

Clp
trunk

Generated by Doxygen 1.8.5

Mon Oct 21 2013 19:00:45

Contents

1	Hierarchical Index	1
1.1	Class Hierarchy	1
2	Class Index	8
2.1	Class List	8
3	File Index	12
3.1	File List	13
4	Class Documentation	15
4.1	AbcDualRowDantzig Class Reference	15
4.1.1	Detailed Description	16
4.1.2	Constructor & Destructor Documentation	17
4.1.3	Member Function Documentation	17
4.2	AbcDualRowPivot Class Reference	18
4.2.1	Detailed Description	19
4.2.2	Constructor & Destructor Documentation	20
4.2.3	Member Function Documentation	20
4.2.4	Member Data Documentation	22
4.3	AbcDualRowSteepest Class Reference	22
4.3.1	Detailed Description	23
4.3.2	Member Enumeration Documentation	24
4.3.3	Constructor & Destructor Documentation	24
4.3.4	Member Function Documentation	24
4.4	AbcMatrix Class Reference	26
4.4.1	Detailed Description	31
4.4.2	Constructor & Destructor Documentation	31
4.4.3	Member Function Documentation	31
4.4.4	Member Data Documentation	39
4.5	AbcMatrix2 Class Reference	41
4.5.1	Detailed Description	42
4.5.2	Constructor & Destructor Documentation	42
4.5.3	Member Function Documentation	42
4.5.4	Member Data Documentation	43
4.6	AbcMatrix3 Class Reference	43
4.6.1	Detailed Description	44
4.6.2	Constructor & Destructor Documentation	45

4.6.3	Member Function Documentation	45
4.6.4	Member Data Documentation	45
4.7	AbcNonLinearCost Class Reference	46
4.7.1	Detailed Description	48
4.7.2	Constructor & Destructor Documentation	48
4.7.3	Member Function Documentation	48
4.8	AbcPrimalColumnDantzig Class Reference	51
4.8.1	Detailed Description	52
4.8.2	Constructor & Destructor Documentation	52
4.8.3	Member Function Documentation	52
4.9	AbcPrimalColumnPivot Class Reference	53
4.9.1	Detailed Description	54
4.9.2	Constructor & Destructor Documentation	54
4.9.3	Member Function Documentation	54
4.9.4	Member Data Documentation	56
4.10	AbcPrimalColumnSteepest Class Reference	57
4.10.1	Detailed Description	58
4.10.2	Member Enumeration Documentation	58
4.10.3	Constructor & Destructor Documentation	59
4.10.4	Member Function Documentation	59
4.11	AbcSimplex Class Reference	61
4.11.1	Detailed Description	73
4.11.2	Member Enumeration Documentation	74
4.11.3	Constructor & Destructor Documentation	74
4.11.4	Member Function Documentation	75
4.11.5	Friends And Related Function Documentation	92
4.11.6	Member Data Documentation	93
4.12	AbcSimplexDual Class Reference	102
4.12.1	Detailed Description	104
4.12.2	Member Function Documentation	104
4.13	AbcSimplexFactorization Class Reference	109
4.13.1	Detailed Description	112
4.13.2	Constructor & Destructor Documentation	112
4.13.3	Member Function Documentation	112
4.14	AbcSimplexPrimal Class Reference	118
4.14.1	Detailed Description	120
4.14.2	Member Function Documentation	120

4.15	AbcTolerancesEtc Class Reference	123
4.15.1	Detailed Description	124
4.15.2	Constructor & Destructor Documentation	124
4.15.3	Member Function Documentation	125
4.15.4	Member Data Documentation	125
4.16	AbcWarmStart Class Reference	126
4.16.1	Detailed Description	128
4.16.2	Constructor & Destructor Documentation	128
4.16.3	Member Function Documentation	129
4.16.4	Member Data Documentation	130
4.17	AbcWarmStartOrganizer Class Reference	131
4.17.1	Detailed Description	132
4.17.2	Constructor & Destructor Documentation	132
4.17.3	Member Function Documentation	133
4.17.4	Member Data Documentation	133
4.18	ampl_info Struct Reference	134
4.18.1	Detailed Description	134
4.18.2	Member Data Documentation	135
4.19	blockStruct Struct Reference	137
4.19.1	Detailed Description	138
4.19.2	Member Data Documentation	138
4.20	blockStruct3 Struct Reference	138
4.20.1	Detailed Description	138
4.20.2	Member Data Documentation	138
4.21	ClpNode::branchState Struct Reference	139
4.21.1	Detailed Description	139
4.21.2	Member Data Documentation	139
4.22	CbcOrClpParam Class Reference	140
4.22.1	Detailed Description	142
4.22.2	Constructor & Destructor Documentation	142
4.22.3	Member Function Documentation	143
4.23	ClpCholeskyBase Class Reference	147
4.23.1	Detailed Description	150
4.23.2	Constructor & Destructor Documentation	150
4.23.3	Member Function Documentation	150
4.23.4	Member Data Documentation	154
4.24	ClpCholeskyDense Class Reference	157

4.24.1 Detailed Description	158
4.24.2 Constructor & Destructor Documentation	158
4.24.3 Member Function Documentation	158
4.25 ClpCholeskyDenseC Struct Reference	160
4.25.1 Detailed Description	160
4.25.2 Member Data Documentation	160
4.26 ClpCholeskyMumps Class Reference	161
4.26.1 Detailed Description	162
4.26.2 Constructor & Destructor Documentation	162
4.26.3 Member Function Documentation	162
4.27 ClpCholeskyTaucs Class Reference	163
4.27.1 Detailed Description	163
4.27.2 Constructor & Destructor Documentation	164
4.27.3 Member Function Documentation	164
4.28 ClpCholeskyUfl Class Reference	165
4.28.1 Detailed Description	165
4.28.2 Constructor & Destructor Documentation	166
4.28.3 Member Function Documentation	166
4.29 ClpCholeskyWssmp Class Reference	167
4.29.1 Detailed Description	167
4.29.2 Constructor & Destructor Documentation	167
4.29.3 Member Function Documentation	168
4.30 ClpCholeskyWssmpKKT Class Reference	168
4.30.1 Detailed Description	169
4.30.2 Constructor & Destructor Documentation	169
4.30.3 Member Function Documentation	170
4.31 ClpConstraint Class Reference	170
4.31.1 Detailed Description	172
4.31.2 Constructor & Destructor Documentation	172
4.31.3 Member Function Documentation	172
4.31.4 Member Data Documentation	174
4.32 ClpConstraintLinear Class Reference	174
4.32.1 Detailed Description	176
4.32.2 Constructor & Destructor Documentation	176
4.32.3 Member Function Documentation	176
4.33 ClpConstraintQuadratic Class Reference	177
4.33.1 Detailed Description	179

4.33.2	Constructor & Destructor Documentation	179
4.33.3	Member Function Documentation	179
4.34	ClpDataSave Class Reference	181
4.34.1	Detailed Description	181
4.34.2	Constructor & Destructor Documentation	181
4.34.3	Member Function Documentation	182
4.34.4	Member Data Documentation	182
4.35	ClpDisasterHandler Class Reference	183
4.35.1	Detailed Description	184
4.35.2	Constructor & Destructor Documentation	184
4.35.3	Member Function Documentation	184
4.35.4	Member Data Documentation	185
4.36	ClpDualRowDantzig Class Reference	185
4.36.1	Detailed Description	186
4.36.2	Constructor & Destructor Documentation	186
4.36.3	Member Function Documentation	186
4.37	ClpDualRowPivot Class Reference	187
4.37.1	Detailed Description	188
4.37.2	Constructor & Destructor Documentation	188
4.37.3	Member Function Documentation	189
4.37.4	Member Data Documentation	190
4.38	ClpDualRowSteepest Class Reference	190
4.38.1	Detailed Description	192
4.38.2	Member Enumeration Documentation	192
4.38.3	Constructor & Destructor Documentation	192
4.38.4	Member Function Documentation	192
4.39	ClpDummyMatrix Class Reference	194
4.39.1	Detailed Description	196
4.39.2	Constructor & Destructor Documentation	196
4.39.3	Member Function Documentation	197
4.39.4	Member Data Documentation	200
4.40	ClpDynamicExampleMatrix Class Reference	200
4.40.1	Detailed Description	202
4.40.2	Constructor & Destructor Documentation	203
4.40.3	Member Function Documentation	203
4.40.4	Member Data Documentation	205
4.41	ClpDynamicMatrix Class Reference	206

4.41.1	Detailed Description	211
4.41.2	Member Enumeration Documentation	211
4.41.3	Constructor & Destructor Documentation	211
4.41.4	Member Function Documentation	212
4.41.5	Member Data Documentation	217
4.42	ClpEventHandler Class Reference	221
4.42.1	Detailed Description	222
4.42.2	Member Enumeration Documentation	223
4.42.3	Constructor & Destructor Documentation	224
4.42.4	Member Function Documentation	224
4.42.5	Member Data Documentation	224
4.43	ClpFactorization Class Reference	225
4.43.1	Detailed Description	228
4.43.2	Constructor & Destructor Documentation	228
4.43.3	Member Function Documentation	228
4.44	ClpGubDynamicMatrix Class Reference	234
4.44.1	Detailed Description	237
4.44.2	Member Enumeration Documentation	237
4.44.3	Constructor & Destructor Documentation	237
4.44.4	Member Function Documentation	238
4.44.5	Member Data Documentation	241
4.45	ClpGubMatrix Class Reference	243
4.45.1	Detailed Description	247
4.45.2	Constructor & Destructor Documentation	247
4.45.3	Member Function Documentation	248
4.45.4	Member Data Documentation	253
4.46	ClpHashValue Class Reference	256
4.46.1	Detailed Description	257
4.46.2	Constructor & Destructor Documentation	257
4.46.3	Member Function Documentation	257
4.46.4	Member Data Documentation	258
4.47	ClpInterior Class Reference	258
4.47.1	Detailed Description	265
4.47.2	Constructor & Destructor Documentation	265
4.47.3	Member Function Documentation	266
4.47.4	Friends And Related Function Documentation	272
4.47.5	Member Data Documentation	272

4.48 ClpLinearObjective Class Reference	279
4.48.1 Detailed Description	280
4.48.2 Constructor & Destructor Documentation	281
4.48.3 Member Function Documentation	281
4.49 ClpLsqp Class Reference	282
4.49.1 Detailed Description	283
4.49.2 Constructor & Destructor Documentation	284
4.49.3 Member Function Documentation	284
4.49.4 Member Data Documentation	285
4.50 ClpMatrixBase Class Reference	286
4.50.1 Detailed Description	290
4.50.2 Constructor & Destructor Documentation	291
4.50.3 Member Function Documentation	291
4.50.4 Member Data Documentation	301
4.51 ClpMessage Class Reference	302
4.51.1 Detailed Description	303
4.51.2 Constructor & Destructor Documentation	303
4.52 ClpModel Class Reference	303
4.52.1 Detailed Description	315
4.52.2 Constructor & Destructor Documentation	315
4.52.3 Member Function Documentation	315
4.52.4 Member Data Documentation	337
4.53 ClpNetworkBasis Class Reference	343
4.53.1 Detailed Description	344
4.53.2 Constructor & Destructor Documentation	344
4.53.3 Member Function Documentation	344
4.54 ClpNetworkMatrix Class Reference	345
4.54.1 Detailed Description	347
4.54.2 Constructor & Destructor Documentation	348
4.54.3 Member Function Documentation	348
4.54.4 Member Data Documentation	352
4.55 ClpNode Class Reference	353
4.55.1 Detailed Description	355
4.55.2 Constructor & Destructor Documentation	356
4.55.3 Member Function Documentation	356
4.55.4 Member Data Documentation	358
4.56 ClpNodeStuff Class Reference	360

4.56.1	Detailed Description	362
4.56.2	Constructor & Destructor Documentation	362
4.56.3	Member Function Documentation	362
4.56.4	Member Data Documentation	363
4.57	ClpNonLinearCost Class Reference	366
4.57.1	Detailed Description	368
4.57.2	Constructor & Destructor Documentation	368
4.57.3	Member Function Documentation	368
4.58	ClpObjective Class Reference	371
4.58.1	Detailed Description	373
4.58.2	Constructor & Destructor Documentation	373
4.58.3	Member Function Documentation	373
4.58.4	Member Data Documentation	375
4.59	ClpPackedMatrix Class Reference	375
4.59.1	Detailed Description	380
4.59.2	Constructor & Destructor Documentation	380
4.59.3	Member Function Documentation	380
4.59.4	Member Data Documentation	389
4.60	ClpPackedMatrix2 Class Reference	389
4.60.1	Detailed Description	390
4.60.2	Constructor & Destructor Documentation	390
4.60.3	Member Function Documentation	391
4.60.4	Member Data Documentation	391
4.61	ClpPackedMatrix3 Class Reference	392
4.61.1	Detailed Description	393
4.61.2	Constructor & Destructor Documentation	393
4.61.3	Member Function Documentation	393
4.61.4	Member Data Documentation	394
4.62	ClpPdco Class Reference	395
4.62.1	Detailed Description	395
4.62.2	Member Function Documentation	396
4.63	ClpPdcoBase Class Reference	396
4.63.1	Detailed Description	397
4.63.2	Constructor & Destructor Documentation	398
4.63.3	Member Function Documentation	398
4.63.4	Member Data Documentation	399
4.64	ClpPlusMinusOneMatrix Class Reference	399

4.64.1 Detailed Description	402
4.64.2 Constructor & Destructor Documentation	402
4.64.3 Member Function Documentation	403
4.64.4 Member Data Documentation	408
4.65 ClpPredictorCorrector Class Reference	409
4.65.1 Detailed Description	410
4.65.2 Member Function Documentation	411
4.66 ClpPresolve Class Reference	412
4.66.1 Detailed Description	414
4.66.2 Constructor & Destructor Documentation	414
4.66.3 Member Function Documentation	414
4.67 ClpPrimalColumnDantzig Class Reference	418
4.67.1 Detailed Description	419
4.67.2 Constructor & Destructor Documentation	419
4.67.3 Member Function Documentation	419
4.68 ClpPrimalColumnPivot Class Reference	420
4.68.1 Detailed Description	421
4.68.2 Constructor & Destructor Documentation	422
4.68.3 Member Function Documentation	422
4.68.4 Member Data Documentation	424
4.69 ClpPrimalColumnSteepest Class Reference	424
4.69.1 Detailed Description	426
4.69.2 Member Enumeration Documentation	426
4.69.3 Constructor & Destructor Documentation	427
4.69.4 Member Function Documentation	427
4.70 ClpPrimalQuadraticDantzig Class Reference	430
4.70.1 Detailed Description	430
4.70.2 Constructor & Destructor Documentation	431
4.70.3 Member Function Documentation	431
4.71 ClpQuadraticObjective Class Reference	431
4.71.1 Detailed Description	433
4.71.2 Constructor & Destructor Documentation	433
4.71.3 Member Function Documentation	434
4.72 ClpSimplex Class Reference	436
4.72.1 Detailed Description	451
4.72.2 Member Enumeration Documentation	452
4.72.3 Constructor & Destructor Documentation	452

4.72.4	Member Function Documentation	453
4.72.5	Friends And Related Function Documentation	479
4.72.6	Member Data Documentation	479
4.73	ClpSimplexDual Class Reference	489
4.73.1	Detailed Description	490
4.73.2	Member Function Documentation	490
4.74	ClpSimplexNonlinear Class Reference	494
4.74.1	Detailed Description	495
4.74.2	Member Function Documentation	495
4.75	ClpSimplexOther Class Reference	497
4.75.1	Detailed Description	498
4.75.2	Member Function Documentation	498
4.76	ClpSimplexPrimal Class Reference	501
4.76.1	Detailed Description	502
4.76.2	Member Function Documentation	502
4.77	ClpSimplexProgress Class Reference	505
4.77.1	Detailed Description	507
4.77.2	Constructor & Destructor Documentation	507
4.77.3	Member Function Documentation	507
4.77.4	Member Data Documentation	509
4.78	ClpSolve Class Reference	511
4.78.1	Detailed Description	512
4.78.2	Member Enumeration Documentation	513
4.78.3	Constructor & Destructor Documentation	513
4.78.4	Member Function Documentation	513
4.79	ClpTrustedData Struct Reference	517
4.79.1	Detailed Description	517
4.79.2	Member Data Documentation	517
4.80	CoinAbcAnyFactorization Class Reference	517
4.80.1	Detailed Description	521
4.80.2	Constructor & Destructor Documentation	521
4.80.3	Member Function Documentation	522
4.80.4	Member Data Documentation	528
4.81	CoinAbcDenseFactorization Class Reference	530
4.81.1	Detailed Description	532
4.81.2	Constructor & Destructor Documentation	533
4.81.3	Member Function Documentation	533

4.81.4	Friends And Related Function Documentation	536
4.81.5	Member Data Documentation	536
4.82	CoinAbcStack Struct Reference	537
4.82.1	Detailed Description	537
4.82.2	Member Data Documentation	537
4.83	CoinAbcStatistics Struct Reference	537
4.83.1	Detailed Description	537
4.83.2	Member Data Documentation	538
4.84	CoinAbcThreadInfo Struct Reference	538
4.84.1	Detailed Description	538
4.84.2	Member Data Documentation	538
4.85	CoinAbcTypeFactorization Class Reference	539
4.85.1	Detailed Description	550
4.85.2	Constructor & Destructor Documentation	551
4.85.3	Member Function Documentation	551
4.85.4	Friends And Related Function Documentation	566
4.85.5	Member Data Documentation	566
4.86	ClpHashValue::CoinHashLink Struct Reference	578
4.86.1	Detailed Description	578
4.86.2	Member Data Documentation	578
4.87	dualColumnResult Struct Reference	578
4.87.1	Detailed Description	579
4.87.2	Member Data Documentation	579
4.88	Idiot Class Reference	580
4.88.1	Detailed Description	582
4.88.2	Constructor & Destructor Documentation	582
4.88.3	Member Function Documentation	582
4.89	IdiotResult Struct Reference	585
4.89.1	Detailed Description	586
4.89.2	Member Data Documentation	586
4.90	Info Struct Reference	586
4.90.1	Detailed Description	587
4.90.2	Member Data Documentation	587
4.91	MyEventHandler Class Reference	587
4.91.1	Detailed Description	588
4.91.2	Constructor & Destructor Documentation	588
4.91.3	Member Function Documentation	588

4.92	MyMessageHandler Class Reference	589
4.92.1	Detailed Description	590
4.92.2	Constructor & Destructor Documentation	590
4.92.3	Member Function Documentation	590
4.92.4	Member Data Documentation	591
4.93	Options Struct Reference	591
4.93.1	Detailed Description	592
4.93.2	Member Data Documentation	592
4.94	OsiClpDisasterHandler Class Reference	593
4.94.1	Detailed Description	594
4.94.2	Constructor & Destructor Documentation	594
4.94.3	Member Function Documentation	595
4.94.4	Member Data Documentation	596
4.95	OsiClpSolverInterface Class Reference	596
4.95.1	Detailed Description	608
4.95.2	Constructor & Destructor Documentation	608
4.95.3	Member Function Documentation	609
4.95.4	Friends And Related Function Documentation	629
4.95.5	Member Data Documentation	629
4.96	Outfo Struct Reference	633
4.96.1	Detailed Description	634
4.96.2	Member Data Documentation	634
4.97	ClpSimplexOther::parametricsData Struct Reference	634
4.97.1	Detailed Description	635
4.97.2	Member Data Documentation	635
4.98	AbcSimplexPrimal::pivotStruct Struct Reference	636
4.98.1	Detailed Description	637
4.98.2	Member Data Documentation	637
4.99	scatterStruct Struct Reference	638
4.99.1	Detailed Description	638
4.99.2	Member Data Documentation	638
5	File Documentation	639
5.1	src/AbcCommon.hpp File Reference	639
5.2	src/AbcDualRowDantzig.hpp File Reference	639
5.3	src/AbcDualRowPivot.hpp File Reference	639
5.4	src/AbcDualRowSteepest.hpp File Reference	639

