRowPivot_

    dual row pivot choice
- ClpPrimalColumnPivot ∗ primalColumnPivot_

    primal column pivot choice
- int ∗ pivotVariable_

    Basic variables pivoting on which rows.
- ClpFactorization ∗ factorization_

    factorization
- double ∗ savedSolution_

    Saved version of solution.
- int numberTimesOptimal_

    Number of times code has tentatively thought optimal.
- ClpDisasterHandler ∗ disasterArea_

    Disaster handler.
- int changeMade_

    If change has been made (first attempt at stopping looping)
- int algorithm_

    Algorithm >0 == Primal, <0 == Dual.
- int forceFactorization_

    Now for some reliability aids This forces re-factorization early.
- int perturbation_

    Perturbation: -50 to +50 - perturb by this power of ten (-6 sounds good) 100 - auto perturb if takes too long (1.0e-6 largest nonzero) 101 - we are perturbed 102 - don't try perturbing again default is 100.
- unsigned char ∗ saveStatus_

    Saved status regions.
- ClpNonLinearCost ∗ nonLinearCost_

    Very wasteful way of dealing with infeasibilities in primal.
- int lastBadIteration_

So we know when to be cautious.

- int lastFlaggedIteration_

  So we know when to open up again.

- int numberFake_

  Can be used for count of fake bounds (dual) or fake costs (primal)

- int numberChanged_

  Can be used for count of changed costs (dual) or changed bounds (primal)

- int progressFlag_

  Progress flag - at present 0 bit says artificials out, 1 free in.

- int firstFree_

  First free/super-basic variable (-1 if none)

- int numberExtraRows_

  Number of extra rows.

- int maximumBasic_

  Maximum number of basic variables - can be more than number of rows if GUB.

- int dontFactorizePivots_

  If may skip final factorize then allow up to this pivots (default 20)

- double incomingInfeasibility_

  For advanced use.

- double **allowedInfeasibility_**
- int automaticScale_

  Automatic scaling of objective and rhs and bounds.

- int maximumPerturbationSize_

  Maximum perturbation array size (take out when code rewritten)

- double ∗ perturbationArray_

  Perturbation array (maximumPerturbationSize_)

- ClpSimplex ∗ baseModel_

  A copy of model with certain state - normally without cuts.

- ClpSimplexProgress progress_

  For dealing with all issues of cycling etc.

- int **abcState_**
- int numberDegeneratePivots_

  Number of degenerate pivots since last perturbed.

- int spareIntArray_ [4]

  Spare int array for passing information [0]!=0 switches on.

- double spareDoubleArray_ [4]

  Spare double array for passing information [0]!=0 switches on.

- class OsiClpSolverInterface

  Allow OsiClp certain perks.

- class OsiCLPSolverInterface

  And OsiCLP.

Additional Inherited Members

4.72.1 Detailed Description

This solves LPs using the simplex method.

It inherits from ClpModel and all its arrays are created at algorithm time. Originally I tried to work with model arrays but for simplicity of coding I changed to single arrays with structural variables then row variables. Some coding is still based on old style and needs cleaning up.

For a description of algorithms:

for dual see ClpSimplexDual.hpp and at top of ClpSimplexDual.cpp for primal see ClpSimplexPrimal.hpp and at top of ClpSimplexPrimal.cpp

There is an algorithm data member. + for primal variations and - for dual variations

Definition at line 70 of file ClpSimplex.hpp.

4.72.2 Member Enumeration Documentation

4.72.2.1 enum **ClpSimplex::Status**

enums for status of various sorts.

First 4 match **CoinWarmStartBasis**, isFixed means fixed at lower bound and out of basis

Definition at line 78 of file ClpSimplex.hpp.

4.72.3 Constructor & Destructor Documentation

4.72.3.1 ClpSimplex::ClpSimplex (const **ClpSimplex** & *rhs*, int *scalingMode* = `-1`)

Copy constructor.

May scale depending on mode -1 leave mode as is 0 -off, 1 equilibrium, 2 geometric, 3, auto, 4 dynamic(later)

4.72.3.2 ClpSimplex::ClpSimplex (const **ClpModel** & *rhs*, int *scalingMode* = `-1`)

Copy constructor from model.

May scale depending on mode -1 leave mode as is 0 -off, 1 equilibrium, 2 geometric, 3, auto, 4 dynamic(later)

4.72.3.3 ClpSimplex::ClpSimplex (const **ClpModel** * *wholeModel*, int *numberRows*, const int * *whichRows*, int *numberColumns*, const int * *whichColumns*, bool *dropNames* = `true`, bool *dropIntegers* = `true`, bool *fixOthers* = `false`)

Subproblem constructor.

A subset of whole model is created from the row and column lists given. The new order is given by list order and duplicates are allowed. Name and integer information can be dropped Can optionally modify rhs to take into account variables NOT in list in this case duplicates are not allowed (also see getbackSolution)

4.72.3.4 ClpSimplex::ClpSimplex (const ClpSimplex * *wholeModel*, int *numberRows*, const int * *whichRows*, int *numberColumns*, const int * *whichColumns*, bool *dropNames* = `true`, bool *dropIntegers* = `true`, bool *fixOthers* = `false`)

Subproblem constructor.

A subset of whole model is created from the row and column lists given. The new order is given by list order and duplicates are allowed. Name and integer information can be dropped Can optionally modify rhs to take into account variables NOT in list in this case duplicates are not allowed (also see getbackSolution)

4.72.3.5 ClpSimplex::ClpSimplex (ClpSimplex * *wholeModel*, int *numberColumns*, const int * *whichColumns*)

This constructor modifies original ClpSimplex and stores original stuff in created ClpSimplex.

It is only to be used in conjunction with originalModel

4.72.4 Member Function Documentation

4.72.4.1 void ClpSimplex::originalModel (ClpSimplex * *miniModel*)

This copies back stuff from miniModel and then deletes miniModel.

Only to be used with mini constructor

4.72.4.2 void ClpSimplex::loadProblem (const ClpMatrixBase & *matrix*, const double * *collb*, const double * *colub*, const double * *obj*, const double * *rowlb*, const double * *rowub*, const double * *rowObjective* = `NULL`)

Loads a problem (the constraints on the rows are given by lower and upper bounds).

If a pointer is 0 then the following values are the default:

- `colub`: all columns have upper bound infinity
- `collb`: all columns have lower bound 0
- `rowub`: all rows have upper bound infinity
- `rowlb`: all rows have lower bound -infinity
- `obj`: all variables have 0 objective coefficient

4.72.4.3 void ClpSimplex::loadProblem (const int *numcols*, const int *numrows*, const CoinBigIndex * *start*, const int * *index*, const double * *value*, const double * *collb*, const double * *colub*, const double * *obj*, const double * *rowlb*, const double * *rowub*, const double * *rowObjective* = `NULL`)

Just like the other loadProblem() method except that the matrix is given in a standard column major ordered format (without gaps).

4.72.4.4 int ClpSimplex::loadProblem (CoinModel & *modelObject*, bool *keepSolution* = `false`)

This loads a model from a coinModel object - returns number of errors.

If keepSolution true and size is same as current then keeps current status and solution

4.72.4.5 int ClpSimplex::readLp (const char * *filename*, const double *epsilon* = 1e-5)

Read file in LP format from file with name filename.

See class **CoinLpIO** for description of this format.

4.72.4.6 void ClpSimplex::borrowModel (ClpModel & *otherModel*)

Borrow model.

This is so we dont have to copy large amounts of data around. It assumes a derived class wants to overwrite an empty model with a real one - while it does an algorithm. This is same as ClpModel one, but sets scaling on etc.

4.72.4.7 int ClpSimplex::initialSolve (ClpSolve & *options*)

General solve algorithm which can do presolve.

See ClpSolve.hpp for options

4.72.4.8 int ClpSimplex::dual (int *ifValuesPass* = 0, int *startFinishOptions* = 0)

Dual algorithm - see ClpSimplexDual.hpp for method.

ifValuesPass==2 just does values pass and then stops.

startFinishOptions - bits 1 - do not delete work areas and factorization at end 2 - use old factorization if same number of rows 4 - skip as much initialization of work areas as possible (based on whatsChanged in clpmodel.hpp) ** work in progress maybe other bits later

4.72.4.9 int ClpSimplex::primal (int *ifValuesPass* = 0, int *startFinishOptions* = 0)

Primal algorithm - see ClpSimplexPrimal.hpp for method.

ifValuesPass==2 just does values pass and then stops.

startFinishOptions - bits 1 - do not delete work areas and factorization at end 2 - use old factorization if same number of rows 4 - skip as much initialization of work areas as possible (based on whatsChanged in clpmodel.hpp) ** work in progress maybe other bits later

4.72.4.10 int ClpSimplex::nonlinearSLP (int *numberPasses*, double *deltaTolerance*)

Solves nonlinear problem using SLP - may be used as crash for other algorithms when number of iterations small.

Also exits if all problematical variables are changing less than deltaTolerance

4.72.4.11 int ClpSimplex::nonlinearSLP (int *numberConstraints*, ClpConstraint ** *constraints*, int *numberPasses*, double *deltaTolerance*)

Solves problem with nonlinear constraints using SLP - may be used as crash for other algorithms when number of iterations small.

Also exits if all problematical variables are changing less than deltaTolerance

4.72.4.12 int ClpSimplex::barrier (bool *crossover* = `true`)

Solves using barrier (assumes you have good cholesky factor code).

Does crossover to simplex if asked

4.72.4.13 int ClpSimplex::reducedGradient (int *phase* = `0`)

Solves non-linear using reduced gradient.

Phase = 0 get feasible, =1 use solution

4.72.4.14 int ClpSimplex::loadProblem (CoinStructuredModel & *modelObject*, bool *originalOrder* = `true`, bool *keepSolution* = `false`)

This loads a model from a **CoinStructuredModel** object - returns number of errors.

If originalOrder then keep to order stored in blocks, otherwise first column/rows correspond to first block - etc. If keep↩ Solution true and size is same as current then keeps current status and solution

4.72.4.15 int ClpSimplex::cleanup (int *cleanupScaling*)

When scaling is on it is possible that the scaled problem is feasible but the unscaled is not.

Clp returns a secondary status code to that effect. This option allows for a cleanup. If you use it I would suggest 1. This only affects actions when scaled optimal 0 - no action 1 - clean up using dual if primal infeasibility 2 - clean up using dual if dual infeasibility 3 - clean up using dual if primal or dual infeasibility 11,12,13 - as 1,2,3 but use primal

return code as dual/primal

4.72.4.16 int ClpSimplex::dualRanging (int *numberCheck*, const int ∗ *which*, double ∗ *costIncrease*, int ∗ *sequenceIncrease*, double ∗ *costDecrease*, int ∗ *sequenceDecrease*, double ∗ *valueIncrease* = `NULL`, double ∗ *valueDecrease* = `NULL`)

Dual ranging.

This computes increase/decrease in cost for each given variable and corresponding sequence numbers which would change basis. Sequence numbers are 0..numberColumns and numberColumns.. for artificials/slacks. For non-basic variables the information is trivial to compute and the change in cost is just minus the reduced cost and the sequence number will be that of the non-basic variables. For basic variables a ratio test is between the reduced costs for non-basic variables and the row of the tableau corresponding to the basic variable. The increase/decrease value is always >= 0.0

Up to user to provide correct length arrays where each array is of length numberCheck. which contains list of variables for which information is desired. All other arrays will be filled in by function. If fifth entry in which is variable 7 then fifth entry in output arrays will be information for variable 7.

If valueIncrease/Decrease not NULL (both must be NULL or both non NULL) then these are filled with the value of variable if such a change in cost were made (the existing bounds are ignored)

Returns non-zero if infeasible unbounded etc

4.72.4.17 int ClpSimplex::primalRanging (int *numberCheck*, const int ∗ *which*, double ∗ *valueIncrease*, int ∗ *sequenceIncrease*, double ∗ *valueDecrease*, int ∗ *sequenceDecrease*)

Primal ranging.

This computes increase/decrease in value for each given variable and corresponding sequence numbers which would change basis. Sequence numbers are 0..numberColumns and numberColumns.. for artificials/slacks. This should only be used for non-basic variabls as otherwise information is pretty useless For basic variables the sequence number will be that of the basic variables.

Up to user to provide correct length arrays where each array is of length numberCheck. which contains list of variables for which information is desired. All other arrays will be filled in by function. If fifth entry in which is variable 7 then fifth entry in output arrays will be information for variable 7.

Returns non-zero if infeasible unbounded etc

4.72.4.18 int ClpSimplex::modifyCoefficientsAndPivot (int *number*, const int ∗ *which*, const CoinBigIndex ∗ *start*, const int ∗ *row*, const double ∗ *newCoefficient*, const unsigned char ∗ *newStatus* = `NULL`, const double ∗ *newLower* = `NULL`, const double ∗ *newUpper* = `NULL`, const double ∗ *newObjective* = `NULL`)

Modifies coefficients etc and if necessary pivots in and out.

All at same status will be done (basis may go singular). User can tell which others have been done (i.e. if status matches). If called from outside will change status and return 0. If called from event handler returns non-zero if user has to take action. indices>=numberColumns are slacks (obviously no coefficients) status array is (char) Status enum

4.72.4.19 int ClpSimplex::outDuplicateRows (int *numberLook*, int ∗ *whichRows*, bool *noOverlaps* = `false`, double *tolerance* = `-1.0`, double *cleanUp* = `0.0`)

Take out duplicate rows (includes scaled rows and intersections).

On exit whichRows has rows to delete - return code is number can be deleted or -1 if would be infeasible. If tolerance is -1.0 use primalTolerance for equality rows and infeasibility If cleanUp not zero then spend more time trying to leave more stable row and make row bounds exact multiple of cleanUp if close enough

4.72.4.20 int ClpSimplex::writeBasis (const char ∗ *filename*, bool *writeValues* = `false`, int *formatType* = `0`) const

Write the basis in MPS format to the specified file.

If writeValues true writes values of structurals (and adds VALUES to end of NAME card)

Row and column names may be null. formatType is

- 0 - normal
- 1 - extra accuracy
- 2 - IEEE hex (later)

Returns non-zero on I/O error

4.72.4.21 int ClpSimplex::tightenPrimalBounds (double *factor* = `0.0`, int *doTight* = `0`, bool *tightIntegers* = `false`)

Tightens primal bounds to make dual faster.

Unless fixed or doTight>10, bounds are slightly looser than they could be. This is to make dual go faster and is probably not needed with a presolve. Returns non-zero if problem infeasible.

Fudge for branch and bound - put bounds on columns of factor ∗ largest value (at continuous) - should improve stability in branch and bound on infeasible branches (0.0 is off)

4.72.4.22 int ClpSimplex::crash (double *gap*, int *pivot*)

Crash - at present just aimed at dual, returns -2 if dual preferred and crash basis created -1 if dual preferred and all slack basis preferred 0 if basis going in was not all slack 1 if primal preferred and all slack basis preferred 2 if primal preferred and crash basis created.

if gap between bounds <="gap" variables can be flipped (If pivot -1 then can be made super basic!)

If "pivot" is -1 No pivoting - always primal 0 No pivoting (so will just be choice of algorithm) 1 Simple pivoting e.g. gub 2 Mini iterations

4.72.4.23 int ClpSimplex::strongBranching (int *numberVariables*, const int ∗ *variables*, double ∗ *newLower*, double ∗ *newUpper*, double ∗∗ *outputSolution*, int ∗ *outputStatus*, int ∗ *outputIterations*, bool *stopOnFirstInfeasible* = `true`, bool *alwaysFinish* = `false`, int *startFinishOptions* = `0`)

For strong branching.

On input lower and upper are new bounds while on output they are change in objective function values (>1.0e50 infeasible). Return code is 0 if nothing interesting, -1 if infeasible both ways and +1 if infeasible one way (check values to see which one(s)) Solutions are filled in as well - even down, odd up - also status and number of iterations

4.72.4.24 int ClpSimplex::fathomMany (void ∗ *stuff*)

Do up to N deep - returns -1 - no solution nNodes_ valid nodes >= if solution and that node gives solution ClpNode array is 2∗∗N long.

Values for N and array are in stuff (nNodes_ also in stuff)

4.72.4.25 int ClpSimplex::pivot ( )

Pivot in a variable and out a variable.

Returns 0 if okay, 1 if inaccuracy forced re-factorization, -1 if would be singular. Also updates primal/dual infeasibilities. Assumes sequenceIn_ and pivotRow_ set and also directionIn and Out.

4.72.4.26 int ClpSimplex::primalPivotResult ( )

Pivot in a variable and choose an outgoing one.

Assumes primal feasible - will not go through a bound. Returns step length in theta Returns ray in ray_ (or NULL if no pivot) Return codes as before but -1 means no acceptable pivot

4.72.4.27 int ClpSimplex::dualPivotResultPart1 ( )

Pivot out a variable and choose an incoing one.

Assumes dual feasible - will not go through a reduced cost. Returns step length in theta Return codes as before but -1 means no acceptable pivot

4.72.4.28 int ClpSimplex::startup (int *ifValuesPass*, int *startFinishOptions* = `0`)

Common bits of coding for dual and primal.

Return 0 if okay, 1 if bad matrix, 2 if very bad factorization startFinishOptions - bits 1 - do not delete work areas and factorization at end 2 - use old factorization if same number of rows 4 - skip as much initialization of work areas as possible (based on whatsChanged in clpmodel.hpp) ** work in progress maybe other bits later

4.72.4.29 bool ClpSimplex::statusOfProblem (bool *initial* = false)

Factorizes and returns true if optimal.

Used by user

4.72.4.30 int ClpSimplex::perturbation ( ) const [inline]

Amount of print out: 0 - none 1 - just final 2 - just factorizations 3 - as 2 plus a bit more 4 - verbose above that 8,16,32 etc just for selective debug.

Perturbation: 50 - switch on perturbation 100 - auto perturb if takes too long (1.0e-6 largest nonzero) 101 - we are perturbed 102 - don't try perturbing again default is 100 others are for playing

Definition at line 642 of file ClpSimplex.hpp.

4.72.4.31 int ClpSimplex::saveModel (const char * *fileName*)

Save model to file, returns 0 if success.

This is designed for use outside algorithms so does not save iterating arrays etc. It does not save any messaging information. Does not save scaling values. It does not know about all types of virtual functions.

4.72.4.32 void ClpSimplex::checkSolution (int *setToBounds* = 0)

Just check solution (for external use) - sets sum of infeasibilities etc.

If setToBounds 0 then primal column values not changed and used to compute primal row activity values. If 1 or 2 then status used - so all nonbasic variables set to indicated bound and if any values changed (or ==2) basic values re-computed.

4.72.4.33 void ClpSimplex::checkSolutionInternal ( )

Just check solution (for internal use) - sets sum of infeasibilities etc.

4.72.4.34 int ClpSimplex::getSolution (const double * *rowActivities,* const double * *columnActivities*)

Given an existing factorization computes and checks primal and dual solutions.

Uses input arrays for variables at bounds. Returns feasibility states

4.72.4.35 int ClpSimplex::getSolution ( )

Given an existing factorization computes and checks primal and dual solutions.

Uses current problem arrays for bounds. Returns feasibility states

4.72.4.36 int ClpSimplex::createPiecewiseLinearCosts (const int * *starts*, const double * *lower*, const double * *gradient*)

Constructs a non linear cost from list of non-linearities (columns only) First lower of each column is taken as real lower Last lower is taken as real upper and cost ignored.

Returns nonzero if bad data e.g. lowers not monotonic

4.72.4.37 int ClpSimplex::internalFactorize (int *solveType*)

Factorizes using current basis.

solveType - 1 iterating, 0 initial, -1 external If 10 added then in primal values pass Return codes are as from Clp↩ Factorization unless initial factorization when total number of singularities is returned. Special case is numberRows_+1 -> all slack basis.

4.72.4.38 void ClpSimplex::computeDuals (double * *givenDjs*)

Computes duals from scratch.

If givenDjs then allows for nonzero basic djs

4.72.4.39 int ClpSimplex::housekeeping (double *objectiveChange*) [protected]

This does basis housekeeping and does values for in/out variables.

Can also decide to re-factorize

4.72.4.40 double ClpSimplex::scaleObjective (double *value*) [protected]

If input negative scales objective so maximum <= -value and returns scale factor used.

If positive unscales and also redoes dual stuff

4.72.4.41 void ClpSimplex::setValuesPassAction (double *incomingInfeasibility*, double *allowedInfeasibility*)

For advanced use.

When doing iterative solves things can get nasty so on values pass if incoming solution has largest infeasibility < incomingInfeasibility throw out variables from basis until largest infeasibility < allowedInfeasibility or incoming largest infeasibility. If allowedInfeasibility>= incomingInfeasibility this is always possible altough you may end up with an all slack basis.

Defaults are 1.0,10.0

4.72.4.42 int ClpSimplex::cleanFactorization (int *ifValuesPass*)

Get a clean factorization - i.e.

throw out singularities may do more later

4.72.4.43 void ClpSimplex::setDisasterHandler (ClpDisasterHandler * *handler*) [inline]

Objective value.

Set disaster handler

Definition at line 880 of file ClpSimplex.hpp.

4.72.4.44 int ClpSimplex::gutsOfSolution (double * *givenDuals,* const double * *givenPrimals,* bool *valuesPass =* false) [protected]

May change basis and then returns number changed.

Computation of solutions may be overriden by given pi and solution

4.72.4.45 bool ClpSimplex::createRim (int *what,* bool *makeRowCopy =* false*,* int *startFinishOptions =* 0) [protected]

puts in format I like (rowLower,rowUpper) also see StandardMatrix 1 bit does rows (now and columns), (2 bit does column bounds), 4 bit does objective(s).

8 bit does solution scaling in 16 bit does rowArray and columnArray indexed vectors and makes row copy if wanted, also sets columnStart_ etc Also creates scaling arrays if needed. It does scaling if needed. 16 also moves solutions etc in to work arrays On 16 returns false if problem "bad" i.e. matrix or bounds bad If startFinishOptions is -1 then called by user in getSolution so do arrays but keep pivotVariable_

4.72.4.46 void ClpSimplex::deleteRim (int *getRidOfFactorizationData =* 2) [protected]

releases above arrays and does solution scaling out.

May also get rid of factorization data - 0 get rid of nothing, 1 get rid of arrays, 2 also factorization

4.72.4.47 double* ClpSimplex::solutionRegion (int *section*) const [inline]

Return row or column sections - not as much needed as it once was.

These just map into single arrays

Definition at line 1018 of file ClpSimplex.hpp.

4.72.4.48 void ClpSimplex::setInitialDenseFactorization (bool *onOff*)

Normally the first factorization does sparse coding because the factorization could be singular.

This allows initial dense factorization when it is known to be safe

4.72.4.49 void ClpSimplex::createStatus ( )

Set up status array (can be used by OsiClp).

Also can be used to set up all slack basis

4.72.4.50 double* ClpSimplex::infeasibilityRay (bool *fullRay =* false) const

Infeasibility/unbounded ray (NULL returned if none/wrong) Up to user to use delete [] on these arrays.

4.72.4.51 int ClpSimplex::numberExtraRows ( ) const [inline]

Number of extra rows.

These are ones which will be dynamically created each iteration. This is for GUB but may have other uses.

Definition at line 1354 of file ClpSimplex.hpp.

4.72.4.52 void ClpSimplex::setColumnLower (int *elementIndex*, double *elementValue*)

Set a single column lower bound
Use -DBL_MAX for -infinity.

4.72.4.53 void ClpSimplex::setColumnUpper (int *elementIndex*, double *elementValue*)

Set a single column upper bound
Use DBL_MAX for infinity.

4.72.4.54 void ClpSimplex::setColumnSetBounds (const int ∗ *indexFirst*, const int ∗ *indexLast*, const double ∗ *boundList*)

Set the bounds on a number of columns simultaneously
The default implementation just invokes setColLower() and setColUpper() over and over again.