5.4.1	Macro Definition Documentation	639
5.5	src/AbcMatrix.hpp File Reference	640
5.5.1	Macro Definition Documentation	640
5.6	src/AbcNonLinearCost.hpp File Reference	640
5.6.1	Macro Definition Documentation	641
5.6.2	Function Documentation	641
5.7	src/AbcPrimalColumnDantzig.hpp File Reference	642
5.8	src/AbcPrimalColumnPivot.hpp File Reference	642
5.8.1	Macro Definition Documentation	642
5.9	src/AbcPrimalColumnSteepest.hpp File Reference	643
5.10	src/AbcSimplex.hpp File Reference	643
5.10.1	Macro Definition Documentation	644
5.10.2	Function Documentation	646
5.11	src/AbcSimplexDual.hpp File Reference	646
5.12	src/AbcSimplexFactorization.hpp File Reference	647
5.13	src/AbcSimplexPrimal.hpp File Reference	647
5.14	src/AbcWarmStart.hpp File Reference	647
5.14.1	Macro Definition Documentation	648
5.15	src/CbcOrClpParam.hpp File Reference	648
5.15.1	Macro Definition Documentation	651
5.15.2	Enumeration Type Documentation	651
5.15.3	Function Documentation	656
5.16	src/Clp_ampl.h File Reference	657
5.16.1	Function Documentation	657
5.17	src/Clp_C_Interface.h File Reference	657
5.17.1	Typedef Documentation	664
5.17.2	Function Documentation	664
5.18	src/ClpCholeskyBase.hpp File Reference	674
5.18.1	Macro Definition Documentation	675
5.18.2	Typedef Documentation	675
5.19	src/ClpCholeskyDense.hpp File Reference	675
5.19.1	Function Documentation	676
5.20	src/ClpCholeskyMumps.hpp File Reference	677
5.20.1	Typedef Documentation	677
5.21	src/ClpCholeskyTaucs.hpp File Reference	677
5.22	src/ClpCholeskyUfl.hpp File Reference	677
5.22.1	Typedef Documentation	678

5.23	src/ClpCholeskyWssmp.hpp File Reference	678
5.24	src/ClpCholeskyWssmpKKT.hpp File Reference	678
5.25	src/ClpConfig.h File Reference	678
5.26	src/ClpConstraint.hpp File Reference	678
5.27	src/ClpConstraintLinear.hpp File Reference	679
5.28	src/ClpConstraintQuadratic.hpp File Reference	679
5.29	src/ClpDualRowDantzig.hpp File Reference	679
5.30	src/ClpDualRowPivot.hpp File Reference	679
5.31	src/ClpDualRowSteepest.hpp File Reference	679
5.32	src/ClpDummyMatrix.hpp File Reference	680
5.33	src/ClpDynamicExampleMatrix.hpp File Reference	680
5.34	src/ClpDynamicMatrix.hpp File Reference	680
5.35	src/ClpEventHandler.hpp File Reference	680
5.36	src/ClpFactorization.hpp File Reference	681
5.36.1	Macro Definition Documentation	681
5.37	src/ClpGubDynamicMatrix.hpp File Reference	681
5.38	src/ClpGubMatrix.hpp File Reference	681
5.39	src/ClpHelperFunctions.hpp File Reference	682
5.39.1	Macro Definition Documentation	682
5.39.2	Function Documentation	682
5.40	src/ClpInterior.hpp File Reference	683
5.40.1	Macro Definition Documentation	683
5.40.2	Function Documentation	684
5.41	src/ClpLinearObjective.hpp File Reference	684
5.42	src/ClpLsqrr.hpp File Reference	684
5.43	src/ClpMatrixBase.hpp File Reference	684
5.43.1	Macro Definition Documentation	685
5.44	src/ClpMessage.hpp File Reference	685
5.44.1	Enumeration Type Documentation	686
5.45	src/ClpModel.hpp File Reference	688
5.45.1	Macro Definition Documentation	689
5.46	src/ClpNetworkBasis.hpp File Reference	690
5.46.1	Macro Definition Documentation	691
5.47	src/ClpNetworkMatrix.hpp File Reference	691
5.48	src/ClpNode.hpp File Reference	691
5.49	src/ClpNonLinearCost.hpp File Reference	691
5.49.1	Macro Definition Documentation	692

5.49.2	Function Documentation	693
5.50	src/ClpObjective.hpp File Reference	693
5.51	src/ClpPackedMatrix.hpp File Reference	693
5.51.1	Macro Definition Documentation	694
5.52	src/ClpParameters.hpp File Reference	694
5.52.1	Enumeration Type Documentation	694
5.52.2	Function Documentation	695
5.53	src/ClpPdco.hpp File Reference	696
5.54	src/ClpPdcoBase.hpp File Reference	696
5.55	src/ClpPlusMinusOneMatrix.hpp File Reference	696
5.56	src/ClpPredictorCorrector.hpp File Reference	696
5.57	src/ClpPresolve.hpp File Reference	697
5.58	src/ClpPrimalColumnDantzig.hpp File Reference	697
5.59	src/ClpPrimalColumnPivot.hpp File Reference	697
5.59.1	Macro Definition Documentation	697
5.60	src/ClpPrimalColumnSteepest.hpp File Reference	697
5.61	src/ClpPrimalQuadraticDantzig.hpp File Reference	698
5.62	src/ClpQuadraticObjective.hpp File Reference	698
5.63	src/ClpSimplex.hpp File Reference	698
5.63.1	Macro Definition Documentation	699
5.63.2	Function Documentation	699
5.64	src/ClpSimplexDual.hpp File Reference	699
5.65	src/ClpSimplexNonlinear.hpp File Reference	699
5.66	src/ClpSimplexOther.hpp File Reference	700
5.67	src/ClpSimplexPrimal.hpp File Reference	700
5.68	src/ClpSolve.hpp File Reference	700
5.68.1	Macro Definition Documentation	701
5.69	src/CoinAbcBaseFactorization.hpp File Reference	701
5.69.1	Macro Definition Documentation	701
5.70	src/CoinAbcCommon.hpp File Reference	702
5.70.1	Macro Definition Documentation	703
5.70.2	Typedef Documentation	705
5.70.3	Function Documentation	706
5.71	src/CoinAbcCommonFactorization.hpp File Reference	706
5.71.1	Macro Definition Documentation	707
5.71.2	Function Documentation	708
5.72	src/CoinAbcDenseFactorization.hpp File Reference	709

5.72.1	Macro Definition Documentation	709
5.73	src/CoinAbcFactorization.hpp File Reference	709
5.73.1	Macro Definition Documentation	710
5.74	src/CoinAbcHelperFunctions.hpp File Reference	710
5.74.1	Macro Definition Documentation	714
5.74.2	Typedef Documentation	715
5.74.3	Function Documentation	715
5.75	src/config_clp_default.h File Reference	719
5.75.1	Macro Definition Documentation	719
5.76	src/config_default.h File Reference	719
5.76.1	Macro Definition Documentation	720
5.77	src/Idiot.hpp File Reference	720
5.77.1	Macro Definition Documentation	720
5.78	src/MyEventHandler.hpp File Reference	720
5.79	src/MyMessageHandler.hpp File Reference	721
5.79.1	Typedef Documentation	721
5.80	src/OsiClp/OsiClpSolverInterface.hpp File Reference	721
5.80.1	Function Documentation	722
5.80.2	Variable Documentation	722

Index

723

1 Hierarchical Index

1.1 Class Hierarchy

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

AbcDualRowPivot	18
AbcDualRowDantzig	15
AbcDualRowSteepest	22
AbcMatrix	26
AbcMatrix2	41
AbcMatrix3	43
AbcNonLinearCost	46
AbcPrimalColumnPivot	53

AbcPrimalColumnDantzig	51
AbcPrimalColumnSteepest	57
AbcSimplexFactorization	109
AbcTolerancesEtc	123
AbcWarmStartOrganizer	131
<code>std::allocator< T ></code>	
ampl_info	134
<code>std::array< T ></code>	
<code>std::auto_ptr< T ></code>	
<code>std::basic_string< Char ></code>	
<code>std::string</code>	
<code>std::wstring</code>	
<code>std::basic_string< char ></code>	
<code>std::basic_string< wchar_t ></code>	
<code>std::bitset< Bits ></code>	
blockStruct	137
blockStruct3	138
ClpNode::branchState	139
CbcOrClpParam	140
ClpCholeskyBase	147
ClpCholeskyDense	157
ClpCholeskyMumps	161
ClpCholeskyTaucs	163
ClpCholeskyUfl	165
ClpCholeskyWssmp	167
ClpCholeskyWssmpKKT	168
ClpCholeskyDenseC	160
ClpConstraint	170
ClpConstraintLinear	174
ClpConstraintQuadratic	177
ClpDataSave	181
ClpDisasterHandler	183
OsiClpDisasterHandler	593
ClpDualRowPivot	187

ClpDualRowDantzig	185
ClpDualRowSteepest	190
ClpEventHandler	221
MyEventHandler	587
ClpFactorization	225
ClpHashValue	256
ClpLsq	282
ClpMatrixBase	286
ClpDummyMatrix	194
ClpNetworkMatrix	345
ClpPackedMatrix	375
ClpDynamicMatrix	206
ClpDynamicExampleMatrix	200
ClpGubMatrix	243
ClpGubDynamicMatrix	234
ClpPlusMinusOneMatrix	399
ClpModel	303
ClpInterior	258
ClpPdco	395
ClpPredictorCorrector	409
ClpSimplex	436
AbcSimplex	61
AbcSimplexDual	102
AbcSimplexPrimal	118
ClpSimplexDual	489
ClpSimplexOther	497
ClpSimplexPrimal	501
ClpSimplexNonlinear	494
ClpNetworkBasis	343
ClpNode	353

ClpNodeStuff	360
ClpNonLinearCost	366
ClpObjective	371
ClpLinearObjective	279
ClpQuadraticObjective	431
ClpPackedMatrix2	389
ClpPackedMatrix3	392
ClpPdcoBase	396
ClpPresolve	412
ClpPrimalColumnPivot	420
ClpPrimalColumnDantzig	418
ClpPrimalColumnSteepest	424
ClpPrimalQuadraticDantzig	430
ClpSimplexProgress	505
ClpSolve	511
ClpTrustedData	517
CoinAbcAnyFactorization	517
CoinAbcDenseFactorization	530
CoinAbcTypeFactorization	539
CoinAbcStack	537
CoinAbcStatistics	537
CoinAbcThreadInfo	538
ClpHashValue::CoinHashLink	578
CoinMessageHandler	
MyMessageHandler	589
CoinMessages	
ClpMessage	302
CoinWarmStartBasis	
AbcWarmStart	126
std::complex	
std::deque< T >::const_iterator	
std::list< T >::const_iterator	
std::forward_list< T >::const_iterator	

```

std::map< K, T >::const_iterator
std::wstring::const_iterator
std::unordered_map< K, T >::const_iterator
std::basic_string< Char >::const_iterator
std::multimap< K, T >::const_iterator
std::unordered_multimap< K, T >::const_iterator
std::set< K >::const_iterator
std::string::const_iterator
std::unordered_set< K >::const_iterator
std::multiset< K >::const_iterator
std::unordered_multiset< K >::const_iterator
std::vector< T >::const_iterator
std::list< T >::const_reverse_iterator
std::forward_list< T >::const_reverse_iterator
std::map< K, T >::const_reverse_iterator
std::deque< T >::const_reverse_iterator
std::string::const_reverse_iterator
std::wstring::const_reverse_iterator
std::unordered_map< K, T >::const_reverse_iterator
std::multimap< K, T >::const_reverse_iterator
std::unordered_multimap< K, T >::const_reverse_iterator
std::basic_string< Char >::const_reverse_iterator
std::set< K >::const_reverse_iterator
std::unordered_set< K >::const_reverse_iterator
std::multiset< K >::const_reverse_iterator
std::vector< T >::const_reverse_iterator
std::unordered_multiset< K >::const_reverse_iterator
std::deque< T >
std::deque< StdVectorDouble >

```

dualColumnResult

578

```

std::error_category
std::error_code
std::error_condition
std::exception
    std::bad_alloc
    std::bad_cast
    std::bad_exception
    std::bad_typeid
    std::ios_base::failure
    std::logic_error
        std::domain_error
        std::invalid_argument
        std::length_error
        std::out_of_range
    std::runtime_error
        std::overflow_error
        std::range_error
        std::underflow_error
std::forward_list< T >

```

Idiot

580

IdiotResult

585

Info**586**

```
std::ios_base
  basic_ios< char >
  basic_ios< wchar_t >
  std::basic_ios
    basic_istream< char >
    basic_istream< wchar_t >
    basic_ostream< char >
    basic_ostream< wchar_t >
    std::basic_istream
      basic_ifstream< char >
      basic_ifstream< wchar_t >
      basic_iostream< char >
      basic_iostream< wchar_t >
      basic_istreamstream< char >
      basic_istreamstream< wchar_t >
      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_istreamstream
        std::istreamstream
        std::wistreamstream
      std::istream
      std::wistream
    std::basic_ostream
      basic_iostream< char >
      basic_iostream< wchar_t >
      basic_ofstream< char >
      basic_ofstream< wchar_t >
      basic_ostreamstream< char >
      basic_ostreamstream< wchar_t >
      std::basic_iostream
      std::basic_ofstream
        std::ofstream
        std::wofstream
      std::basic_ostreamstream
        std::ostreamstream
        std::wostringstream
      std::ostream
      std::wostream
    std::ios
    std::wios
  std::wstring::iterator
```

```

std::list< T >::iterator
std::map< K, T >::iterator
std::unordered_map< K, T >::iterator
std::unordered_set< K >::iterator
std::basic_string< Char >::iterator
std::string::iterator
std::unordered_multimap< K, T >::iterator
std::set< K >::iterator
std::multiset< K >::iterator
std::multimap< K, T >::iterator
std::forward_list< T >::iterator
std::unordered_multiset< K >::iterator
std::deque< T >::iterator
std::vector< T >::iterator
std::list< T >
std::map< K, T >
std::multimap< K, T >
std::multiset< K >

```

Options 591

OsiSolverInterface

OsiClpSolverInterface 596

Outfo 633

ClpSimplexOther::parametricsData 634

AbcSimplexPrimal::pivotStruct 636

```

std::priority_queue< T >
std::queue< T >
std::list< T >::reverse_iterator
std::unordered_set< K >::reverse_iterator
std::deque< T >::reverse_iterator
std::vector< T >::reverse_iterator
std::multimap< K, T >::reverse_iterator
std::unordered_multimap< K, T >::reverse_iterator
std::set< K >::reverse_iterator
std::string::reverse_iterator
std::unordered_map< K, T >::reverse_iterator
std::multiset< K >::reverse_iterator
std::map< K, T >::reverse_iterator
std::basic_string< Char >::reverse_iterator
std::forward_list< T >::reverse_iterator
std::wstring::reverse_iterator
std::unordered_multiset< K >::reverse_iterator

```

scatterStruct 638

```

std::set< K >
std::smart_ptr< T >
std::stack< T >
std::system_error
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< std::string >
std::weak_ptr< T >
K
T

```

2 Class Index

2.1 Class List

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

AbcDualRowDantzig	
Dual Row Pivot Dantzig Algorithm Class	15
AbcDualRowPivot	
Dual Row Pivot Abstract Base Class	18
AbcDualRowSteepest	
Dual Row Pivot Steepest Edge Algorithm Class	22
AbcMatrix	26
AbcMatrix2	41
AbcMatrix3	43
AbcNonLinearCost	46
AbcPrimalColumnDantzig	
Primal Column Pivot Dantzig Algorithm Class	51
AbcPrimalColumnPivot	
Primal Column Pivot Abstract Base Class	53
AbcPrimalColumnSteepest	
Primal Column Pivot Steepest Edge Algorithm Class	57
AbcSimplex	61
AbcSimplexDual	
This solves LPs using the dual simplex method	102
AbcSimplexFactorization	
This just implements AbcFactorization when an AbcMatrix object is passed	109
AbcSimplexPrimal	
This solves LPs using the primal simplex method	118
AbcTolerancesEtc	123

AbcWarmStart	
As CoinWarmStartBasis but with alternatives (Also uses Clp status meaning for slacks)	126
AbcWarmStartOrganizer	131
ampl_info	134
blockStruct	137
blockStruct3	138
ClpNode::branchState	139
CbcOrClpParam	
Very simple class for setting parameters	140
ClpCholeskyBase	
Base class for Clp Cholesky factorization Will do better factorization	147
ClpCholeskyDense	157
ClpCholeskyDenseC	160
ClpCholeskyMumps	
Mumps class for Clp Cholesky factorization	161
ClpCholeskyTaucs	
Taucs class for Clp Cholesky factorization	163
ClpCholeskyUfl	
Ufl class for Clp Cholesky factorization	165
ClpCholeskyWssmp	
Wssmp class for Clp Cholesky factorization	167
ClpCholeskyWssmpKKT	
WssmpKKT class for Clp Cholesky factorization	168
ClpConstraint	
Constraint Abstract Base Class	170
ClpConstraintLinear	
Linear Constraint Class	174
ClpConstraintQuadratic	
Quadratic Constraint Class	177
ClpDataSave	
This is a tiny class where data can be saved round calls	181
ClpDisasterHandler	
Base class for Clp disaster handling	183
ClpDualRowDantzig	
Dual Row Pivot Dantzig Algorithm Class	185
ClpDualRowPivot	
Dual Row Pivot Abstract Base Class	187

ClpDualRowSteepest	
Dual Row Pivot Steepest Edge Algorithm Class	190
ClpDummyMatrix	
This implements a dummy matrix as derived from ClpMatrixBase	194
ClpDynamicExampleMatrix	
This implements a dynamic matrix when we have a limit on the number of "interesting rows"	200
ClpDynamicMatrix	
This implements a dynamic matrix when we have a limit on the number of "interesting rows"	206
ClpEventHandler	
Base class for Clp event handling	221
ClpFactorization	
This just implements CoinFactorization when an ClpMatrixBase object is passed	225
ClpGubDynamicMatrix	
This implements Gub rows plus a ClpPackedMatrix	234
ClpGubMatrix	
This implements Gub rows plus a ClpPackedMatrix	243
ClpHashValue	256
ClpInterior	
This solves LPs using interior point methods	258
ClpLinearObjective	
Linear Objective Class	279
ClpLsqqr	
This class implements LSQR	282
ClpMatrixBase	
Abstract base class for Clp Matrices	286
ClpMessage	
This deals with Clp messages (as against Osi messages etc)	302
ClpModel	303
ClpNetworkBasis	
This deals with Factorization and Updates for network structures	343
ClpNetworkMatrix	
This implements a simple network matrix as derived from ClpMatrixBase	345
ClpNode	353
ClpNodeStuff	360
ClpNonLinearCost	366
ClpObjective	
Objective Abstract Base Class	371

ClpPackedMatrix	375
ClpPackedMatrix2	389
ClpPackedMatrix3	392
ClpPdco This solves problems in Primal Dual Convex Optimization	395
ClpPdcoBase Abstract base class for tailoring everything for Pcdco	396
ClpPlusMinusOneMatrix This implements a simple +- one matrix as derived from ClpMatrixBase	399
ClpPredictorCorrector This solves LPs using the predictor-corrector method due to Mehrotra	409
ClpPresolve This is the Clp interface to CoinPresolve	412
ClpPrimalColumnDantzig Primal Column Pivot Dantzig Algorithm Class	418
ClpPrimalColumnPivot Primal Column Pivot Abstract Base Class	420
ClpPrimalColumnSteepest Primal Column Pivot Steepest Edge Algorithm Class	424
ClpPrimalQuadraticDantzig Primal Column Pivot Dantzig Algorithm Class	430
ClpQuadraticObjective Quadratic Objective Class	431
ClpSimplex This solves LPs using the simplex method	436
ClpSimplexDual This solves LPs using the dual simplex method	489
ClpSimplexNonlinear This solves non-linear LPs using the primal simplex method	494
ClpSimplexOther This is for Simplex stuff which is neither dual nor primal	497
ClpSimplexPrimal This solves LPs using the primal simplex method	501
ClpSimplexProgress For saving extra information to see if looping	505
ClpSolve This is a very simple class to guide algorithms	511

ClpTrustedData	
For a structure to be used by trusted code	517
CoinAbcAnyFactorization	
Abstract base class which also has some scalars so can be used from Dense or Simp	517
CoinAbcDenseFactorization	
This deals with Factorization and Updates This is a simple dense version so other people can write a better one	530
CoinAbcStack	537
CoinAbcStatistics	537
CoinAbcThreadInfo	538
CoinAbcTypeFactorization	539
ClpHashValue::CoinHashLink	
Data	578
dualColumnResult	578
Idiot	
This class implements a very silly algorithm	580
IdiotResult	
For use internally	585
Info	
***** DATA to be moved into protected section of ClpInterior	586
MyEventHandler	
This is so user can trap events and do useful stuff	587
MyMessageHandler	589
Options	
***** DATA to be moved into protected section of ClpInterior	591
OsiClpDisasterHandler	593
OsiClpSolverInterface	
Clp Solver Interface	596
Outfo	
***** DATA to be moved into protected section of ClpInterior	633
ClpSimplexOther::parametricsData	634
AbcSimplexPrimal::pivotStruct	636
scatterStruct	638

3 File Index

3.1 File List

Here is a list of all files with brief descriptions:

src/AbcCommon.hpp	639
src/AbcDualRowDantzig.hpp	639
src/AbcDualRowPivot.hpp	639
src/AbcDualRowSteepest.hpp	639
src/AbcMatrix.hpp	640
src/AbcNonLinearCost.hpp	640
src/AbcPrimalColumnDantzig.hpp	642
src/AbcPrimalColumnPivot.hpp	642
src/AbcPrimalColumnSteepest.hpp	643
src/AbcSimplex.hpp	643
src/AbcSimplexDual.hpp	646
src/AbcSimplexFactorization.hpp	647
src/AbcSimplexPrimal.hpp	647
src/AbcWarmStart.hpp	647
src/CbcOrClpParam.hpp	648
src/Clp_ampl.h	657
src/Clp_C_Interface.h	657
src/ClpCholeskyBase.hpp	674
src/ClpCholeskyDense.hpp	675
src/ClpCholeskyMumps.hpp	677
src/ClpCholeskyTaucs.hpp	677
src/ClpCholeskyUfl.hpp	677
src/ClpCholeskyWssmp.hpp	678
src/ClpCholeskyWssmpKKT.hpp	678
src/ClpConfig.h	678
src/ClpConstraint.hpp	678
src/ClpConstraintLinear.hpp	679
src/ClpConstraintQuadratic.hpp	679

src/ClpDualRowDantzig.hpp	679
src/ClpDualRowPivot.hpp	679
src/ClpDualRowSteepest.hpp	679
src/ClpDummyMatrix.hpp	680
src/ClpDynamicExampleMatrix.hpp	680
src/ClpDynamicMatrix.hpp	680
src/ClpEventHandler.hpp	680
src/ClpFactorization.hpp	681
src/ClpGubDynamicMatrix.hpp	681
src/ClpGubMatrix.hpp	681
src/ClpHelperFunctions.hpp	682
src/ClpInterior.hpp	683
src/ClpLinearObjective.hpp	684
src/ClpLsqr.hpp	684
src/ClpMatrixBase.hpp	684
src/ClpMessage.hpp	685
src/ClpModel.hpp	688
src/ClpNetworkBasis.hpp	690
src/ClpNetworkMatrix.hpp	691
src/ClpNode.hpp	691
src/ClpNonLinearCost.hpp	691
src/ClpObjective.hpp	693
src/ClpPackedMatrix.hpp	693
src/ClpParameters.hpp	694
src/ClpPdco.hpp	696
src/ClpPdcoBase.hpp	696
src/ClpPlusMinusOneMatrix.hpp	696
src/ClpPredictorCorrector.hpp	696
src/ClpPresolve.hpp	697
src/ClpPrimalColumnDantzig.hpp	697

src/ClpPrimalColumnPivot.hpp	697
src/ClpPrimalColumnSteepest.hpp	697
src/ClpPrimalQuadraticDantzig.hpp	698
src/ClpQuadraticObjective.hpp	698
src/ClpSimplex.hpp	698
src/ClpSimplexDual.hpp	699
src/ClpSimplexNonlinear.hpp	699
src/ClpSimplexOther.hpp	700
src/ClpSimplexPrimal.hpp	700
src/ClpSolve.hpp	700
src/CoinAbcBaseFactorization.hpp	701
src/CoinAbcCommon.hpp	702
src/CoinAbcCommonFactorization.hpp	706
src/CoinAbcDenseFactorization.hpp	709
src/CoinAbcFactorization.hpp	709
src/CoinAbcHelperFunctions.hpp	710
src/config_clp_default.h	719
src/config_default.h	719
src/ldiot.hpp	720
src/MyEventHandler.hpp	720
src/MyMessageHandler.hpp	721
src/OsiClp/OsiClpSolverInterface.hpp	721

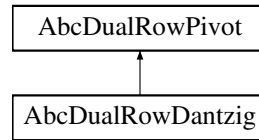
4 Class Documentation

4.1 AbcDualRowDantzig Class Reference

Dual Row Pivot Dantzig Algorithm Class.

```
#include <AbcDualRowDantzig.hpp>
```

Inheritance diagram for AbcDualRowDantzig:



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 &)
- virtual void [updateWeights2](#) (CoinIndexedVector &input, CoinIndexedVector &)
Actually updates weights.
- virtual void [updatePrimalSolution](#) (CoinIndexedVector &input, double theta)
Updates primal solution (and maybe list of candidates) Uses input vector which it deletes Computes change in objective function.
- 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.

Constructors and destructors

- [AbcDualRowDantzig](#) ()
Default Constructor.
- [AbcDualRowDantzig](#) (const [AbcDualRowDantzig](#) &)
Copy constructor.
- [AbcDualRowDantzig](#) & [operator=](#) (const [AbcDualRowDantzig](#) &rhs)
Assignment operator.
- virtual [~AbcDualRowDantzig](#) ()
Destructor.
- virtual [AbcDualRowPivot](#) * [clone](#) (bool copyData=true) const
Clone.

Additional Inherited Members

4.1.1 Detailed Description

Dual Row Pivot Dantzig Algorithm Class.

This is simplest choice - choose largest infeasibility

Definition at line 19 of file AbcDualRowDantzig.hpp.

4.1.2 Constructor & Destructor Documentation

4.1.2.1 `AbcDualRowDantzig::AbcDualRowDantzig ()`

Default Constructor.

4.1.2.2 `AbcDualRowDantzig::AbcDualRowDantzig (const AbcDualRowDantzig &)`

Copy constructor.

4.1.2.3 `virtual AbcDualRowDantzig::~~AbcDualRowDantzig () [virtual]`

Destructor.

4.1.3 Member Function Documentation

4.1.3.1 `virtual int AbcDualRowDantzig::pivotRow () [virtual]`

Returns pivot row, -1 if none.

Implements [AbcDualRowPivot](#).

4.1.3.2 `virtual double AbcDualRowDantzig::updateWeights (CoinIndexedVector & input, CoinIndexedVector & updatedColumn) [virtual]`

Updates weights and returns pivot alpha.

Also does FT update

Implements [AbcDualRowPivot](#).

4.1.3.3 `virtual double AbcDualRowDantzig::updateWeights1 (CoinIndexedVector & input, CoinIndexedVector & updateColumn) [virtual]`

Does most of work for weights and returns pivot alpha.

Also does FT update

Implements [AbcDualRowPivot](#).

4.1.3.4 `virtual void AbcDualRowDantzig::updateWeightsOnly (CoinIndexedVector &) [inline],[virtual]`

Implements [AbcDualRowPivot](#).

Definition at line 33 of file `AbcDualRowDantzig.hpp`.

4.1.3.5 `virtual void AbcDualRowDantzig::updateWeights2 (CoinIndexedVector & input, CoinIndexedVector &) [inline],[virtual]`

Actually updates weights.

Implements [AbcDualRowPivot](#).

Definition at line 35 of file `AbcDualRowDantzig.hpp`.

4.1.3.6 `virtual void AbcDualRowDantzig::updatePrimalSolution (CoinIndexedVector & input, double theta) [virtual]`

Updates primal solution (and maybe list of candidates) Uses input vector which it deletes Computes change in objective function.

Implements [AbcDualRowPivot](#).

4.1.3.7 `virtual void AbcDualRowDantzig::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 from [AbcDualRowPivot](#).

4.1.3.8 `virtual void AbcDualRowDantzig::recomputeInfeasibilities () [virtual]`

Recompute infeasibilities.

Reimplemented from [AbcDualRowPivot](#).

4.1.3.9 `AbcDualRowDantzig& AbcDualRowDantzig::operator= (const AbcDualRowDantzig & rhs)`

Assignment operator.

4.1.3.10 `virtual AbcDualRowPivot* AbcDualRowDantzig::clone (bool copyData = true) const [virtual]`

Clone.

Implements [AbcDualRowPivot](#).

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

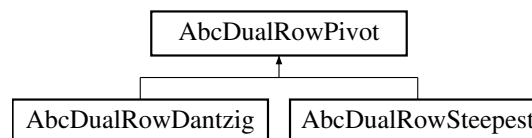
- [src/AbcDualRowDantzig.hpp](#)

4.2 AbcDualRowPivot Class Reference

Dual Row Pivot Abstract Base Class.

```
#include <AbcDualRowPivot.hpp>
```

Inheritance 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 Constructor & Destructor Documentation

4.2.2.1 `AbcDualRowPivot::AbcDualRowPivot ()`

Default Constructor.

4.2.2.2 `AbcDualRowPivot::AbcDualRowPivot (const AbcDualRowPivot &)`

Copy constructor.

4.2.2.3 `virtual AbcDualRowPivot::~~AbcDualRowPivot () [virtual]`

Destructor.

4.2.3 Member Function Documentation

4.2.3.1 `virtual int AbcDualRowPivot::pivotRow () [pure virtual]`

Returns pivot row, -1 if none.

Implemented in [AbcDualRowSteepest](#), and [AbcDualRowDantzig](#).

4.2.3.2 `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](#).

4.2.3.3 `virtual void AbcDualRowPivot::updateWeightsOnly (CoinIndexedVector & input) [pure virtual]`

Implemented in [AbcDualRowSteepest](#), and [AbcDualRowDantzig](#).

4.2.3.4 `virtual double AbcDualRowPivot::updateWeights (CoinIndexedVector & input, CoinIndexedVector & updateColumn) [pure virtual]`

Implemented in [AbcDualRowSteepest](#), and [AbcDualRowDantzig](#).

4.2.3.5 `virtual void AbcDualRowPivot::updateWeights2 (CoinIndexedVector & input, CoinIndexedVector & updateColumn) [pure virtual]`

Actually updates weights.

Implemented in [AbcDualRowSteepest](#), and [AbcDualRowDantzig](#).

4.2.3.6 `virtual void AbcDualRowPivot::updatePrimalSolution (CoinIndexedVector & updateColumn, double theta) [pure virtual]`

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.

Implemented in [AbcDualRowSteepest](#), and [AbcDualRowDantzig](#).

4.2.3.7 `virtual void AbcDualRowPivot::updatePrimalSolutionAndWeights (CoinIndexedVector & weightsVector, CoinIndexedVector & updateColumn, double theta) [virtual]`

Reimplemented in [AbcDualRowSteepest](#).

4.2.3.8 `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](#).

4.2.3.9 `virtual void AbcDualRowPivot::recomputeInfeasibilities () [virtual]`

Recompute infeasibilities.

Reimplemented in [AbcDualRowSteepest](#), and [AbcDualRowDantzig](#).

4.2.3.10 `virtual void AbcDualRowPivot::checkAccuracy () [virtual]`

checks accuracy and may re-initialize (may be empty)

4.2.3.11 `virtual void AbcDualRowPivot::clearArrays () [virtual]`

Gets rid of all arrays (may be empty)

Reimplemented in [AbcDualRowSteepest](#).

4.2.3.12 `virtual bool AbcDualRowPivot::looksOptimal () const [inline],[virtual]`

Returns true if would not find any row.

Reimplemented in [AbcDualRowSteepest](#).

Definition at line 69 of file `AbcDualRowPivot.hpp`.

4.2.3.13 `AbcDualRowPivot& AbcDualRowPivot::operator= (const AbcDualRowPivot & rhs)`

Assignment operator.

4.2.3.14 `virtual AbcDualRowPivot* AbcDualRowPivot::clone (bool copyData =true) const [pure virtual]`

Clone.

Implemented in [AbcDualRowSteepest](#), and [AbcDualRowDantzig](#).

4.2.3.15 `AbcSimplex* AbcDualRowPivot::model () [inline]`

Returns model.

Definition at line 97 of file `AbcDualRowPivot.hpp`.

4.2.3.16 `void AbcDualRowPivot::setModel (AbcSimplex * newmodel) [inline]`

Sets model (normally to NULL)

Definition at line 102 of file `AbcDualRowPivot.hpp`.

4.2.3.17 `int AbcDualRowPivot::type () [inline]`

Returns type (above 63 is extra information)

Definition at line 107 of file `AbcDualRowPivot.hpp`.

4.2.4 Member Data Documentation

4.2.4.1 `AbcSimplex* AbcDualRowPivot::model_ [protected]`

Pointer to model.

Definition at line 119 of file `AbcDualRowPivot.hpp`.

4.2.4.2 `int AbcDualRowPivot::type_ [protected]`

Type of row pivot algorithm.

Definition at line 121 of file `AbcDualRowPivot.hpp`.

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

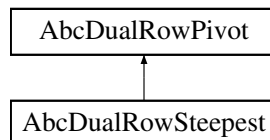
- [src/AbcDualRowPivot.hpp](#)

4.3 AbcDualRowSteepest Class Reference

Dual Row Pivot Steepest Edge Algorithm Class.

```
#include <AbcDualRowSteepest.hpp>
```

Inheritance diagram for `AbcDualRowSteepest`:



Public Types

- enum `Persistence` { `normal` = 0x00, `keep` = 0x01 }
- 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 Member Enumeration Documentation

4.3.2.1 enum `AbcDualRowSteepest::Persistence`

enums for persistence

Enumerator

normal

keep

Definition at line 69 of file `AbcDualRowSteepest.hpp`.

4.3.3 Constructor & Destructor Documentation

4.3.3.1 `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.

4.3.3.2 `AbcDualRowSteepest::AbcDualRowSteepest (const AbcDualRowSteepest &)`

Copy constructor.

4.3.3.3 `virtual AbcDualRowSteepest::~AbcDualRowSteepest () [virtual]`

Destructor.

4.3.4 Member Function Documentation

4.3.4.1 `virtual int AbcDualRowSteepest::pivotRow () [virtual]`

Returns pivot row, -1 if none.

Implements [AbcDualRowPivot](#).

4.3.4.2 `virtual double AbcDualRowSteepest::updateWeights (CoinIndexedVector & input, CoinIndexedVector & updatedColumn) [virtual]`

Updates weights and returns pivot alpha.

Also does FT update

Implements [AbcDualRowPivot](#).

4.3.4.3 `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](#).

4.3.4.4 `virtual void AbcDualRowSteepest::updateWeightsOnly (CoinIndexedVector & input) [virtual]`

Implements [AbcDualRowPivot](#).

4.3.4.5 `virtual void AbcDualRowSteepest::updateWeights2 (CoinIndexedVector & input, CoinIndexedVector & updateColumn) [virtual]`

Actually updates weights.

Implements [AbcDualRowPivot](#).

4.3.4.6 `virtual void AbcDualRowSteepest::updatePrimalSolution (CoinIndexedVector & input, double theta) [virtual]`

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

Implements [AbcDualRowPivot](#).

4.3.4.7 `virtual void AbcDualRowSteepest::updatePrimalSolutionAndWeights (CoinIndexedVector & weightsVector, CoinIndexedVector & updateColumn, double theta) [virtual]`

Reimplemented from [AbcDualRowPivot](#).

4.3.4.8 `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](#).

4.3.4.9 `virtual void AbcDualRowSteepest::recomputeInfeasibilities () [virtual]`

Recompute infeasibilities.

Reimplemented from [AbcDualRowPivot](#).

4.3.4.10 `virtual void AbcDualRowSteepest::clearArrays () [virtual]`

Gets rid of all arrays.

Reimplemented from [AbcDualRowPivot](#).

4.3.4.11 `virtual bool AbcDualRowSteepest::looksOptimal () const [virtual]`

Returns true if would not find any row.

Reimplemented from [AbcDualRowPivot](#).

4.3.4.12 `AbcDualRowSteepest& AbcDualRowSteepest::operator= (const AbcDualRowSteepest & rhs)`

Assignment operator.

4.3.4.13 `void AbcDualRowSteepest::fill (const AbcDualRowSteepest & rhs)`

Fill most values.

4.3.4.14 `virtual AbcDualRowPivot* AbcDualRowSteepest::clone (bool copyData = true) const` [virtual]

Clone.

Implements [AbcDualRowPivot](#).

4.3.4.15 `int AbcDualRowSteepest::mode () const` [inline]

Mode.

Definition at line 104 of file `AbcDualRowSteepest.hpp`.

4.3.4.16 `void AbcDualRowSteepest::setPersistence (Persistence life)` [inline]

Set/ get persistence.

Definition at line 108 of file `AbcDualRowSteepest.hpp`.

4.3.4.17 `Persistence AbcDualRowSteepest::persistence () const` [inline]

Definition at line 111 of file `AbcDualRowSteepest.hpp`.

4.3.4.18 `CoinIndexedVector* AbcDualRowSteepest::infeasible () const` [inline]

Infeasible vector.

Definition at line 115 of file `AbcDualRowSteepest.hpp`.

4.3.4.19 `CoinIndexedVector* AbcDualRowSteepest::weights () const` [inline]

Weights vector.

Definition at line 118 of file `AbcDualRowSteepest.hpp`.

4.3.4.20 `AbcSimplex* AbcDualRowSteepest::model () const` [inline]

Model.

Definition at line 121 of file `AbcDualRowSteepest.hpp`.

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

- [src/AbcDualRowSteepest.hpp](#)

4.4 AbcMatrix Class Reference

```
#include <AbcMatrix.hpp>
```

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 numberOfRowsAlreadyScaled)
Scales and creates row copy.
- void [createRowCopy](#) ()
Creates row copy.
- void [takeOutOfUseful](#) (int sequence, [CoinIndexedVector](#) &spare)
Take out of useful.
- void [putIntoUseful](#) (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 &sparseColumn1, 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 &sparseColumn1, 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 &sparseColumn1, 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 $x \times *A$ in z but just for indices Already in z .*
- void [transposeTimes](#) (const CoinIndexedVector &x, CoinIndexedVector &z) const
*Return $-x \times *A$ 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 numberOfRows, const int *whichRows, int numberColumns, const int *whichColumns)
Subset constructor (without gaps).
- `AbcMatrix` (const `CoinPackedMatrix` &wholeModel, int numberOfRows, const int *whichRows, int numberColumns, 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 ()

Destructor.

4.4.2.3 AbcMatrix::AbcMatrix (const AbcMatrix &)

The copy constructor.

4.4.2.4 AbcMatrix::AbcMatrix (const CoinPackedMatrix &)

The copy constructor from an CoinPackedMatrix.

4.4.2.5 AbcMatrix::AbcMatrix (const AbcMatrix & wholeModel, int numberOfRows, const int * whichRows, int numberColumns, const int * whichColumns)

Subset constructor (without gaps).

Duplicates are allowed and order is as given

4.4.2.6 AbcMatrix::AbcMatrix (const CoinPackedMatrix & wholeModel, int numberOfRows, const int * whichRows, int numberColumns, const int * whichColumns)

4.4.3 Member Function Documentation

4.4.3.1 `CoinPackedMatrix* AbcMatrix::getPackedMatrix () const [inline]`

Return a complete `CoinPackedMatrix`.

Definition at line 28 of file `AbcMatrix.hpp`.

4.4.3.2 `bool AbcMatrix::isColOrdered () const [inline]`

Whether the packed matrix is column major ordered or not.

Definition at line 32 of file `AbcMatrix.hpp`.

4.4.3.3 `CoinBigIndex AbcMatrix::getNumElements () const [inline]`

Number of entries in the packed matrix.

Definition at line 36 of file `AbcMatrix.hpp`.

4.4.3.4 `int AbcMatrix::getNumCols () const [inline]`

Number of columns.

Definition at line 40 of file `AbcMatrix.hpp`.

4.4.3.5 `int AbcMatrix::getNumRows () const [inline]`

Number of rows.

Definition at line 44 of file `AbcMatrix.hpp`.

4.4.3.6 `void AbcMatrix::setModel (AbcSimplex * model)`

Sets model.

4.4.3.7 `const double* AbcMatrix::getElements () const [inline]`

A vector containing the elements in the packed matrix.

Definition at line 50 of file `AbcMatrix.hpp`.

4.4.3.8 `double* AbcMatrix::getMutableElements () const [inline]`

Mutable elements.

Definition at line 54 of file `AbcMatrix.hpp`.

4.4.3.9 `const int* AbcMatrix::getIndices () const [inline]`

A vector containing the minor indices of the elements in the packed matrix.

Definition at line 58 of file `AbcMatrix.hpp`.

4.4.3.10 `int* AbcMatrix::getMutableIndices () const [inline]`

A vector containing the minor indices of the elements in the packed matrix.

Definition at line 62 of file `AbcMatrix.hpp`.

4.4.3.11 `const CoinBigIndex* AbcMatrix::getVectorStarts () const [inline]`

Starts.

Definition at line 66 of file AbcMatrix.hpp.

4.4.3.12 `CoinBigIndex* AbcMatrix::getMutableVectorStarts () const` `[inline]`

Definition at line 69 of file AbcMatrix.hpp.

4.4.3.13 `const int* AbcMatrix::getVectorLengths () const` `[inline]`

The lengths of the major-dimension vectors.

Definition at line 73 of file AbcMatrix.hpp.