Parameters

index↩ First,indexLast	pointers to the beginning and after the end of the array of the indices of the variables whose *either* bound changes
boundList	the new lower/upper bound pairs for the variables

4.72.4.55 void ClpSimplex::setColLower (int *elementIndex*, double *elementValue*) [inline]

Set a single column lower bound
Use -DBL_MAX for -infinity.

Definition at line 1435 of file ClpSimplex.hpp.

4.72.4.56 void ClpSimplex::setColUpper (int *elementIndex*, double *elementValue*) [inline]

Set a single column upper bound
Use DBL_MAX for infinity.

Definition at line 1440 of file ClpSimplex.hpp.

4.72.4.57 void ClpSimplex::setColSetBounds (const int ∗ *indexFirst*, const int ∗ *indexLast*, const double ∗ *boundList*) [inline]

Set the bounds on a number of columns simultaneously

Parameters

index←→ First,indexLast	pointers to the beginning and after the end of the array of the indices of the variables whose *either* bound changes
boundList	the new lower/upper bound pairs for the variables

Definition at line 1456 of file ClpSimplex.hpp.

4.72.4.58 void ClpSimplex::setRowLower (int *elementIndex*, double *elementValue*)

Set a single row lower bound
Use -DBL_MAX for -infinity.

4.72.4.59 void ClpSimplex::setRowUpper (int *elementIndex*, double *elementValue*)

Set a single row upper bound
Use DBL_MAX for infinity.

4.72.4.60 void ClpSimplex::setRowSetBounds (const int ∗ *indexFirst*, const int ∗ *indexLast*, const double ∗ *boundList*)

Set the bounds on a number of rows simultaneously

Parameters

index←→ First,indexLast	pointers to the beginning and after the end of the array of the indices of the constraints whose *either* bound changes
boundList	the new lower/upper bound pairs for the constraints

4.72.5 Friends And Related Function Documentation

4.72.5.1 void ClpSimplexUnitTest (const std::string & *mpsDir*) [friend]

A function that tests the methods in the ClpSimplex class.

The only reason for it not to be a member method is that this way it doesn't have to be compiled into the library. And that's a gain, because the library should be compiled with optimization on, but this method should be compiled with debugging.

It also does some testing of ClpFactorization class

4.72.6 Member Data Documentation

4.72.6.1 ClpNonLinearCost∗ ClpSimplex::nonLinearCost_ [protected]

Very wasteful way of dealing with infeasibilities in primal.

However it will allow non-linearities and use of dual analysis. If it doesn't work it can easily be replaced.

Definition at line 1654 of file ClpSimplex.hpp.

4.72.6.2 int ClpSimplex::numberExtraRows_ [protected]

Number of extra rows.

These are ones which will be dynamically created each iteration. This is for GUB but may have other uses.

Definition at line 1670 of file ClpSimplex.hpp.

4.72.6.3 double ClpSimplex::incomingInfeasibility_ [protected]

For advanced use.

When doing iterative solves things can get nasty so on values pass if incoming solution has largest infeasibility $<$ incomingInfeasibility throw out variables from basis until largest infeasibility $<$ allowedInfeasibility. if allowed←Infeasibility$>=$ incomingInfeasibility this is always possible altough you may end up with an all slack basis.

Defaults are 1.0,10.0

Definition at line 1685 of file ClpSimplex.hpp.

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

- ClpSimplex.hpp

4.73 ClpSimplexDual Class Reference

This solves LPs using the dual simplex method.

`#include <ClpSimplexDual.hpp>`

Inheritance diagram for ClpSimplexDual:

Collaboration diagram for ClpSimplexDual:

Public Member Functions

Description of algorithm

- int dual (int ifValuesPass, int startFinishOptions=0)
  Dual algorithm.
- int strongBranching (int numberVariables, const int ∗variables, double ∗newLower, double ∗newUpper, double ∗∗outputSolution, int ∗outputStatus, int ∗outputIterations, bool stopOnFirstInfeasible=true, bool always←Finish=false, int startFinishOptions=0)
  For strong branching.
- ClpFactorization ∗ setupForStrongBranching (char ∗arrays, int numberRows, int numberColumns, bool solve←Lp=false)
  This does first part of StrongBranching.
- void cleanupAfterStrongBranching (ClpFactorization ∗factorization)
  This cleans up after strong branching.

Functions used in dual

- int whileIterating (double ∗&givenPi, int ifValuesPass)
  This has the flow between re-factorizations Broken out for clarity and will be used by strong branching.
- int updateDualsInDual (**CoinIndexedVector** ∗rowArray, **CoinIndexedVector** ∗columnArray, **CoinIndexed←Vector** ∗outputArray, double theta, double &objectiveChange, bool fullRecompute)

The duals are updated by the given arrays.

- void updateDualsInValuesPass (**CoinIndexedVector** ∗rowArray, **CoinIndexedVector** ∗columnArray, double theta)

  The duals are updated by the given arrays.
- void flipBounds (**CoinIndexedVector** ∗rowArray, **CoinIndexedVector** ∗columnArray)

  While updateDualsInDual sees what effect is of flip this does actual flipping.
- double dualColumn (**CoinIndexedVector** ∗rowArray, **CoinIndexedVector** ∗columnArray, **CoinIndexedVector** ∗spareArray, **CoinIndexedVector** ∗spareArray2, double accpetablePivot, CoinBigIndex ∗dubiousWeights)

  Row array has row part of pivot row Column array has column part.
- int dualColumn0 (const **CoinIndexedVector** ∗rowArray, const **CoinIndexedVector** ∗columnArray, **Coin**←↩ **IndexedVector** ∗spareArray, double acceptablePivot, double &upperReturn, double &bestReturn, double &badFree)

  Does first bit of dualColumn.
- void checkPossibleValuesMove (**CoinIndexedVector** ∗rowArray, **CoinIndexedVector** ∗columnArray, double acceptablePivot)

  Row array has row part of pivot row Column array has column part.
- void checkPossibleCleanup (**CoinIndexedVector** ∗rowArray, **CoinIndexedVector** ∗columnArray, double acceptablePivot)

  Row array has row part of pivot row Column array has column part.
- void doEasyOnesInValuesPass (double ∗givenReducedCosts)

  This sees if we can move duals in dual values pass.
- void dualRow (int alreadyChosen)

  Chooses dual pivot row Would be faster with separate region to scan and will have this (with square of infeasibility) when steepest For easy problems we can just choose one of the first rows we look at.
- int changeBounds (int initialize, **CoinIndexedVector** ∗outputArray, double &changeCost)

  Checks if any fake bounds active - if so returns number and modifies updatedDualBound_ and everything.
- bool changeBound (int iSequence)

  As changeBounds but just changes new bounds for a single variable.
- void originalBound (int iSequence)

  Restores bound to original bound.
- int checkUnbounded (**CoinIndexedVector** ∗ray, **CoinIndexedVector** ∗spare, double changeCost)

  Checks if tentative optimal actually means unbounded in dual Returns -3 if not, 2 if is unbounded.
- void statusOfProblemInDual (int &lastCleaned, int type, double ∗givenDjs, ClpDataSave &saveData, int if←↩ ValuesPass)

  Refactorizes if necessary Checks if finished.
- int perturb ()

  Perturbs problem (method depends on perturbation()) returns nonzero if should go to dual.
- int fastDual (bool alwaysFinish=false)

  Fast iterations.
- int numberAtFakeBound ()

  Checks number of variables at fake bounds.
- int pivotResultPart1 ()

  Pivot in a variable and choose an outgoing one.
- int nextSuperBasic ()

  Get next free , -1 if none.
- int startupSolve (int ifValuesPass, double ∗saveDuals, int startFinishOptions)

  Startup part of dual (may be extended to other algorithms) returns 0 if good, 1 if bad.
- void **finishSolve** (int startFinishOptions)
- void **gutsOfDual** (int ifValuesPass, double ∗&saveDuals, int initialStatus, ClpDataSave &saveData)
- void **resetFakeBounds** (int type)

Additional Inherited Members

4.73.1 Detailed Description

This solves LPs using the dual simplex method.

It inherits from ClpSimplex. It has no data of its own and is never created - only cast from a ClpSimplex object at algorithm time.

Definition at line 23 of file ClpSimplexDual.hpp.

4.73.2 Member Function Documentation

4.73.2.1 int ClpSimplexDual::dual (int *ifValuesPass*, int *startFinishOptions* = 0)

Dual algorithm.

Method

It tries to be a single phase approach with a weight of 1.0 being given to getting optimal and a weight of updatedDual←Bound_ being given to getting dual feasible. In this version I have used the idea that this weight can be thought of as a fake bound. If the distance between the lower and upper bounds on a variable is less than the feasibility weight then we are always better off flipping to other bound to make dual feasible. If the distance is greater then we make up a fake bound updatedDualBound_ away from one bound. If we end up optimal or primal infeasible, we check to see if bounds okay. If so we have finished, if not we increase updatedDualBound_ and continue (after checking if unbounded). I am undecided about free variables - there is coding but I am not sure about it. At present I put them in basis anyway.

The code is designed to take advantage of sparsity so arrays are seldom zeroed out from scratch or gone over in their entirety. The only exception is a full scan to find outgoing variable for Dantzig row choice. For steepest edge we keep an updated list of infeasibilities (actually squares). On easy problems we don't need full scan - just pick first reasonable.

One problem is how to tackle degeneracy and accuracy. At present I am using the modification of costs which I put in OSL and some of what I think is the dual analog of Gill et al. I am still not sure of the exact details.

The flow of dual is three while loops as follows:

while (not finished) {

while (not clean solution) {

Factorize and/or clean up solution by flipping variables so dual feasible. If looks finished check fake dual bounds. Repeat until status is iterating (-1) or finished (0,1,2)

}

while (status==-1) {

Iterate until no pivot in or out or time to re-factorize.

Flow is:

choose pivot row (outgoing variable). if none then we are primal feasible so looks as if done but we need to break and check bounds etc.

Get pivot row in tableau

```
Choose incoming column.  If we don't find one then we look
```

primal infeasible so break and check bounds etc. (Also the pivot tolerance is larger after any iterations so that may be reason)

```
If we do find incoming column, we may have to adjust costs to
```

keep going forwards (anti-degeneracy). Check pivot will be stable and if unstable throw away iteration and break to re-factorize. If minor error re-factorize after iteration.

Update everything (this may involve flipping variables to stay dual feasible.

}

}

TODO's (or maybe not)

At present we never check we are going forwards. I overdid that in OSL so will try and make a last resort.

Needs partial scan pivot out option.

May need other anti-degeneracy measures, especially if we try and use loose tolerances as a way to solve in fewer iterations.

I like idea of dynamic scaling. This gives opportunity to decouple different implications of scaling for accuracy, iteration count and feasibility tolerance.

for use of exotic parameter startFinishoptions see Clpsimplex.hpp

4.73.2.2 int ClpSimplexDual::strongBranching (int *numberVariables*, const int ∗ *variables*, double ∗ *newLower*, double ∗ *newUpper*, double ∗∗ *outputSolution*, int ∗ *outputStatus*, int ∗ *outputIterations*, bool *stopOnFirstInfeasible* = `true`, bool *alwaysFinish* = `false`, int *startFinishOptions* = `0`)

For strong branching.

On input lower and upper are new bounds while on output they are change in objective function values (>1.0e50 infeasible). Return code is 0 if nothing interesting, -1 if infeasible both ways and +1 if infeasible one way (check values to see which one(s)) Solutions are filled in as well - even down, odd up - also status and number of iterations

4.73.2.3 int ClpSimplexDual::whileIterating (double ∗& *givenPi*, int *ifValuesPass*)

This has the flow between re-factorizations Broken out for clarity and will be used by strong branching.

Reasons to come out: -1 iterations etc -2 inaccuracy -3 slight inaccuracy (and done iterations) +0 looks optimal (might be unbounded - but we will investigate) +1 looks infeasible +3 max iterations

If givenPi not NULL then in values pass

4.73.2.4 int ClpSimplexDual::updateDualsInDual (CoinIndexedVector ∗ *rowArray*, CoinIndexedVector ∗ *columnArray*, CoinIndexedVector ∗ *outputArray*, double *theta*, double & *objectiveChange*, bool *fullRecompute*)

The duals are updated by the given arrays.

Returns number of infeasibilities. After rowArray and columnArray will just have those which have been flipped. Variables may be flipped between bounds to stay dual feasible. The output vector has movement of primal solution (row length array)

4.73.2.5 void ClpSimplexDual::updateDualsInValuesPass (CoinIndexedVector ∗ *rowArray*, CoinIndexedVector ∗ *columnArray*, double *theta*)

The duals are updated by the given arrays.

This is in values pass - so no changes to primal is made

4.73.2.6 double ClpSimplexDual::dualColumn (CoinIndexedVector * *rowArray*, CoinIndexedVector * *columnArray*, CoinIndexedVector * *spareArray*, CoinIndexedVector * *spareArray2*, double *accpetablePivot*, CoinBigIndex * *dubiousWeights*)

Row array has row part of pivot row Column array has column part.

This chooses pivot column. Spare arrays are used to save pivots which will go infeasible We will check for basic so spare array will never overflow. If necessary will modify costs For speed, we may need to go to a bucket approach when many variables are being flipped. Returns best possible pivot value

4.73.2.7 void ClpSimplexDual::checkPossibleValuesMove (CoinIndexedVector * *rowArray*, CoinIndexedVector * *columnArray*, double *acceptablePivot*)

Row array has row part of pivot row Column array has column part.

This sees what is best thing to do in dual values pass if sequenceIn==sequenceOut can change dual on chosen row and leave variable in basis

4.73.2.8 void ClpSimplexDual::checkPossibleCleanup (CoinIndexedVector * *rowArray*, CoinIndexedVector * *columnArray*, double *acceptablePivot*)

Row array has row part of pivot row Column array has column part.

This sees what is best thing to do in branch and bound cleanup If sequenceIn_ $<$ 0 then can't do anything

4.73.2.9 void ClpSimplexDual::doEasyOnesInValuesPass (double * *givenReducedCosts*)

This sees if we can move duals in dual values pass.

This is done before any pivoting

4.73.2.10 void ClpSimplexDual::dualRow (int *alreadyChosen*)

Chooses dual pivot row Would be faster with separate region to scan and will have this (with square of infeasibility) when steepest For easy problems we can just choose one of the first rows we look at.

If alreadyChosen $>$=0 then in values pass and that row has been selected

4.73.2.11 int ClpSimplexDual::changeBounds (int *initialize*, CoinIndexedVector * *outputArray*, double & *changeCost*)

Checks if any fake bounds active - if so returns number and modifies updatedDualBound_ and everything.

Free variables will be left as free Returns number of bounds changed if $>$=0 Returns -1 if not initialize and no effect Fills in changeVector which can be used to see if unbounded and cost of change vector If 2 sets to original (just changed)

4.73.2.12 bool ClpSimplexDual::changeBound (int *iSequence*)

As changeBounds but just changes new bounds for a single variable.

Returns true if change

4.73.2.13 void ClpSimplexDual::statusOfProblemInDual (int & *lastCleaned,* int *type,* double * *givenDjs,* ClpDataSave & *saveData,* int *ifValuesPass*)

Refactorizes if necessary Checks if finished.

Updates status. lastCleaned refers to iteration at which some objective/feasibility cleaning too place.

type - 0 initial so set up save arrays etc

- 1 normal -if good update save

2 restoring from saved

4.73.2.14 int ClpSimplexDual::fastDual (bool *alwaysFinish* = `false`)

Fast iterations.

Misses out a lot of initialization. Normally stops on maximum iterations, first re-factorization or tentative optimum. If looks interesting then continues as normal. Returns 0 if finished properly, 1 otherwise.

4.73.2.15 int ClpSimplexDual::numberAtFakeBound ( )

Checks number of variables at fake bounds.

This is used by fastDual so can exit gracefully before end

4.73.2.16 int ClpSimplexDual::pivotResultPart1 ( )

Pivot in a variable and choose an outgoing one.

Assumes dual feasible - will not go through a reduced cost. Returns step length in theta Return codes as before but -1 means no acceptable pivot

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

- ClpSimplexDual.hpp

4.74 ClpSimplexNonlinear Class Reference

This solves non-linear LPs using the primal simplex method.

`#include <ClpSimplexNonlinear.hpp>`

Inheritance diagram for ClpSimplexNonlinear:

Collaboration diagram for ClpSimplexNonlinear:

Public Member Functions

Description of algorithm

- int primal ()

    Primal algorithms for reduced gradient At present we have two algorithms:
- int primalSLP (int numberPasses, double deltaTolerance, int otherOptions=0)

Primal algorithm for quadratic Using a semi-trust region approach as for pooling problem This is in because I have it lying around.

- int primalDualCuts (char ∗rowsIn, int startUp, int algorithm)

  May use a cut approach for solving any LP.
- int primalSLP (int numberConstraints, ClpConstraint ∗∗constraints, int numberPasses, double deltaTolerance)

  Primal algorithm for nonlinear constraints Using a semi-trust region approach as for pooling problem This is in because I have it lying around.
- void directionVector (**CoinIndexedVector** ∗longArray, **CoinIndexedVector** ∗spare1, **CoinIndexedVector** ∗spare2, int mode, double &normFlagged, double &normUnflagged, int &numberNonBasic)

  Creates direction vector.
- int whileIterating (int &pivotMode)

  Main part.
- int pivotColumn (**CoinIndexedVector** ∗longArray, **CoinIndexedVector** ∗rowArray, **CoinIndexedVector** ∗columnArray, **CoinIndexedVector** ∗spare, int &pivotMode, double &solutionError, double ∗array1)

  longArray has direction pivotMode - 0 - use all dual infeasible variables 1 - largest dj while >= 10 trying startup phase Returns 0 - can do normal iteration (basis change) 1 - no basis change 2 - if wants singleton 3 - if time to re-factorize If sequenceIn_ >=0 then that will be incoming variable
- void statusOfProblemInPrimal (int &lastCleaned, int type, ClpSimplexProgress ∗progress, bool doFactorization, double &bestObjectiveWhenFlagged)

  Refactorizes if necessary Checks if finished.
- int pivotNonlinearResult ()

  Do last half of an iteration.

Additional Inherited Members

4.74.1 Detailed Description

This solves non-linear LPs using the primal simplex method.

It inherits from ClpSimplexPrimal. It has no data of its own and is never created - only cast from a ClpSimplexPrimal object at algorithm time. If needed create new class and pass around

Definition at line 28 of file ClpSimplexNonlinear.hpp.

4.74.2 Member Function Documentation

4.74.2.1 int ClpSimplexNonlinear::primal ( )

Primal algorithms for reduced gradient At present we have two algorithms:

A reduced gradient method.

4.74.2.2 int ClpSimplexNonlinear::primalSLP (int *numberConstraints,* **ClpConstraint** ∗∗ *constraints,* int *numberPasses,* double *deltaTolerance*)

Primal algorithm for nonlinear constraints Using a semi-trust region approach as for pooling problem This is in because I have it lying around.

4.74.2.3 void ClpSimplexNonlinear::directionVector (**CoinIndexedVector** ∗ *longArray,* **CoinIndexedVector** ∗ *spare1,* **CoinIndexedVector** ∗ *spare2,* int *mode,* double & *normFlagged,* double & *normUnflagged,* int & *numberNonBasic*)

Creates direction vector.

note longArray is long enough for rows and columns. If numberNonBasic 0 then is updated otherwise mode is ignored and those are used. Norms are only for those > 1.0e3∗dualTolerance If mode is nonzero then just largest dj

4.74.2.4 void ClpSimplexNonlinear::statusOfProblemInPrimal (int & *lastCleaned,* int *type,* ClpSimplexProgress ∗ *progress,* bool *doFactorization,* double & *bestObjectiveWhenFlagged*)

Refactorizes if necessary Checks if finished.

Updates status. lastCleaned refers to iteration at which some objective/feasibility cleaning too place.

type - 0 initial so set up save arrays etc

- 1 normal -if good update save

2 restoring from saved

4.74.2.5 int ClpSimplexNonlinear::pivotNonlinearResult ( )

Do last half of an iteration.

Return codes Reasons to come out normal mode -1 normal -2 factorize now - good iteration -3 slight inaccuracy - refactorize - iteration done -4 inaccuracy - refactorize - no iteration -5 something flagged - go round again +2 looks unbounded +3 max iterations (iteration done)

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

- ClpSimplexNonlinear.hpp

4.75 ClpSimplexOther Class Reference

This is for Simplex stuff which is neither dual nor primal.

`#include <ClpSimplexOther.hpp>`

Inheritance diagram for ClpSimplexOther:

Collaboration diagram for ClpSimplexOther:

Classes

- struct parametricsData

Methods

- void dualRanging (int numberCheck, const int ∗which, double ∗costIncrease, int ∗sequenceIncrease, double ∗costDecrease, int ∗sequenceDecrease, double ∗valueIncrease=NULL, double ∗valueDecrease=NULL)

  Dual ranging.
- void primalRanging (int numberCheck, const int ∗which, double ∗valueIncrease, int ∗sequenceIncrease, double ∗valueDecrease, int ∗sequenceDecrease)

  Primal ranging.
- int parametrics (double startingTheta, double &endingTheta, double reportIncrement, const double ∗changeLowerBound, const double ∗changeUpperBound, const double ∗changeLowerRhs, const double ∗changeUpperRhs, const double ∗changeObjective)

Parametrics This is an initial slow version.

- int parametrics (const char ∗dataFile)

  Version of parametrics which reads from file See CbcClpParam.cpp for details of format Returns -2 if unable to open file.
- int parametrics (double startingTheta, double &endingTheta, const double ∗changeLowerBound, const double ∗changeUpperBound, const double ∗changeLowerRhs, const double ∗changeUpperRhs)

  Parametrics This is an initial slow version.
- int **parametricsObj** (double startingTheta, double &endingTheta, const double ∗changeObjective)
- double bestPivot (bool justColumns=false)

  Finds best possible pivot.
- int writeBasis (const char ∗filename, bool writeValues=false, int formatType=0) const

  Write the basis in MPS format to the specified file.
- int readBasis (const char ∗filename)

  Read a basis from the given filename.
- ClpSimplex ∗ dualOfModel (double fractionRowRanges=1.0, double fractionColumnRanges=1.0) const

  Creates dual of a problem if looks plausible (defaults will always create model) fractionRowRanges is fraction of rows allowed to have ranges fractionColumnRanges is fraction of columns allowed to have ranges.
- int restoreFromDual (const ClpSimplex ∗dualProblem, bool checkAccuracy=false)

  Restores solution from dualized problem non-zero return code indicates minor problems.
- int setInDual (ClpSimplex ∗dualProblem)

  Sets solution in dualized problem non-zero return code indicates minor problems.
- ClpSimplex ∗ crunch (double ∗rhs, int ∗whichRows, int ∗whichColumns, int &nBound, bool moreBounds=false, bool tightenBounds=false)

  Does very cursory presolve.
- void afterCrunch (const ClpSimplex &small, const int ∗whichRows, const int ∗whichColumns, int nBound)

  After very cursory presolve.
- ClpSimplex ∗ gubVersion (int ∗whichRows, int ∗whichColumns, int neededGub, int factorizationFrequency=50)

  Returns gub version of model or NULL whichRows has to be numberRows whichColumns has to be number←Rows+numberColumns.
- void setGubBasis (ClpSimplex &original, const int ∗whichRows, const int ∗whichColumns)

  Sets basis from original.
- void getGubBasis (ClpSimplex &original, const int ∗whichRows, const int ∗whichColumns) const

  Restores basis to original.
- void cleanupAfterPostsolve ()

  Quick try at cleaning up duals if postsolve gets wrong.
- int tightenIntegerBounds (double ∗rhsSpace)

  Tightens integer bounds - returns number tightened or -1 if infeasible.
- int expandKnapsack (int knapsackRow, int &numberOutput, double ∗buildObj, CoinBigIndex ∗buildStart, int ∗buildRow, double ∗buildElement, int reConstruct=-1) const

  Expands out all possible combinations for a knapsack If buildObj NULL then just computes space needed - returns number elements On entry numberOutput is maximum allowed, on exit it is number needed or -1 (as will be number elements) if maximum exceeded.

Additional Inherited Members

4.75.1 Detailed Description

This is for Simplex stuff which is neither dual nor primal.

It inherits from ClpSimplex. It has no data of its own and is never created - only cast from a ClpSimplex object at algorithm time.

Definition at line 23 of file ClpSimplexOther.hpp.

4.75.2 Member Function Documentation

4.75.2.1 void ClpSimplexOther::dualRanging (int *numberCheck*, const int * *which*, double * *costIncrease*, int * *sequenceIncrease*, double * *costDecrease*, int * *sequenceDecrease*, double * *valueIncrease* = NULL, double * *valueDecrease* = NULL)

Dual ranging.

This computes increase/decrease in cost for each given variable and corresponding sequence numbers which would change basis. Sequence numbers are 0..numberColumns and numberColumns.. for artificials/slacks. For non-basic variables the information is trivial to compute and the change in cost is just minus the reduced cost and the sequence number will be that of the non-basic variables. For basic variables a ratio test is between the reduced costs for non-basic variables and the row of the tableau corresponding to the basic variable. The increase/decrease value is always >= 0.0

Up to user to provide correct length arrays where each array is of length numberCheck. which contains list of variables for which information is desired. All other arrays will be filled in by function. If fifth entry in which is variable 7 then fifth entry in output arrays will be information for variable 7.

If valueIncrease/Decrease not NULL (both must be NULL or both non NULL) then these are filled with the value of variable if such a change in cost were made (the existing bounds are ignored)

When here - guaranteed optimal

4.75.2.2 void ClpSimplexOther::primalRanging (int *numberCheck*, const int * *which*, double * *valueIncrease*, int * *sequenceIncrease*, double * *valueDecrease*, int * *sequenceDecrease*)

Primal ranging.

This computes increase/decrease in value for each given variable and corresponding sequence numbers which would change basis. Sequence numbers are 0..numberColumns and numberColumns.. for artificials/slacks. This should only be used for non-basic variabls as otherwise information is pretty useless For basic variables the sequence number will be that of the basic variables.

Up to user to provide correct length arrays where each array is of length numberCheck. which contains list of variables for which information is desired. All other arrays will be filled in by function. If fifth entry in which is variable 7 then fifth entry in output arrays will be information for variable 7.

When here - guaranteed optimal

4.75.2.3 int ClpSimplexOther::parametrics (double *startingTheta*, double & *endingTheta*, double *reportIncrement*, const double * *changeLowerBound*, const double * *changeUpperBound*, const double * *changeLowerRhs*, const double * *changeUpperRhs*, const double * *changeObjective*)

Parametrics This is an initial slow version.

The code uses current bounds + theta * change (if change array not NULL) and similarly for objective. It starts at startingTheta and returns ending theta in endingTheta. If reportIncrement 0.0 it will report on any movement If reportIncrement >0.0 it will report at startingTheta+k*reportIncrement. If it can not reach input endingTheta return code will be 1 for infeasible, 2 for unbounded, if error on ranges -1, otherwise 0. Normal report is just theta and objective but if event handler exists it may do more On exit endingTheta is maximum reached (can be used for next startingTheta)

4.75.2.4 int ClpSimplexOther::parametrics (double *startingTheta*, double & *endingTheta*, const double * *changeLowerBound*, const double * *changeUpperBound*, const double * *changeLowerRhs*, const double * *changeUpperRhs*)

Parametrics This is an initial slow version.

The code uses current bounds + theta * change (if change array not NULL) It starts at startingTheta and returns ending theta in endingTheta. If it can not reach input endingTheta return code will be 1 for infeasible, 2 for unbounded, if error on ranges -1, otherwise 0. Event handler may do more On exit endingTheta is maximum reached (can be used for next startingTheta)

4.75.2.5 int ClpSimplexOther::writeBasis (const char * *filename*, bool *writeValues* = false, int *formatType* = 0) const

Write the basis in MPS format to the specified file.

If writeValues true writes values of structurals (and adds VALUES to end of NAME card)

Row and column names may be null. formatType is

- 0 - normal
- 1 - extra accuracy
- 2 - IEEE hex (later)

Returns non-zero on I/O error

4.75.2.6 ClpSimplex* ClpSimplexOther::crunch (double * *rhs*, int * *whichRows*, int * *whichColumns*, int & *nBound*, bool *moreBounds* = false, bool *tightenBounds* = false)

Does very cursory presolve.

rhs is numberRows, whichRows is 3*numberRows and whichColumns is 2*numberColumns.

4.75.2.7 void ClpSimplexOther::afterCrunch (const ClpSimplex & *small*, const int * *whichRows*, const int * *whichColumns*, int *nBound*)

After very cursory presolve.

rhs is numberRows, whichRows is 3*numberRows and whichColumns is 2*numberColumns.

4.75.2.8 int ClpSimplexOther::expandKnapsack (int *knapsackRow*, int & *numberOutput*, double * *buildObj*, CoinBigIndex * *buildStart*, int * *buildRow*, double * *buildElement*, int *reConstruct* = -1) const

Expands out all possible combinations for a knapsack If buildObj NULL then just computes space needed - returns number elements On entry numberOutput is maximum allowed, on exit it is number needed or -1 (as will be number elements) if maximum exceeded.

numberOutput will have at least space to return values which reconstruct input. Rows returned will be original rows but no entries will be returned for any rows all of whose entries are in knapsack. So up to user to allow for this. If reConstruct >=0 then returns number of entrie which make up item "reConstruct" in expanded knapsack. Values in buildRow and buildElement;

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

- ClpSimplexOther.hpp

4.76 ClpSimplexPrimal Class Reference

This solves LPs using the primal simplex method.

`#include <ClpSimplexPrimal.hpp>`

Inheritance diagram for ClpSimplexPrimal:

Collaboration diagram for ClpSimplexPrimal:

Public Member Functions

Description of algorithm

- int primal (int ifValuesPass=0, int startFinishOptions=0)
  Primal algorithm.

For advanced users

- void alwaysOptimal (bool onOff)
  Do not change infeasibility cost and always say optimal.
- bool **alwaysOptimal** () const
- void exactOutgoing (bool onOff)
  Normally outgoing variables can go out to slightly negative values (but within tolerance) - this is to help stability and and degeneracy.
- bool **exactOutgoing** () const

Functions used in primal

- int whileIterating (int valuesOption)
  This has the flow between re-factorizations.
- int pivotResult (int ifValuesPass=0)
  Do last half of an iteration.
- int updatePrimalsInPrimal (**CoinIndexedVector** ∗rowArray, double theta, double &objectiveChange, int valuesPass)
  The primals are updated by the given array.
- void primalRow (**CoinIndexedVector** ∗rowArray, **CoinIndexedVector** ∗rhsArray, **CoinIndexedVector** ∗spareArray, int valuesPass)
  Row array has pivot column This chooses pivot row.
- void primalColumn (**CoinIndexedVector** ∗updateArray, **CoinIndexedVector** ∗spareRow1, **CoinIndexed**←**Vector** ∗spareRow2, **CoinIndexedVector** ∗spareColumn1, **CoinIndexedVector** ∗spareColumn2)
  Chooses primal pivot column updateArray has cost updates (also use pivotRow_ from last iteration) Would be faster with separate region to scan and will have this (with square of infeasibility) when steepest For easy problems we can just choose one of the first columns we look at.
- int checkUnbounded (**CoinIndexedVector** ∗ray, **CoinIndexedVector** ∗spare, double changeCost)
  Checks if tentative optimal actually means unbounded in primal Returns -3 if not, 2 if is unbounded.
- void statusOfProblemInPrimal (int &lastCleaned, int type, ClpSimplexProgress ∗progress, bool doFactorization, int ifValuesPass, ClpSimplex ∗saveModel=NULL)
  Refactorizes if necessary Checks if finished.
- void perturb (int type)
  Perturbs problem (method depends on perturbation())
- bool unPerturb ()
  Take off effect of perturbation and say whether to try dual.
- int unflag ()
  Unflag all variables and return number unflagged.
- int nextSuperBasic (int superBasicType, **CoinIndexedVector** ∗columnArray)
  Get next superbasic -1 if none, Normal type is 1 If type is 3 then initializes sorted list if 2 uses list.
- void primalRay (**CoinIndexedVector** ∗rowArray)
  Create primal ray.
- void clearAll ()
  Clears all bits and clears rowArray[1] etc.
- int lexSolve ()
  Sort of lexicographic resolve.

Additional Inherited Members

4.76.1 Detailed Description

This solves LPs using the primal simplex method.

It inherits from ClpSimplex. It has no data of its own and is never created - only cast from a ClpSimplex object at algorithm time.

Definition at line 23 of file ClpSimplexPrimal.hpp.

4.76.2 Member Function Documentation

4.76.2.1 int ClpSimplexPrimal::primal (int *ifValuesPass* = 0, int *startFinishOptions* = 0)

Primal algorithm.

Method

It tries to be a single phase approach with a weight of 1.0 being given to getting optimal and a weight of infeasibility↩ Cost_ being given to getting primal feasible. In this version I have tried to be clever in a stupid way. The idea of fake bounds in dual seems to work so the primal analogue would be that of getting bounds on reduced costs (by a presolve approach) and using these for being above or below feasible region. I decided to waste memory and keep these explicitly. This allows for non-linear costs! I have not tested non-linear costs but will be glad to do something if a reasonable example is provided.

The code is designed to take advantage of sparsity so arrays are seldom zeroed out from scratch or gone over in their entirety. The only exception is a full scan to find incoming variable for Dantzig row choice. For steepest edge we keep an updated list of dual infeasibilities (actually squares). On easy problems we don't need full scan - just pick first reasonable. This method has not been coded.

One problem is how to tackle degeneracy and accuracy. At present I am using the modification of costs which I put in OSL and which was extended by Gill et al. I am still not sure whether we will also need explicit perturbation.

The flow of primal is three while loops as follows:

while (not finished) {

while (not clean solution) {

Factorize and/or clean up solution by changing bounds so primal feasible. If looks finished check fake primal bounds. Repeat until status is iterating (-1) or finished (0,1,2)

}

while (status==-1) {

Iterate until no pivot in or out or time to re-factorize.

Flow is:

choose pivot column (incoming variable). if none then we are primal feasible so looks as if done but we need to break and check bounds etc.

Get pivot column in tableau

```
Choose outgoing row.  If we don't find one then we look
```

primal unbounded so break and check bounds etc. (Also the pivot tolerance is larger after any iterations so that may be reason)

```
If we do find outgoing row, we may have to adjust costs to
```

keep going forwards (anti-degeneracy). Check pivot will be stable and if unstable throw away iteration and break to re-factorize. If minor error re-factorize after iteration.

Update everything (this may involve changing bounds on variables to stay primal feasible.

}

}

TODO's (or maybe not)

At present we never check we are going forwards. I overdid that in OSL so will try and make a last resort.

Needs partial scan pivot in option.

May need other anti-degeneracy measures, especially if we try and use loose tolerances as a way to solve in fewer iterations.

I like idea of dynamic scaling. This gives opportunity to decouple different implications of scaling for accuracy, iteration count and feasibility tolerance.

for use of exotic parameter startFinishoptions see Clpsimplex.hpp

4.76.2.2 void ClpSimplexPrimal::exactOutgoing (bool *onOff*)

Normally outgoing variables can go out to slightly negative values (but within tolerance) - this is to help stability and and degeneracy.

This can be switched off

4.76.2.3 int ClpSimplexPrimal::whileIterating (int *valuesOption*)

This has the flow between re-factorizations.

Returns a code to say where decision to exit was made Problem status set to:

-2 re-factorize -4 Looks optimal/infeasible -5 Looks unbounded +3 max iterations

valuesOption has original value of valuesPass

4.76.2.4 int ClpSimplexPrimal::pivotResult (int *ifValuesPass =* 0)

Do last half of an iteration.

This is split out so people can force incoming variable. If solveType_ is 2 then this may re-factorize while normally it would exit to re-factorize. Return codes Reasons to come out (normal mode/user mode): -1 normal -2 factorize now - good iteration/ NA -3 slight inaccuracy - refactorize - iteration done/ same but factor done -4 inaccuracy - refactorize - no iteration/ NA -5 something flagged - go round again/ pivot not possible +2 looks unbounded +3 max iterations (iteration done)

With solveType_ ==2 this should Pivot in a variable and choose an outgoing one. Assumes primal feasible - will not go through a bound. Returns step length in theta Returns ray in ray_

4.76.2.5 int ClpSimplexPrimal::updatePrimalsInPrimal (CoinIndexedVector ∗ *rowArray,* double *theta,* double & *objectiveChange,* int *valuesPass*)

The primals are updated by the given array.

Returns number of infeasibilities. After rowArray will have cost changes for use next iteration

4.76.2.6 void ClpSimplexPrimal::primalRow (CoinIndexedVector ∗ *rowArray,* CoinIndexedVector ∗ *rhsArray,* CoinIndexedVector ∗ *spareArray,* int *valuesPass*)

Row array has pivot column This chooses pivot row.

Rhs array is used for distance to next bound (for speed) For speed, we may need to go to a bucket approach when many variables go through bounds If valuesPass non-zero then compute dj for direction

4.76.2.7 void ClpSimplexPrimal::statusOfProblemInPrimal (int & *lastCleaned,* int *type,* ClpSimplexProgress ∗ *progress,* bool *doFactorization,* int *ifValuesPass,* ClpSimplex ∗ *saveModel* = NULL)

Refactorizes if necessary Checks if finished.

Updates status. lastCleaned refers to iteration at which some objective/feasibility cleaning too place.

type - 0 initial so set up save arrays etc

- 1 normal -if good update save

2 restoring from saved saveModel is normally NULL but may not be if doing Sprint

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

- ClpSimplexPrimal.hpp

4.77 ClpSimplexProgress Class Reference

For saving extra information to see if looping.

`#include <ClpSolve.hpp>`

Collaboration diagram for ClpSimplexProgress:

Public Member Functions

Constructors and destructor and copy

- ClpSimplexProgress ()
  Default constructor.
- ClpSimplexProgress (ClpSimplex ∗model)
  Constructor from model.
- ClpSimplexProgress (const ClpSimplexProgress &)
  Copy constructor.
- ClpSimplexProgress & operator= (const ClpSimplexProgress &rhs)
  Assignment operator. This copies the data.
- ∼ClpSimplexProgress ()
  Destructor.
- void reset ()
  Resets as much as possible.
- void fillFromModel (ClpSimplex ∗model)
  Fill from model.

Check progress

- int looping ()

  Returns -1 if okay, -n+1 (n number of times bad) if bad but action taken, >=0 if give up and use as problem status.
- void startCheck ()

  Start check at beginning of whileIterating.
- int cycle (int in, int out, int wayIn, int wayOut)

  Returns cycle length in whileIterating.
- double lastObjective (int back=1) const

  Returns previous objective (if -1) - current if (0)
- void setInfeasibility (double value)

  Set real primal infeasibility and move back.
- double lastInfeasibility (int back=1) const

  Returns real primal infeasibility (if -1) - current if (0)
- int numberInfeasibilities (int back=1) const

  Returns number of primal infeasibilities (if -1) - current if (0)
- void modifyObjective (double value)

  Modify objective e.g. if dual infeasible in dual.
- int lastIterationNumber (int back=1) const

  Returns previous iteration number (if -1) - current if (0)
- void clearIterationNumbers ()

  clears all iteration numbers (to switch off panic)
- void newOddState ()

  Odd state.
- void **endOddState** ()
- void **clearOddState** ()
- int **oddState** () const
- int badTimes () const

  number of bad times
- void **clearBadTimes** ()
- int reallyBadTimes () const

  number of really bad times
- void **incrementReallyBadTimes** ()
- int timesFlagged () const

  number of times flagged
- void **clearTimesFlagged** ()
- void **incrementTimesFlagged** ()

Public Attributes

Data

- double objective_ [CLP_PROGRESS]

  Objective values.
- double infeasibility_ [CLP_PROGRESS]

  Sum of infeasibilities for algorithm.
- double realInfeasibility_ [CLP_PROGRESS]

  Sum of real primal infeasibilities for primal.
- double initialWeight_

  Initial weight for weights.
- int in_ [CLP_CYCLE]

  For cycle checking.
- int **out_** [CLP_CYCLE]
- char **way_** [CLP_CYCLE]

- ClpSimplex * model_

    Pointer back to model so we can get information.
- int numberInfeasibilities_ [CLP_PROGRESS]

    Number of infeasibilities.
- int iterationNumber_ [CLP_PROGRESS]

    Iteration number at which occurred.
- int numberTimes_

    Number of times checked (so won't stop too early)
- int numberBadTimes_

    Number of times it looked like loop.
- int numberReallyBadTimes_

    Number really bad times.
- int numberTimesFlagged_

    Number of times no iterations as flagged.
- int oddState_

    If things are in an odd state.

4.77.1 Detailed Description

For saving extra information to see if looping.

Definition at line 261 of file ClpSolve.hpp.

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

- ClpSolve.hpp

4.78 ClpSolve Class Reference

This is a very simple class to guide algorithms.

```
#include <ClpSolve.hpp>
```

Public Types

- enum SolveType

    enums for solve function

Public Member Functions

Constructors and destructor and copy

- ClpSolve ()

    Default constructor.
- ClpSolve (SolveType method, PresolveType presolveType, int numberPasses, int options[6], int extraInfo[6], int independentOptions[3])

    Constructor when you really know what you are doing.
- void generateCpp (FILE *fp)

    Generates code for above constructor.
- ClpSolve (const ClpSolve &)

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

  Assignment operator. This copies the data.
- ∼ClpSolve ()

  Destructor.

Functions most useful to user

- void setSpecialOption (int which, int value, int extraInfo=-1)

  Special options - bits 0 4 - use crash (default allslack in dual, idiot in primal) 8 - all slack basis in primal 2 16 - switch off interrupt handling 3 32 - do not try and make plus minus one matrix 64 - do not use sprint even if problem looks good.
- int **getSpecialOption** (int which) const
- void setSolveType (SolveType method, int extraInfo=-1)

  Solve types.
- SolveType **getSolveType** ()
- void **setPresolveType** (PresolveType amount, int extraInfo=-1)
- PresolveType **getPresolveType** ()
- int **getPresolvePasses** () const
- int getExtraInfo (int which) const

  Extra info for idiot (or sprint)
- void setInfeasibleReturn (bool trueFalse)

  Say to return at once if infeasible, default is to solve.
- bool **infeasibleReturn** () const
- bool doDual () const

  Whether we want to do dual part of presolve.
- void **setDoDual** (bool doDual_)
- bool doSingleton () const

  Whether we want to do singleton part of presolve.
- void **setDoSingleton** (bool doSingleton_)
- bool doDoubleton () const

  Whether we want to do doubleton part of presolve.
- void **setDoDoubleton** (bool doDoubleton_)
- bool doTripleton () const

  Whether we want to do tripleton part of presolve.
- void **setDoTripleton** (bool doTripleton_)
- bool doTighten () const

  Whether we want to do tighten part of presolve.
- void **setDoTighten** (bool doTighten_)
- bool doForcing () const

  Whether we want to do forcing part of presolve.
- void **setDoForcing** (bool doForcing_)
- bool doImpliedFree () const

  Whether we want to do impliedfree part of presolve.
- void **setDoImpliedFree** (bool doImpliedfree)
- bool doDupcol () const

  Whether we want to do dupcol part of presolve.
- void **setDoDupcol** (bool doDupcol_)
- bool doDuprow () const

  Whether we want to do duprow part of presolve.
- void **setDoDuprow** (bool doDuprow_)
- bool doSingletonColumn () const

  Whether we want to do singleton column part of presolve.
- void **setDoSingletonColumn** (bool doSingleton_)
- bool doKillSmall () const

Whether we want to kill small substitutions.
- void **setDoKillSmall** (bool doKill)
- int presolveActions () const

  Set whole group.
- void **setPresolveActions** (int action)
- int substitution () const

  Largest column for substitution (normally 3)
- void **setSubstitution** (int value)
- void **setIndependentOption** (int type, int value)
- int **independentOption** (int type) const

4.78.1 Detailed Description

This is a very simple class to guide algorithms.

It is used to tidy up passing parameters to initialSolve and maybe for output from that

Definition at line 20 of file ClpSolve.hpp.

4.78.2 Member Function Documentation

4.78.2.1 void ClpSolve::setSpecialOption (int *which*, int *value*, int *extraInfo* = `-1`)

Special options - bits 0 4 - use crash (default allslack in dual, idiot in primal) 8 - all slack basis in primal 2 16 - switch off interrupt handling 3 32 - do not try and make plus minus one matrix 64 - do not use sprint even if problem looks good.

which translation is: which: 0 - startup in Dual (nothing if basis exists).: 0 - no basis 1 - crash 2 - use initiative about idiot! but no crash 1 - startup in Primal (nothing if basis exists): 0 - use initiative 1 - use crash 2 - use idiot and look at further info 3 - use sprint and look at further info 4 - use all slack 5 - use initiative but no idiot 6 - use initiative but no sprint 7 - use initiative but no crash 8 - do allslack or idiot 9 - do allslack or sprint 10 - slp before 11 - no nothing and primal(0) 2 - interrupt handling - 0 yes, 1 no (for threadsafe) 3 - whether to make +- 1matrix - 0 yes, 1 no 4 - for barrier 0 - dense cholesky 1 - Wssmp allowing some long columns 2 - Wssmp not allowing long columns 3 - Wssmp using KKT 4 - Using Florida ordering 8 - bit set to do scaling 16 - set to be aggressive with gamma/delta? 32 - Use KKT 5 - for presolve 1 - switch off dual stuff 6 - extra switches

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

- ClpSolve.hpp

4.79 ClpTrustedData Struct Reference

For a structure to be used by trusted code.

```
#include <ClpParameters.hpp>
```

4.79.1 Detailed Description

For a structure to be used by trusted code.

Definition at line 121 of file ClpParameters.hpp.

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

- ClpParameters.hpp

4.80 CoinAbcAnyFactorization Class Reference

Abstract base class which also has some scalars so can be used from Dense or Simp.

`#include <CoinAbcDenseFactorization.hpp>`

Inheritance diagram for CoinAbcAnyFactorization:

Public Member Functions

Constructors and destructor and copy

- CoinAbcAnyFactorization ()

  Default constructor.
- CoinAbcAnyFactorization (const CoinAbcAnyFactorization &other)

  Copy constructor.
- virtual ∼CoinAbcAnyFactorization ()

  Destructor.
- CoinAbcAnyFactorization & operator= (const CoinAbcAnyFactorization &other)

  = copy
- virtual CoinAbcAnyFactorization ∗ clone () const =0

  Clone.

general stuff such as status

- int status () const

  Returns status.
- void setStatus (int value)

  Sets status.
- int pivots () const

  Returns number of pivots since factorization.
- void setPivots (int value)

  Sets number of pivots since factorization.
- int numberSlacks () const

  Returns number of slacks.
- void setNumberSlacks (int value)

  Sets number of slacks.
- void setNumberRows (int value)

  Set number of Rows after factorization.
- int numberRows () const

  Number of Rows after factorization.
- CoinSimplexInt numberDense () const

  Number of dense rows after factorization.
- int numberGoodColumns () const

  Number of good columns in factorization.
- void relaxAccuracyCheck (double value)

  Allows change of pivot accuracy check 1.0 == none >1.0 relaxed.
- double **getAccuracyCheck** () const
- int maximumPivots () const

  Maximum number of pivots between factorizations.
- virtual void maximumPivots (int value)

  Set maximum pivots.