4.4.3.14 `int* AbcMatrix::getMutableVectorLengths () const` `[inline]`

The lengths of the major-dimension vectors.

Definition at line 77 of file AbcMatrix.hpp.

4.4.3.15 `CoinBigIndex* AbcMatrix::rowStart () const`

Row starts.

4.4.3.16 `CoinBigIndex* AbcMatrix::rowEnd () const`

Row ends.

4.4.3.17 `double* AbcMatrix::rowElements () const`

Row elements.

4.4.3.18 `CoinSimplexInt* AbcMatrix::rowColumns () const`

Row columns.

4.4.3.19 `CoinPackedMatrix* AbcMatrix::reverseOrderedCopy () const`

Returns a new matrix in reverse order without gaps.

4.4.3.20 `CoinBigIndex AbcMatrix::countBasis (const int * whichColumn, int & numberColumnBasic)`

Returns number of elements in column part of basis.

4.4.3.21 `void AbcMatrix::fillBasis (const int * whichColumn, int & numberColumnBasic, int * row, int * start, int * rowCount, int * columnCount, CoinSimplexDouble * element)`

Fills in column part of basis.

4.4.3.22 `void AbcMatrix::fillBasis (const int * whichColumn, int & numberColumnBasic, int * row, int * start, int * rowCount, int * columnCount, long double * element)`

Fills in column part of basis.

4.4.3.23 `void AbcMatrix::scale (int numberRowsAlreadyScaled)`

Scales and creates row copy.

4.4.3.24 `void AbcMatrix::createRowCopy ()`

Creates row copy.

4.4.3.25 void `AbcMatrix::takeOutOfUseful (int sequence, CoinIndexedVector & spare)`

Take out of useful.

4.4.3.26 void `AbcMatrix::putIntofUseful (int sequence, CoinIndexedVector & spare)`

Put into useful.

4.4.3.27 void `AbcMatrix::inOutUseful (int sequenceIn, int sequenceOut)`

Put in and out for useful.

4.4.3.28 void `AbcMatrix::makeAllUseful (CoinIndexedVector & spare)`

Make all useful.

4.4.3.29 void `AbcMatrix::sortUseful (CoinIndexedVector & spare)`

Sort into useful.

4.4.3.30 void `AbcMatrix::moveLargestToStart ()`

Move largest in column to beginning (not used as doesn't help factorization)

4.4.3.31 void `AbcMatrix::unpack (CoinIndexedVector & rowArray, int column) const`

Unpacks a column into an CoinIndexedVector.

4.4.3.32 void `AbcMatrix::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.

4.4.3.33 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()`

4.4.3.34 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()`

4.4.3.35 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()`

4.4.3.36 `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()`

4.4.3.37 `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()`

4.4.3.38 `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()`

4.4.3.39 `int AbcMatrix::transposeTimesNonBasic (double scalar, const CoinIndexedVector & x, CoinIndexedVector & z) const`

Return $x * scalar * A$ 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](#)

4.4.3.40 `double AbcMatrix::dualColumn1 (const CoinIndexedVector & update, CoinPartitionedVector & tableauRow, CoinPartitionedVector & candidateList) const`

gets sorted tableau row and a possible value of theta

4.4.3.41 `double AbcMatrix::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

4.4.3.42 `double AbcMatrix::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

4.4.3.43 `double AbcMatrix::dualColumn1Row2 (double upperThetaSlack, int & freeSequence, const CoinIndexedVector & update, CoinPartitionedVector & tableauRow, CoinPartitionedVector & candidateList) const`

gets sorted tableau row and a possible value of theta

4.4.3.44 `double AbcMatrix::dualColumn1Row1 (double upperThetaSlack, int & freeSequence, const CoinIndexedVector & update, CoinPartitionedVector & tableauRow, CoinPartitionedVector & candidateList) const`

gets sorted tableau row and a possible value of theta

4.4.3.45 `void AbcMatrix::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

4.4.3.46 `void AbcMatrix::rebalance () const`

rebalance for parallel

4.4.3.47 `int AbcMatrix::pivotColumnDantzig (const CoinIndexedVector & updates, CoinPartitionedVector & spare) const`

Get sequenceIn when Dantzig.

4.4.3.48 `int AbcMatrix::pivotColumnDantzig (int iBlock, bool doByRow, const CoinIndexedVector & updates, CoinPartitionedVector & spare, double & bestValue) const`

Get sequenceIn when Dantzig (One block)

4.4.3.49 `int AbcMatrix::primalColumnRow (int iBlock, bool doByRow, const CoinIndexedVector & update, CoinPartitionedVector & tableauRow) const`

gets tableau row - returns number of slacks in block

4.4.3.50 `int AbcMatrix::primalColumnRowAndDjs (int iBlock, const CoinIndexedVector & updateTableau, const CoinIndexedVector & updateDjs, CoinPartitionedVector & tableauRow) const`

gets tableau row and dj row - returns number of slacks in block

4.4.3.51 `int AbcMatrix::chooseBestDj (int iBlock, const CoinIndexedVector & infeasibilities, const double * weights) const`

Chooses best weighted dj.

4.4.3.52 `int AbcMatrix::primalColumnDouble (int iBlock, CoinPartitionedVector & updateForTableauRow, CoinPartitionedVector & updateForDjs, const CoinIndexedVector & updateForWeights, CoinPartitionedVector & spareColumn1, double * infeasibilities, double referenceln, 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

4.4.3.53 `int AbcMatrix::primalColumnSparseDouble (int iBlock, CoinPartitionedVector & updateForTableauRow, CoinPartitionedVector & updateForDjs, const CoinIndexedVector & updateForWeights, CoinPartitionedVector & spareColumn1, double * infeasibilities, double referenceln, 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

4.4.3.54 `int AbcMatrix::primalColumnDouble (CoinPartitionedVector & updateForTableauRow, CoinPartitionedVector & updateForDjs, const CoinIndexedVector & updateForWeights, CoinPartitionedVector & sparseColumn1, 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

4.4.3.55 `void AbcMatrix::primalColumnSubset (int iBlock, const CoinIndexedVector & update, const CoinPartitionedVector & tableauRow, CoinPartitionedVector & weights) const`

gets subset updates

4.4.3.56 `void AbcMatrix::partialPricing (double startFraction, double endFraction, int & bestSequence, int & numberWanted)`

Partial pricing.

4.4.3.57 `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

4.4.3.58 `void AbcMatrix::transposeTimes (const CoinIndexedVector & x, CoinIndexedVector & z) const`

Return $-x * A$ in *z*

4.4.3.59 `CoinPackedMatrix* AbcMatrix::matrix () const [inline]`

Returns CoinPackedMatrix (non const)

Definition at line 289 of file AbcMatrix.hpp.

4.4.3.60 `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.

4.4.3.61 `void AbcMatrix::setMinimumObjectsScan (int value) [inline]`

Definition at line 297 of file AbcMatrix.hpp.

4.4.3.62 `int AbcMatrix::minimumGoodReducedCosts () const [inline]`

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

Definition at line 301 of file AbcMatrix.hpp.

4.4.3.63 `void AbcMatrix::setMinimumGoodReducedCosts (int value) [inline]`

Definition at line 304 of file AbcMatrix.hpp.

4.4.3.64 `double AbcMatrix::startFraction () const [inline]`

Current start of search space in matrix (as fraction)

Definition at line 308 of file AbcMatrix.hpp.

4.4.3.65 `void AbcMatrix::setStartFraction (double value) [inline]`

Definition at line 311 of file AbcMatrix.hpp.

4.4.3.66 `double AbcMatrix::endFraction () const [inline]`

Current end of search space in matrix (as fraction)

Definition at line 315 of file AbcMatrix.hpp.

4.4.3.67 `void AbcMatrix::setEndFraction (double value) [inline]`

Definition at line 318 of file AbcMatrix.hpp.

4.4.3.68 `double AbcMatrix::savedBestDj () const [inline]`

Current best reduced cost.

Definition at line 322 of file AbcMatrix.hpp.

4.4.3.69 `void AbcMatrix::setSavedBestDj (double value) [inline]`

Definition at line 325 of file AbcMatrix.hpp.

4.4.3.70 `int AbcMatrix::originalWanted () const [inline]`

Initial number of negative reduced costs wanted.

Definition at line 329 of file AbcMatrix.hpp.

4.4.3.71 `void AbcMatrix::setOriginalWanted (int value) [inline]`

Definition at line 332 of file AbcMatrix.hpp.

4.4.3.72 `int AbcMatrix::currentWanted () const [inline]`

Current number of negative reduced costs which we still need.

Definition at line 336 of file AbcMatrix.hpp.

4.4.3.73 `void AbcMatrix::setCurrentWanted (int value) [inline]`

Definition at line 339 of file AbcMatrix.hpp.

4.4.3.74 `int AbcMatrix::savedBestSequence () const [inline]`

Current best sequence.

Definition at line 343 of file AbcMatrix.hpp.

4.4.3.75 `void AbcMatrix::setSavedBestSequence (int value) [inline]`

Definition at line 346 of file AbcMatrix.hpp.

4.4.3.76 `int* AbcMatrix::startColumnBlock () const [inline]`

Start of each column block.

Definition at line 350 of file AbcMatrix.hpp.

4.4.3.77 `const int* AbcMatrix::blockStart () const [inline]`

Start of each block (in stored)

Definition at line 353 of file AbcMatrix.hpp.

4.4.3.78 `bool AbcMatrix::gotRowCopy () const [inline]`

Definition at line 355 of file AbcMatrix.hpp.

4.4.3.79 `int AbcMatrix::blockStart (int block) const [inline]`

Start of each block (in stored)

Definition at line 358 of file AbcMatrix.hpp.

4.4.3.80 `int AbcMatrix::numberColumnBlocks () const [inline]`

Number of actual column blocks.

Definition at line 361 of file AbcMatrix.hpp.

4.4.3.81 `int AbcMatrix::numberRowBlocks () const [inline]`

Number of actual row blocks.

Definition at line 364 of file AbcMatrix.hpp.

4.4.3.82 `AbcMatrix& AbcMatrix::operator= (const AbcMatrix &)`

4.4.3.83 `void AbcMatrix::copy (const AbcMatrix * from)`

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

4.4.4 Member Data Documentation

4.4.4.1 `CoinPackedMatrix* AbcMatrix::matrix_ [protected]`

Data.

Definition at line 403 of file AbcMatrix.hpp.

4.4.4.2 `AbcSimplex* AbcMatrix::model_ [mutable],[protected]`

Model.

Definition at line 405 of file AbcMatrix.hpp.

4.4.4.3 `CoinBigIndex* AbcMatrix::rowStart_ [protected]`

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.

Definition at line 419 of file AbcMatrix.hpp.

4.4.4.4 `double* AbcMatrix::element_ [protected]`

Values by row.

Definition at line 421 of file AbcMatrix.hpp.

4.4.4.5 `int* AbcMatrix::column_` [protected]

Columns.

Definition at line 423 of file AbcMatrix.hpp.

4.4.4.6 `int AbcMatrix::startColumnBlock_[NUMBER_COLUMN_BLOCKS+1]` [mutable], [protected]

Start of each column block.

Definition at line 425 of file AbcMatrix.hpp.

4.4.4.7 `int AbcMatrix::blockStart_[NUMBER_ROW_BLOCKS+1]` [protected]

Start of each block (in stored)

Definition at line 427 of file AbcMatrix.hpp.

4.4.4.8 `int AbcMatrix::numberColumnBlocks_` [mutable], [protected]

Number of actual column blocks.

Definition at line 429 of file AbcMatrix.hpp.

4.4.4.9 `int AbcMatrix::numberRowBlocks_` [protected]

Number of actual row blocks.

Definition at line 431 of file AbcMatrix.hpp.

4.4.4.10 `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.

4.4.4.11 `double AbcMatrix::endFraction_` [protected]

Current end of search space in matrix (as fraction)

Definition at line 455 of file AbcMatrix.hpp.

4.4.4.12 `double AbcMatrix::savedBestDj_` [protected]

Best reduced cost so far.

Definition at line 457 of file AbcMatrix.hpp.

4.4.4.13 `int AbcMatrix::originalWanted_` [protected]

Initial number of negative reduced costs wanted.

Definition at line 459 of file AbcMatrix.hpp.

4.4.4.14 `int AbcMatrix::currentWanted_` [protected]

Current number of negative reduced costs which we still need.

Definition at line 461 of file AbcMatrix.hpp.

4.4.4.15 int AbcMatrix::savedBestSequence_ [protected]

Saved best sequence in pricing.

Definition at line 463 of file AbcMatrix.hpp.

4.4.4.16 int AbcMatrix::minimumObjectsScan_ [protected]

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

Definition at line 465 of file AbcMatrix.hpp.

4.4.4.17 int AbcMatrix::minimumGoodReducedCosts_ [protected]

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

Definition at line 467 of file AbcMatrix.hpp.

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

- [src/AbcMatrix.hpp](#)

4.5 AbcMatrix2 Class Reference

```
#include <AbcMatrix.hpp>
```

Public Member Functions

Useful methods

- void [transposeTimes](#) (const [AbcSimplex](#) *model, const CoinPackedMatrix *rowCopy, const CoinIndexedVector &x, CoinIndexedVector &sparseArray, 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 ()`

Destructor.

4.5.2.4 `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](#)

4.5.3.2 `bool AbcMatrix2::usefullInfo () const [inline]`

Returns true if copy has useful information.

Definition at line 509 of file `AbcMatrix.hpp`.

4.5.3.3 AbcMatrix2& AbcMatrix2::operator= (const AbcMatrix2 &)**4.5.4 Member Data Documentation****4.5.4.1 int AbcMatrix2::numberBlocks_ [protected]**

Number of blocks.

Definition at line 538 of file AbcMatrix.hpp.

4.5.4.2 int AbcMatrix2::numberRows_ [protected]

Number of rows.

Definition at line 540 of file AbcMatrix.hpp.

4.5.4.3 int* AbcMatrix2::offset_ [protected]

Column offset for each block (plus one at end)

Definition at line 542 of file AbcMatrix.hpp.

4.5.4.4 unsigned short* AbcMatrix2::count_ [mutable], [protected]

Counts of elements in each part of row.

Definition at line 544 of file AbcMatrix.hpp.

4.5.4.5 CoinBigIndex* AbcMatrix2::rowStart_ [mutable], [protected]

Row starts.

Definition at line 546 of file AbcMatrix.hpp.

4.5.4.6 unsigned short* AbcMatrix2::column_ [protected]

columns within block

Definition at line 548 of file AbcMatrix.hpp.

4.5.4.7 double* AbcMatrix2::work_ [protected]

work arrays

Definition at line 550 of file AbcMatrix.hpp.

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

- src/[AbcMatrix.hpp](#)

4.6 AbcMatrix3 Class Reference

```
#include <AbcMatrix.hpp>
```

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 ()`

Destructor.

4.6.2.4 `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](#)

4.6.3.2 `void AbcMatrix3::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.

4.6.3.3 `AbcMatrix3& AbcMatrix3::operator= (const AbcMatrix3 &)`**4.6.3.4 `void AbcMatrix3::sortBlocks (const AbcSimplex * model)`**

Sort blocks.

4.6.3.5 `void AbcMatrix3::swapOne (const AbcSimplex * model, const AbcMatrix * matrix, int iColumn)`

Swap one variable.

4.6.4 Member Data Documentation**4.6.4.1 `int AbcMatrix3::numberBlocks_ [protected]`**

Number of blocks.

Definition at line 617 of file `AbcMatrix.hpp`.

4.6.4.2 `int AbcMatrix3::numberColumns_ [protected]`

Number of columns.

Definition at line 619 of file `AbcMatrix.hpp`.

4.6.4.3 `int* AbcMatrix3::column_` [protected]

Column indices and reverse lookup (within block)

Definition at line 621 of file `AbcMatrix.hpp`.

4.6.4.4 `CoinBigIndex* AbcMatrix3::start_` [protected]

Starts for odd/long vectors.

Definition at line 623 of file `AbcMatrix.hpp`.

4.6.4.5 `int* AbcMatrix3::row_` [protected]

Rows.

Definition at line 625 of file `AbcMatrix.hpp`.

4.6.4.6 `double* AbcMatrix3::element_` [protected]

Elements.

Definition at line 627 of file `AbcMatrix.hpp`.

4.6.4.7 `blockStruct* AbcMatrix3::block_` [protected]

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

Definition at line 629 of file `AbcMatrix.hpp`.

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

- [src/AbcMatrix.hpp](#)

4.7 `AbcNonLinearCost` Class Reference

```
#include <AbcNonLinearCost.hpp>
```

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 $\alpha < 0.0$ 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 70 of file `AbcNonLinearCost.hpp`.

4.7.2 Constructor & Destructor Documentation

4.7.2.1 `AbcNonLinearCost::AbcNonLinearCost ()`

Default constructor.

4.7.2.2 `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 .

4.7.2.3 `AbcNonLinearCost::~~AbcNonLinearCost ()`

Destructor.

4.7.2.4 `AbcNonLinearCost::AbcNonLinearCost (const AbcNonLinearCost &)`

4.7.3 Member Function Documentation

4.7.3.1 `AbcNonLinearCost& AbcNonLinearCost::operator= (const AbcNonLinearCost &)`

4.7.3.2 `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 \leq oldTolerance away

4.7.3.3 `void AbcNonLinearCost::checkInfeasibilities (int numberInArray, const int * index)`

Changes infeasible costs for each variable. The indices are row indices and need converting to sequences.

4.7.3.4 `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`

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

Goes through one bound for each variable.

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

4.7.3.6 `void AbcNonLinearCost::goBack (int numberInArray, const int * index, double * rhs)`

Takes off last iteration (i.e.

offsets closer to 0)

4.7.3.7 `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

4.7.3.8 `void AbcNonLinearCost::zapCosts ()`

Temporary zeroing of feasible costs.

4.7.3.9 `void AbcNonLinearCost::refreshCosts (const double * columnCosts)`

Refreshes costs always makes row costs zero.

4.7.3.10 `void AbcNonLinearCost::feasibleBounds ()`

Puts feasible bounds into lower and upper.

4.7.3.11 `void AbcNonLinearCost::refresh ()`

Refresh - assuming regions OK.

4.7.3.12 `void AbcNonLinearCost::refreshFromPerturbed (double tolerance)`

Refresh - from original.

4.7.3.13 `double AbcNonLinearCost::setOne (int sequence, double solutionValue)`

Sets bounds and cost for one variable Returns change in cost May need to be inline for speed.

4.7.3.14 `double AbcNonLinearCost::setOneBasic (int iRow, double solutionValue)`

Sets bounds and cost for one variable Returns change in cost May need to be inline for speed.

4.7.3.15 `int AbcNonLinearCost::setOneOutgoing (int sequence, double & solutionValue)`

Sets bounds and cost for outgoing variable may change value Returns direction.

4.7.3.16 `double AbcNonLinearCost::nearest (int iRow, double solutionValue)`

Returns nearest bound.

4.7.3.17 `double AbcNonLinearCost::changeInCost (int , double alpha) const [inline]`

Returns change in cost - one down if $\alpha > 0.0$, up if < 0.0 Value is current - new.

Definition at line 156 of file AbcNonLinearCost.hpp.

4.7.3.18 `double AbcNonLinearCost::changeUpInCost (int) const [inline]`

Definition at line 159 of file AbcNonLinearCost.hpp.

4.7.3.19 `double AbcNonLinearCost::changeDownInCost (int) const [inline]`

Definition at line 162 of file AbcNonLinearCost.hpp.

4.7.3.20 `double AbcNonLinearCost::changeInCost (int iRow, double alpha, double & rhs) [inline]`

This also updates next bound.

Definition at line 166 of file AbcNonLinearCost.hpp.

4.7.3.21 `int AbcNonLinearCost::numberInfeasibilities () const [inline]`

Number of infeasibilities.

Definition at line 205 of file AbcNonLinearCost.hpp.

4.7.3.22 `double AbcNonLinearCost::changeInCost () const [inline]`

Change in cost.

Definition at line 209 of file AbcNonLinearCost.hpp.

4.7.3.23 `double AbcNonLinearCost::feasibleCost () const [inline]`

Feasible cost.

Definition at line 213 of file AbcNonLinearCost.hpp.

4.7.3.24 `double AbcNonLinearCost::feasibleReportCost () const`

Feasible cost with offset and direction (i.e. for reporting)

4.7.3.25 `double AbcNonLinearCost::sumInfeasibilities () const [inline]`

Sum of infeasibilities.

Definition at line 219 of file AbcNonLinearCost.hpp.

4.7.3.26 `double AbcNonLinearCost::largestInfeasibility () const [inline]`

Largest infeasibility.

Definition at line 223 of file AbcNonLinearCost.hpp.

4.7.3.27 `double AbcNonLinearCost::averageTheta () const [inline]`

Average theta.