- double pivotTolerance () const

  Pivot tolerance.
- void **pivotTolerance** (double value)
- double minimumPivotTolerance () const

  Minimum pivot tolerance.
- void **minimumPivotTolerance** (double value)
- virtual CoinFactorizationDouble ∗ **pivotRegion** () const
- double areaFactor () const

  Area factor.
- void **areaFactor** (CoinSimplexDouble value)
- double zeroTolerance () const

  Zero tolerance.
- void **zeroTolerance** (double value)
- virtual CoinFactorizationDouble ∗ elements () const

  Returns array to put basis elements in.
- virtual int ∗ pivotRow () const

  Returns pivot row.
- virtual CoinFactorizationDouble ∗ workArea () const

  Returns work area.
- virtual int ∗ intWorkArea () const

  Returns int work area.
- virtual int ∗ numberInRow () const

  Number of entries in each row.
- virtual int ∗ numberInColumn () const

  Number of entries in each column.
- virtual CoinBigIndex ∗ starts () const

  Returns array to put basis starts in.
- virtual int ∗ permuteBack () const

  Returns permute back.
- virtual void goSparse ()

  Sees whether to go sparse.
- virtual void **checkMarkArrays** () const
- int solveMode () const

  Get solve mode e.g.
- void setSolveMode (int value)

  Set solve mode e.g.
- virtual bool wantsTableauColumn () const

  Returns true if wants tableauColumn in replaceColumn.
- virtual void setUsefulInformation (const int ∗info, int whereFrom)

  Useful information for factorization 0 - iteration number whereFrom is 0 for factorize and 1 for replaceColumn.
- virtual void clearArrays ()

  Get rid of all memory.

virtual general stuff such as permutation

- virtual int ∗ indices () const =0

  Returns array to put basis indices in.
- virtual int ∗ permute () const =0

  Returns permute in.
- virtual int ∗ pivotColumn () const

  Returns pivotColumn or permute.
- virtual int numberElements () const =0

  Total number of elements in factorization.

Do factorization - public

- virtual void getAreas (int numberRows, int numberColumns, CoinBigIndex maximumL, CoinBigIndex maximumU)=0
  Gets space for a factorization.
- virtual void preProcess ()=0
  PreProcesses column ordered copy of basis.
- virtual int factor (AbcSimplex ∗model)=0
  Does most of factorization returning status 0 - OK -99 - needs more memory -1 - singular - use numberGoodColumns and redo.
- virtual void postProcess (const int ∗sequence, int ∗pivotVariable)=0
  Does post processing on valid factorization - putting variables on correct rows.
- virtual void makeNonSingular (int ∗sequence)=0
  Makes a non-singular basis by replacing variables.

rank one updates which do exist

- virtual double checkReplacePart1 (**CoinIndexedVector** ∗, int)
  Checks if can replace one Column to basis, returns update alpha Fills in region for use later partial update already in U.
- virtual double **checkReplacePart1** (**CoinIndexedVector** ∗, **CoinIndexedVector** ∗, int)
- virtual void **checkReplacePart1a** (**CoinIndexedVector** ∗, int)
- virtual double **checkReplacePart1b** (**CoinIndexedVector** ∗, int)
- virtual int checkReplacePart2 (int pivotRow, double btranAlpha, double ftranAlpha, double ftAlpha, double acceptablePivot=1.0e-8)=0
  Checks if can replace one Column to basis, returns 0=OK, 1=Probably OK, 2=singular, 3=no room, 5 max pivots.
- virtual void replaceColumnPart3 (const AbcSimplex ∗model, **CoinIndexedVector** ∗regionSparse, **CoinIndexedVector** ∗tableauColumn, int pivotRow, double alpha)=0
  Replaces one Column to basis, partial update already in U.
- virtual void replaceColumnPart3 (const AbcSimplex ∗model, **CoinIndexedVector** ∗regionSparse, **CoinIndexedVector** ∗tableauColumn, **CoinIndexedVector** ∗partialUpdate, int pivotRow, double alpha)=0
  Replaces one Column to basis, partial update in vector.

various uses of factorization (return code number elements)

which user may want to know about

- virtual int updateColumnFT (**CoinIndexedVector** &regionSparse)=0
  Updates one column (FTRAN) from unpacked regionSparse Tries to do FT update number returned is negative if no room.
- virtual int **updateColumnFTPart1** (**CoinIndexedVector** &regionSparse)=0
- virtual void **updateColumnFTPart2** (**CoinIndexedVector** &regionSparse)=0
- virtual void **updateColumnFT** (**CoinIndexedVector** &regionSparseFT, **CoinIndexedVector** &partialUpdate, int which)=0
- virtual int updateColumn (**CoinIndexedVector** &regionSparse) const =0
  This version has same effect as above with FTUpdate==false so number returned is always >=0.
- virtual int updateTwoColumnsFT (**CoinIndexedVector** &regionFT, **CoinIndexedVector** &regionOther)=0
  does FTRAN on two unpacked columns
- virtual int updateColumnTranspose (**CoinIndexedVector** &regionSparse) const =0
  Updates one column (BTRAN) from unpacked regionSparse.
- virtual void updateFullColumn (**CoinIndexedVector** &regionSparse) const =0
  This version does FTRAN on array when indices not set up.
- virtual void updateFullColumnTranspose (**CoinIndexedVector** &regionSparse) const =0
  Updates one column (BTRAN) from unpacked regionSparse.
- virtual void updateWeights (**CoinIndexedVector** &regionSparse) const =0
  Updates one column for dual steepest edge weights (FTRAN)
- virtual void updateColumnCpu (**CoinIndexedVector** &regionSparse, int whichCpu) const
  Updates one column (FTRAN)
- virtual void updateColumnTransposeCpu (**CoinIndexedVector** &regionSparse, int whichCpu) const
  Updates one column (BTRAN)

Protected Attributes

data

- double pivotTolerance_
  Pivot tolerance.
- double minimumPivotTolerance_
  Minimum pivot tolerance.
- double areaFactor_
  Area factor.
- double zeroTolerance_
  Zero tolerance.
- double relaxCheck_
  Relax check on accuracy in replaceColumn.
- CoinBigIndex factorElements_
  Number of elements after factorization.
- int numberRows_
  Number of Rows in factorization.
- int numberDense_
  Number of dense rows in factorization.
- int numberGoodU_
  Number factorized in U (not row singletons)
- int maximumPivots_
  Maximum number of pivots before factorization.
- int numberPivots_
  Number pivots since last factorization.
- int numberSlacks_
  Number slacks.
- int status_
  Status of factorization.
- int maximumRows_
  Maximum rows ever (i.e. use to copy arrays etc)
- int ∗ pivotRow_
  Pivot row.
- CoinFactorizationDouble ∗ elements_
  Elements of factorization and updates length is maxR∗maxR+maxSpace will always be long enough so can have nR∗nR ints in maxSpace.
- CoinFactorizationDouble ∗ workArea_
  Work area of numberRows_.
- int solveMode_
  Solve mode e.g.

4.80.1 Detailed Description

Abstract base class which also has some scalars so can be used from Dense or Simp.

Definition at line 24 of file CoinAbcDenseFactorization.hpp.

4.80.2 Member Function Documentation

4.80.2.1 int CoinAbcAnyFactorization::solveMode ( ) const [inline]

Get solve mode e.g.

0 C++ code, 1 Lapack, 2 choose If 4 set then values pass if 8 set then has iterated

Definition at line 148 of file CoinAbcDenseFactorization.hpp.

4.80.2.2 void CoinAbcAnyFactorization::setSolveMode (int *value*) [inline]

Set solve mode e.g.

0 C++ code, 1 Lapack, 2 choose If 4 set then values pass if 8 set then has iterated

Definition at line 154 of file CoinAbcDenseFactorization.hpp.

4.80.3 Member Data Documentation

4.80.3.1 int CoinAbcAnyFactorization::solveMode_ [protected]

Solve mode e.g.

0 C++ code, 1 Lapack, 2 choose If 4 set then values pass if 8 set then has iterated

Definition at line 382 of file CoinAbcDenseFactorization.hpp.

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

- CoinAbcDenseFactorization.hpp

4.81 CoinAbcDenseFactorization Class Reference

This deals with Factorization and Updates This is a simple dense version so other people can write a better one.

`#include <CoinAbcDenseFactorization.hpp>`

Inheritance diagram for CoinAbcDenseFactorization:

Collaboration diagram for CoinAbcDenseFactorization:

Public Member Functions

- void gutsOfDestructor ()

  The real work of desstructor.
- void gutsOfInitialize ()

  The real work of constructor.
- void gutsOfCopy (const CoinAbcDenseFactorization &other)

  The real work of copy.

Constructors and destructor and copy

- CoinAbcDenseFactorization ()

  Default constructor.
- CoinAbcDenseFactorization (const CoinAbcDenseFactorization &other)

  Copy constructor.
- virtual ∼CoinAbcDenseFactorization ()

  Destructor.
- CoinAbcDenseFactorization & operator= (const CoinAbcDenseFactorization &other)

  = copy
- virtual CoinAbcAnyFactorization ∗ clone () const

  Clone.

Do factorization - public

- virtual void getAreas (int numberRows, int numberColumns, CoinBigIndex maximumL, CoinBigIndex maximumU)

  Gets space for a factorization.
- virtual void preProcess ()

  PreProcesses column ordered copy of basis.
- virtual int factor (AbcSimplex ∗model)

  Does most of factorization returning status 0 - OK -99 - needs more memory -1 - singular - use numberGoodColumns and redo.
- virtual void postProcess (const int ∗sequence, int ∗pivotVariable)

  Does post processing on valid factorization - putting variables on correct rows.
- virtual void makeNonSingular (int ∗sequence)

  Makes a non-singular basis by replacing variables.

general stuff such as number of elements

- virtual int numberElements () const

  Total number of elements in factorization.
- double maximumCoefficient () const

  Returns maximum absolute value in factorization.

rank one updates which do exist

- virtual int replaceColumn (**CoinIndexedVector** ∗regionSparse, int pivotRow, double pivotCheck, bool skipBtranU=false, double acceptablePivot=1.0e-8)

  Replaces one Column to basis, returns 0=OK, 1=Probably OK, 2=singular, 3=no room If skipBtranU is false will do btran part partial update already in U.
- virtual int checkReplacePart2 (int pivotRow, double btranAlpha, double ftranAlpha, double ftAlpha, double acceptablePivot=1.0e-8)

  Checks if can replace one Column to basis, returns 0=OK, 1=Probably OK, 2=singular, 3=no room, 5 max pivots.
- virtual void replaceColumnPart3 (const AbcSimplex ∗model, **CoinIndexedVector** ∗regionSparse, **CoinIndexedVector** ∗tableauColumn, int pivotRow, double alpha)

  Replaces one Column to basis, partial update already in U.
- virtual void replaceColumnPart3 (const AbcSimplex ∗model, **CoinIndexedVector** ∗regionSparse, **CoinIndexedVector** ∗tableauColumn, **CoinIndexedVector** ∗, int pivotRow, double alpha)

  Replaces one Column to basis, partial update in vector.

various uses of factorization (return code number elements)

which user may want to know about

- virtual int updateColumnFT (**CoinIndexedVector** &regionSparse)

  Updates one column (FTRAN) from unpacked regionSparse Tries to do FT update number returned is negative if no room.
- virtual int **updateColumnFTPart1** (**CoinIndexedVector** &regionSparse)
- virtual void **updateColumnFTPart2** (**CoinIndexedVector** &)
- virtual void **updateColumnFT** (**CoinIndexedVector** &regionSparseFT, **CoinIndexedVector** &, int)
- virtual int updateColumn (**CoinIndexedVector** &regionSparse) const

  This version has same effect as above with FTUpdate==false so number returned is always >=0.
- virtual int updateTwoColumnsFT (**CoinIndexedVector** &regionFT, **CoinIndexedVector** &regionOther)

  does FTRAN on two unpacked columns
- virtual int updateColumnTranspose (**CoinIndexedVector** &regionSparse) const

  Updates one column (BTRAN) from unpacked regionSparse.

- virtual void updateFullColumn (**CoinIndexedVector** &regionSparse) const
  This version does FTRAN on array when indices not set up.
- virtual void updateFullColumnTranspose (**CoinIndexedVector** &regionSparse) const
  Updates one column (BTRAN) from unpacked regionSparse.
- virtual void updateWeights (**CoinIndexedVector** &regionSparse) const
  Updates one column for dual steepest edge weights (FTRAN)

various uses of factorization

∗∗∗ Below this user may not want to know about

which user may not want to know about (left over from my LP code)

- void clearArrays ()
  Get rid of all memory.
- virtual int ∗ indices () const
  Returns array to put basis indices in.
- virtual int ∗ permute () const
  Returns permute in.

Protected Member Functions

- int checkPivot (double saveFromU, double oldPivot) const
  Returns accuracy status of replaceColumn returns 0=OK, 1=Probably OK, 2=singular.

Protected Attributes

- CoinBigIndex maximumSpace_
  Maximum length of iterating area.
- CoinSimplexInt maximumRowsAdjusted_
  Use for array size to get multiple of 8.

4.81.1 Detailed Description

This deals with Factorization and Updates This is a simple dense version so other people can write a better one.

I am assuming that 32 bits is enough for number of rows or columns, but CoinBigIndex may be redefined to get 64 bits.

Definition at line 394 of file CoinAbcDenseFactorization.hpp.

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

- CoinAbcDenseFactorization.hpp

4.82 CoinAbcStack Struct Reference

4.82.1 Detailed Description

Definition at line 71 of file CoinAbcCommonFactorization.hpp.

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

- CoinAbcCommonFactorization.hpp

4.83 CoinAbcStatistics Struct Reference

4.83.1 Detailed Description

Definition at line 32 of file CoinAbcCommonFactorization.hpp.

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

- CoinAbcCommonFactorization.hpp

4.84 CoinAbcTypeFactorization Class Reference

Inheritance diagram for CoinAbcTypeFactorization:

Collaboration diagram for CoinAbcTypeFactorization:

Public Member Functions

Constructors and destructor and copy

- CoinAbcTypeFactorization ()
  Default constructor.
- CoinAbcTypeFactorization (const CoinAbcTypeFactorization &other)
  Copy constructor.
- CoinAbcTypeFactorization (const **CoinFactorization** &other)
  Copy constructor.
- virtual ∼CoinAbcTypeFactorization ()
  Destructor.
- virtual CoinAbcAnyFactorization ∗ clone () const
  Clone.
- void almostDestructor ()
  Delete all stuff (leaves as after CoinAbcFactorization())
- void show_self () const
  Debug show object (shows one representation)
- void sort () const
  Debug - sort so can compare.
- CoinAbcTypeFactorization & operator= (const CoinAbcTypeFactorization &other)
  = copy

Do factorization

- CoinSimplexDouble conditionNumber () const
  Condition number - product of pivots after factorization.

general stuff such as permutation or status

- CoinSimplexInt ∗ permute () const
  Returns address of permute region.
- virtual CoinSimplexInt ∗ indices () const
  Returns array to put basis indices in.
- virtual CoinSimplexInt ∗ pivotColumn () const

Returns address of pivotColumn region (also used for permuting)

- virtual CoinFactorizationDouble ∗ pivotRegion () const

  Returns address of pivot region.
- CoinBigIndex ∗ startRowL () const

  Start of each row in L.
- CoinBigIndex ∗ startColumnL () const

  Start of each column in L.
- CoinSimplexInt ∗ indexColumnL () const

  Index of column in row for L.
- CoinSimplexInt ∗ indexRowL () const

  Row indices of L.
- CoinFactorizationDouble ∗ elementByRowL () const

  Elements in L (row copy)
- CoinSimplexInt ∗ pivotLinkedBackwards () const

  Forward and backward linked lists (numberRows_+2)
- CoinSimplexInt ∗ **pivotLinkedForwards** () const
- CoinSimplexInt ∗ **pivotLOrder** () const
- CoinSimplexInt ∗ firstCount () const

  For equal counts in factorization.
- CoinSimplexInt ∗ nextCount () const

  Next Row/Column with count.
- CoinSimplexInt ∗ lastCount () const

  Previous Row/Column with count.
- CoinSimplexInt numberRowsExtra () const

  Number of Rows after iterating.
- CoinBigIndex numberL () const

  Number in L.
- CoinBigIndex baseL () const

  Base of L.
- CoinSimplexInt maximumRowsExtra () const

  Maximum of Rows after iterating.
- virtual CoinBigIndex numberElements () const

  Total number of elements in factorization.
- CoinSimplexInt numberForrestTomlin () const

  Length of FT vector.
- CoinSimplexDouble adjustedAreaFactor () const

  Returns areaFactor but adjusted for dense.
- CoinSimplexInt messageLevel () const

  Level of detail of messages.
- void **messageLevel** (CoinSimplexInt value)
- virtual void maximumPivots (CoinSimplexInt value)

  Set maximum pivots.
- CoinSimplexInt denseThreshold () const

  Gets dense threshold.
- void setDenseThreshold (CoinSimplexInt value)

  Sets dense threshold.
- CoinSimplexDouble maximumCoefficient () const

  Returns maximum absolute value in factorization.
- bool spaceForForrestTomlin () const

  True if FT update and space.

some simple stuff

- CoinBigIndex numberElementsU () const
  Returns number in U area.
- void setNumberElementsU (CoinBigIndex value)
  Setss number in U area.
- CoinBigIndex lengthAreaU () const
  Returns length of U area.
- CoinBigIndex numberElementsL () const
  Returns number in L area.
- CoinBigIndex lengthAreaL () const
  Returns length of L area.
- CoinBigIndex numberElementsR () const
  Returns number in R area.
- CoinBigIndex numberCompressions () const
  Number of compressions done.
- virtual CoinBigIndex ∗ starts () const
  Returns pivot row.
- virtual CoinSimplexInt ∗ numberInRow () const
  Number of entries in each row.
- virtual CoinSimplexInt ∗ numberInColumn () const
  Number of entries in each column.
- virtual CoinFactorizationDouble ∗ elements () const
  Returns array to put basis elements in.
- CoinBigIndex ∗ startColumnR () const
  Start of columns for R.
- CoinFactorizationDouble ∗ elementU () const
  Elements of U.
- CoinSimplexInt ∗ indexRowU () const
  Row indices of U.
- CoinBigIndex ∗ startColumnU () const
  Start of each column in U.
- double ∗ denseVector (**CoinIndexedVector** ∗vector) const
  Returns double ∗ associated with vector.
- double ∗ **denseVector** (**CoinIndexedVector** &vector) const
- const double ∗ denseVector (const **CoinIndexedVector** ∗vector) const
  Returns double ∗ associated with vector.
- const double ∗ **denseVector** (const **CoinIndexedVector** &vector) const
- void toLongArray (**CoinIndexedVector** ∗vector, int which) const
  To a work array and associate vector.
- void fromLongArray (**CoinIndexedVector** ∗vector) const
  From a work array and dis-associate vector.
- void fromLongArray (int which) const
  From a work array and dis-associate vector.
- void scan (**CoinIndexedVector** ∗vector) const
  Scans region to find nonzeros.

rank one updates which do exist

Array persistence flag If 0 then as now (delete/new) 1 then only do arrays if bigger needed 2 as 1 but give a bit extra if bigger needed

- virtual double checkReplacePart1 (**CoinIndexedVector** ∗regionSparse, int pivotRow)
  Checks if can replace one Column to basis, returns update alpha Fills in region for use later partial update already in U.

- virtual double checkReplacePart1 (**CoinIndexedVector** ∗regionSparse, **CoinIndexedVector** ∗partialUpdate, int pivotRow)

  Checks if can replace one Column to basis, returns update alpha Fills in region for use later partial update in vector.
- virtual int checkReplacePart2 (int pivotRow, CoinSimplexDouble btranAlpha, double ftranAlpha, double ftAlpha, double acceptablePivot=1.0e-8)

  Checks if can replace one Column to basis, returns 0=OK, 1=Probably OK, 2=singular, 3=no room, 5 max pivots.
- virtual void replaceColumnPart3 (const AbcSimplex ∗model, **CoinIndexedVector** ∗regionSparse, **CoinIndexedVector** ∗tableauColumn, int pivotRow, double alpha)

  Replaces one Column to basis, partial update already in U.
- virtual void replaceColumnPart3 (const AbcSimplex ∗model, **CoinIndexedVector** ∗regionSparse, **CoinIndexedVector** ∗tableauColumn, **CoinIndexedVector** ∗partialUpdate, int pivotRow, double alpha)

  Replaces one Column to basis, partial update in vector.
- void updatePartialUpdate (**CoinIndexedVector** &partialUpdate)

  Update partial Ftran by R update.
- virtual bool wantsTableauColumn () const

  Returns true if wants tableauColumn in replaceColumn.
- int replaceColumnU (**CoinIndexedVector** ∗regionSparse, CoinBigIndex ∗deletedPosition, CoinSimplexInt ∗deletedColumns, CoinSimplexInt pivotRow)

  Combines BtranU and store which elements are to be deleted returns number to be deleted.

various uses of factorization (return code number elements)

∗∗∗ *Below this user may not want to know about*

which user may not want to know about (left over from my LP code)

- virtual CoinSimplexInt updateColumnFT (**CoinIndexedVector** &regionSparse)

  Later take out return codes (apart from +- 1 on FT)
- virtual int **updateColumnFTPart1** (**CoinIndexedVector** &regionSparse)
- virtual void **updateColumnFTPart2** (**CoinIndexedVector** &regionSparse)
- virtual void updateColumnFT (**CoinIndexedVector** &regionSparseFT, **CoinIndexedVector** &partialUpdate, int which)

  Updates one column (FTRAN) Tries to do FT update puts partial update in vector.
- virtual CoinSimplexInt updateColumn (**CoinIndexedVector** &regionSparse) const

  This version has same effect as above with FTUpdate==false so number returned is always >=0.
- virtual CoinSimplexInt updateTwoColumnsFT (**CoinIndexedVector** &regionFT, **CoinIndexedVector** &regionOther)

  Updates one column (FTRAN) from region2 Tries to do FT update number returned is negative if no room.
- virtual CoinSimplexInt updateColumnTranspose (**CoinIndexedVector** &regionSparse) const

  Updates one column (BTRAN) from regionSparse2 regionSparse starts as zero and is zero at end Note - if regionSparse2 packed on input - will be packed on output.
- virtual void updateFullColumn (**CoinIndexedVector** &regionSparse) const

  Updates one full column (FTRAN)
- virtual void updateFullColumnTranspose (**CoinIndexedVector** &regionSparse) const

  Updates one full column (BTRAN)
- virtual void updateWeights (**CoinIndexedVector** &regionSparse) const

  Updates one column for dual steepest edge weights (FTRAN)
- virtual void updateColumnCpu (**CoinIndexedVector** &regionSparse, int whichCpu) const

  Updates one column (FTRAN)
- virtual void updateColumnTransposeCpu (**CoinIndexedVector** &regionSparse, int whichCpu) const

  Updates one column (BTRAN)
- void **unpack** (**CoinIndexedVector** ∗regionFrom, **CoinIndexedVector** ∗regionTo) const
- void **pack** (**CoinIndexedVector** ∗regionFrom, **CoinIndexedVector** ∗regionTo) const
- void goSparse ()

makes a row copy of L for speed and to allow very sparse problems
- void **goSparse2** ()
- virtual void **checkMarkArrays** () const
- CoinSimplexInt sparseThreshold () const

  get sparse threshold
- void sparseThreshold (CoinSimplexInt value)

  set sparse threshold
- void clearArrays ()

  Get rid of all memory.

used by ClpFactorization

- void checkSparse ()

  See if worth going sparse.
- void gutsOfDestructor (CoinSimplexInt type=1)

  The real work of constructors etc 0 just scalars, 1 bit normal.
- void gutsOfInitialize (CoinSimplexInt type)

  1 bit - tolerances etc, 2 more, 4 dummy arrays
- void **gutsOfCopy** (const CoinAbcTypeFactorization &other)
- void resetStatistics ()

  Reset all sparsity etc statistics.
- void **printRegion** (const **CoinIndexedVector** &vector, const char ∗where) const

used by factorization

- virtual void getAreas (CoinSimplexInt numberRows, CoinSimplexInt numberColumns, CoinBigIndex maximumL, CoinBigIndex maximumU)

  Gets space for a factorization, called by constructors.
- virtual void preProcess ()

  PreProcesses column ordered copy of basis.
- void **preProcess** (CoinSimplexInt)
- double preProcess3 ()

  Return largest element.
- void **preProcess4** ()
- virtual CoinSimplexInt factor (AbcSimplex ∗model)

  Does most of factorization.
- virtual void postProcess (const CoinSimplexInt ∗sequence, CoinSimplexInt ∗pivotVariable)

  Does post processing on valid factorization - putting variables on correct rows.
- virtual void makeNonSingular (CoinSimplexInt ∗sequence)

  Makes a non-singular basis by replacing variables.
- CoinSimplexInt replaceColumnPFI (**CoinIndexedVector** ∗regionSparse, CoinSimplexInt pivotRow, Coin↩ SimplexDouble alpha)

  Replaces one Column to basis for PFI returns 0=OK, 1=Probably OK, 2=singular, 3=no room.
- CoinSimplexInt factorSparse ()

  Does sparse phase of factorization return code is <0 error, 0= finished.
- CoinSimplexInt factorDense ()

  Does dense phase of factorization return code is <0 error, 0= finished.
- bool pivotOneOtherRow (CoinSimplexInt pivotRow, CoinSimplexInt pivotColumn)

  Pivots when just one other row so faster?
- bool pivotRowSingleton (CoinSimplexInt pivotRow, CoinSimplexInt pivotColumn)

Does one pivot on Row Singleton in factorization.

- void pivotColumnSingleton (CoinSimplexInt pivotRow, CoinSimplexInt pivotColumn)

    Does one pivot on Column Singleton in factorization (can't return false)

- void afterPivot (CoinSimplexInt pivotRow, CoinSimplexInt pivotColumn)

    After pivoting.

- int wantToGoDense ()

    After pivoting - returns true if need to go dense.

- bool getColumnSpace (CoinSimplexInt iColumn, CoinSimplexInt extraNeeded)

    Gets space for one Column with given length, may have to do compression (returns True if successful), also moves existing vector, extraNeeded is over and above present.

- bool reorderU ()

    Reorders U so contiguous and in order (if there is space) Returns true if it could.

- bool getColumnSpaceIterateR (CoinSimplexInt iColumn, CoinFactorizationDouble value, CoinSimplexInt iRow)

    getColumnSpaceIterateR.

- CoinBigIndex getColumnSpaceIterate (CoinSimplexInt iColumn, CoinFactorizationDouble value, CoinSimplexInt iRow)

    getColumnSpaceIterate.

- bool getRowSpace (CoinSimplexInt iRow, CoinSimplexInt extraNeeded)

    Gets space for one Row with given length, may have to do compression (returns True if successful), also moves existing vector.

- bool getRowSpaceIterate (CoinSimplexInt iRow, CoinSimplexInt extraNeeded)

    Gets space for one Row with given length while iterating, may have to do compression (returns True if successful), also moves existing vector.

- void checkConsistency ()

    Checks that row and column copies look OK.

- void addLink (CoinSimplexInt index, CoinSimplexInt count)

    Adds a link in chain of equal counts.

- void deleteLink (CoinSimplexInt index)

    Deletes a link in chain of equal counts.

- void modifyLink (CoinSimplexInt index, CoinSimplexInt count)

    Modifies links in chain of equal counts.

- void separateLinks ()

    Separate out links with same row/column count.

- void **separateLinks** (CoinSimplexInt, CoinSimplexInt)
- void cleanup ()

    Cleans up at end of factorization.

- void doAddresses ()

    Set up addresses from arrays.

- void updateColumnL (**CoinIndexedVector** ∗region, CoinAbcStatistics &statistics) const

    Updates part of column (FTRANL)

- void updateColumnLDensish (**CoinIndexedVector** ∗region) const

    Updates part of column (FTRANL) when densish.

- void updateColumnLDense (**CoinIndexedVector** ∗region) const

    Updates part of column (FTRANL) when dense (i.e. do as inner products)

- void updateColumnLSparse (**CoinIndexedVector** ∗region) const

    Updates part of column (FTRANL) when sparse.

- void updateColumnR (**CoinIndexedVector** ∗region, CoinAbcStatistics &statistics) const

    Updates part of column (FTRANR) without FT update.

- bool storeFT (const **CoinIndexedVector** ∗regionFT)

  Store update after doing L and R - retuns false if no room.
- void updateColumnU (**CoinIndexedVector** ∗region, CoinAbcStatistics &statistics) const

  Updates part of column (FTRANU)
- void updateColumnUSparse (**CoinIndexedVector** ∗regionSparse) const

  Updates part of column (FTRANU) when sparse.
- void updateColumnUDensish (**CoinIndexedVector** ∗regionSparse) const

  Updates part of column (FTRANU)
- void updateColumnUDense (**CoinIndexedVector** ∗regionSparse) const

  Updates part of column (FTRANU) when dense (i.e. do as inner products)
- void updateTwoColumnsUDensish (CoinSimplexInt &numberNonZero1, CoinFactorizationDouble ∗COIN_RESTRICT region1, CoinSimplexInt ∗COIN_RESTRICT index1, CoinSimplexInt &numberNonZero2, CoinFactorizationDouble ∗COIN_RESTRICT region2, CoinSimplexInt ∗COIN_RESTRICT index2) const

  Updates part of 2 columns (FTRANU) real work.
- void updateColumnPFI (**CoinIndexedVector** ∗regionSparse) const

  Updates part of column PFI (FTRAN) (after rest)
- void updateColumnTransposePFI (**CoinIndexedVector** ∗region) const

  Updates part of column transpose PFI (BTRAN) (before rest)
- void updateColumnTransposeU (**CoinIndexedVector** ∗region, CoinSimplexInt smallestIndex, CoinAbcStatistics &statistics) const

  Updates part of column transpose (BTRANU), assumes index is sorted i.e.
- void updateColumnTransposeUDensish (**CoinIndexedVector** ∗region, CoinSimplexInt smallestIndex) const

  Updates part of column transpose (BTRANU) when densish, assumes index is sorted i.e.
- void updateColumnTransposeUSparse (**CoinIndexedVector** ∗region) const

  Updates part of column transpose (BTRANU) when sparse, assumes index is sorted i.e.
- void updateColumnTransposeUByColumn (**CoinIndexedVector** ∗region, CoinSimplexInt smallestIndex) const

  Updates part of column transpose (BTRANU) by column assumes index is sorted i.e.
- void updateColumnTransposeR (**CoinIndexedVector** ∗region, CoinAbcStatistics &statistics) const

  Updates part of column transpose (BTRANR)
- void updateColumnTransposeRDensish (**CoinIndexedVector** ∗region) const

  Updates part of column transpose (BTRANR) when dense.
- void updateColumnTransposeRSparse (**CoinIndexedVector** ∗region) const

  Updates part of column transpose (BTRANR) when sparse.
- void updateColumnTransposeL (**CoinIndexedVector** ∗region, CoinAbcStatistics &statistics) const

  Updates part of column transpose (BTRANL)
- void updateColumnTransposeLDensish (**CoinIndexedVector** ∗region) const

  Updates part of column transpose (BTRANL) when densish by column.
- void updateColumnTransposeLByRow (**CoinIndexedVector** ∗region) const

  Updates part of column transpose (BTRANL) when densish by row.
- void updateColumnTransposeLSparse (**CoinIndexedVector** ∗region) const

  Updates part of column transpose (BTRANL) when sparse (by Row)
- CoinSimplexInt checkPivot (CoinSimplexDouble saveFromU, CoinSimplexDouble oldPivot) const

  Returns accuracy status of replaceColumn returns 0=OK, 1=Probably OK, 2=singular.
- int pivot (CoinSimplexInt pivotRow, CoinSimplexInt pivotColumn, CoinBigIndex pivotRowPosition, CoinBigIndex pivotColumnPosition, CoinFactorizationDouble ∗COIN_RESTRICT work, CoinSimplexUnsignedInt ∗COIN_RESTRICT workArea2, CoinSimplexInt increment2, int ∗COIN_RESTRICT markRow)

  0 fine, -99 singular, 2 dense
- int **pivot** (CoinSimplexInt &pivotRow, CoinSimplexInt &pivotColumn, CoinBigIndex pivotRowPosition, CoinBigIndex pivotColumnPosition, int ∗COIN_RESTRICT markRow)

data

- CoinSimplexInt * **pivotColumnAddress_**
- CoinSimplexInt * **permuteAddress_**
- CoinFactorizationDouble * **pivotRegionAddress_**
- CoinFactorizationDouble * **elementUAddress_**
- CoinSimplexInt * **indexRowUAddress_**
- CoinSimplexInt * **numberInColumnAddress_**
- CoinSimplexInt * **numberInColumnPlusAddress_**
- CoinBigIndex * **startColumnUAddress_**
- CoinBigIndex * **convertRowToColumnUAddress_**
- CoinBigIndex * **convertColumnToRowUAddress_**
- CoinFactorizationDouble * **elementRowUAddress_**
- CoinBigIndex * **startRowUAddress_**
- CoinSimplexInt * **numberInRowAddress_**
- CoinSimplexInt * **indexColumnUAddress_**
- CoinSimplexInt * **firstCountAddress_**
- CoinSimplexInt * nextCountAddress_

    Next Row/Column with count.
- CoinSimplexInt * lastCountAddress_

    Previous Row/Column with count.
- CoinSimplexInt * **nextColumnAddress_**
- CoinSimplexInt * **lastColumnAddress_**
- CoinSimplexInt * **nextRowAddress_**
- CoinSimplexInt * **lastRowAddress_**
- CoinSimplexInt * **saveColumnAddress_**
- CoinCheckZero * **markRowAddress_**
- CoinSimplexInt * **listAddress_**
- CoinFactorizationDouble * **elementLAddress_**
- CoinSimplexInt * **indexRowLAddress_**
- CoinBigIndex * **startColumnLAddress_**
- CoinBigIndex * **startRowLAddress_**
- CoinSimplexInt * **pivotLinkedBackwardsAddress_**
- CoinSimplexInt * **pivotLinkedForwardsAddress_**
- CoinSimplexInt * **pivotLOrderAddress_**
- CoinBigIndex * **startColumnRAddress_**
- CoinFactorizationDouble * elementRAddress_

    Elements of R.
- CoinSimplexInt * indexRowRAddress_

    Row indices for R.
- CoinSimplexInt * **indexColumnLAddress_**
- CoinFactorizationDouble * **elementByRowLAddress_**
- CoinFactorizationDouble * **denseAreaAddress_**
- CoinFactorizationDouble * **workAreaAddress_**
- CoinSimplexUnsignedInt * **workArea2Address_**
- CoinSimplexInt * **sparseAddress_**
- CoinSimplexInt numberRowsExtra_

    Number of Rows after iterating.
- CoinSimplexInt maximumRowsExtra_

    Maximum number of Rows after iterating.

- CoinSimplexInt numberRowsSmall_

  Size of small inverse.
- CoinSimplexInt numberGoodL_

  Number factorized in L.
- CoinSimplexInt numberRowsLeft_

  Number Rows left (numberRows-numberGood)
- CoinBigIndex totalElements_

  Number of elements in U (to go) or while iterating total overall.
- CoinBigIndex firstZeroed_

  First place in funny copy zeroed out.
- CoinSimplexInt sparseThreshold_

  Below this use sparse technology - if 0 then no L row copy.
- CoinSimplexInt numberR_

  Number in R.
- CoinBigIndex lengthR_

  Length of R stuff.
- CoinBigIndex lengthAreaR_

  length of area reserved for R
- CoinBigIndex numberL_

  Number in L.
- CoinBigIndex baseL_

  Base of L.
- CoinBigIndex lengthL_

  Length of L.
- CoinBigIndex lengthAreaL_

  Length of area reserved for L.
- CoinSimplexInt numberU_

  Number in U.
- CoinBigIndex maximumU_

  Maximum space used in U.
- CoinBigIndex lengthU_

  Length of U.
- CoinBigIndex lengthAreaU_

  Length of area reserved for U.
- CoinBigIndex lastEntryByColumnU_

  Last entry by column for U.
- CoinBigIndex lastEntryByRowU_

  Last entry by row for U.
- CoinSimplexInt numberTrials_

  Number of trials before rejection.
- CoinSimplexInt leadingDimension_

  Leading dimension for dense.
- **CoinIntArrayWithLength** pivotColumn_

  Pivot order for each Column.
- **CoinIntArrayWithLength** permute_

  Permutation vector for pivot row order.
- **CoinBigIndexArrayWithLength** startRowU_

Start of each Row as pointer.
- **CoinIntArrayWithLength** numberInRow_

  Number in each Row.
- **CoinIntArrayWithLength** numberInColumn_

  Number in each Column.
- **CoinIntArrayWithLength** numberInColumnPlus_

  Number in each Column including pivoted.
- **CoinIntArrayWithLength** firstCount_

  First Row/Column with count of k, can tell which by offset - Rows then Columns.
- **CoinIntArrayWithLength** nextColumn_

  Next Column in memory order.
- **CoinIntArrayWithLength** lastColumn_

  Previous Column in memory order.
- **CoinIntArrayWithLength** nextRow_

  Next Row in memory order.
- **CoinIntArrayWithLength** lastRow_

  Previous Row in memory order.
- **CoinIntArrayWithLength** saveColumn_

  Columns left to do in a single pivot.
- **CoinIntArrayWithLength** markRow_

  Marks rows to be updated.
- **CoinIntArrayWithLength** indexColumnU_

  Base address for U (may change)
- **CoinFactorizationDoubleArrayWithLength** pivotRegion_

  Inverses of pivot values.
- **CoinFactorizationDoubleArrayWithLength** elementU_

  Elements of U.
- **CoinIntArrayWithLength** indexRowU_

  Row indices of U.
- **CoinBigIndexArrayWithLength** startColumnU_

  Start of each column in U.
- **CoinBigIndexArrayWithLength** convertRowToColumnU_

  Converts rows to columns in U.
- **CoinBigIndexArrayWithLength** convertColumnToRowU_

  Converts columns to rows in U.
- **CoinFactorizationDoubleArrayWithLength** elementRowU_

  Elements of U by row.
- **CoinFactorizationDoubleArrayWithLength** elementL_

  Elements of L.
- **CoinIntArrayWithLength** indexRowL_

  Row indices of L.
- **CoinBigIndexArrayWithLength** startColumnL_

  Start of each column in L.
- **CoinFactorizationDoubleArrayWithLength** denseArea_

  Dense area.
- **CoinFactorizationDoubleArrayWithLength** workArea_

  First work area.

- **CoinUnsignedIntArrayWithLength** workArea2_

  Second work area.
- **CoinBigIndexArrayWithLength** startRowL_

  Start of each row in L.
- **CoinIntArrayWithLength** indexColumnL_

  Index of column in row for L.
- **CoinFactorizationDoubleArrayWithLength** elementByRowL_

  Elements in L (row copy)
- **CoinIntArrayWithLength** sparse_

  Sparse regions.
- CoinSimplexInt messageLevel_

  Detail in messages.
- CoinBigIndex numberCompressions_

  Number of compressions done.
- CoinSimplexInt **lastSlack_**
- double ftranCountInput_

  To decide how to solve.
- double **ftranCountAfterL_**
- double **ftranCountAfterR_**
- double **ftranCountAfterU_**
- double **ftranAverageAfterL_**
- double **ftranAverageAfterR_**
- double **ftranAverageAfterU_**
- CoinSimplexInt **numberFtranCounts_**
- CoinSimplexInt maximumRows_

  Maximum rows (ever) (here to use double alignment)
- double **ftranFTCountInput_**
- double **ftranFTCountAfterL_**
- double **ftranFTCountAfterR_**
- double **ftranFTCountAfterU_**
- double **ftranFTAverageAfterL_**
- double **ftranFTAverageAfterR_**
- double **ftranFTAverageAfterU_**
- CoinSimplexInt **numberFtranFTCounts_**
- CoinSimplexInt denseThreshold_

  Dense threshold (here to use double alignment)
- double **btranCountInput_**
- double **btranCountAfterU_**
- double **btranCountAfterR_**
- double **btranCountAfterL_**
- double **btranAverageAfterU_**
- double **btranAverageAfterR_**
- double **btranAverageAfterL_**
- CoinSimplexInt **numberBtranCounts_**
- CoinSimplexInt maximumMaximumPivots_

  Maximum maximum pivots.
- double ftranFullCountInput_

  To decide how to solve.
- double **ftranFullCountAfterL_**

- double **ftranFullCountAfterR_**
- double **ftranFullCountAfterU_**
- double **ftranFullAverageAfterL_**
- double **ftranFullAverageAfterR_**
- double **ftranFullAverageAfterU_**
- CoinSimplexInt **numberFtranFullCounts_**
- CoinSimplexInt initialNumberRows_

    Rows first time nonzero.
- double btranFullCountInput_

    To decide how to solve.
- double **btranFullCountAfterL_**
- double **btranFullCountAfterR_**
- double **btranFullCountAfterU_**
- double **btranFullAverageAfterL_**
- double **btranFullAverageAfterR_**
- double **btranFullAverageAfterU_**
- CoinSimplexInt **numberBtranFullCounts_**
- CoinSimplexInt state_

    State of saved version and what can be done 0 - nothing saved 1 - saved and can go back to previous save by unwinding 2 - saved - getting on for a full copy higher bits - see ABC_FAC....
- CoinBigIndex sizeSparseArray_

    Size in bytes of a sparseArray.
- bool **gotLCopy** () const
- void **setNoGotLCopy** ()
- void **setYesGotLCopy** ()
- bool **gotRCopy** () const
- void **setNoGotRCopy** ()
- void **setYesGotRCopy** ()
- bool **gotUCopy** () const
- void **setNoGotUCopy** ()
- void **setYesGotUCopy** ()
- bool **gotSparse** () const
- void **setNoGotSparse** ()
- void **setYesGotSparse** ()

Additional Inherited Members

4.84.1 Detailed Description

Definition at line 28 of file CoinAbcBaseFactorization.hpp.

4.84.2 Member Function Documentation

4.84.2.1 CoinSimplexInt* CoinAbcTypeFactorization::firstCount ( ) const [inline]

For equal counts in factorization.

First Row/Column with count of k, can tell which by offset - Rows then Columns actually comes before nextCount

Definition at line 143 of file CoinAbcBaseFactorization.hpp.

4.84.2.2 virtual CoinBigIndex* CoinAbcTypeFactorization::starts ( ) const [inline], [virtual]

Returns pivot row.

Returns work area Returns CoinSimplexInt work area Returns array to put basis starts in

Reimplemented from CoinAbcAnyFactorization.

Definition at line 250 of file CoinAbcBaseFactorization.hpp.

4.84.2.3 virtual CoinSimplexInt CoinAbcTypeFactorization::updateColumnFT (CoinIndexedVector & *regionSparse*) [virtual]

Later take out return codes (apart from +- 1 on FT)

Updates one column (FTRAN) from regionSparse2 Tries to do FT update number returned is negative if no room regionSparse starts as zero and is zero at end. Note - if regionSparse2 packed on input - will be packed on output

Implements CoinAbcAnyFactorization.

4.84.2.4 virtual CoinSimplexInt CoinAbcTypeFactorization::updateTwoColumnsFT (CoinIndexedVector & *regionFT*, CoinIndexedVector & *regionOther*) [virtual]

Updates one column (FTRAN) from region2 Tries to do FT update number returned is negative if no room.

Also updates region3 region1 starts as zero and is zero at end

Implements CoinAbcAnyFactorization.

4.84.2.5 bool CoinAbcTypeFactorization::getColumnSpaceIterateR (CoinSimplexInt *iColumn*, CoinFactorizationDouble *value*, CoinSimplexInt *iRow*) [protected]

getColumnSpaceIterateR.

Gets space for one extra R element in Column may have to do compression (returns true) also moves existing vector

4.84.2.6 CoinBigIndex CoinAbcTypeFactorization::getColumnSpaceIterate (CoinSimplexInt *iColumn*, CoinFactorizationDouble *value*, CoinSimplexInt *iRow*) [protected]

getColumnSpaceIterate.

Gets space for one extra U element in Column may have to do compression (returns true) also moves existing vector. Returns -1 if no memory or where element was put Used by replaceRow (turns off R version)

4.84.2.7 void CoinAbcTypeFactorization::updateColumnTransposeU (CoinIndexedVector * *region*, CoinSimplexInt *smallestIndex*, CoinAbcStatistics & *statistics*) const [protected]

Updates part of column transpose (BTRANU), assumes index is sorted i.e.

region is correct

4.84.2.8 void CoinAbcTypeFactorization::updateColumnTransposeUDensish (CoinIndexedVector * *region*, CoinSimplexInt *smallestIndex*) const [protected]

Updates part of column transpose (BTRANU) when densish, assumes index is sorted i.e.

region is correct

4.84.2.9 void CoinAbcTypeFactorization::updateColumnTransposeUSparse (CoinIndexedVector * *region*) const [protected]

Updates part of column transpose (BTRANU) when sparse, assumes index is sorted i.e.

region is correct

4.84.2.10 void CoinAbcTypeFactorization::updateColumnTransposeUByColumn (CoinIndexedVector * *region,* CoinSimplexInt *smallestIndex*) const [protected]

Updates part of column transpose (BTRANU) by column assumes index is sorted i.e.

region is correct

4.84.2.11 CoinSimplexInt CoinAbcTypeFactorization::replaceColumnPFI (CoinIndexedVector * *regionSparse,* CoinSimplexInt *pivotRow,* CoinSimplexDouble *alpha*)

Replaces one Column to basis for PFI returns 0=OK, 1=Probably OK, 2=singular, 3=no room.

In this case region is not empty - it is incoming variable (updated)

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

- CoinAbcBaseFactorization.hpp

4.85 ClpHashValue::CoinHashLink Struct Reference

Data.

`#include <ClpNode.hpp>`

4.85.1 Detailed Description

Data.

Definition at line 335 of file ClpNode.hpp.

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

- ClpNode.hpp

4.86 dualColumnResult Struct Reference

4.86.1 Detailed Description

Definition at line 23 of file AbcSimplexDual.hpp.

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

- AbcSimplexDual.hpp

4.87 Idiot Class Reference

This class implements a very silly algorithm.

`#include <Idiot.hpp>`

Public Member Functions

- void solve2 (**CoinMessageHandler** *handler, const **CoinMessages** *messages)

    Stuff for internal use.

Constructors and destructor

Just a pointer to model is kept

- Idiot ()

    Default constructor.
- Idiot (**OsiSolverInterface** &model)

    Constructor with model.
- Idiot (const Idiot &)

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

    Assignment operator. This copies the data.
- ~Idiot ()

    Destructor.

Algorithmic calls

- void solve ()

    Get an approximate solution with the idiot code.
- void crash (int numberPass, **CoinMessageHandler** *handler, const **CoinMessages** *messages, bool do↩ Crossover=true)

    Lightweight "crash".
- void crossOver (int mode)

    Use simplex to get an optimal solution mode is how many steps the simplex crossover should take to arrive to an extreme point: 0 - chosen,all ever used, all 1 - chosen, all 2 - all 3 - do not do anything - maybe basis.

Gets and sets of most useful data

- double getStartingWeight () const

    Starting weight - small emphasizes feasibility, default 1.0e-4.
- void **setStartingWeight** (double value)
- double getWeightFactor () const

    Weight factor - weight multiplied by this when changes, default 0.333.
- void **setWeightFactor** (double value)
- double getFeasibilityTolerance () const

    Feasibility tolerance - problem essentially feasible if individual infeasibilities less than this.
- void **setFeasibilityTolerance** (double value)
- double getReasonablyFeasible () const

    Reasonably feasible.
- void **setReasonablyFeasible** (double value)
- double getExitInfeasibility () const

    Exit infeasibility - exit if sum of infeasibilities less than this.