Definition at line 227 of file AbcNonLinearCost.hpp.

4.7.3.28 `void AbcNonLinearCost::setAverageTheta (double value) [inline]`

Definition at line 230 of file AbcNonLinearCost.hpp.

4.7.3.29 `void AbcNonLinearCost::setChangeInCost (double value) [inline]`

Definition at line 233 of file AbcNonLinearCost.hpp.

4.7.3.30 `unsigned char* AbcNonLinearCost::statusArray () const` `[inline]`

Definition at line 238 of file `AbcNonLinearCost.hpp`.

4.7.3.31 `int AbcNonLinearCost::getCurrentStatus (int sequence)` `[inline]`

Definition at line 241 of file `AbcNonLinearCost.hpp`.

4.7.3.32 `void AbcNonLinearCost::validate ()`

For debug.

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

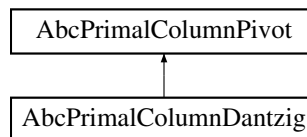
- [src/AbcNonLinearCost.hpp](#)

4.8 **AbcPrimalColumnDantzig Class Reference**

Primal Column Pivot Dantzig Algorithm Class.

```
#include <AbcPrimalColumnDantzig.hpp>
```

Inheritance diagram for `AbcPrimalColumnDantzig`:



Public Member Functions

Algorithmic methods

- virtual int [pivotColumn](#) (CoinPartitionedVector *updates, CoinPartitionedVector *spareRow2, CoinPartitionedVector *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 Constructor & Destructor Documentation

4.8.2.1 `AbcPrimalColumnDantzig::AbcPrimalColumnDantzig ()`

Default Constructor.

4.8.2.2 `AbcPrimalColumnDantzig::AbcPrimalColumnDantzig (const AbcPrimalColumnDantzig &)`

Copy constructor.

4.8.2.3 `virtual AbcPrimalColumnDantzig::~~AbcPrimalColumnDantzig () [virtual]`

Destructor.

4.8.3 Member Function Documentation

4.8.3.1 `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](#).

4.8.3.2 `virtual void AbcPrimalColumnDantzig::saveWeights (AbcSimplex * model, int) [inline],[virtual]`

Just sets model.

Implements [AbcPrimalColumnPivot](#).

Definition at line 38 of file `AbcPrimalColumnDantzig.hpp`.

4.8.3.3 `AbcPrimalColumnDantzig& AbcPrimalColumnDantzig::operator= (const AbcPrimalColumnDantzig & rhs)`

Assignment operator.

4.8.3.4 `virtual AbcPrimalColumnPivot* AbcPrimalColumnDantzig::clone (bool copyData = true) const [virtual]`

Clone.

Implements [AbcPrimalColumnPivot](#).

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

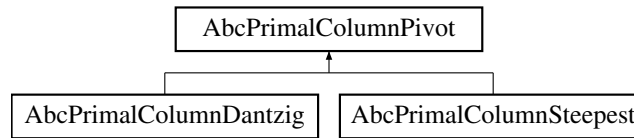
- [src/AbcPrimalColumnDantzig.hpp](#)

4.9 **AbcPrimalColumnPivot Class Reference**

Primal Column Pivot Abstract Base Class.

```
#include <AbcPrimalColumnPivot.hpp>
```

Inheritance diagram for AbcPrimalColumnPivot:



Public Member Functions

Algorithmic methods

- virtual int [pivotColumn](#) (CoinPartitionedVector *updates, CoinPartitionedVector *spareRow2, CoinPartitionedVector *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 numberOfRows-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 Constructor & Destructor Documentation

4.9.2.1 `AbcPrimalColumnPivot::AbcPrimalColumnPivot ()`

Default Constructor.

4.9.2.2 `AbcPrimalColumnPivot::AbcPrimalColumnPivot (const AbcPrimalColumnPivot &)`

Copy constructor.

4.9.2.3 `virtual AbcPrimalColumnPivot::~~AbcPrimalColumnPivot () [virtual]`

Destructor.

4.9.3 Member Function Documentation

4.9.3.1 `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](#).

4.9.3.2 `virtual void AbcPrimalColumnPivot::updateWeights (CoinIndexedVector * input)` `[virtual]`

Updates weights - part 1 (may be empty)

Reimplemented in [AbcPrimalColumnSteepest](#).

4.9.3.3 `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](#).

4.9.3.4 `virtual int AbcPrimalColumnPivot::pivotRow (double & way)` `[inline],[virtual]`

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.

Definition at line 75 of file `AbcPrimalColumnPivot.hpp`.

4.9.3.5 `virtual void AbcPrimalColumnPivot::clearArrays ()` `[virtual]`

Gets rid of all arrays (may be empty)

Reimplemented in [AbcPrimalColumnSteepest](#).

4.9.3.6 `virtual bool AbcPrimalColumnPivot::looksOptimal () const` `[inline],[virtual]`

Returns true if would not find any column.

Reimplemented in [AbcPrimalColumnSteepest](#).

Definition at line 82 of file `AbcPrimalColumnPivot.hpp`.

4.9.3.7 `virtual void AbcPrimalColumnPivot::setLooksOptimal (bool flag)` `[inline],[virtual]`

Sets optimality flag (for advanced use)

Definition at line 86 of file `AbcPrimalColumnPivot.hpp`.

4.9.3.8 `AbcPrimalColumnPivot& AbcPrimalColumnPivot::operator= (const AbcPrimalColumnPivot & rhs)`

Assignment operator.

4.9.3.9 `virtual AbcPrimalColumnPivot* AbcPrimalColumnPivot::clone (bool copyData = true) const` [pure virtual]

Clone.

Implemented in [AbcPrimalColumnSteepest](#), and [AbcPrimalColumnDantzig](#).

4.9.3.10 `AbcSimplex* AbcPrimalColumnPivot::model ()` [inline]

Returns model.

Definition at line 114 of file `AbcPrimalColumnPivot.hpp`.

4.9.3.11 `void AbcPrimalColumnPivot::setModel (AbcSimplex * newmodel)` [inline]

Sets model.

Definition at line 118 of file `AbcPrimalColumnPivot.hpp`.

4.9.3.12 `int AbcPrimalColumnPivot::type ()` [inline]

Returns type (above 63 is extra information)

Definition at line 123 of file `AbcPrimalColumnPivot.hpp`.

4.9.3.13 `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

4.9.3.14 `virtual void AbcPrimalColumnPivot::switchOffSprint ()` [virtual]

Switch off sprint idea.

4.9.3.15 `virtual void AbcPrimalColumnPivot::maximumPivotsChanged ()` [inline],[virtual]

Called when maximum pivots changes.

Reimplemented in [AbcPrimalColumnSteepest](#).

Definition at line 134 of file `AbcPrimalColumnPivot.hpp`.

4.9.4 Member Data Documentation

4.9.4.1 `AbcSimplex* AbcPrimalColumnPivot::model_` [protected]

Pointer to model.

Definition at line 144 of file `AbcPrimalColumnPivot.hpp`.

4.9.4.2 `int AbcPrimalColumnPivot::type_` [protected]

Type of column pivot algorithm.

Definition at line 146 of file `AbcPrimalColumnPivot.hpp`.

4.9.4.3 `bool AbcPrimalColumnPivot::looksOptimal_` [protected]

Says if looks optimal (normally computed)

Definition at line 148 of file AbcPrimalColumnPivot.hpp.

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

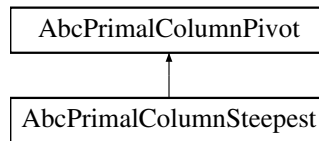
- src/[AbcPrimalColumnPivot.hpp](#)

4.10 **AbcPrimalColumnSteepest Class Reference**

Primal Column Pivot Steepest Edge Algorithm Class.

```
#include <AbcPrimalColumnSteepest.hpp>
```

Inheritance diagram for AbcPrimalColumnSteepest:



Public Types

- enum [Persistence](#) { [normal](#) = 0x00, [keep](#) = 0x01 }
- 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 Member Enumeration Documentation

4.10.2.1 enum `AbcPrimalColumnSteepest::Persistence`

enums for persistence

Enumerator

normal

keep

Definition at line 108 of file AbcPrimalColumnSteepest.hpp.

4.10.3 Constructor & Destructor Documentation

4.10.3.1 **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.

4.10.3.2 **AbcPrimalColumnSteepest::AbcPrimalColumnSteepest (const AbcPrimalColumnSteepest & *rhs*)**

Copy constructor.

4.10.3.3 **virtual AbcPrimalColumnSteepest::~~AbcPrimalColumnSteepest () [virtual]**

Destructor.

4.10.4 Member Function Documentation

4.10.4.1 **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](#).

4.10.4.2 **void AbcPrimalColumnSteepest::justDjs (CoinIndexedVector * *updates*, CoinIndexedVector * *spareColumn1*)**

Just update djs.

4.10.4.3 **int AbcPrimalColumnSteepest::partialPricing (CoinIndexedVector * *updates*, int *numberWanted*, int *numberLook*)**

Update djs doing partial pricing (dantzig)

4.10.4.4 **void AbcPrimalColumnSteepest::djsAndDevex (CoinIndexedVector * *updates*, CoinIndexedVector * *spareRow2*, CoinIndexedVector * *spareColumn1*)**

Update djs, weights for Devex using djs.

4.10.4.5 **void AbcPrimalColumnSteepest::djsAndDevex2 (CoinIndexedVector * *updates*, CoinIndexedVector * *spareColumn1*)**

Update djs, weights for Devex using pivot row.

4.10.4.6 `void AbcPrimalColumnSteepest::justDevex (CoinIndexedVector * updates, CoinIndexedVector * spareColumn1)`

Update weights for Devex.

4.10.4.7 `int AbcPrimalColumnSteepest::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.

4.10.4.8 `virtual void AbcPrimalColumnSteepest::updateWeights (CoinIndexedVector * input) [virtual]`

Updates weights - part 1 - also checks accuracy.

Reimplemented from [AbcPrimalColumnPivot](#).

4.10.4.9 `void AbcPrimalColumnSteepest::checkAccuracy (int sequence, double relativeTolerance, CoinIndexedVector * rowArray1)`

Checks accuracy - just for debug.

4.10.4.10 `void AbcPrimalColumnSteepest::initializeWeights ()`

Initialize weights.

4.10.4.11 `virtual void AbcPrimalColumnSteepest::saveWeights (AbcSimplex * model, int mode) [virtual]`

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)

Implements [AbcPrimalColumnPivot](#).

4.10.4.12 `virtual void AbcPrimalColumnSteepest::unrollWeights () [virtual]`

Gets rid of last update.

4.10.4.13 `virtual void AbcPrimalColumnSteepest::clearArrays () [virtual]`

Gets rid of all arrays.

Reimplemented from [AbcPrimalColumnPivot](#).

4.10.4.14 `virtual bool AbcPrimalColumnSteepest::looksOptimal () const [virtual]`

Returns true if would not find any column.

Reimplemented from [AbcPrimalColumnPivot](#).

4.10.4.15 `virtual void AbcPrimalColumnSteepest::maximumPivotsChanged () [virtual]`

Called when maximum pivots changes.

Reimplemented from [AbcPrimalColumnPivot](#).

4.10.4.16 `int AbcPrimalColumnSteepest::mode () const [inline]`

Mode.

Definition at line 101 of file `AbcPrimalColumnSteepest.hpp`.

4.10.4.17 **AbcPrimalColumnSteepest& AbcPrimalColumnSteepest::operator= (const AbcPrimalColumnSteepest & rhs)**

Assignment operator.

4.10.4.18 **virtual AbcPrimalColumnPivot* AbcPrimalColumnSteepest::clone (bool *copyData* = true) const** [virtual]

Clone.

Implements [AbcPrimalColumnPivot](#).

4.10.4.19 **bool AbcPrimalColumnSteepest::reference (int *i*) const** [inline]

reference would be faster using [AbcSimplex](#)'s `status_`, but I prefer to keep modularity.

Definition at line 143 of file `AbcPrimalColumnSteepest.hpp`.

4.10.4.20 **void AbcPrimalColumnSteepest::setReference (int *i*, bool *trueFalse*)** [inline]

Definition at line 146 of file `AbcPrimalColumnSteepest.hpp`.

4.10.4.21 **void AbcPrimalColumnSteepest::setPersistence (Persistence *life*)** [inline]

Set/ get persistence.

Definition at line 155 of file `AbcPrimalColumnSteepest.hpp`.

4.10.4.22 **Persistence AbcPrimalColumnSteepest::persistence () const** [inline]

Definition at line 158 of file `AbcPrimalColumnSteepest.hpp`.

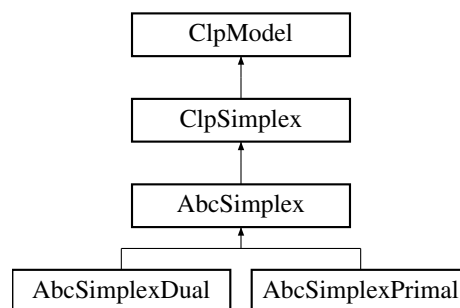
The documentation for this class was generated from the following file:

- [src/AbcPrimalColumnSteepest.hpp](#)

4.11 AbcSimplex Class Reference

```
#include <AbcSimplex.hpp>
```

Inheritance diagram for `AbcSimplex`:



Public Types

- enum [Status](#) {
[atLowerBound](#) = 0x00, [atUpperBound](#) = 0x01, [isFree](#) = 0x04, [superBasic](#) = 0x05,
[basic](#) = 0x06, [isFixed](#) = 0x07 }

enums for status of various sorts.

- enum `FakeBound` { `noFake` = 0x00, `lowerFake` = 0x01, `upperFake` = 0x02, `bothFake` = 0x03 }

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` ()
Clean up status.
- 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 numberOfRows, int numberOfColumns, bool doMore)
Initializes arrays.
- void `gutsOfResize` (int numberOfRows, int numberOfColumns)
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 70 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

Enumerator

atLowerBound
atUpperBound
isFree
superBasic
basic
isFixed

Definition at line 82 of file `AbcSimplex.hpp`.

4.11.2.2 enum `AbcSimplex::FakeBound`

Enumerator

noFake
lowerFake
upperFake
bothFake

Definition at line 91 of file `AbcSimplex.hpp`.

4.11.3 Constructor & Destructor Documentation

4.11.3.1 `AbcSimplex::AbcSimplex (bool emptyMessages = false)`

Default constructor.

4.11.3.2 `AbcSimplex::AbcSimplex (const AbcSimplex & rhs)`

Copy constructor.

4.11.3.3 `AbcSimplex::AbcSimplex (const ClpSimplex & rhs)`

Copy constructor from model.

4.11.3.4 `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 optionally modify rhs to take into account variables NOT in list in this case duplicates are not allowed (also see `getbackSolution`)

4.11.3.5 **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`)

4.11.3.6 **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`

4.11.3.7 **AbcSimplex::AbcSimplex** (**const ClpSimplex** * *clpSimplex*)

This constructor copies from [ClpSimplex](#).

4.11.3.8 **AbcSimplex::~~AbcSimplex** ()

Destructor.

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::putBackSolution** (**ClpSimplex** * *simplex*)

Put back solution into [ClpSimplex](#).

4.11.4.3 **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

4.11.4.4 **void AbcSimplex::deleteBaseModel** ()

Switch off base model.

4.11.4.5 **AbcSimplex* AbcSimplex::baseModel** () **const** [inline]

See if we have base model.

Definition at line 154 of file `AbcSimplex.hpp`.

4.11.4.6 **void AbcSimplex::setToBaseModel** (**AbcSimplex** * *model* = **NULL**)

Reset to base model (just size and arrays needed) If `model` `NULL` use internal copy.

4.11.4.7 **AbcSimplex& AbcSimplex::operator= (const AbcSimplex & rhs)**

Assignment operator. This copies the data.

4.11.4.8 **int AbcSimplex::dual ()**

Dual algorithm - see [AbcSimplexDual.hpp](#) for method.

4.11.4.9 **int AbcSimplex::doAbcDual ()**

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

Primal algorithm - see [AbcSimplexPrimal.hpp](#) for method.

4.11.4.11 **int AbcSimplex::doAbcPrimal (int ifValuesPass)**

4.11.4.12 **CoinWarmStartBasis* AbcSimplex::getBasis () const**

Returns a basis (to be deleted by user)

4.11.4.13 **void AbcSimplex::setFactorization (AbcSimplexFactorization & factorization)**

Passes in factorization.

4.11.4.14 **AbcSimplexFactorization* AbcSimplex::swapFactorization (AbcSimplexFactorization * factorization)**

Swaps factorization.

4.11.4.15 **AbcSimplexFactorization* AbcSimplex::getEmptyFactorization ()**

Gets clean and emptyish factorization.

4.11.4.16 **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)

4.11.4.17 **void AbcSimplex::setDualRowPivotAlgorithm (AbcDualRowPivot & choice)**

Sets row pivot choice algorithm in dual.

4.11.4.18 **void AbcSimplex::setPrimalColumnPivotAlgorithm (AbcPrimalColumnPivot & choice)**

Sets column pivot choice algorithm in primal.

4.11.4.19 **void AbcSimplex::defaultFactorizationFrequency ()**

If user left factorization frequency then compute.

4.11.4.20 **AbcSimplexFactorization* AbcSimplex::factorization () const** `[inline]`

factorization

Definition at line 207 of file AbcSimplex.hpp.

4.11.4.21 `int AbcSimplex::factorizationFrequency () const`

Factorization frequency.

4.11.4.22 `void AbcSimplex::setFactorizationFrequency (int value)`

4.11.4.23 `int AbcSimplex::maximumAbcNumberRows () const [inline]`

Maximum rows.

Definition at line 220 of file AbcSimplex.hpp.

4.11.4.24 `int AbcSimplex::maximumNumberTotal () const [inline]`

Maximum Total.

Definition at line 223 of file AbcSimplex.hpp.

4.11.4.25 `int AbcSimplex::maximumTotal () const [inline]`

Definition at line 225 of file AbcSimplex.hpp.

4.11.4.26 `bool AbcSimplex::isObjectiveLimitTestValid () const`

Return true if the objective limit test can be relied upon.

4.11.4.27 `int AbcSimplex::numberTotal () const [inline]`

Number of variables (includes spare rows)

Definition at line 230 of file AbcSimplex.hpp.

4.11.4.28 `int AbcSimplex::numberTotalWithoutFixed () const [inline]`

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

Definition at line 233 of file AbcSimplex.hpp.

4.11.4.29 `CoinPartitionedVector* AbcSimplex::usefulArray (int index) [inline]`

Useful arrays (0,1,2,3,4,5,6,7)

Definition at line 236 of file AbcSimplex.hpp.

4.11.4.30 `CoinPartitionedVector* AbcSimplex::usefulArray (int index) const [inline]`

Definition at line 239 of file AbcSimplex.hpp.

4.11.4.31 `int AbcSimplex::getSolution ()`

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

Uses current problem arrays for bounds. Returns feasibility states

4.11.4.32 `void AbcSimplex::setClpSimplexObjectiveValue ()`

Sets objectiveValue_ from rawObjectiveValue_.

4.11.4.33 `void AbcSimplex::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.

4.11.4.34 `double AbcSimplex::minimizationObjectiveValue () const [inline]`

Gets objective value with all offsets but as for minimization.

Definition at line 262 of file AbcSimplex.hpp.

4.11.4.35 `double AbcSimplex::currentDualTolerance () const [inline]`

Current dualTolerance (will end up as dualTolerance_)

Definition at line 265 of file AbcSimplex.hpp.

4.11.4.36 `void AbcSimplex::setCurrentDualTolerance (double value) [inline]`

Definition at line 267 of file AbcSimplex.hpp.

4.11.4.37 `AbcNonLinearCost* AbcSimplex::abcNonLinearCost () const [inline]`

Return pointer to details of costs.

Definition at line 271 of file AbcSimplex.hpp.

4.11.4.38 `double* AbcSimplex::perturbationSaved () const [inline]`

Perturbation (fixed) - is just scaled random numbers.

Definition at line 275 of file AbcSimplex.hpp.

4.11.4.39 `double AbcSimplex::acceptablePivot () const [inline]`

Acceptable pivot for this iteration.

Definition at line 278 of file AbcSimplex.hpp.

4.11.4.40 `int AbcSimplex::ordinaryVariables () const [inline]`

Set to 1 if no free or super basic.

Definition at line 281 of file AbcSimplex.hpp.

4.11.4.41 `int AbcSimplex::numberOrdinary () const [inline]`

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

Definition at line 284 of file AbcSimplex.hpp.

4.11.4.42 `void AbcSimplex::setNumberOrdinary (int number) [inline]`

Set number of ordinary (lo/up) in tableau row.

Definition at line 287 of file AbcSimplex.hpp.

4.11.4.43 `double AbcSimplex::currentDualBound () const [inline]`

Current dualBound (will end up as dualBound_)

Definition at line 290 of file AbcSimplex.hpp.

4.11.4.44 **AbcDualRowPivot*** **AbcSimplex::dualRowPivot** () const [inline]

dual row pivot choice

Definition at line 293 of file AbcSimplex.hpp.

4.11.4.45 **AbcPrimalColumnPivot*** **AbcSimplex::primalColumnPivot** () const [inline]

primal column pivot choice

Definition at line 297 of file AbcSimplex.hpp.

4.11.4.46 **AbcMatrix*** **AbcSimplex::abcMatrix** () const [inline]

Abc Matrix.

Definition at line 301 of file AbcSimplex.hpp.

4.11.4.47 **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 [AbcSimplex-Factorization](#) unless initial factorization when total number of singularities is returned. Special case is *numberRows_+1* -> all slack basis. if initial should be before permute in *pivotVariable* may be same as *toExternal*

4.11.4.48 **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

4.11.4.49 **void** **AbcSimplex::permuteBasis** ()

deals with new basis and puts in *abcPivotVariable_*

4.11.4.50 **void** **AbcSimplex::permuteOut** (int *whatsWanted*)

Permutes out - bit settings same as *stateOfProblem*.

4.11.4.51 **ClpDataSave** **AbcSimplex::saveData** ()

Save data.

4.11.4.52 **void** **AbcSimplex::restoreData** (**ClpDataSave** *saved*)

Restore data.

4.11.4.53 `void AbcSimplex::cleanStatus ()`

Clean up status.

4.11.4.54 `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

4.11.4.55 `int AbcSimplex::computePrimals (CoinIndexedVector * array1, CoinIndexedVector * array2)`

Computes primals from scratch. Returns number of refinements.

4.11.4.56 `void AbcSimplex::computeObjective ()`

Computes nonbasic cost and total cost.

4.11.4.57 `void AbcSimplex::setMultipleSequenceIn (int sequenceIn[4])`

set multiple sequence in

4.11.4.58 `void AbcSimplex::unpack (CoinIndexedVector & rowArray) const [inline]`

Unpacks one column of the matrix into indexed array Uses `sequenceIn_`.

Definition at line 353 of file `AbcSimplex.hpp`.

4.11.4.59 `void AbcSimplex::unpack (CoinIndexedVector & rowArray, int sequence) const`

Unpacks one column of the matrix into indexed array.

4.11.4.60 `int AbcSimplex::housekeeping ()`

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

Can also decide to re-factorize

4.11.4.61 `void AbcSimplex::checkPrimalSolution (bool justBasic)`

This sets largest infeasibility and most infeasible and sum and number of infeasibilities (Primal)

4.11.4.62 `void AbcSimplex::checkDualSolution ()`

This sets largest infeasibility and most infeasible and sum and number of infeasibilities (Dual)

4.11.4.63 `void AbcSimplex::checkDualSolutionPlusFake ()`

This sets largest infeasibility and most infeasible and sum and number of infeasibilities AND `sumFakeInfeasibilites_` (Dual)

4.11.4.64 `void AbcSimplex::checkBothSolutions ()`

This sets sum and number of infeasibilities (Dual and Primal)

4.11.4.65 `int AbcSimplex::gutsOfSolution (int type)`

Computes solutions - 1 do duals, 2 do primals, 3 both (returns number of refinements)

4.11.4.66 `int AbcSimplex::gutsOfPrimalSolution (int type)`

Computes solutions - 1 do duals, 2 do primals, 3 both (returns number of refinements)

4.11.4.67 `void AbcSimplex::saveGoodStatus ()`

Saves good status etc.

4.11.4.68 `void AbcSimplex::restoreGoodStatus (int type)`

Restores previous good status and says trouble.

4.11.4.69 `void AbcSimplex::refreshCosts ()`

After modifying first copy refreshes second copy and marks as updated.

4.11.4.70 `void AbcSimplex::refreshLower (unsigned int type = ~ (ROW_LOWER_SAME|COLUMN_UPPER_SAME))`

4.11.4.71 `void AbcSimplex::refreshUpper (unsigned int type = ~ (ROW_LOWER_SAME|COLUMN_LOWER_SAME))`

4.11.4.72 `void AbcSimplex::setupPointers (int maxRows, int maxColumns)`

Sets up all extra pointers.

4.11.4.73 `void AbcSimplex::copyFromSaved (int type = 31)`

Copies all saved versions to working versions and may do something for perturbation.

4.11.4.74 `void AbcSimplex::fillPerturbation (int start, int number)`

fills in perturbationSaved_ from start with 0.5+random

4.11.4.75 `void AbcSimplex::checkArrays (int ignoreEmpty = 0) const`

For debug - prints summary of arrays which are out of kilter.

4.11.4.76 `void AbcSimplex::checkDjs (int type = 1) const`

For debug - summarizes dj situation (1 recomputes duals first, 2 checks duals as well)

4.11.4.77 `void AbcSimplex::checkSolutionBasic () const`

For debug - checks solutionBasic.

4.11.4.78 `void AbcSimplex::checkMoveBack (bool checkDuals)`

For debug - moves solution back to external and computes stuff (always checks djs)

4.11.4.79 `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 although you may end up with an all slack basis.

Defaults are 1.0,10.0

4.11.4.80 `int` `AbcSimplex::cleanFactorization` (`int` *ifValuesPass*)

Get a clean factorization - i.e.

throw out singularities may do more later

4.11.4.81 `void` `AbcSimplex::moveStatusToClp` (`ClpSimplex` * *clpModel*)

Move status and solution to [ClpSimplex](#).

4.11.4.82 `void` `AbcSimplex::moveStatusFromClp` (`ClpSimplex` * *clpModel*)

Move status and solution from [ClpSimplex](#).

4.11.4.83 `double` `AbcSimplex::clpObjectiveValue` () `const` [`inline`]

Objective value.

Definition at line 429 of file `AbcSimplex.hpp`.

4.11.4.84 `int*` `AbcSimplex::pivotVariable` () `const` [`inline`]

Basic variables pivoting on which rows may be same as `toExternal` but may be as at invert.

Definition at line 434 of file `AbcSimplex.hpp`.

4.11.4.85 `int` `AbcSimplex::stateOfProblem` () `const` [`inline`]

State of problem.

Definition at line 438 of file `AbcSimplex.hpp`.

4.11.4.86 `void` `AbcSimplex::setStateOfProblem` (`int` *value*) [`inline`]

State of problem.

Definition at line 441 of file `AbcSimplex.hpp`.

4.11.4.87 `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 451 of file `AbcSimplex.hpp`.

4.11.4.88 `double*` `AbcSimplex::scaleToExternal` () `const` [`inline`]

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

Definition at line 455 of file `AbcSimplex.hpp`.

4.11.4.89 `double*` `AbcSimplex::rowScale2` () `const` [`inline`]

corresponds to `rowScale` etc

Definition at line 458 of file `AbcSimplex.hpp`.

4.11.4.90 `double*` `AbcSimplex::inverseRowScale2` () `const` [`inline`]

Definition at line 460 of file `AbcSimplex.hpp`.

4.11.4.91 `double* AbcSimplex::inverseColumnScale2 () const [inline]`

Definition at line 462 of file AbcSimplex.hpp.

4.11.4.92 `double* AbcSimplex::columnScale2 () const [inline]`

Definition at line 464 of file AbcSimplex.hpp.

4.11.4.93 `int AbcSimplex::arrayForDualColumn () const [inline]`

Definition at line 466 of file AbcSimplex.hpp.

4.11.4.94 `double AbcSimplex::upperTheta () const [inline]`

upper theta from dual column

Definition at line 469 of file AbcSimplex.hpp.

4.11.4.95 `int AbcSimplex::arrayForReplaceColumn () const [inline]`

Definition at line 471 of file AbcSimplex.hpp.

4.11.4.96 `int AbcSimplex::arrayForFlipBounds () const [inline]`

Definition at line 473 of file AbcSimplex.hpp.

4.11.4.97 `int AbcSimplex::arrayForFlipRhs () const [inline]`

Definition at line 475 of file AbcSimplex.hpp.

4.11.4.98 `int AbcSimplex::arrayForBtran () const [inline]`

Definition at line 477 of file AbcSimplex.hpp.

4.11.4.99 `int AbcSimplex::arrayForFtran () const [inline]`

Definition at line 479 of file AbcSimplex.hpp.

4.11.4.100 `int AbcSimplex::arrayForTableauRow () const [inline]`

Definition at line 481 of file AbcSimplex.hpp.

4.11.4.101 `double AbcSimplex::valueIncomingDual () const`

value of incoming variable (in Dual)

4.11.4.102 `const double* AbcSimplex::getColSolution () const`

Get pointer to array[[getNumCols\(\)](#)] of primal solution vector.

4.11.4.103 `const double* AbcSimplex::getRowPrice () const`

Get pointer to array[[getNumRows\(\)](#)] of dual prices.

4.11.4.104 `const double* AbcSimplex::getReducedCost () const`

Get a pointer to array[[getNumCols\(\)](#)] of reduced costs.

4.11.4.105 `const double* AbcSimplex::getRowActivity () const`

Get pointer to array[[getNumRows\(\)](#)] of row activity levels (constraint matrix times the solution vector).

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

May change basis and then returns number changed.

Computation of solutions may be overridden by given pi and solution

4.11.4.107 `void AbcSimplex::gutsOfDelete (int type)`

Does most of deletion for arrays etc(0 just null arrays, 1 delete first)

4.11.4.108 `void AbcSimplex::gutsOfCopy (const AbcSimplex & rhs)`

Does most of copying.

4.11.4.109 `void AbcSimplex::gutsOfInitialize (int numberRows, int numberColumns, bool doMore)`

Initializes arrays.

4.11.4.110 `void AbcSimplex::gutsOfResize (int numberRows, int numberColumns)`

resizes arrays

4.11.4.111 `void AbcSimplex::translate (int type)`

Translates [ClpModel](#) to [AbcSimplex](#) See DO_ bits in stateOfProblem_ for type e.g.

DO_BASIS_AND_ORDER

4.11.4.112 `void AbcSimplex::moveToBasic (int which = 15)`

Moves basic stuff to basic area.

4.11.4.113 `double* AbcSimplex::solutionRegion () const` `[inline]`

Return region.

Definition at line 526 of file [AbcSimplex.hpp](#).

4.11.4.114 `double* AbcSimplex::djRegion () const` `[inline]`

Definition at line 529 of file [AbcSimplex.hpp](#).

4.11.4.115 `double* AbcSimplex::lowerRegion () const` `[inline]`

Definition at line 532 of file [AbcSimplex.hpp](#).

4.11.4.116 `double* AbcSimplex::upperRegion () const` `[inline]`

Definition at line 535 of file [AbcSimplex.hpp](#).

4.11.4.117 `double* AbcSimplex::costRegion () const` `[inline]`

Definition at line 538 of file [AbcSimplex.hpp](#).

4.11.4.118 `double* AbcSimplex::solutionRegion (int which) const` `[inline]`

Return region.

Definition at line 542 of file AbcSimplex.hpp.

4.11.4.119 `double* AbcSimplex::djRegion (int which) const` `[inline]`

Definition at line 545 of file AbcSimplex.hpp.

4.11.4.120 `double* AbcSimplex::lowerRegion (int which) const` `[inline]`

Definition at line 548 of file AbcSimplex.hpp.

4.11.4.121 `double* AbcSimplex::upperRegion (int which) const` `[inline]`

Definition at line 551 of file AbcSimplex.hpp.

4.11.4.122 `double* AbcSimplex::costRegion (int which) const` `[inline]`

Definition at line 554 of file AbcSimplex.hpp.

4.11.4.123 `double* AbcSimplex::solutionBasic () const` `[inline]`

Return region.

Definition at line 558 of file AbcSimplex.hpp.

4.11.4.124 `double* AbcSimplex::djBasic () const` `[inline]`

Definition at line 561 of file AbcSimplex.hpp.

4.11.4.125 `double* AbcSimplex::lowerBasic () const` `[inline]`

Definition at line 564 of file AbcSimplex.hpp.

4.11.4.126 `double* AbcSimplex::upperBasic () const` `[inline]`

Definition at line 567 of file AbcSimplex.hpp.

4.11.4.127 `double* AbcSimplex::costBasic () const` `[inline]`

Definition at line 570 of file AbcSimplex.hpp.

4.11.4.128 `double* AbcSimplex::abcPerturbation () const` `[inline]`

Perturbation.

Definition at line 574 of file AbcSimplex.hpp.

4.11.4.129 `double* AbcSimplex::fakeDjs () const` `[inline]`

Fake djs.

Definition at line 577 of file AbcSimplex.hpp.

4.11.4.130 `unsigned char* AbcSimplex::internalStatus () const` `[inline]`

Definition at line 579 of file AbcSimplex.hpp.

4.11.4.131 **AbcSimplex::Status** AbcSimplex::getInternalStatus (int *sequence*) const [inline]

Definition at line 581 of file AbcSimplex.hpp.

4.11.4.132 **AbcSimplex::Status** AbcSimplex::getInternalColumnStatus (int *sequence*) const [inline]

Definition at line 584 of file AbcSimplex.hpp.

4.11.4.133 void AbcSimplex::setInternalStatus (int *sequence*, **AbcSimplex::Status** *newstatus*) [inline]

Definition at line 587 of file AbcSimplex.hpp.

4.11.4.134 void AbcSimplex::setInternalColumnStatus (int *sequence*, **AbcSimplex::Status** *newstatus*) [inline]

Definition at line 592 of file AbcSimplex.hpp.

4.11.4.135 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

4.11.4.136 bool AbcSimplex::initialDenseFactorization () const

4.11.4.137 int AbcSimplex::sequenceIn () const [inline]

Return sequence In or Out.

Definition at line 604 of file AbcSimplex.hpp.

4.11.4.138 int AbcSimplex::sequenceOut () const [inline]

Definition at line 607 of file AbcSimplex.hpp.

4.11.4.139 void AbcSimplex::setSequenceIn (int *sequence*) [inline]

Set sequenceIn or Out.

Definition at line 611 of file AbcSimplex.hpp.

4.11.4.140 void AbcSimplex::setSequenceOut (int *sequence*) [inline]

Definition at line 614 of file AbcSimplex.hpp.

4.11.4.141 int AbcSimplex::isColumn (int *sequence*) const [inline]

Returns 1 if sequence indicates column.

Definition at line 634 of file AbcSimplex.hpp.

4.11.4.142 int AbcSimplex::sequenceWithin (int *sequence*) const [inline]

Returns sequence number within section.

Definition at line 638 of file AbcSimplex.hpp.

4.11.4.143 int AbcSimplex::lastPivotRow () const [inline]

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

Definition at line 642 of file AbcSimplex.hpp.

4.11.4.144 `int AbcSimplex::firstFree () const [inline]`

First Free_.

Definition at line 645 of file AbcSimplex.hpp.

4.11.4.145 `int AbcSimplex::lastFirstFree () const [inline]`

Last firstFree_.

Definition at line 648 of file AbcSimplex.hpp.

4.11.4.146 `int AbcSimplex::freeSequenceIn () const [inline]`

Free chosen vector.

Definition at line 651 of file AbcSimplex.hpp.

4.11.4.147 `double AbcSimplex::currentAcceptablePivot () const [inline]`

Acceptable pivot for this iteration.

Definition at line 654 of file AbcSimplex.hpp.

4.11.4.148 `int AbcSimplex::fakeSuperBasic (int iSequence) [inline]`

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.

Definition at line 663 of file AbcSimplex.hpp.

4.11.4.149 `double AbcSimplex::solution (int sequence) [inline]`

Return row or column values.

Definition at line 695 of file AbcSimplex.hpp.

4.11.4.150 `double& AbcSimplex::solutionAddress (int sequence) [inline]`

Return address of row or column values.

Definition at line 699 of file AbcSimplex.hpp.

4.11.4.151 `double AbcSimplex::reducedCost (int sequence) [inline]`

Definition at line 702 of file AbcSimplex.hpp.

4.11.4.152 `double& AbcSimplex::reducedCostAddress (int sequence) [inline]`

Definition at line 705 of file AbcSimplex.hpp.

4.11.4.153 `double AbcSimplex::lower (int sequence) [inline]`

Definition at line 708 of file AbcSimplex.hpp.

4.11.4.154 `double& AbcSimplex::lowerAddress (int sequence) [inline]`

Return address of row or column lower bound.

Definition at line 712 of file `AbcSimplex.hpp`.

4.11.4.155 `double AbcSimplex::upper (int sequence) [inline]`

Definition at line 715 of file `AbcSimplex.hpp`.

4.11.4.156 `double& AbcSimplex::upperAddress (int sequence) [inline]`

Return address of row or column upper bound.

Definition at line 719 of file `AbcSimplex.hpp`.

4.11.4.157 `double AbcSimplex::cost (int sequence) [inline]`

Definition at line 722 of file `AbcSimplex.hpp`.

4.11.4.158 `double& AbcSimplex::costAddress (int sequence) [inline]`

Return address of row or column cost.

Definition at line 726 of file `AbcSimplex.hpp`.

4.11.4.159 `double AbcSimplex::originalLower (int iSequence) const [inline]`

Return original lower bound.

Definition at line 730 of file `AbcSimplex.hpp`.

4.11.4.160 `double AbcSimplex::originalUpper (int iSequence) const [inline]`

Return original lower bound.

Definition at line 736 of file `AbcSimplex.hpp`.

4.11.4.161 `AbcSimplexProgress* AbcSimplex::abcProgress () [inline]`

For dealing with all issues of cycling etc.

Definition at line 742 of file `AbcSimplex.hpp`.

4.11.4.162 `void AbcSimplex::clearArraysPublic (int which) [inline]`

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

Definition at line 747 of file `AbcSimplex.hpp`.

4.11.4.163 `int AbcSimplex::getAvailableArrayPublic () const [inline]`

Returns first available empty array (and sets flag) when no possibility of going parallel.

Definition at line 751 of file `AbcSimplex.hpp`.

4.11.4.164 `void AbcSimplex::clearArrays (int which)`

Clears an array and says available (-1 does all)

4.11.4.165 `void AbcSimplex::clearArrays (CoinPartitionedVector * which)`

Clears an array and says available.

4.11.4.166 `int AbcSimplex::getAvailableArray () const`

Returns first available empty array (and sets flag)

4.11.4.167 `void AbcSimplex::setUsedArray (int which) const` `[inline]`

Say array going to be used.

Definition at line 777 of file AbcSimplex.hpp.

4.11.4.168 `void AbcSimplex::setAvailableArray (int which) const` `[inline]`

Say array going available.

Definition at line 780 of file AbcSimplex.hpp.

4.11.4.169 `void AbcSimplex::swapPrimalStuff ()`

Swaps primal stuff.

4.11.4.170 `void AbcSimplex::swapDualStuff (int lastSequenceOut, int lastDirectionOut)`

Swaps dual stuff.

4.11.4.171 `void AbcSimplex::swap (int pivotRow, int nonBasicPosition, Status newStatus)` `[protected]`

Swaps two variables and does status.

4.11.4.172 `void AbcSimplex::setFakeBound (int sequence, FakeBound fakeBound)` `[inline]`, `[protected]`

Definition at line 793 of file AbcSimplex.hpp.

4.11.4.173 `FakeBound AbcSimplex::getFakeBound (int sequence) const` `[inline]`, `[protected]`

Definition at line 798 of file AbcSimplex.hpp.

4.11.4.174 `bool AbcSimplex::atFakeBound (int sequence) const` `[protected]`

4.11.4.175 `void AbcSimplex::setPivoted (int sequence)` `[inline]`, `[protected]`

Definition at line 802 of file AbcSimplex.hpp.

4.11.4.176 `void AbcSimplex::clearPivoted (int sequence)` `[inline]`, `[protected]`

Definition at line 805 of file AbcSimplex.hpp.

4.11.4.177 `bool AbcSimplex::pivoted (int sequence) const` `[inline]`, `[protected]`

Definition at line 808 of file AbcSimplex.hpp.

4.11.4.178 `void AbcSimplex::swap (int pivotRow, int nonBasicPosition)`

Swaps two variables.

4.11.4.179 `void AbcSimplex::setFlagged (int sequence)`

To flag a variable.

4.11.4.180 `void AbcSimplex::clearFlagged (int sequence) [inline]`

Definition at line 816 of file AbcSimplex.hpp.

4.11.4.181 `bool AbcSimplex::flagged (int sequence) const [inline]`

Definition at line 819 of file AbcSimplex.hpp.

4.11.4.182 `void AbcSimplex::setActive (int iRow) [inline],[protected]`

To say row active in primal pivot row choice.