- void **setExitInfeasibility** (double value)
- int getMajorIterations () const

    Major iterations.
- void **setMajorIterations** (int value)
- int getMinorIterations () const

    Minor iterations.
- void **setMinorIterations** (int value)
- int **getMinorIterations0** () const
- void **setMinorIterations0** (int value)
- int getReduceIterations () const

    Reduce weight after this many major iterations.
- void **setReduceIterations** (int value)
- int getLogLevel () const

    Amount of information - default of 1 should be okay.
- void **setLogLevel** (int value)
- int getLightweight () const

    How lightweight - 0 not, 1 yes, 2 very lightweight.
- void **setLightweight** (int value)
- int getStrategy () const

    strategy
- void **setStrategy** (int value)
- double getDropEnoughFeasibility () const

    Fine tuning - okay if feasibility drop this factor.
- void **setDropEnoughFeasibility** (double value)
- double getDropEnoughWeighted () const

    Fine tuning - okay if weighted obj drop this factor.
- void **setDropEnoughWeighted** (double value)
- void setModel (**OsiSolverInterface** ∗model)

    Set model.

4.87.1 Detailed Description

This class implements a very silly algorithm.

It has no merit apart from the fact that it gets an approximate solution to some classes of problems. Better if vaguely homogeneous. It works on problems where volume algorithm works and often gets a better primal solution but it has no dual solution.

It can also be used as a "crash" to get a problem started. This is probably its most useful function.

It is based on the idea that algorithms with terrible convergence properties may be okay at first. Throw in some random dubious tricks and the resulting code may be worth keeping as long as you don't look at it.

Definition at line 48 of file Idiot.hpp.

4.87.2 Member Function Documentation

4.87.2.1 void Idiot::crossOver (int *mode*)

Use simplex to get an optimal solution mode is how many steps the simplex crossover should take to arrive to an extreme point: 0 - chosen,all ever used, all 1 - chosen, all 2 - all 3 - do not do anything - maybe basis.

- 16 do presolves

4.87.2.2 double Idiot::getFeasibilityTolerance ( ) const [inline]

Feasibility tolerance - problem essentially feasible if individual infeasibilities less than this.

default 0.1

Definition at line 113 of file Idiot.hpp.

4.87.2.3 double Idiot::getReasonablyFeasible ( ) const [inline]

Reasonably feasible.

Dubious method concentrates more on objective when sum of infeasibilities less than this. Very dubious default value of (Number of rows)/20

Definition at line 122 of file Idiot.hpp.

4.87.2.4 double Idiot::getExitInfeasibility ( ) const [inline]

Exit infeasibility - exit if sum of infeasibilities less than this.

Default -1.0 (i.e. switched off)

Definition at line 130 of file Idiot.hpp.

4.87.2.5 int Idiot::getMajorIterations ( ) const [inline]

Major iterations.

stop after this number. Default 30. Use 2-5 for "crash" 50-100 for serious crunching

Definition at line 138 of file Idiot.hpp.

4.87.2.6 int Idiot::getMinorIterations ( ) const [inline]

Minor iterations.

Do this number of tiny steps before deciding whether to change weights etc. Default - dubious sqrt(Number of Rows). Good numbers 105 to 405 say (5 is dubious method of making sure idiot is not trying to be clever which it may do every 10 minor iterations)

Definition at line 150 of file Idiot.hpp.

4.87.2.7 int Idiot::getReduceIterations ( ) const [inline]

Reduce weight after this many major iterations.

It may get reduced before this but this is a maximum. Default 3. 3-10 plausible.

Definition at line 166 of file Idiot.hpp.

4.87.2.8 void Idiot::solve2 (CoinMessageHandler * *handler*, const CoinMessages * *messages*)

Stuff for internal use.

Does actual work

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

- Idiot.hpp

4.88 IdiotResult Struct Reference

for use internally

`#include <Idiot.hpp>`

4.88.1 Detailed Description

for use internally

Definition at line 22 of file Idiot.hpp.

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

- Idiot.hpp

4.89 Info Struct Reference

******** DATA to be moved into protected section of ClpInterior

`#include <ClpInterior.hpp>`

4.89.1 Detailed Description

******** DATA to be moved into protected section of ClpInterior

Definition at line 27 of file ClpInterior.hpp.

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

- ClpInterior.hpp

4.90 MyEventHandler Class Reference

This is so user can trap events and do useful stuff.

`#include <MyEventHandler.hpp>`

Inheritance diagram for MyEventHandler:

Collaboration diagram for MyEventHandler:

Public Member Functions

Overrides

- virtual int event (Event whichEvent)

This can do whatever it likes.

Constructors, destructor etc

- MyEventHandler ()
  Default constructor.
- MyEventHandler (ClpSimplex ∗model)
  Constructor with pointer to model (redundant as setEventHandler does)
- virtual ∼MyEventHandler ()
  Destructor.
- MyEventHandler (const MyEventHandler &rhs)
  The copy constructor.
- MyEventHandler & operator= (const MyEventHandler &rhs)
  Assignment.
- virtual ClpEventHandler ∗ clone () const
  Clone.

Additional Inherited Members

4.90.1 Detailed Description

This is so user can trap events and do useful stuff.

This is used in Clp/Test/unitTest.cpp

ClpSimplex model_ is available as well as anything else you care to pass in

Definition at line 18 of file MyEventHandler.hpp.

4.90.2 Constructor & Destructor Documentation

4.90.2.1 MyEventHandler::MyEventHandler ( )

Default constructor.

4.90.2.2 MyEventHandler::MyEventHandler (const MyEventHandler & *rhs*)

The copy constructor.

4.90.3 Member Function Documentation

4.90.3.1 virtual int MyEventHandler::event (Event *whichEvent*) [virtual]

This can do whatever it likes.

If return code -1 then carries on if 0 sets ClpModel::status() to 5 (stopped by event) and will return to user. At present if <-1 carries on and if >0 acts as if 0 - this may change. For ClpSolve 2 -> too big return status of -2 and -> too small 3

Reimplemented from ClpEventHandler.

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

- MyEventHandler.hpp

4.91 MyMessageHandler Class Reference

Inheritance diagram for MyMessageHandler:

Collaboration diagram for MyMessageHandler:

Public Member Functions

Overrides

- virtual int **print** ()

set and get

- const ClpSimplex ∗ model () const
  Model.
- void **setModel** (ClpSimplex ∗model)
- const std::deque< StdVectorDouble > & getFeasibleExtremePoints () const
  Get queue of feasible extreme points.
- void clearFeasibleExtremePoints ()
  Empty queue of feasible extreme points.

Constructors, destructor

- MyMessageHandler ()
  Default constructor.
- MyMessageHandler (ClpSimplex ∗model, FILE ∗userPointer=NULL)
  Constructor with pointer to model.
- virtual ∼MyMessageHandler ()
  Destructor.

Copy method

- MyMessageHandler (const MyMessageHandler &)
  The copy constructor.
- MyMessageHandler (const **CoinMessageHandler** &)
  The copy constructor from an CoinSimplexMessageHandler.
- MyMessageHandler & **operator=** (const MyMessageHandler &)
- virtual **CoinMessageHandler** ∗ clone () const
  Clone.

Protected Attributes

Data members

The data members are protected to allow access for derived classes.

- ClpSimplex ∗ model_
  Pointer back to model.
- std::deque< StdVectorDouble > feasibleExtremePoints_
  Saved extreme points.
- int iterationNumber_
  Iteration number so won't do same one twice.

4.91.1 Detailed Description

Definition at line 28 of file MyMessageHandler.hpp.

4.91.2 Constructor & Destructor Documentation

4.91.2.1 MyMessageHandler::MyMessageHandler ( )

Default constructor.

4.91.2.2 MyMessageHandler::MyMessageHandler (const MyMessageHandler &)

The copy constructor.

4.91.2.3 MyMessageHandler::MyMessageHandler (const CoinMessageHandler &)

The copy constructor from an CoinSimplexMessageHandler.

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

- MyMessageHandler.hpp

4.92 Options Struct Reference

******** DATA to be moved into protected section of ClpInterior

`#include <ClpInterior.hpp>`

4.92.1 Detailed Description

******** DATA to be moved into protected section of ClpInterior

Definition at line 44 of file ClpInterior.hpp.

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

- ClpInterior.hpp

4.93 OsiClpDisasterHandler Class Reference

Inheritance diagram for OsiClpDisasterHandler:

Collaboration diagram for OsiClpDisasterHandler:

Public Member Functions

Virtual methods that the derived classe should provide.

- virtual void intoSimplex ()

Into simplex.
- virtual bool check () const

  Checks if disaster.
- virtual void saveInfo ()

  saves information for next attempt
- virtual int typeOfDisaster ()

  Type of disaster 0 can fix, 1 abort.

Constructors, destructor

- OsiClpDisasterHandler (OsiClpSolverInterface ∗model=NULL)

  Default constructor.
- virtual ∼OsiClpDisasterHandler ()

  Destructor.
- **OsiClpDisasterHandler** (const OsiClpDisasterHandler &)
- OsiClpDisasterHandler & **operator=** (const OsiClpDisasterHandler &)
- virtual ClpDisasterHandler ∗ clone () const

  Clone.

Sets/gets

- void setOsiModel (OsiClpSolverInterface ∗model)

  set model.
- OsiClpSolverInterface ∗ osiModel () const

  Get model.
- void setWhereFrom (int value)

  Set where from.
- int whereFrom () const

  Get where from.
- void setPhase (int value)

  Set phase.
- int phase () const

  Get phase.
- bool inTrouble () const

  are we in trouble

Protected Attributes

Data members

The data members are protected to allow access for derived classes.

- OsiClpSolverInterface ∗ osiModel_

  Pointer to model.
- int whereFrom_

  Where from 0 dual (resolve) 1 crunch 2 primal (resolve) 4 dual (initialSolve) 6 primal (initialSolve)
- int phase_

  phase 0 initial 1 trying continuing with back in and maybe different perturb 2 trying continuing with back in and different scaling 3 trying dual from all slack 4 trying primal from previous stored basis
- bool inTrouble_

  Are we in trouble.

4.93.1 Detailed Description

Definition at line 1420 of file OsiClpSolverInterface.hpp.

4.93.2 Constructor & Destructor Documentation

4.93.2.1 OsiClpDisasterHandler::OsiClpDisasterHandler (OsiClpSolverInterface * *model* = NULL)

Default constructor.

4.93.3 Member Function Documentation

4.93.3.1 void OsiClpDisasterHandler::setOsiModel (OsiClpSolverInterface * *model*)

set model.

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

- OsiClpSolverInterface.hpp

4.94 OsiClpSolverInterface Class Reference

Clp Solver Interface.

`#include <OsiClpSolverInterface.hpp>`

Inheritance diagram for OsiClpSolverInterface:

Collaboration diagram for OsiClpSolverInterface:

Public Member Functions

- virtual void setObjSense (double s)

  Set objective function sense (1 for min (default), -1 for max,)
- virtual void setColSolution (const double *colsol)

  Set the primal solution column values.
- virtual void setRowPrice (const double *rowprice)

  Set dual solution vector.

Solve methods

- virtual void initialSolve ()

  Solve initial LP relaxation.
- virtual void resolve ()

  Resolve an LP relaxation after problem modification.
- virtual void resolveGub (int needed)

  Resolve an LP relaxation after problem modification (try GUB)
- virtual void branchAndBound ()

  Invoke solver's built-in enumeration algorithm.
- void crossover (int options, int basis)

Solve when primal column and dual row solutions are near-optimal options - 0 no presolve (use primal and dual) 1 presolve (just use primal) 2 no presolve (just use primal) basis - 0 use all slack basis 1 try and put some in basis.

OsiSimplexInterface methods

Methods for the Osi Simplex API.

The current implementation should work for both minimisation and maximisation in mode 1 (tableau access). In mode 2 (single pivot), only minimisation is supported as of 100907.

- virtual int canDoSimplexInterface () const
  Simplex API capability.
- virtual void enableFactorization () const
  Enables simplex mode 1 (tableau access)
- virtual void disableFactorization () const
  Undo any setting changes made by enableFactorization.
- virtual bool basisIsAvailable () const
  Returns true if a basis is available AND problem is optimal.
- virtual void getBasisStatus (int ∗cstat, int ∗rstat) const
  The following two methods may be replaced by the methods of **OsiSolverInterface** *using OsiWarmStartBasis if:*
- virtual int setBasisStatus (const int ∗cstat, const int ∗rstat)
  Set the status of structural/artificial variables and factorize, update solution etc.
- virtual void getReducedGradient (double ∗columnReducedCosts, double ∗duals, const double ∗c) const
  Get the reduced gradient for the cost vector c.
- virtual void getBInvARow (int row, double ∗z, double ∗slack=NULL) const
  Get a row of the tableau (slack part in slack if not NULL)
- virtual void getBInvARow (int row, **CoinIndexedVector** ∗z, **CoinIndexedVector** ∗slack=NULL, bool keep←Scaled=false) const
  Get a row of the tableau (slack part in slack if not NULL) If keepScaled is true then scale factors not applied after so user has to use coding similar to what is in this method.
- virtual void getBInvRow (int row, double ∗z) const
  Get a row of the basis inverse.
- virtual void getBInvACol (int col, double ∗vec) const
  Get a column of the tableau.
- virtual void getBInvACol (int col, **CoinIndexedVector** ∗vec) const
  Get a column of the tableau.
- virtual void getBInvACol (**CoinIndexedVector** ∗vec) const
  Update (i.e.
- virtual void getBInvCol (int col, double ∗vec) const
  Get a column of the basis inverse.
- virtual void getBasics (int ∗index) const
  Get basic indices (order of indices corresponds to the order of elements in a vector retured by getBInvACol() and getBInvCol()).
- virtual void enableSimplexInterface (bool doingPrimal)
  Enables simplex mode 2 (individual pivot control)
- void copyEnabledSuff (OsiClpSolverInterface &rhs)
  Copy across enabled stuff from one solver to another.
- virtual void disableSimplexInterface ()
  Undo setting changes made by enableSimplexInterface.
- void copyEnabledStuff (ClpSimplex &rhs)
  Copy across enabled stuff from one solver to another.
- virtual int pivot (int colIn, int colOut, int outStatus)
  Perform a pivot by substituting a colIn for colOut in the basis.
- virtual int primalPivotResult (int colIn, int sign, int &colOut, int &outStatus, double &t, **CoinPackedVector** ∗dx)

Obtain a result of the primal pivot Outputs: colOut – leaving column, outStatus – its status, t – step size, and, if dx!=NULL, ∗dx – primal ray direction.

- virtual int dualPivotResult (int &colIn, int &sign, int colOut, int outStatus, double &t, **CoinPackedVector** ∗dx)

  Obtain a result of the dual pivot (similar to the previous method) Differences: entering variable and a sign of its change are now the outputs, the leaving variable and its statuts – the inputs If dx!=NULL, then ∗dx contains dual ray Return code: same.

Parameter set/get methods

The set methods return true if the parameter was set to the given value, false otherwise.

There can be various reasons for failure: the given parameter is not applicable for the solver (e.g., refactorization frequency for the clp algorithm), the parameter is not yet implemented for the solver or simply the value of the parameter is out of the range the solver accepts. If a parameter setting call returns false check the details of your solver.

The get methods return true if the given parameter is applicable for the solver and is implemented. In this case the value of the parameter is returned in the second argument. Otherwise they return false.

- bool **setIntParam** (OsiIntParam key, int value)
- bool **setDblParam** (OsiDblParam key, double value)
- bool **setStrParam** (OsiStrParam key, const std::string &value)
- bool **getIntParam** (OsiIntParam key, int &value) const
- bool **getDblParam** (OsiDblParam key, double &value) const
- bool **getStrParam** (OsiStrParam key, std::string &value) const
- virtual bool **setHintParam** (OsiHintParam key, bool yesNo=true, OsiHintStrength strength=OsiHintTry, void ∗otherInformation=NULL)

Methods returning info on how the solution process terminated

- virtual bool isAbandoned () const

  Are there a numerical difficulties?
- virtual bool isProvenOptimal () const

  Is optimality proven?
- virtual bool isProvenPrimalInfeasible () const

  Is primal infeasiblity proven?
- virtual bool isProvenDualInfeasible () const

  Is dual infeasiblity proven?
- virtual bool isPrimalObjectiveLimitReached () const

  Is the given primal objective limit reached?
- virtual bool isDualObjectiveLimitReached () const

  Is the given dual objective limit reached?
- virtual bool isIterationLimitReached () const

  Iteration limit reached?

WarmStart related methods

- virtual **CoinWarmStart** ∗ getEmptyWarmStart () const

  Get an empty warm start object.
- virtual **CoinWarmStart** ∗ getWarmStart () const

  Get warmstarting information.
- **CoinWarmStartBasis** ∗ getPointerToWarmStart ()

  Get warmstarting information.
- const **CoinWarmStartBasis** ∗ getConstPointerToWarmStart () const

  Get warmstarting information.
- virtual bool setWarmStart (const **CoinWarmStart** ∗warmstart)

Set warmstarting information.

- virtual **CoinWarmStart** ∗ getPointerToWarmStart (bool &mustDelete)

  Get warm start information.

- void setColumnStatus (int iColumn, ClpSimplex::Status status)

  Set column status in ClpSimplex and warmStart.

Hotstart related methods (primarily used in strong branching).

The user can create a hotstart (a snapshot) of the optimization process then reoptimize over and over again always starting from there.

***NOTE**: between hotstarted optimizations only bound changes are allowed.*

- virtual void markHotStart ()

  Create a hotstart point of the optimization process.

- virtual void solveFromHotStart ()

  Optimize starting from the hotstart.

- virtual void unmarkHotStart ()

  Delete the snapshot.

- int startFastDual (int options)

  Start faster dual - returns negative if problems 1 if infeasible, Options to pass to solver 1 - create external reduced costs for columns 2 - create external reduced costs for rows 4 - create external row activity (columns always done) Above only done if feasible When set resolve does less work.

- void stopFastDual ()

  Stop fast dual.

- void setStuff (double tolerance, double increment)

  Sets integer tolerance and increment.

- **OsiRowCut** ∗ smallModelCut (const double ∗originalLower, const double ∗originalUpper, int numberRowsAtContinuous, const int ∗whichGenerator, int typeCut=0)

  Return a conflict analysis cut from small model.

- **OsiRowCut** ∗ modelCut (const double ∗originalLower, const double ∗originalUpper, int numberRowsAtContinuous, const int ∗whichGenerator, int typeCut=0)

  Return a conflict analysis cut from model If type is 0 then genuine cut, if 1 then only partially processed.

Methods related to querying the input data

- virtual int getNumCols () const

  Get number of columns.

- virtual int getNumRows () const

  Get number of rows.

- virtual int getNumElements () const

  Get number of nonzero elements.

- virtual std::string getRowName (int rowIndex, unsigned maxLen=static_cast< unsigned >(std::string::npos)) const

  Return name of row if one exists or Rnnnnnnn maxLen is currently ignored and only there to match the signature from the base class!

- virtual std::string getColName (int colIndex, unsigned maxLen=static_cast< unsigned >(std::string::npos)) const

  Return name of column if one exists or Cnnnnnnn maxLen is currently ignored and only there to match the signature from the base class!

- virtual const double ∗ getColLower () const

  Get pointer to array[getNumCols()] of column lower bounds.

- virtual const double ∗ getColUpper () const

Get pointer to array[getNumCols()] of column upper bounds.

- virtual const char ∗ getRowSense () const

  Get pointer to array[getNumRows()] of row constraint senses.
- virtual const double ∗ getRightHandSide () const

  Get pointer to array[getNumRows()] of rows right-hand sides.
- virtual const double ∗ getRowRange () const

  Get pointer to array[getNumRows()] of row ranges.
- virtual const double ∗ getRowLower () const

  Get pointer to array[getNumRows()] of row lower bounds.
- virtual const double ∗ getRowUpper () const

  Get pointer to array[getNumRows()] of row upper bounds.
- virtual const double ∗ getObjCoefficients () const

  Get pointer to array[getNumCols()] of objective function coefficients.
- virtual double getObjSense () const

  Get objective function sense (1 for min (default), -1 for max)
- virtual bool isContinuous (int colNumber) const

  Return true if column is continuous.
- virtual bool isBinary (int colIndex) const

  Return true if variable is binary.
- virtual bool isInteger (int colIndex) const

  Return true if column is integer.
- virtual bool isIntegerNonBinary (int colIndex) const

  Return true if variable is general integer.
- virtual bool isFreeBinary (int colIndex) const

  Return true if variable is binary and not fixed at either bound.
- virtual const char ∗ getColType (bool refresh=false) const

  Return array of column length 0 - continuous 1 - binary (may get fixed later) 2 - general integer (may get fixed later)
- bool isOptionalInteger (int colIndex) const

  Return true if column is integer but does not have to be declared as such.
- void setOptionalInteger (int index)

  Set the index-th variable to be an optional integer variable.
- virtual const **CoinPackedMatrix** ∗ getMatrixByRow () const

  Get pointer to row-wise copy of matrix.
- virtual const **CoinPackedMatrix** ∗ getMatrixByCol () const

  Get pointer to column-wise copy of matrix.
- virtual **CoinPackedMatrix** ∗ getMutableMatrixByCol () const

  Get pointer to mutable column-wise copy of matrix.
- virtual double getInfinity () const

  Get solver's value for infinity.

Methods related to querying the solution

- virtual const double ∗ getColSolution () const

  Get pointer to array[getNumCols()] of primal solution vector.
- virtual const double ∗ getRowPrice () const

  Get pointer to array[getNumRows()] of dual prices.
- virtual const double ∗ getReducedCost () const

  Get a pointer to array[getNumCols()] of reduced costs.
- virtual const double ∗ getRowActivity () const

  Get pointer to array[getNumRows()] of row activity levels (constraint matrix times the solution vector.
- virtual double getObjValue () const

  Get objective function value.

- virtual int getIterationCount () const

  Get how many iterations it took to solve the problem (whatever "iteration" mean to the solver.
- virtual std::vector< double ∗ > getDualRays (int maxNumRays, bool fullRay=false) const

  Get as many dual rays as the solver can provide.
- virtual std::vector< double ∗ > getPrimalRays (int maxNumRays) const

  Get as many primal rays as the solver can provide.

Changing bounds on variables and constraints

- virtual void setObjCoeff (int elementIndex, double elementValue)

  Set an objective function coefficient.
- virtual void setColLower (int elementIndex, double elementValue)

  Set a single column lower bound
  Use -DBL_MAX for -infinity.
- virtual void setColUpper (int elementIndex, double elementValue)

  Set a single column upper bound
  Use DBL_MAX for infinity.
- virtual void setColBounds (int elementIndex, double lower, double upper)

  Set a single column lower and upper bound.
- virtual void setColSetBounds (const int ∗indexFirst, const int ∗indexLast, const double ∗boundList)

  Set the bounds on a number of columns simultaneously
  The default implementation just invokes setColLower() and setColUpper() over and over again.
- virtual void setRowLower (int elementIndex, double elementValue)

  Set a single row lower bound
  Use -DBL_MAX for -infinity.
- virtual void setRowUpper (int elementIndex, double elementValue)

  Set a single row upper bound
  Use DBL_MAX for infinity.
- virtual void setRowBounds (int elementIndex, double lower, double upper)

  Set a single row lower and upper bound.
- virtual void setRowType (int index, char sense, double rightHandSide, double range)

  Set the type of a single row
- virtual void setRowSetBounds (const int ∗indexFirst, const int ∗indexLast, const double ∗boundList)

  Set the bounds on a number of rows simultaneously
  The default implementation just invokes setRowLower() and setRowUpper() over and over again.
- virtual void setRowSetTypes (const int ∗indexFirst, const int ∗indexLast, const char ∗senseList, const double ∗rhsList, const double ∗rangeList)

  Set the type of a number of rows simultaneously
  The default implementation just invokes setRowType() over and over again.
- virtual void setObjective (const double ∗array)

  Set the objective coefficients for all columns array [getNumCols()] is an array of values for the objective.
- virtual void setColLower (const double ∗array)

  Set the lower bounds for all columns array [getNumCols()] is an array of values for the objective.
- virtual void setColUpper (const double ∗array)

  Set the upper bounds for all columns array [getNumCols()] is an array of values for the objective.
- virtual void setRowName (int rowIndex, std::string name)

  Set name of row.
- virtual void setColName (int colIndex, std::string name)

  Set name of column.