Definition at line 824 of file AbcSimplex.hpp.

4.11.4.183 `void AbcSimplex::clearActive (int iRow) [inline],[protected]`

Definition at line 827 of file AbcSimplex.hpp.

4.11.4.184 `bool AbcSimplex::active (int iRow) const [inline],[protected]`

Definition at line 830 of file AbcSimplex.hpp.

4.11.4.185 `void AbcSimplex::createStatus ()`

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

Also can be used to set up all slack basis

4.11.4.186 `void AbcSimplex::crash (int type)`

Does sort of crash.

4.11.4.187 `void AbcSimplex::putStuffInBasis (int type)`

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

4.11.4.188 `void AbcSimplex::allSlackBasis ()`

Sets up all slack basis and resets solution to as it was after initial load or readMps.

4.11.4.189 `void AbcSimplex::checkConsistentPivots () const`

For debug - check pivotVariable consistent.

4.11.4.190 `void AbcSimplex::printStuff () const`

Print stuff.

4.11.4.191 `int AbcSimplex::startup (int ifValuesPass)`

Common bits of coding for dual and primal.

4.11.4.192 `double AbcSimplex::rawObjectiveValue () const [inline]`

Raw objective value (so always minimize in primal)

Definition at line 855 of file AbcSimplex.hpp.

4.11.4.193 void AbcSimplex::computeObjectiveValue (bool *useWorkingSolution* = false)

Compute objective value from solution and put in objectiveValue_.

4.11.4.194 double AbcSimplex::computeInternalObjectiveValue ()

Compute minimization objective value from internal solution without perturbation.

4.11.4.195 void AbcSimplex::moveInfo (const AbcSimplex & *rhs*, bool *justStatus* = false)

Move status and solution across.

4.11.4.196 void AbcSimplex::setObjectiveCoefficient (int *elementIndex*, double *elementValue*)

Set an objective function coefficient.

4.11.4.197 void AbcSimplex::setObjCoeff (int *elementIndex*, double *elementValue*) [inline]

Set an objective function coefficient.

Definition at line 891 of file AbcSimplex.hpp.

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

Set a single column lower bound

Use -DBL_MAX for -infinity.

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

Set a single column upper bound

Use DBL_MAX for infinity.

4.11.4.200 void AbcSimplex::setColumnBounds (int *elementIndex*, double *lower*, double *upper*)

Set a single column lower and upper bound.

4.11.4.201 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

<i>indexFirst, index-Last</i>	pointers to the beginning and after the end of the array of the indices of the variables whose <i>either</i> bound changes
<i>boundList</i>	the new lower/upper bound pairs for the variables

4.11.4.202 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.

4.11.4.203 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`.

4.11.4.204 void `AbcSimplex::setColBounds` (int *elementIndex*, double *newlower*, double *newupper*) [inline]

Set a single column lower and upper bound.

Definition at line 931 of file `AbcSimplex.hpp`.

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

Set the bounds on a number of columns simultaneously

Parameters

<i>indexFirst, index-Last</i>	pointers to the beginning and after the end of the array of the indices of the variables whose <i>either</i> bound changes
<i>boundList</i>	the new lower/upper bound pairs for the variables

Definition at line 942 of file `AbcSimplex.hpp`.

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

Set a single row lower bound

Use -DBL_MAX for -infinity.

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

Set a single row upper bound

Use DBL_MAX for infinity.

4.11.4.208 void `AbcSimplex::setRowBounds` (int *elementIndex*, double *lower*, double *upper*)

Set a single row lower and upper bound.

4.11.4.209 void `AbcSimplex::setRowSetBounds` (const int * *indexFirst*, const int * *indexLast*, const double * *boundList*)

Set the bounds on a number of rows simultaneously

Parameters

<i>indexFirst, index-Last</i>	pointers to the beginning and after the end of the array of the indices of the constraints whose <i>either</i> bound changes
<i>boundList</i>	the new lower/upper bound pairs for the constraints

4.11.4.210 void `AbcSimplex::resize` (int *newNumberRows*, int *newNumberColumns*)

Resizes rim part of model.

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::sumNonBasicCosts_ [protected]`

Sum of nonbasic costs.

Definition at line 984 of file [AbcSimplex.hpp](#).

4.11.6.2 `double AbcSimplex::rawObjectiveValue_ [protected]`

Sum of costs (raw objective value)

Definition at line 986 of file [AbcSimplex.hpp](#).

4.11.6.3 `double AbcSimplex::objectiveOffset_ [protected]`

Objective offset (from offset_)

Definition at line 988 of file [AbcSimplex.hpp](#).

4.11.6.4 `double AbcSimplex::perturbationFactor_ [protected]`

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

Definition at line 993 of file [AbcSimplex.hpp](#).

4.11.6.5 `double AbcSimplex::currentDualTolerance_ [protected]`

Current dualTolerance (will end up as dualTolerance_)

Definition at line 995 of file [AbcSimplex.hpp](#).

4.11.6.6 `double AbcSimplex::currentDualBound_ [protected]`

Current dualBound (will end up as dualBound_)

Definition at line 997 of file [AbcSimplex.hpp](#).

4.11.6.7 `double AbcSimplex::largestGap_ [protected]`

Largest gap.

Definition at line 999 of file [AbcSimplex.hpp](#).

4.11.6.8 `double AbcSimplex::lastDualBound_ [protected]`

Last dual bound.

Definition at line 1001 of file [AbcSimplex.hpp](#).

4.11.6.9 double `AbcSimplex::sumFakeInfeasibilities_` [protected]

Sum of infeasibilities when using fake perturbation tolerance.

Definition at line 1003 of file `AbcSimplex.hpp`.

4.11.6.10 double `AbcSimplex::lastPrimalError_` [protected]

Last primal error.

Definition at line 1005 of file `AbcSimplex.hpp`.

4.11.6.11 double `AbcSimplex::lastDualError_` [protected]

Last dual error.

Definition at line 1007 of file `AbcSimplex.hpp`.

4.11.6.12 double `AbcSimplex::currentAcceptablePivot_` [protected]

Acceptable pivot for this iteration.

Definition at line 1009 of file `AbcSimplex.hpp`.

4.11.6.13 double `AbcSimplex::movement_` [protected]

Movement of variable.

Definition at line 1011 of file `AbcSimplex.hpp`.

4.11.6.14 double `AbcSimplex::objectiveChange_` [protected]

Objective change.

Definition at line 1013 of file `AbcSimplex.hpp`.

4.11.6.15 double `AbcSimplex::btranAlpha_` [protected]

Btran alpha.

Definition at line 1015 of file `AbcSimplex.hpp`.

4.11.6.16 double `AbcSimplex::ftAlpha_` [protected]

FT alpha.

Definition at line 1020 of file `AbcSimplex.hpp`.

4.11.6.17 double `AbcSimplex::minimumThetaMovement_` [protected]

Minimum theta movement.

Definition at line 1022 of file `AbcSimplex.hpp`.

4.11.6.18 double `AbcSimplex::initialSumInfeasibilities_` [protected]

Initial sum of infeasibilities.

Definition at line 1024 of file `AbcSimplex.hpp`.

4.11.6.19 int AbcSimplex::stateOfIteration_

Where we are in iteration.

Definition at line 1027 of file AbcSimplex.hpp.

4.11.6.20 int AbcSimplex::lastFirstFree_ [protected]

Last firstFree_.

Definition at line 1030 of file AbcSimplex.hpp.

4.11.6.21 int AbcSimplex::freeSequenceIn_ [protected]

Free chosen vector.

Definition at line 1032 of file AbcSimplex.hpp.

4.11.6.22 int AbcSimplex::maximumAbcNumberRows_ [protected]

Maximum number rows.

Definition at line 1034 of file AbcSimplex.hpp.

4.11.6.23 int AbcSimplex::maximumAbcNumberColumns_ [protected]

Maximum number columns.

Definition at line 1036 of file AbcSimplex.hpp.

4.11.6.24 int AbcSimplex::maximumNumberTotal_ [protected]

Maximum numberTotal.

Definition at line 1038 of file AbcSimplex.hpp.

4.11.6.25 int AbcSimplex::numberFlagged_ [protected]

Current number of variables flagged.

Definition at line 1040 of file AbcSimplex.hpp.

4.11.6.26 int AbcSimplex::normalDualColumnIteration_ [protected]

Iteration at which to do relaxed dualColumn.

Definition at line 1042 of file AbcSimplex.hpp.

4.11.6.27 int AbcSimplex::stateDualColumn_ [protected]

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.

Definition at line 1048 of file AbcSimplex.hpp.

4.11.6.28 int AbcSimplex::numberTotal_ [protected]

Number of variables (includes spare rows)

Definition at line 1055 of file AbcSimplex.hpp.

4.11.6.29 `int AbcSimplex::numberTotalWithoutFixed_` [protected]

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

Definition at line 1057 of file `AbcSimplex.hpp`.

4.11.6.30 `int AbcSimplex::startAtLowerOther_` [protected]

Start of variables at lower bound with upper.

Definition at line 1061 of file `AbcSimplex.hpp`.

4.11.6.31 `int AbcSimplex::startAtUpperNoOther_` [protected]

Start of variables at upper bound with no lower.

Definition at line 1063 of file `AbcSimplex.hpp`.

4.11.6.32 `int AbcSimplex::startAtUpperOther_` [protected]

Start of variables at upper bound with lower.

Definition at line 1065 of file `AbcSimplex.hpp`.

4.11.6.33 `int AbcSimplex::startOther_` [protected]

Start of superBasic, free or awkward bounds variables.

Definition at line 1067 of file `AbcSimplex.hpp`.

4.11.6.34 `int AbcSimplex::startFixed_` [protected]

Start of fixed variables.

Definition at line 1069 of file `AbcSimplex.hpp`.

4.11.6.35 `int AbcSimplex::stateOfProblem_` [mutable], [protected]

Definition at line 1107 of file `AbcSimplex.hpp`.

4.11.6.36 `int AbcSimplex::numberOrdinary_` [protected]

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

Definition at line 1115 of file `AbcSimplex.hpp`.

4.11.6.37 `int AbcSimplex::ordinaryVariables_` [protected]

Set to 1 if no free or super basic.

Definition at line 1117 of file `AbcSimplex.hpp`.

4.11.6.38 `int AbcSimplex::numberFreeNonBasic_` [protected]

Number of free nonbasic variables.

Definition at line 1119 of file `AbcSimplex.hpp`.

4.11.6.39 `int AbcSimplex::lastCleaned_` [protected]

Last time cleaned up.

Definition at line 1121 of file AbcSimplex.hpp.

4.11.6.40 `int AbcSimplex::lastPivotRow_` `[protected]`

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

Definition at line 1123 of file AbcSimplex.hpp.

4.11.6.41 `int AbcSimplex::swappedAlgorithm_` `[protected]`

Nonzero (probably 10) if swapped algorithms.

Definition at line 1125 of file AbcSimplex.hpp.

4.11.6.42 `int AbcSimplex::initialNumberInfeasibilities_` `[protected]`

Initial number of infeasibilities.

Definition at line 1127 of file AbcSimplex.hpp.

4.11.6.43 `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.

4.11.6.44 `double* AbcSimplex::scaleToExternal_` `[protected]`

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

Definition at line 1137 of file AbcSimplex.hpp.

4.11.6.45 `double* AbcSimplex::columnUseScale_` `[protected]`

use this instead of columnScale

Definition at line 1139 of file AbcSimplex.hpp.

4.11.6.46 `double* AbcSimplex::inverseColumnUseScale_` `[protected]`

use this instead of inverseColumnScale

Definition at line 1141 of file AbcSimplex.hpp.

4.11.6.47 `double* AbcSimplex::offset_` `[protected]`

Primal offset (in external order) So internal value is (external-offset)*scaleFromExternal.

Definition at line 1145 of file AbcSimplex.hpp.

4.11.6.48 `double* AbcSimplex::offsetRhs_` `[protected]`

Offset for accumulated offsets*matrix.

Definition at line 1147 of file AbcSimplex.hpp.

4.11.6.49 `double* AbcSimplex::tempArray_` `[protected]`

Useful array of numberTotal length.

Definition at line 1149 of file AbcSimplex.hpp.

4.11.6.50 `unsigned char* AbcSimplex::internalStatus_` [protected]

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

Definition at line 1154 of file AbcSimplex.hpp.

4.11.6.51 `unsigned char* AbcSimplex::internalStatusSaved_` [protected]

Saved status.

Definition at line 1156 of file AbcSimplex.hpp.

4.11.6.52 `double* AbcSimplex::abcPerturbation_` [protected]

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

Definition at line 1159 of file AbcSimplex.hpp.

4.11.6.53 `double* AbcSimplex::perturbationSaved_` [protected]

saved perturbation

Definition at line 1161 of file AbcSimplex.hpp.

4.11.6.54 `double* AbcSimplex::perturbationBasic_` [protected]

basic perturbation

Definition at line 1163 of file AbcSimplex.hpp.

4.11.6.55 `AbcMatrix* AbcSimplex::abcMatrix_` [protected]

Working matrix.

Definition at line 1165 of file AbcSimplex.hpp.

4.11.6.56 `double* AbcSimplex::abcLower_` [protected]

Working scaled copy of lower bounds has original scaled copy at end.

Definition at line 1168 of file AbcSimplex.hpp.

4.11.6.57 `double* AbcSimplex::abcUpper_` [protected]

Working scaled copy of upper bounds has original scaled copy at end.

Definition at line 1171 of file AbcSimplex.hpp.

4.11.6.58 `double* AbcSimplex::abcCost_` [protected]

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.

Definition at line 1177 of file AbcSimplex.hpp.

4.11.6.59 `double* AbcSimplex::abcSolution_` [protected]

Working scaled primal solution may have saved from last factorization at end.

Definition at line 1180 of file AbcSimplex.hpp.

4.11.6.60 `double* AbcSimplex::abcDj_` [protected]

Working scaled dual solution may have saved from last factorization at end.

Definition at line 1183 of file AbcSimplex.hpp.

4.11.6.61 `double* AbcSimplex::lowerSaved_` [protected]

Saved scaled copy of lower bounds.

Definition at line 1185 of file AbcSimplex.hpp.

4.11.6.62 `double* AbcSimplex::upperSaved_` [protected]

Saved scaled copy of upper bounds.

Definition at line 1187 of file AbcSimplex.hpp.

4.11.6.63 `double* AbcSimplex::costSaved_` [protected]

Saved scaled copy of objective.

Definition at line 1189 of file AbcSimplex.hpp.

4.11.6.64 `double* AbcSimplex::solutionSaved_` [protected]

Saved scaled primal solution.

Definition at line 1191 of file AbcSimplex.hpp.

4.11.6.65 `double* AbcSimplex::djSaved_` [protected]

Saved scaled dual solution.

Definition at line 1193 of file AbcSimplex.hpp.

4.11.6.66 `double* AbcSimplex::lowerBasic_` [protected]

Working scaled copy of basic lower bounds.

Definition at line 1195 of file AbcSimplex.hpp.

4.11.6.67 `double* AbcSimplex::upperBasic_` [protected]

Working scaled copy of basic upper bounds.

Definition at line 1197 of file AbcSimplex.hpp.

4.11.6.68 `double* AbcSimplex::costBasic_` [protected]

Working scaled copy of basic objective.

Definition at line 1199 of file AbcSimplex.hpp.

4.11.6.69 `double* AbcSimplex::solutionBasic_` [protected]

Working scaled basic primal solution.

Definition at line 1201 of file AbcSimplex.hpp.

4.11.6.70 **double* AbcSimplex::djBasic_** [protected]

Working scaled basic dual solution (want it to be zero)

Definition at line 1203 of file AbcSimplex.hpp.

4.11.6.71 **AbcDualRowPivot* AbcSimplex::abcDualRowPivot_** [protected]

dual row pivot choice

Definition at line 1205 of file AbcSimplex.hpp.

4.11.6.72 **AbcPrimalColumnPivot* AbcSimplex::abcPrimalColumnPivot_** [protected]

primal column pivot choice

Definition at line 1207 of file AbcSimplex.hpp.

4.11.6.73 **int* AbcSimplex::abcPivotVariable_** [protected]

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

Definition at line 1211 of file AbcSimplex.hpp.

4.11.6.74 **int* AbcSimplex::reversePivotVariable_** [protected]

Reverse abcPivotVariable_ for moving around.

Definition at line 1213 of file AbcSimplex.hpp.

4.11.6.75 **AbcSimplexFactorization* AbcSimplex::abcFactorization_** [protected]

factorization

Definition at line 1215 of file AbcSimplex.hpp.

4.11.6.76 **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.

4.11.6.77 **ClpSimplex* AbcSimplex::clpModel_** [protected]

A copy of model as [ClpSimplex](#) with certain state.

Definition at line 1229 of file AbcSimplex.hpp.

4.11.6.78 **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.

4.11.6.79 **CoinPartitionedVector AbcSimplex::usefulArray_[ABC_NUMBER_USEFUL]** [mutable], [protected]

Definition at line 1240 of file AbcSimplex.hpp.

4.11.6.80 `AbcSimplexProgress AbcSimplex::abcProgress_` `[protected]`

For dealing with all issues of cycling etc.

Definition at line 1242 of file AbcSimplex.hpp.

4.11.6.81 `ClpDataSave AbcSimplex::saveData_` `[protected]`

For saving stuff at beginning.

Definition at line 1244 of file AbcSimplex.hpp.

4.11.6.82 `double AbcSimplex::upperTheta_` `[protected]`

upper theta from dual column

Definition at line 1246 of file AbcSimplex.hpp.

4.11.6.83 `int AbcSimplex::multipleSequenceIn_[4]` `[protected]`

Multiple sequence in.

Definition at line 1248 of file AbcSimplex.hpp.

4.11.6.84 `int AbcSimplex::arrayForDualColumn_`

Definition at line 1250 of file AbcSimplex.hpp.

4.11.6.85 `int AbcSimplex::arrayForReplaceColumn_`

Definition at line 1251 of file AbcSimplex.hpp.

4.11.6.86 `int AbcSimplex::arrayForFlipBounds_`

Definition at line 1252 of file AbcSimplex.hpp.

4.11.6.87 `int AbcSimplex::arrayForFlipRhs_`

Definition at line 1253 of file AbcSimplex.hpp.

4.11.6.88 `int AbcSimplex::arrayForBtran_`

Definition at line 1254 of file AbcSimplex.hpp.

4.11.6.89 `int AbcSimplex::arrayForFtran_`

Definition at line 1255 of file AbcSimplex.hpp.

4.11.6.90 `int AbcSimplex::arrayForTableauRow_`

Definition at line 1256 of file AbcSimplex.hpp.

4.11.6.91 `int AbcSimplex::numberFlipped_` `[protected]`

Definition at line 1258 of file AbcSimplex.hpp.

4.11.6.92 `int AbcSimplex::numberDisasters_` `[protected]`

Definition at line 1259 of file AbcSimplex.hpp.

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

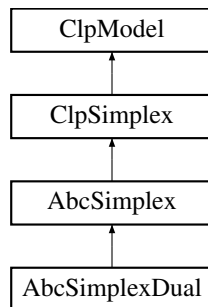
- [src/AbcSimplex.hpp](#)

4.12 AbcSimplexDual Class Reference

This solves LPs using the dual simplex method.

```
#include <AbcSimplexDual.hpp>
```

Inheritance 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 alwaysFinish=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 [updatedDualBound_](#) 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 AbcSimplexFactorization* AbcSimplexDual::setupForStrongBranching ( char * arrays, int numberRows, int
      numberColumns, bool solveLp = false )
```

This does first part of StrongBranching.

```
4.12.2.4 void AbcSimplexDual::cleanupAfterStrongBranching ( AbcSimplexFactorization * factorization )
```

This cleans up after strong branching.

```
4.12.2.5 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 [CipSimplexDual](#))

```
4.12.2.6 void AbcSimplexDual::whileIterating2 ( )
```

4.12.2.7 `int` `AbcSimplexDual::whileIteratingParallel` (`int` *numberIterations*)

4.12.2.8 `int` `AbcSimplexDual::whileIterating3` ()

4.12.2.9 `void` `AbcSimplexDual::updatePrimalSolution` ()

4.12.2.10 `int` `AbcSimplexDual::noPivotRow` ()

4.12.2.11 `int` `AbcSimplexDual::noPivotColumn` ()

4.12.2.12 `void` `AbcSimplexDual::dualPivotColumn` ()

4.12.2.13 `void` `AbcSimplexDual::createDualPricingVectorSerial` ()

Create dual pricing vector.