Integrality related changing methods

- virtual void setContinuous (int index)

Set the index-th variable to be a continuous variable.
- virtual void setInteger (int index)

  Set the index-th variable to be an integer variable.
- virtual void setContinuous (const int ∗indices, int len)

  Set the variables listed in indices (which is of length len) to be continuous variables.
- virtual void setInteger (const int ∗indices, int len)

  Set the variables listed in indices (which is of length len) to be integer variables.
- int numberSOS () const

  Number of SOS sets.
- const **CoinSet** ∗ setInfo () const

  SOS set info.
- virtual int findIntegersAndSOS (bool justCount)

  Identify integer variables and SOS and create corresponding objects.

Methods to expand a problem.

Note that if a column is added then by default it will correspond to a continuous variable.

- virtual void **addCol** (const **CoinPackedVectorBase** &vec, const double collb, const double colub, const double obj)
- virtual void addCol (const **CoinPackedVectorBase** &vec, const double collb, const double colub, const double obj, std::string name)

  Add a named column (primal variable) to the problem.
- virtual void addCol (int numberElements, const int ∗rows, const double ∗elements, const double collb, const double colub, const double obj)

  Add a column (primal variable) to the problem.
- virtual void addCol (int numberElements, const int ∗rows, const double ∗elements, const double collb, const double colub, const double obj, std::string name)

  Add a named column (primal variable) to the problem.
- virtual void **addCols** (const int numcols, const **CoinPackedVectorBase** ∗const ∗cols, const double ∗collb, const double ∗colub, const double ∗obj)
- virtual void **addCols** (const int numcols, const int ∗columnStarts, const int ∗rows, const double ∗elements, const double ∗collb, const double ∗colub, const double ∗obj)
- virtual void **deleteCols** (const int num, const int ∗colIndices)
- virtual void **addRow** (const **CoinPackedVectorBase** &vec, const double rowlb, const double rowub)
- virtual void addRow (const **CoinPackedVectorBase** &vec, const double rowlb, const double rowub, std::string name)

  Add a named row (constraint) to the problem.
- virtual void **addRow** (const **CoinPackedVectorBase** &vec, const char rowsen, const double rowrhs, const double rowrng)
- virtual void addRow (int numberElements, const int ∗columns, const double ∗element, const double rowlb, const double rowub)

  Add a row (constraint) to the problem.
- virtual void addRow (const **CoinPackedVectorBase** &vec, const char rowsen, const double rowrhs, const double rowrng, std::string name)

  Add a named row (constraint) to the problem.
- virtual void **addRows** (const int numrows, const **CoinPackedVectorBase** ∗const ∗rows, const double ∗rowlb, const double ∗rowub)
- virtual void **addRows** (const int numrows, const **CoinPackedVectorBase** ∗const ∗rows, const char ∗rowsen, const double ∗rowrhs, const double ∗rowrng)
- virtual void **addRows** (const int numrows, const int ∗rowStarts, const int ∗columns, const double ∗element, const double ∗rowlb, const double ∗rowub)
- void **modifyCoefficient** (int row, int column, double newElement, bool keepZero=false)
- virtual void **deleteRows** (const int num, const int ∗rowIndices)
- virtual void saveBaseModel ()

If solver wants it can save a copy of "base" (continuous) model here.
- virtual void restoreBaseModel (int numberRows)

  Strip off rows to get to this number of rows.
- virtual void applyRowCuts (int numberCuts, const **OsiRowCut** ∗cuts)

  Apply a collection of row cuts which are all effective.
- virtual void applyRowCuts (int numberCuts, const **OsiRowCut** ∗∗cuts)

  Apply a collection of row cuts which are all effective.
- virtual ApplyCutsReturnCode applyCuts (const **OsiCuts** &cs, double effectivenessLb=0.0)

  Apply a collection of cuts.

Methods to input a problem

- virtual void loadProblem (const **CoinPackedMatrix** &matrix, const double ∗collb, const double ∗colub, const double ∗obj, const double ∗rowlb, const double ∗rowub)

  Load in an problem by copying the arguments (the constraints on the rows are given by lower and upper bounds).
- virtual void assignProblem (**CoinPackedMatrix** ∗&matrix, double ∗&collb, double ∗&colub, double ∗&obj, double ∗&rowlb, double ∗&rowub)

  Load in an problem by assuming ownership of the arguments (the constraints on the rows are given by lower and upper bounds).
- virtual void loadProblem (const **CoinPackedMatrix** &matrix, const double ∗collb, const double ∗colub, const double ∗obj, const char ∗rowsen, const double ∗rowrhs, const double ∗rowrng)

  Load in an problem by copying the arguments (the constraints on the rows are given by sense/rhs/range triplets).
- virtual void assignProblem (**CoinPackedMatrix** ∗&matrix, double ∗&collb, double ∗&colub, double ∗&obj, char ∗&rowsen, double ∗&rowrhs, double ∗&rowrng)

  Load in an problem by assuming ownership of the arguments (the constraints on the rows are given by sense/rhs/range triplets).
- virtual void loadProblem (const ClpMatrixBase &matrix, const double ∗collb, const double ∗colub, const double ∗obj, const double ∗rowlb, const double ∗rowub)

  Just like the other loadProblem() methods except that the matrix is given as a ClpMatrixBase.
- virtual void loadProblem (const int numcols, const int numrows, const CoinBigIndex ∗start, const int ∗index, const double ∗value, const double ∗collb, const double ∗colub, const double ∗obj, const double ∗rowlb, const double ∗rowub)

  Just like the other loadProblem() methods except that the matrix is given in a standard column major ordered format (without gaps).
- virtual void loadProblem (const int numcols, const int numrows, const CoinBigIndex ∗start, const int ∗index, const double ∗value, const double ∗collb, const double ∗colub, const double ∗obj, const char ∗rowsen, const double ∗rowrhs, const double ∗rowrng)

  Just like the other loadProblem() methods except that the matrix is given in a standard column major ordered format (without gaps).
- virtual int loadFromCoinModel (**CoinModel** &modelObject, bool keepSolution=false)

  This loads a model from a coinModel object - returns number of errors.
- virtual int readMps (const char ∗filename, const char ∗extension="mps")

  Read an mps file from the given filename (defaults to Osi reader) - returns number of errors (see OsiMpsReader class)
- int readMps (const char ∗filename, bool keepNames, bool allowErrors)

  Read an mps file from the given filename returns number of errors (see OsiMpsReader class)
- virtual int readMps (const char ∗filename, const char ∗extension, int &numberSets, **CoinSet** ∗∗&sets)

  Read an mps file.
- virtual void writeMps (const char ∗filename, const char ∗extension="mps", double objSense=0.0) const

  Write the problem into an mps file of the given filename.
- virtual int writeMpsNative (const char ∗filename, const char ∗∗rowNames, const char ∗∗columnNames, int formatType=0, int numberAcross=2, double objSense=0.0) const

  Write the problem into an mps file of the given filename, names may be null.
- virtual int readLp (const char ∗filename, const double epsilon=1e-5)

  Read file in LP format (with names)

- virtual void writeLp (const char ∗filename, const char ∗extension="lp", double epsilon=1e-5, int numberAcross=10, int decimals=5, double objSense=0.0, bool useRowNames=true) const

  Write the problem into an Lp file of the given filename.
- virtual void writeLp (FILE ∗fp, double epsilon=1e-5, int numberAcross=10, int decimals=5, double objSense=0.0, bool useRowNames=true) const

  Write the problem into the file pointed to by the parameter fp.
- virtual void replaceMatrixOptional (const **CoinPackedMatrix** &matrix)

  I (JJF) am getting annoyed because I can't just replace a matrix.
- virtual void replaceMatrix (const **CoinPackedMatrix** &matrix)

  And if it does matter (not used at present)

Message handling (extra for Clp messages).

Normally I presume you would want the same language.

If not then you could use underlying model pointer

- virtual void passInMessageHandler (**CoinMessageHandler** ∗handler)

  Pass in a message handler.
- void newLanguage (**CoinMessages::Language** language)

  Set language.
- void **setLanguage** (**CoinMessages::Language** language)
- void setLogLevel (int value)

  Set log level (will also set underlying solver's log level)
- void generateCpp (FILE ∗fp)

  Create C++ lines to get to current state.

Clp specific public interfaces

- ClpSimplex ∗ getModelPtr () const

  Get pointer to Clp model.
- ClpSimplex ∗ swapModelPtr (ClpSimplex ∗newModel)

  Set pointer to Clp model and return old.
- unsigned int specialOptions () const

  Get special options.
- void **setSpecialOptions** (unsigned int value)
- int lastAlgorithm () const

  Last algorithm used , 1 = primal, 2 = dual other unknown.
- void setLastAlgorithm (int value)

  Set last algorithm used , 1 = primal, 2 = dual other unknown.
- int cleanupScaling () const

  Get scaling action option.
- void setCleanupScaling (int value)

  Set Scaling option When scaling is on it is possible that the scaled problem is feasible but the unscaled is not.
- double smallestElementInCut () const

  Get smallest allowed element in cut.
- void setSmallestElementInCut (double value)

  Set smallest allowed element in cut.
- double smallestChangeInCut () const

  Get smallest change in cut.
- void setSmallestChangeInCut (double value)

  Set smallest change in cut.
- void setSolveOptions (const ClpSolve &options)

  Pass in initial solve options.
- virtual int tightenBounds (int lightweight=0)

Tighten bounds - lightweight or very lightweight 0 - normal, 1 lightweight but just integers, 2 lightweight and all.
- int infeasibleOtherWay (char *whichWay)

  See if any integer variables make infeasible other way.
- virtual CoinBigIndex getSizeL () const

  Return number of entries in L part of current factorization.
- virtual CoinBigIndex getSizeU () const

  Return number of entries in U part of current factorization.
- const OsiClpDisasterHandler * disasterHandler () const

  Get disaster handler.
- void passInDisasterHandler (OsiClpDisasterHandler *handler)

  Pass in disaster handler.
- ClpLinearObjective * fakeObjective () const

  Get fake objective.
- void setFakeObjective (ClpLinearObjective *fakeObjective)

  Set fake objective (and take ownership)
- void setFakeObjective (double *fakeObjective)

  Set fake objective.
- void setupForRepeatedUse (int senseOfAdventure=0, int printOut=0)

  Set up solver for repeated use by Osi interface.
- virtual void synchronizeModel ()

  Synchronize model (really if no cuts in tree)
- void setSpecialOptionsMutable (unsigned int value) const

  Set special options in underlying clp solver.

Constructors and destructors

- OsiClpSolverInterface ()

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

  Clone.
- OsiClpSolverInterface (const OsiClpSolverInterface &)

  Copy constructor.
- OsiClpSolverInterface (ClpSimplex *rhs, bool reallyOwn=false)

  Borrow constructor - only delete one copy.
- void releaseClp ()

  Releases so won't error.
- OsiClpSolverInterface & operator= (const OsiClpSolverInterface &rhs)

  Assignment operator.
- virtual ~OsiClpSolverInterface ()

  Destructor.
- virtual void reset ()

  Resets as if default constructor.

Protected Attributes

Protected member data

- ClpSimplex * modelPtr_

  Clp model represented by this class instance.

Cached information derived from the OSL model

- char * rowsense_

  Pointer to dense vector of row sense indicators.
- double * rhs_

  Pointer to dense vector of row right-hand side values.
- double * rowrange_

  Pointer to dense vector of slack upper bounds for range constraints (undefined for non-range rows)
- **CoinWarmStartBasis** * ws_

  A pointer to the warmstart information to be used in the hotstarts.
- double * rowActivity_

  also save row and column information for hot starts only used in hotstarts so can be casual
- double * **columnActivity_**
- ClpNodeStuff stuff_

  Stuff for fast dual.
- int numberSOS_

  Number of SOS sets.
- **CoinSet** * setInfo_

  SOS set info.
- ClpSimplex * smallModel_

  Alternate model (hot starts) - but also could be permanent and used for crunch.
- ClpFactorization * factorization_

  factorization for hot starts
- double smallestElementInCut_

  Smallest allowed element in cut.
- double smallestChangeInCut_

  Smallest change in cut.
- double largestAway_

  Largest amount continuous away from bound.
- char * spareArrays_

  Arrays for hot starts.
- **CoinWarmStartBasis** basis_

  Warmstart information to be used in resolves.
- int itlimOrig_

  The original iteration limit before hotstarts started.
- int lastAlgorithm_

  Last algorithm used.
- bool notOwned_

  To say if destructor should delete underlying model.
- **CoinPackedMatrix** * matrixByRow_

  Pointer to row-wise copy of problem matrix coefficients.
- **CoinPackedMatrix** * matrixByRowAtContinuous_

  Pointer to row-wise copy of continuous problem matrix coefficients.
- char * integerInformation_

  Pointer to integer information.
- int * whichRange_

  Pointer to variables for which we want range information The number is in [0] memory is not owned by OsiClp.
- bool fakeMinInSimplex_

  Faking min to get proper dual solution signs in simplex API.
- double * linearObjective_

  Linear objective.
- ClpDataSave saveData_

  To save data in OsiSimplex stuff.
- ClpSolve solveOptions_

Options for initialSolve.

- int cleanupScaling_

  Scaling option When scaling is on it is possible that the scaled problem is feasible but the unscaled is not.

- unsigned int specialOptions_

  Special options 0x80000000 off 0 simple stuff for branch and bound 1 try and keep work regions as much as possible 2 do not use any perturbation 4 allow exit before re-factorization 8 try and re-use factorization if no cuts 16 use standard strong branching rather than clp's 32 Just go to first factorization in fast dual 64 try and tighten bounds in crunch 128 Model will only change in column bounds 256 Clean up model before hot start 512 Give user direct access to Clp regions in getBInvARow etc (i.e., do not unscale, and do not return result in getBInv parameters; you have to know where to look for the answer) 1024 Don't "borrow" model in initialSolve 2048 Don't crunch 4096 quick check for optimality Bits above 8192 give where called from in Cbc At present 0 is normal, 1 doing fast hotstarts, 2 is can do quick check 65536 Keep simple i.e.

- ClpSimplex ∗ baseModel_

  Copy of model when option 131072 set.

- int lastNumberRows_

  Number of rows when last "scaled".

- ClpSimplex ∗ continuousModel_

  Continuous model.

- OsiClpDisasterHandler ∗ disasterHandler_

  Possible disaster handler.

- ClpLinearObjective ∗ fakeObjective_

  Fake objective.

- **CoinDoubleArrayWithLength** rowScale_

  Row scale factors (has inverse at end)

- **CoinDoubleArrayWithLength** columnScale_

  Column scale factors (has inverse at end)

Friends

- void OsiClpSolverInterfaceUnitTest (const std::string &mpsDir, const std::string &netlibDir)

  A function that tests the methods in the OsiClpSolverInterface class.

Protected methods

- void setBasis (const **CoinWarmStartBasis** &basis)

  Sets up working basis as a copy of input and puts in as basis.

- void setBasis ()

  Just puts current basis_ into ClpSimplex model.

- **CoinWarmStartDiff** ∗ getBasisDiff (const unsigned char ∗statusArray) const

  Warm start difference from basis_ to statusArray.

- **CoinWarmStartBasis** ∗ getBasis (const unsigned char ∗statusArray) const

  Warm start from statusArray.

- void deleteScaleFactors ()

  Delete all scale factor stuff and reset option.

- const double ∗ upRange () const

  If doing fast hot start then ranges are computed.

- const double ∗ **downRange** () const
- void passInRanges (int ∗array)

  Pass in range array.

- void setSOSData (int numberSOS, const char ∗type, const int ∗start, const int ∗indices, const double ∗weights=NULL)

  Pass in sos stuff from AMPl.
- void computeLargestAway ()

  Compute largest amount any at continuous away from bound.
- double largestAway () const

  Get largest amount continuous away from bound.
- void setLargestAway (double value)

  Set largest amount continuous away from bound.
- void lexSolve ()

  Sort of lexicographic resolve.
- virtual void applyRowCut (const **OsiRowCut** &rc)

  Apply a row cut (append to constraint matrix).
- virtual void applyColCut (const **OsiColCut** &cc)

  Apply a column cut (adjust one or more bounds).
- void gutsOfDestructor ()

  The real work of a copy constructor (used by copy and assignment)
- void freeCachedResults () const

  Deletes all mutable stuff.
- void freeCachedResults0 () const

  Deletes all mutable stuff for row ranges etc.
- void freeCachedResults1 () const

  Deletes all mutable stuff for matrix etc.
- void extractSenseRhsRange () const

  A method that fills up the rowsense_, rhs_ and rowrange_ arrays.
- void **fillParamMaps** ()
- **CoinWarmStartBasis** getBasis (ClpSimplex ∗model) const

  Warm start.
- void setBasis (const **CoinWarmStartBasis** &basis, ClpSimplex ∗model)

  Sets up working basis as a copy of input.
- void crunch ()

  Crunch down problem a bit.
- void redoScaleFactors (int numberRows, const CoinBigIndex ∗starts, const int ∗indices, const double ∗elements)

  Extend scale factors.

4.94.1 Detailed Description

Clp Solver Interface.

Instantiation of OsiClpSolverInterface for the Model Algorithm.

Definition at line 38 of file OsiClpSolverInterface.hpp.

4.94.2 Member Function Documentation

4.94.2.1 virtual int OsiClpSolverInterface::canDoSimplexInterface ( ) const [virtual]

Simplex API capability.

Returns

- 0 if no simplex API
- 1 if can just do getBInv etc
- 2 if has all OsiSimplex methods

Reimplemented from **OsiSolverInterface**.

4.94.2.2 virtual void OsiClpSolverInterface::enableFactorization ( ) const [virtual]

Enables simplex mode 1 (tableau access)

Tells solver that calls to getBInv etc are about to take place. Underlying code may need mutable as this may be called from CglCut::generateCuts which is const. If that is too horrific then each solver e.g. BCP or CBC will have to do something outside main loop.

Reimplemented from **OsiSolverInterface**.

4.94.2.3 virtual bool OsiClpSolverInterface::basisIsAvailable ( ) const [virtual]

Returns true if a basis is available AND problem is optimal.

This should be used to see if the BInvARow type operations are possible and meaningful.

Reimplemented from **OsiSolverInterface**.

4.94.2.4 virtual void OsiClpSolverInterface::getBasisStatus (int * *cstat*, int * *rstat*) const [virtual]

The following two methods may be replaced by the methods of **OsiSolverInterface** using OsiWarmStartBasis if:

1. OsiWarmStartBasis resize operation is implemented more efficiently and
2. It is ensured that effects on the solver are the same

Returns a basis status of the structural/artificial variables At present as warm start i.e 0 free, 1 basic, 2 upper, 3 lower

NOTE artificials are treated as +1 elements so for <= rhs artificial will be at lower bound if constraint is tight

This means that Clpsimplex flips artificials as it works in terms of row activities

Reimplemented from **OsiSolverInterface**.

4.94.2.5 virtual int OsiClpSolverInterface::setBasisStatus (const int * *cstat*, const int * *rstat*) [virtual]

Set the status of structural/artificial variables and factorize, update solution etc.

NOTE artificials are treated as +1 elements so for <= rhs artificial will be at lower bound if constraint is tight

This means that Clpsimplex flips artificials as it works in terms of row activities Returns 0 if OK, 1 if problem is bad e.g. duplicate elements, too large ...

Reimplemented from **OsiSolverInterface**.

4.94.2.6 virtual void OsiClpSolverInterface::getBInvACol (CoinIndexedVector * *vec*) const [virtual]

Update (i.e.

ftran) the vector passed in. Unscaling is applied after - can't be applied before

4.94.2.7 virtual void OsiClpSolverInterface::enableSimplexInterface (bool *doingPrimal*) [virtual]

Enables simplex mode 2 (individual pivot control)

This method is supposed to ensure that all typical things (like reduced costs, etc.) are updated when individual pivots are executed and can be queried by other methods.

Reimplemented from **OsiSolverInterface**.

4.94.2.8 virtual int OsiClpSolverInterface::pivot (int *colIn,* int *colOut,* int *outStatus*) [virtual]

Perform a pivot by substituting a colIn for colOut in the basis.

The status of the leaving variable is given in statOut. Where 1 is to upper bound, -1 to lower bound Return code is 0 for okay, 1 if inaccuracy forced re-factorization (should be okay) and -1 for singular factorization

Reimplemented from **OsiSolverInterface**.

4.94.2.9 virtual int OsiClpSolverInterface::primalPivotResult (int *colIn,* int *sign,* int & *colOut,* int & *outStatus,* double & *t,* CoinPackedVector * *dx*) [virtual]

Obtain a result of the primal pivot Outputs: colOut – leaving column, outStatus – its status, t – step size, and, if dx!=N↩ ULL, *dx – primal ray direction.

Inputs: colIn – entering column, sign – direction of its change (+/-1). Both for colIn and colOut, artificial variables are index by the negative of the row index minus 1. Return code (for now): 0 – leaving variable found, -1 – everything else? Clearly, more informative set of return values is required Primal and dual solutions are updated

Reimplemented from **OsiSolverInterface**.

4.94.2.10 virtual CoinWarmStart* OsiClpSolverInterface::getEmptyWarmStart ( ) const [virtual]

Get an empty warm start object.

This routine returns an empty **CoinWarmStartBasis** object. Its purpose is to provide a way to give a client a warm start basis object of the appropriate type, which can resized and modified as desired.

Implements **OsiSolverInterface**.

4.94.2.11 virtual bool OsiClpSolverInterface::setWarmStart (const CoinWarmStart * *warmstart*) [virtual]

Set warmstarting information.

Return true/false depending on whether the warmstart information was accepted or not.

Implements **OsiSolverInterface**.

4.94.2.12 virtual CoinWarmStart* OsiClpSolverInterface::getPointerToWarmStart (bool & *mustDelete*) [virtual]

Get warm start information.

Return warm start information for the current state of the solver interface. If there is no valid warm start information, an empty warm start object wil be returned. This does not necessarily create an object - may just point to one. must Delete set true if user should delete returned object. OsiClp version always returns pointer and false.

Reimplemented from **OsiSolverInterface**.

4.94.2.13 virtual const char* OsiClpSolverInterface::getRowSense ( ) const [virtual]

Get pointer to array[getNumRows()] of row constraint senses.

- 'L' <= constraint
- 'E' = constraint
- 'G' >= constraint
- 'R' ranged constraint
- 'N' free constraint

Implements **OsiSolverInterface**.

4.94.2.14 virtual const double* OsiClpSolverInterface::getRightHandSide ( ) const [virtual]

Get pointer to array[getNumRows()] of rows right-hand sides.

- if rowsense()[i] == 'L' then rhs()[i] == rowupper()[i]
- if rowsense()[i] == 'G' then rhs()[i] == rowlower()[i]
- if rowsense()[i] == 'R' then rhs()[i] == rowupper()[i]
- if rowsense()[i] == 'N' then rhs()[i] == 0.0

Implements **OsiSolverInterface**.

4.94.2.15 virtual const double* OsiClpSolverInterface::getRowRange ( ) const [virtual]

Get pointer to array[getNumRows()] of row ranges.

- if rowsense()[i] == 'R' then rowrange()[i] == rowupper()[i] - rowlower()[i]
- if rowsense()[i] != 'R' then rowrange()[i] is undefined

Implements **OsiSolverInterface**.

4.94.2.16 virtual bool OsiClpSolverInterface::isInteger (int *colIndex*) const [virtual]

Return true if column is integer.

Note: This function returns true if the the column is binary or a general integer.

Reimplemented from **OsiSolverInterface**.

4.94.2.17 bool OsiClpSolverInterface::isOptionalInteger (int *colIndex*) const

Return true if column is integer but does not have to be declared as such.

Note: This function returns true if the the column is binary or a general integer.

4.94.2.18 virtual int OsiClpSolverInterface::getIterationCount ( ) const [inline], [virtual]

Get how many iterations it took to solve the problem (whatever "iteration" mean to the solver.

Implements **OsiSolverInterface**.

Definition at line 515 of file OsiClpSolverInterface.hpp.

4.94.2.19 virtual std::vector<double*> OsiClpSolverInterface::getDualRays (int *maxNumRays,* bool *fullRay* = false) const [virtual]

Get as many dual rays as the solver can provide.

(In case of proven primal infeasibility there should be at least one.)

The first getNumRows() ray components will always be associated with the row duals (as returned by getRowPrice()). If `fullRay` is true, the final getNumCols() entries will correspond to the ray components associated with the nonbasic variables. If the full ray is requested and the method cannot provide it, it will throw an exception.

NOTE for implementers of solver interfaces:
The double pointers in the vector should point to arrays of length getNumRows() and they should be allocated via new[].

NOTE for users of solver interfaces:
It is the user's responsibility to free the double pointers in the vector using delete[].

Implements **OsiSolverInterface**.

4.94.2.20 virtual std::vector<double*> OsiClpSolverInterface::getPrimalRays (int *maxNumRays*) const [virtual]

Get as many primal rays as the solver can provide.

(In case of proven dual infeasibility there should be at least one.)

NOTE for implementers of solver interfaces:
The double pointers in the vector should point to arrays of length getNumCols() and they should be allocated via new[].

NOTE for users of solver interfaces:
It is the user's responsibility to free the double pointers in the vector using delete[].

Implements **OsiSolverInterface**.

4.94.2.21 virtual void OsiClpSolverInterface::setColLower (int *elementIndex,* double *elementValue*) [virtual]

Set a single column lower bound
Use -DBL_MAX for -infinity.

Implements **OsiSolverInterface**.

4.94.2.22 virtual void OsiClpSolverInterface::setColUpper (int *elementIndex,* double *elementValue*) [virtual]

Set a single column upper bound
Use DBL_MAX for infinity.

Implements **OsiSolverInterface**.

4.94.2.23 virtual void OsiClpSolverInterface::setColSetBounds (const int ∗ *indexFirst,* const int ∗ *indexLast,* const double ∗ *boundList*) [virtual]

Set the bounds on a number of columns simultaneously
The default implementation just invokes setColLower() and setColUpper() over and over again.

Parameters

index←First,indexLast	pointers to the beginning and after the end of the array of the indices of the variables whose *either* bound changes
boundList	the new lower/upper bound pairs for the variables

Reimplemented from **OsiSolverInterface**.

4.94.2.24 virtual void OsiClpSolverInterface::setRowLower (int *elementIndex,* double *elementValue*) [virtual]

Set a single row lower bound
Use -DBL_MAX for -infinity.

Implements **OsiSolverInterface**.

4.94.2.25 virtual void OsiClpSolverInterface::setRowUpper (int *elementIndex,* double *elementValue*) [virtual]

Set a single row upper bound
Use DBL_MAX for infinity.

Implements **OsiSolverInterface**.

4.94.2.26 virtual void OsiClpSolverInterface::setRowSetBounds (const int ∗ *indexFirst,* const int ∗ *indexLast,* const double ∗ *boundList*) [virtual]

Set the bounds on a number of rows simultaneously
The default implementation just invokes setRowLower() and setRowUpper() over and over again.

Parameters

index←→ First,indexLast	pointers to the beginning and after the end of the array of the indices of the constraints whose *either* bound changes
boundList	the new lower/upper bound pairs for the constraints

Reimplemented from **OsiSolverInterface**.

4.94.2.27 virtual void OsiClpSolverInterface::setRowSetTypes (const int * *indexFirst*, const int * *indexLast*, const char * *senseList*, const double * *rhsList*, const double * *rangeList*) [virtual]

Set the type of a number of rows simultaneously
The default implementation just invokes setRowType() over and over again.

Parameters

index←→ First,indexLast	pointers to the beginning and after the end of the array of the indices of the constraints whose *any* characteristics changes
senseList	the new senses
rhsList	the new right hand sides
rangeList	the new ranges

Reimplemented from **OsiSolverInterface**.

4.94.2.28 virtual void OsiClpSolverInterface::setObjective (const double * *array*) [virtual]

Set the objective coefficients for all columns array [getNumCols()] is an array of values for the objective.

This defaults to a series of set operations and is here for speed.

Reimplemented from **OsiSolverInterface**.

4.94.2.29 virtual void OsiClpSolverInterface::setColLower (const double * *array*) [virtual]

Set the lower bounds for all columns array [getNumCols()] is an array of values for the objective.

This defaults to a series of set operations and is here for speed.

Reimplemented from **OsiSolverInterface**.

4.94.2.30 virtual void OsiClpSolverInterface::setColUpper (const double * *array*) [virtual]

Set the upper bounds for all columns array [getNumCols()] is an array of values for the objective.

This defaults to a series of set operations and is here for speed.

Reimplemented from **OsiSolverInterface**.

4.94.2.31 virtual int OsiClpSolverInterface::findIntegersAndSOS (bool *justCount*) [virtual]

Identify integer variables and SOS and create corresponding objects.

Record integer variables and create an **OsiSimpleInteger** object for each one. All existing **OsiSimpleInteger** objects will be destroyed. If the solver supports SOS then do the same for SOS. If justCount then no objects created and we just store numberIntegers_ Returns number of SOS

Reimplemented from **OsiSolverInterface**.

4.94.2.32 virtual void OsiClpSolverInterface::setColSolution (const double * *colsol*) [virtual]

Set the primal solution column values.

colsol[numcols()] is an array of values of the problem column variables. These values are copied to memory owned by the solver object or the solver. They will be returned as the result of colsol() until changed by another call to setColsol() or by a call to any solver routine. Whether the solver makes use of the solution in any way is solver-dependent.

Implements **OsiSolverInterface**.

4.94.2.33 virtual void OsiClpSolverInterface::setRowPrice (const double * *rowprice*) [virtual]

Set dual solution vector.

rowprice[numrows()] is an array of values of the problem row dual variables. These values are copied to memory owned by the solver object or the solver. They will be returned as the result of rowprice() until changed by another call to setRowprice() or by a call to any solver routine. Whether the solver makes use of the solution in any way is solver-dependent.

Implements **OsiSolverInterface**.

4.94.2.34 virtual void OsiClpSolverInterface::addCol (int *numberElements,* const int * *rows,* const double * *elements,* const double *collb,* const double *colub,* const double *obj*) [virtual]

Add a column (primal variable) to the problem.

Reimplemented from **OsiSolverInterface**.

4.94.2.35 virtual void OsiClpSolverInterface::addRow (const CoinPackedVectorBase & *vec,* const double *rowlb,* const double *rowub,* std::string *name*) [virtual]

Add a named row (constraint) to the problem.

The default implementation adds the row, then changes the name. This can surely be made more efficient within an OsiXXX class.

Reimplemented from **OsiSolverInterface**.

4.94.2.36 virtual void OsiClpSolverInterface::addRow (int *numberElements,* const int * *columns,* const double * *element,* const double *rowlb,* const double *rowub*) [virtual]

Add a row (constraint) to the problem.

Reimplemented from **OsiSolverInterface**.

4.94.2.37 virtual void OsiClpSolverInterface::restoreBaseModel (int *numberRows*) [virtual]

Strip off rows to get to this number of rows.

If solver wants it can restore a copy of "base" (continuous) model here

Reimplemented from **OsiSolverInterface**.

4.94.2.38 virtual void OsiClpSolverInterface::applyRowCuts (int *numberCuts*, const **OsiRowCut** * *cuts*) [virtual]

Apply a collection of row cuts which are all effective.

applyCuts seems to do one at a time which seems inefficient.

Reimplemented from **OsiSolverInterface**.

4.94.2.39 virtual void OsiClpSolverInterface::applyRowCuts (int *numberCuts*, const **OsiRowCut** ** *cuts*) [virtual]

Apply a collection of row cuts which are all effective.

applyCuts seems to do one at a time which seems inefficient. This uses array of pointers

Reimplemented from **OsiSolverInterface**.

4.94.2.40 virtual ApplyCutsReturnCode OsiClpSolverInterface::applyCuts (const **OsiCuts** & *cs*, double *effectivenessLb* = 0.0) [virtual]

Apply a collection of cuts.

Only cuts which have an `effectiveness >= effectivenessLb` are applied.

- ReturnCode.getNumineffective() – number of cuts which were not applied because they had an `effectiveness < effectivenessLb`
- ReturnCode.getNuminconsistent() – number of invalid cuts
- ReturnCode.getNuminconsistentWrtIntegerModel() – number of cuts that are invalid with respect to this integer model
- ReturnCode.getNuminfeasible() – number of cuts that would make this integer model infeasible
- ReturnCode.getNumApplied() – number of integer cuts which were applied to the integer model
- cs.size() == getNumineffective() + getNuminconsistent() + getNuminconsistentWrtIntegerModel() + get↩ Numinfeasible() + getNumApplied()

Reimplemented from **OsiSolverInterface**.

4.94.2.41 virtual void OsiClpSolverInterface::loadProblem (const **CoinPackedMatrix** & *matrix*, const double * *collb*, const double * *colub*, const double * *obj*, const double * *rowlb*, const double * *rowub*) [virtual]

Load in an problem by copying the arguments (the constraints on the rows are given by lower and upper bounds).

If a pointer is NULL then the following values are the default:

- `colub`: all columns have upper bound infinity
- `collb`: all columns have lower bound 0
- `rowub`: all rows have upper bound infinity
- `rowlb`: all rows have lower bound -infinity
- `obj`: all variables have 0 objective coefficient

Implements **OsiSolverInterface**.

4.94.2.42 virtual void OsiClpSolverInterface::assignProblem (CoinPackedMatrix ∗& *matrix,* double ∗& *collb,* double ∗& *colub,* double ∗& *obj,* double ∗& *rowlb,* double ∗& *rowub*) [virtual]

Load in an problem by assuming ownership of the arguments (the constraints on the rows are given by lower and upper bounds).

For default values see the previous method.
WARNING: The arguments passed to this method will be freed using the C++ `delete` and `delete[]` functions.

Implements **OsiSolverInterface**.

4.94.2.43 virtual void OsiClpSolverInterface::loadProblem (const CoinPackedMatrix & *matrix,* const double ∗ *collb,* const double ∗ *colub,* const double ∗ *obj,* const char ∗ *rowsen,* const double ∗ *rowrhs,* const double ∗ *rowrng*) [virtual]

Load in an problem by copying the arguments (the constraints on the rows are given by sense/rhs/range triplets).

If a pointer is NULL then the following values are the default:

- `colub`: all columns have upper bound infinity
- `collb`: all columns have lower bound 0
- `obj`: all variables have 0 objective coefficient
- `rowsen`: all rows are >=
- `rowrhs`: all right hand sides are 0
- `rowrng`: 0 for the ranged rows

Implements **OsiSolverInterface**.

4.94.2.44 virtual void OsiClpSolverInterface::assignProblem (CoinPackedMatrix ∗& *matrix,* double ∗& *collb,* double ∗& *colub,* double ∗& *obj,* char ∗& *rowsen,* double ∗& *rowrhs,* double ∗& *rowrng*) [virtual]

Load in an problem by assuming ownership of the arguments (the constraints on the rows are given by sense/rhs/range triplets).

For default values see the previous method.
WARNING: The arguments passed to this method will be freed using the C++ `delete` and `delete[]` functions.

Implements **OsiSolverInterface**.

4.94.2.45 virtual void OsiClpSolverInterface::loadProblem (const ClpMatrixBase & *matrix,* const double ∗ *collb,* const double ∗ *colub,* const double ∗ *obj,* const double ∗ *rowlb,* const double ∗ *rowub*) [virtual]

Just like the other loadProblem() methods except that the matrix is given as a ClpMatrixBase.

4.94.2.46 virtual void OsiClpSolverInterface::loadProblem (const int *numcols,* const int *numrows,* const CoinBigIndex ∗ *start,* const int ∗ *index,* const double ∗ *value,* const double ∗ *collb,* const double ∗ *colub,* const double ∗ *obj,* const double ∗ *rowlb,* const double ∗ *rowub*) [virtual]

Just like the other loadProblem() methods except that the matrix is given in a standard column major ordered format (without gaps).

Implements **OsiSolverInterface**.

4.94.2.47 virtual void OsiClpSolverInterface::loadProblem (const int *numcols*, const int *numrows*, const CoinBigIndex ∗ *start*, const int ∗ *index*, const double ∗ *value*, const double ∗ *collb*, const double ∗ *colub*, const double ∗ *obj*, const char ∗ *rowsen*, const double ∗ *rowrhs*, const double ∗ *rowrng*) [virtual]

Just like the other loadProblem() methods except that the matrix is given in a standard column major ordered format (without gaps).

Implements **OsiSolverInterface**.

4.94.2.48 virtual void OsiClpSolverInterface::writeMps (const char ∗ *filename*, const char ∗ *extension* = "mps", double *objSense* = 0.0) const [virtual]

Write the problem into an mps file of the given filename.

If objSense is non zero then -1.0 forces the code to write a maximization objective and +1.0 to write a minimization one. If 0.0 then solver can do what it wants

Implements **OsiSolverInterface**.

4.94.2.49 virtual int OsiClpSolverInterface::writeMpsNative (const char ∗ *filename*, const char ∗∗ *rowNames*, const char ∗∗ *columnNames*, int *formatType* = 0, int *numberAcross* = 2, double *objSense* = 0.0) const [virtual]

Write the problem into an mps file of the given filename, names may be null.

formatType is 0 - normal 1 - extra accuracy 2 - IEEE hex (later)

Returns non-zero on I/O error

4.94.2.50 virtual void OsiClpSolverInterface::writeLp (const char ∗ *filename*, const char ∗ *extension* = "lp", double *epsilon* = 1e−5, int *numberAcross* = 10, int *decimals* = 5, double *objSense* = 0.0, bool *useRowNames* = true) const [virtual]

Write the problem into an Lp file of the given filename.

If objSense is non zero then -1.0 forces the code to write a maximization objective and +1.0 to write a minimization one. If 0.0 then solver can do what it wants. This version calls writeLpNative with names

Reimplemented from **OsiSolverInterface**.

4.94.2.51 virtual void OsiClpSolverInterface::writeLp (FILE ∗ *fp*, double *epsilon* = 1e−5, int *numberAcross* = 10, int *decimals* = 5, double *objSense* = 0.0, bool *useRowNames* = true) const [virtual]

Write the problem into the file pointed to by the parameter fp.

Other parameters are similar to those of writeLp() with first parameter filename.

Reimplemented from **OsiSolverInterface**.

4.94.2.52 virtual void OsiClpSolverInterface::replaceMatrixOptional (const CoinPackedMatrix & *matrix*) [virtual]

I (JJF) am getting annoyed because I can't just replace a matrix.

The default behavior of this is do nothing so only use where that would not matter e.g. strengthening a matrix for MIP

Reimplemented from **OsiSolverInterface**.

4.94.2.53 virtual void OsiClpSolverInterface::passInMessageHandler (CoinMessageHandler * handler) [virtual]

Pass in a message handler.

It is the client's responsibility to destroy a message handler installed by this routine; it will not be destroyed when the solver interface is destroyed.

Reimplemented from **OsiSolverInterface**.

4.94.2.54 void OsiClpSolverInterface::setCleanupScaling (int value) [inline]

Set Scaling option When scaling is on it is possible that the scaled problem is feasible but the unscaled is not.

Clp returns a secondary status code to that effect. This option allows for a cleanup. If you use it I would suggest 1. This only affects actions when scaled optimal 0 - no action 1 - clean up using dual if primal infeasibility 2 - clean up using dual if dual infeasibility 3 - clean up using dual if primal or dual infeasibility 11,12,13 - as 1,2,3 but use primal

Definition at line 1059 of file OsiClpSolverInterface.hpp.

4.94.2.55 double OsiClpSolverInterface::smallestElementInCut ( ) const [inline]

Get smallest allowed element in cut.

If smaller than this then ignored

Definition at line 1063 of file OsiClpSolverInterface.hpp.

4.94.2.56 void OsiClpSolverInterface::setSmallestElementInCut (double value) [inline]

Set smallest allowed element in cut.

If smaller than this then ignored

Definition at line 1067 of file OsiClpSolverInterface.hpp.

4.94.2.57 double OsiClpSolverInterface::smallestChangeInCut ( ) const [inline]

Get smallest change in cut.

If (upper-lower)*element < this then element is taken out and cut relaxed. (upper-lower) is taken to be at least 1.0 and this is assumed >= smallestElementInCut_

Definition at line 1075 of file OsiClpSolverInterface.hpp.

4.94.2.58 void OsiClpSolverInterface::setSmallestChangeInCut (double value) [inline]

Set smallest change in cut.

If (upper-lower)*element < this then element is taken out and cut relaxed. (upper-lower) is taken to be at least 1.0 and this is assumed >= smallestElementInCut_

Definition at line 1083 of file OsiClpSolverInterface.hpp.

4.94.2.59 void OsiClpSolverInterface::setupForRepeatedUse (int senseOfAdventure = 0, int printOut = 0)

Set up solver for repeated use by Osi interface.

The normal usage does things like keeping factorization around so can be used. Will also do things like keep scaling and row copy of matrix if matrix does not change.

`senseOfAdventure:`

- 0 - safe stuff as above
- 1 - will take more risks - if it does not work then bug which will be fixed
- 2 - don't bother doing most extreme termination checks e.g. don't bother re-factorizing if less than 20 iterations.
- 3 - Actually safer than 1 (mainly just keeps factorization)

`printOut`

- -1 always skip round common messages instead of doing some work
- 0 skip if normal defaults
- 1 leaves

4.94.2.60 void OsiClpSolverInterface::setSpecialOptionsMutable (unsigned int *value*) const

Set special options in underlying clp solver.

Safe as const because modelPtr_ is mutable.

4.94.2.61 virtual void OsiClpSolverInterface::applyRowCut (const **OsiRowCut** & *rc*) `[protected], [virtual]`

Apply a row cut (append to constraint matrix).

Implements **OsiSolverInterface**.

4.94.2.62 virtual void OsiClpSolverInterface::applyColCut (const **OsiColCut** & *cc*) `[protected], [virtual]`

Apply a column cut (adjust one or more bounds).

Implements **OsiSolverInterface**.

4.94.2.63 **CoinWarmStartBasis** OsiClpSolverInterface::getBasis (**ClpSimplex** ∗ *model*) const `[protected]`

Warm start.

NOTE artificials are treated as +1 elements so for <= rhs artificial will be at lower bound if constraint is tight

This means that Clpsimplex flips artificials as it works in terms of row activities

4.94.2.64 void OsiClpSolverInterface::setBasis (const **CoinWarmStartBasis** & *basis*, **ClpSimplex** ∗ *model*) `[protected]`

Sets up working basis as a copy of input.

NOTE artificials are treated as +1 elements so for <= rhs artificial will be at lower bound if constraint is tight

This means that Clpsimplex flips artificials as it works in terms of row activities

4.94.3 Friends And Related Function Documentation

4.94.3.1 void OsiClpSolverInterfaceUnitTest (const std::string & *mpsDir*, const std::string & *netlibDir*) [friend]

A function that tests the methods in the OsiClpSolverInterface class.

4.94.4 Member Data Documentation

4.94.4.1 CoinWarmStartBasis* OsiClpSolverInterface::ws_ [mutable], [protected]

A pointer to the warmstart information to be used in the hotstarts.

This is NOT efficient and more thought should be given to it...

Definition at line 1283 of file OsiClpSolverInterface.hpp.

4.94.4.2 double OsiClpSolverInterface::smallestElementInCut_ [protected]

Smallest allowed element in cut.

If smaller than this then ignored

Definition at line 1300 of file OsiClpSolverInterface.hpp.

4.94.4.3 double OsiClpSolverInterface::smallestChangeInCut_ [protected]

Smallest change in cut.

If (upper-lower)$*$element $<$ this then element is taken out and cut relaxed.

Definition at line 1304 of file OsiClpSolverInterface.hpp.

4.94.4.4 CoinWarmStartBasis OsiClpSolverInterface::basis_ [protected]

Warmstart information to be used in resolves.

Definition at line 1310 of file OsiClpSolverInterface.hpp.

4.94.4.5 int OsiClpSolverInterface::itlimOrig_ [protected]

The original iteration limit before hotstarts started.

Definition at line 1312 of file OsiClpSolverInterface.hpp.

4.94.4.6 int OsiClpSolverInterface::lastAlgorithm_ [mutable], [protected]

Last algorithm used.

Coded as

- 0 invalid
- 1 primal

- 2 dual
- -911 disaster in the algorithm that was attempted
- 999 current solution no longer optimal due to change in problem or basis

Definition at line 1324 of file OsiClpSolverInterface.hpp.

4.94.4.7 double* OsiClpSolverInterface::linearObjective_ [mutable], [protected]

Linear objective.

Normally a pointer to the linear coefficient array in the clp objective. An independent copy when fakeMinInSimplex_ is true, because we need something permanent to point to when getObjCoefficients is called.

Definition at line 1356 of file OsiClpSolverInterface.hpp.

4.94.4.8 int OsiClpSolverInterface::cleanupScaling_ [protected]

Scaling option When scaling is on it is possible that the scaled problem is feasible but the unscaled is not.

Clp returns a secondary status code to that effect. This option allows for a cleanup. If you use it I would suggest 1. This only affects actions when scaled optimal 0 - no action 1 - clean up using dual if primal infeasibility 2 - clean up using dual if dual infeasibility 3 - clean up using dual if primal or dual infeasibility 11,12,13 - as 1,2,3 but use primal

Definition at line 1374 of file OsiClpSolverInterface.hpp.

4.94.4.9 unsigned int OsiClpSolverInterface::specialOptions_ [mutable], [protected]

Special options 0x80000000 off 0 simple stuff for branch and bound 1 try and keep work regions as much as possible 2 do not use any perturbation 4 allow exit before re-factorization 8 try and re-use factorization if no cuts 16 use standard strong branching rather than clp's 32 Just go to first factorization in fast dual 64 try and tighten bounds in crunch 128 Model will only change in column bounds 256 Clean up model before hot start 512 Give user direct access to Clp regions in getBInvARow etc (i.e., do not unscale, and do not return result in getBInv parameters; you have to know where to look for the answer) 1024 Don't "borrow" model in initialSolve 2048 Don't crunch 4096 quick check for optimality Bits above 8192 give where called from in Cbc At present 0 is normal, 1 doing fast hotstarts, 2 is can do quick check 65536 Keep simple i.e.

no crunch etc 131072 Try and keep scaling factors around 262144 Don't try and tighten bounds (funny global cuts) 524288 Fake objective and 0-1 1048576 Don't recompute ray after crunch 2097152

Definition at line 1402 of file OsiClpSolverInterface.hpp.

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

- OsiClpSolverInterface.hpp

4.95 Outfo Struct Reference

******** DATA to be moved into protected section of ClpInterior

`#include <ClpInterior.hpp>`

4.95.1 Detailed Description

******** DATA to be moved into protected section of ClpInterior

Definition at line 35 of file ClpInterior.hpp.

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

- ClpInterior.hpp

4.96 ClpSimplexOther::parametricsData Struct Reference

4.96.1 Detailed Description

Definition at line 107 of file ClpSimplexOther.hpp.

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

- ClpSimplexOther.hpp

4.97 AbcSimplexPrimal::pivotStruct Struct Reference

4.97.1 Detailed Description

Definition at line 210 of file AbcSimplexPrimal.hpp.

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

- AbcSimplexPrimal.hpp

4.98 scatterStruct Struct Reference

4.98.1 Detailed Description

Definition at line 534 of file CoinAbcHelperFunctions.hpp.

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

- CoinAbcHelperFunctions.hpp

File Documentation

Index