4.12.2.14 `int` `AbcSimplexDual::getTableauColumnFlipAndStartReplaceSerial` ()

4.12.2.15 `void` `AbcSimplexDual::getTableauColumnPart1Serial` ()

4.12.2.16 `void` `AbcSimplexDual::getTableauColumnPart2` ()

4.12.2.17 `int` `AbcSimplexDual::checkReplace` ()

4.12.2.18 `void` `AbcSimplexDual::replaceColumnPart3` ()

4.12.2.19 `void` `AbcSimplexDual::checkReplacePart1` ()

4.12.2.20 `void` `AbcSimplexDual::checkReplacePart1a` ()

4.12.2.21 `void` `AbcSimplexDual::checkReplacePart1b` ()

4.12.2.22 `void` `AbcSimplexDual::updateDualsInDual` ()

The duals are updated.

4.12.2.23 `int` `AbcSimplexDual::flipBounds` ()

The duals are updated by the given arrays.

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

4.12.2.24 `void` `AbcSimplexDual::flipBack` (`int` *number*)

Undo a flip.

4.12.2.25 `void` `AbcSimplexDual::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)

4.12.2.26 `double` `AbcSimplexDual::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.

4.12.2.27 `double AbcSimplexDual::dualColumn1B ()`

Do all given tableau row.

4.12.2.28 `void AbcSimplexDual::dualColumn2 ()`

Chooses incoming Puts flipped ones in list If necessary will modify costs.

4.12.2.29 `void AbcSimplexDual::dualColumn2Most (dualColumnResult & result)`4.12.2.30 `void AbcSimplexDual::dualColumn2First (dualColumnResult & result)`4.12.2.31 `void AbcSimplexDual::dualColumn2 (dualColumnResult & result)`

Chooses part of incoming Puts flipped ones in list If necessary will modify costs.

4.12.2.32 `void AbcSimplexDual::checkPossibleCleanup (CoinIndexedVector * array)`

This sees what is best thing to do in branch and bound cleanup If `sequenceIn_ < 0` then can't do anything.

4.12.2.33 `void AbcSimplexDual::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.

4.12.2.34 `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

4.12.2.35 `bool AbcSimplexDual::changeBound (int iSequence)`

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

Returns true if change

4.12.2.36 `void AbcSimplexDual::originalBound (int iSequence)`

Restores bound to original bound.

4.12.2.37 `int AbcSimplexDual::checkUnbounded (CoinIndexedVector & ray, double changeCost)`

Checks if tentative optimal actually means unbounded in dual Returns -3 if not, 2 if is unbounded.

4.12.2.38 `void AbcSimplexDual::statusOfProblemInDual (int type)`

Refactorizes if necessary Checks if finished.

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

type - 0 initial so set up save arrays etc

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

4.12.2.39 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)

4.12.2.40 bool `AbcSimplexDual::checkCutoff` (bool *computeObjective*)

see if cutoff reached

4.12.2.41 int `AbcSimplexDual::bounceTolerances` (int *type*)

Does something about fake tolerances.

4.12.2.42 void `AbcSimplexDual::perturb` (double *factor*)

Perturbs problem.

4.12.2.43 void `AbcSimplexDual::perturbB` (double *factor*, int *type*)

Perturbs problem B.

4.12.2.44 int `AbcSimplexDual::makeNonFreeVariablesDualFeasible` (bool *changeCosts* = false)

Make non free variables dual feasible by moving to a bound.

4.12.2.45 int `AbcSimplexDual::fastDual` (bool *alwaysFinish* = false)

4.12.2.46 int `AbcSimplexDual::numberAtFakeBound` ()

Checks number of variables at fake bounds.

This is used by `fastDual` so can exit gracefully before end

4.12.2.47 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

4.12.2.48 int `AbcSimplexDual::nextSuperBasic` ()

Get next free , -1 if none.

4.12.2.49 void `AbcSimplexDual::startupSolve` ()

Startup part of dual.

4.12.2.50 void `AbcSimplexDual::finishSolve` ()

Ending part of dual.

4.12.2.51 void `AbcSimplexDual::gutsOfDual` ()

4.12.2.52 `int AbcSimplexDual::resetFakeBounds (int type)`

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

- [src/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 numberOfRows=0)`
Default constructor.
- `~AbcSimplexFactorization ()`
Destructor.

Copy method

- `AbcSimplexFactorization (const AbcSimplexFactorization &, int denselfSmaller=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.
- `void checkReplacePart1a (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 checkReplacePart1b (CoinIndexedVector *regionSparse, int pivotRow)`
- `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 ®ionSparseFT)
Updates one column (FTRAN) Tries to do FT update number returned is negative if no room.
- int [updateColumnFTPart1](#) (CoinIndexedVector ®ionSparseFT)
- void [updateColumnFTPart2](#) (CoinIndexedVector ®ionSparseFT)
- void [updateColumnFT](#) (CoinIndexedVector ®ionSparseFT, CoinIndexedVector &partialUpdate, int which)
Updates one column (FTRAN) Tries to do FT update puts partial update in vector.
- int [updateColumn](#) (CoinIndexedVector ®ionSparse) const
Updates one column (FTRAN)
- int [updateTwoColumnsFT](#) (CoinIndexedVector ®ionSparseFT, CoinIndexedVector ®ionSparseOther)
Updates one column (FTRAN) from regionFT Tries to do FT update number returned is negative if no room.
- int [updateColumnTranspose](#) (CoinIndexedVector ®ionSparse) const
Updates one column (BTRAN)
- void [updateColumnCpu](#) (CoinIndexedVector ®ionSparse, int whichCpu) const
Updates one column (FTRAN)
- void [updateColumnTransposeCpu](#) (CoinIndexedVector ®ionSparse, int whichCpu) const
Updates one column (BTRAN)
- void [updateFullColumn](#) (CoinIndexedVector ®ionSparse) const
Updates one full column (FTRAN)
- void [updateFullColumnTranspose](#) (CoinIndexedVector ®ionSparse) const
Updates one full column (BTRAN)
- void [updateWeights](#) (CoinIndexedVector ®ionSparse) 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 [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 22 of file `AbcSimplexFactorization.hpp`.

4.13.2 Constructor & Destructor Documentation

4.13.2.1 `AbcSimplexFactorization::AbcSimplexFactorization (int numberOfRows = 0)`

Default constructor.

4.13.2.2 `AbcSimplexFactorization::~~AbcSimplexFactorization ()`

Destructor.

4.13.2.3 `AbcSimplexFactorization::AbcSimplexFactorization (const AbcSimplexFactorization &, int denseSelfSmaller = 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 `AbcSimplexFactorization& AbcSimplexFactorization::operator= (const AbcSimplexFactorization &)`

4.13.3.3 `void AbcSimplexFactorization::setFactorization (AbcSimplexFactorization & rhs)`

Sets factorization.

4.13.3.4 `double AbcSimplexFactorization::checkReplacePart1 (CoinIndexedVector * regionSparse, int pivotRow) [inline]`

Checks if can replace one Column to basis, returns update alpha Fills in region for use later partial update already in U.

Definition at line 75 of file `AbcSimplexFactorization.hpp`.

4.13.3.5 `double AbcSimplexFactorization::checkReplacePart1 (CoinIndexedVector * regionSparse, CoinIndexedVector * partialUpdate, int pivotRow) [inline]`

Checks if can replace one Column to basis, returns update alpha Fills in region for use later partial update in vector.

Definition at line 86 of file `AbcSimplexFactorization.hpp`.

4.13.3.6 `void AbcSimplexFactorization::checkReplacePart1a (CoinIndexedVector * regionSparse, int pivotRow) [inline]`

Checks if can replace one Column to basis, returns update alpha Fills in region for use later partial update already in U.

Definition at line 94 of file `AbcSimplexFactorization.hpp`.

4.13.3.7 `double AbcSimplexFactorization::checkReplacePart1b (CoinIndexedVector * regionSparse, int pivotRow)` `[inline]`

Definition at line 97 of file AbcSimplexFactorization.hpp.

4.13.3.8 `int AbcSimplexFactorization::checkReplacePart2 (int pivotRow, double btranAlpha, double ftranAlpha, double ftAlpha)`
`[inline]`

Checks if can replace one Column to basis, returns 0=OK, 1=Probably OK, 2=singular, 3=no room, 5 max pivots.

Definition at line 102 of file AbcSimplexFactorization.hpp.

4.13.3.9 `void AbcSimplexFactorization::replaceColumnPart3 (const AbcSimplex * model, CoinIndexedVector * regionSparse, CoinIndexedVector * tableauColumn, int pivotRow, double alpha)`

Replaces one Column to basis, partial update already in U.

4.13.3.10 `void AbcSimplexFactorization::replaceColumnPart3 (const AbcSimplex * model, CoinIndexedVector * regionSparse, CoinIndexedVector * tableauColumn, CoinIndexedVector * partialUpdate, int pivotRow, double alpha)`

Replaces one Column to basis, partial update in vector.

4.13.3.11 `int AbcSimplexFactorization::updateColumnFT (CoinIndexedVector & regionSparseFT)` `[inline]`

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

Definition at line 175 of file AbcSimplexFactorization.hpp.

4.13.3.12 `int AbcSimplexFactorization::updateColumnFTPart1 (CoinIndexedVector & regionSparseFT)` `[inline]`

Definition at line 177 of file AbcSimplexFactorization.hpp.

4.13.3.13 `void AbcSimplexFactorization::updateColumnFTPart2 (CoinIndexedVector & regionSparseFT)` `[inline]`

Definition at line 179 of file AbcSimplexFactorization.hpp.

4.13.3.14 `void AbcSimplexFactorization::updateColumnFT (CoinIndexedVector & regionSparseFT, CoinIndexedVector & partialUpdate, int which)` `[inline]`

Updates one column (FTRAN) Tries to do FT update puts partial update in vector.

Definition at line 184 of file AbcSimplexFactorization.hpp.

4.13.3.15 `int AbcSimplexFactorization::updateColumn (CoinIndexedVector & regionSparse) const` `[inline]`

Updates one column (FTRAN)

Definition at line 189 of file AbcSimplexFactorization.hpp.

4.13.3.16 `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 195 of file AbcSimplexFactorization.hpp.

4.13.3.17 `int AbcSimplexFactorization::updateColumnTranspose (CoinIndexedVector & regionSparse) const` `[inline]`

Updates one column (BTRAN)

Definition at line 199 of file `AbcSimplexFactorization.hpp`.

4.13.3.18 `void AbcSimplexFactorization::updateColumnCpu (CoinIndexedVector & regionSparse, int whichCpu) const`
`[inline]`

Updates one column (FTRAN)

Definition at line 202 of file `AbcSimplexFactorization.hpp`.

4.13.3.19 `void AbcSimplexFactorization::updateColumnTransposeCpu (CoinIndexedVector & regionSparse, int whichCpu) const`
`[inline]`

Updates one column (BTRAN)

Definition at line 205 of file `AbcSimplexFactorization.hpp`.

4.13.3.20 `void AbcSimplexFactorization::updateFullColumn (CoinIndexedVector & regionSparse) const` `[inline]`

Updates one full column (FTRAN)

Definition at line 208 of file `AbcSimplexFactorization.hpp`.

4.13.3.21 `void AbcSimplexFactorization::updateFullColumnTranspose (CoinIndexedVector & regionSparse) const` `[inline]`

Updates one full column (BTRAN)

Definition at line 211 of file `AbcSimplexFactorization.hpp`.

4.13.3.22 `void AbcSimplexFactorization::updateWeights (CoinIndexedVector & regionSparse) const` `[inline]`

Updates one column for dual steepest edge weights (FTRAN)

Definition at line 214 of file `AbcSimplexFactorization.hpp`.

4.13.3.23 `int AbcSimplexFactorization::numberElements () const` `[inline]`

Total number of elements in factorization.

Definition at line 220 of file `AbcSimplexFactorization.hpp`.

4.13.3.24 `int AbcSimplexFactorization::maximumPivots () const` `[inline]`

Maximum number of pivots between factorizations.

Definition at line 224 of file `AbcSimplexFactorization.hpp`.

4.13.3.25 `void AbcSimplexFactorization::maximumPivots (int value)` `[inline]`

Set maximum number of pivots between factorizations.

Definition at line 228 of file `AbcSimplexFactorization.hpp`.

4.13.3.26 `bool AbcSimplexFactorization::usingFT () const` `[inline]`

Returns true if doing FT.

Definition at line 232 of file `AbcSimplexFactorization.hpp`.

4.13.3.27 `int AbcSimplexFactorization::pivots () const` `[inline]`

Returns number of pivots since factorization.

Definition at line 235 of file AbcSimplexFactorization.hpp.

4.13.3.28 void AbcSimplexFactorization::setPivots (int *value*) const [inline]

Sets number of pivots since factorization.

Definition at line 239 of file AbcSimplexFactorization.hpp.

4.13.3.29 double AbcSimplexFactorization::areaFactor () const [inline]

Whether larger areas needed.

Definition at line 243 of file AbcSimplexFactorization.hpp.

4.13.3.30 void AbcSimplexFactorization::areaFactor (double *value*) [inline]

Set whether larger areas needed.

Definition at line 247 of file AbcSimplexFactorization.hpp.

4.13.3.31 double AbcSimplexFactorization::zeroTolerance () const [inline]

Zero tolerance.

Definition at line 251 of file AbcSimplexFactorization.hpp.

4.13.3.32 void AbcSimplexFactorization::zeroTolerance (double *value*) [inline]

Set zero tolerance.

Definition at line 255 of file AbcSimplexFactorization.hpp.

4.13.3.33 void AbcSimplexFactorization::saferTolerances (double *zeroTolerance*, double *pivotTolerance*)

Set tolerances to safer of existing and given.

4.13.3.34 int AbcSimplexFactorization::status () const [inline]

Returns status.

Definition at line 261 of file AbcSimplexFactorization.hpp.

4.13.3.35 void AbcSimplexFactorization::setStatus (int *value*) [inline]

Sets status.

Definition at line 265 of file AbcSimplexFactorization.hpp.

4.13.3.36 int AbcSimplexFactorization::numberDense () const [inline]

Returns number of dense rows.

Definition at line 274 of file AbcSimplexFactorization.hpp.

4.13.3.37 bool AbcSimplexFactorization::timeToRefactorize () const [inline]

Definition at line 277 of file AbcSimplexFactorization.hpp.

4.13.3.38 void AbcSimplexFactorization::clearArrays () [inline]

Get rid of all memory.

Definition at line 281 of file `AbcSimplexFactorization.hpp`.

4.13.3.39 `int AbcSimplexFactorization::numberOfRows () const [inline]`

Number of Rows after factorization.

Definition at line 285 of file `AbcSimplexFactorization.hpp`.

4.13.3.40 `int AbcSimplexFactorization::numberOfSlacks () const [inline]`

Number of slacks at last factorization.

Definition at line 289 of file `AbcSimplexFactorization.hpp`.

4.13.3.41 `double AbcSimplexFactorization::pivotTolerance () const [inline]`

Pivot tolerance.

Definition at line 292 of file `AbcSimplexFactorization.hpp`.

4.13.3.42 `void AbcSimplexFactorization::pivotTolerance (double value) [inline]`

Set pivot tolerance.

Definition at line 296 of file `AbcSimplexFactorization.hpp`.

4.13.3.43 `double AbcSimplexFactorization::minimumPivotTolerance () const [inline]`

Minimum pivot tolerance.

Definition at line 300 of file `AbcSimplexFactorization.hpp`.

4.13.3.44 `void AbcSimplexFactorization::minimumPivotTolerance (double value) [inline]`

Set minimum pivot tolerance.

Definition at line 304 of file `AbcSimplexFactorization.hpp`.

4.13.3.45 `double* AbcSimplexFactorization::pivotRegion () const [inline]`

pivot region

Definition at line 308 of file `AbcSimplexFactorization.hpp`.

4.13.3.46 `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 315 of file `AbcSimplexFactorization.hpp`.

4.13.3.47 `void AbcSimplexFactorization::setDenseThreshold (int number)`

So we can temporarily switch off dense.

4.13.3.48 `int AbcSimplexFactorization::getDenseThreshold () const`

4.13.3.49 `void AbcSimplexFactorization::forceOtherFactorization (int which)`

If nonzero force use of 1,dense 2,small 3,long.

4.13.3.50 `void AbcSimplexFactorization::goDenseOrSmall (int numberOfRows)`

Go over to dense code.

4.13.3.51 `int AbcSimplexFactorization::goDenseThreshold () const [inline]`

Get switch to dense if number rows \leq this.

Definition at line 326 of file AbcSimplexFactorization.hpp.

4.13.3.52 `void AbcSimplexFactorization::setGoDenseThreshold (int value) [inline]`

Set switch to dense if number rows \leq this.

Definition at line 330 of file AbcSimplexFactorization.hpp.

4.13.3.53 `int AbcSimplexFactorization::goSmallThreshold () const [inline]`

Get switch to small if number rows \leq this.

Definition at line 334 of file AbcSimplexFactorization.hpp.

4.13.3.54 `void AbcSimplexFactorization::setGoSmallThreshold (int value) [inline]`

Set switch to small if number rows \leq this.

Definition at line 338 of file AbcSimplexFactorization.hpp.

4.13.3.55 `int AbcSimplexFactorization::goLongThreshold () const [inline]`

Get switch to long/ordered if number rows \geq this.

Definition at line 342 of file AbcSimplexFactorization.hpp.

4.13.3.56 `void AbcSimplexFactorization::setGoLongThreshold (int value) [inline]`

Set switch to long/ordered if number rows \geq this.

Definition at line 346 of file AbcSimplexFactorization.hpp.

4.13.3.57 `int AbcSimplexFactorization::typeOfFactorization () const [inline]`

Returns type.

Definition at line 350 of file AbcSimplexFactorization.hpp.

4.13.3.58 `void AbcSimplexFactorization::synchronize (const ClpFactorization * otherFactorization, const AbcSimplex * model)`

Synchronize stuff.

4.13.3.59 `void AbcSimplexFactorization::goSparse ()`

makes a row copy of L for speed and to allow very sparse problems

4.13.3.60 `void AbcSimplexFactorization::checkMarkArrays () const [inline]`

Definition at line 361 of file AbcSimplexFactorization.hpp.

4.13.3.61 `bool AbcSimplexFactorization::needToReorder () const [inline]`

Says whether to redo pivot order.

Definition at line 365 of file `AbcSimplexFactorization.hpp`.

4.13.3.62 `CoinAbcAnyFactorization* AbcSimplexFactorization::factorization () const [inline]`

Pointer to factorization.

Definition at line 367 of file `AbcSimplexFactorization.hpp`.

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

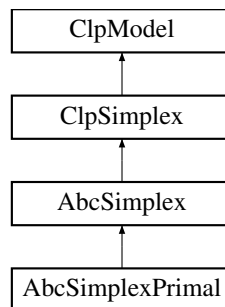
- [src/AbcSimplexFactorization.hpp](#)

4.14 AbcSimplexPrimal Class Reference

This solves LPs using the primal simplex method.

```
#include <AbcSimplexPrimal.hpp>
```

Inheritance 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 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 extraCost[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 Fran 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, CoinPartitionedVector *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 `infeasibilityCost_` 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::alwaysOptimal (bool *onOff*)

Do not change infeasibility cost and always say optimal.

4.14.2.3 bool AbcSimplexPrimal::alwaysOptimal () const

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

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

This can be switched off

4.14.2.5 bool AbcSimplexPrimal::exactOutgoing () const

4.14.2.6 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

4.14.2.7 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_

4.14.2.8 int AbcSimplexPrimal::pivotResult4 (int *ifValuesPass* = 0)

4.14.2.9 `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

4.14.2.10 `void AbcSimplexPrimal::updatePrimalsInPrimal (CoinIndexedVector & rowArray, double theta, bool valuesPass)`

The primals are updated by the given array.

costs are changed

4.14.2.11 `void AbcSimplexPrimal::createUpdateDuals (CoinIndexedVector & rowArray, const double * originalCost, const double extraCost[4], double & objectiveChange, int valuesPass)`

After rowArray will have cost changes for use next iteration.

4.14.2.12 `double AbcSimplexPrimal::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)

4.14.2.13 `void AbcSimplexPrimal::updatePartialUpdate (CoinIndexedVector & partialUpdate)`

Update partial Ftran by R update.

4.14.2.14 `int AbcSimplexPrimal::doFTUpdate (CoinIndexedVector * vector[4])`

Do FT update as separate function for minor iterations (nonzero return code on problems)

4.14.2.15 `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

4.14.2.16 `void AbcSimplexPrimal::primalRow (CoinIndexedVector * rowArray, CoinIndexedVector * rhsArray, CoinIndexedVector * spareArray, pivotStruct & stuff)`

4.14.2.17 `void AbcSimplexPrimal::primalColumn (CoinPartitionedVector * updateArray, CoinPartitionedVector * spareRow2, CoinPartitionedVector * 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.

4.14.2.18 `int AbcSimplexPrimal::checkUnbounded (CoinIndexedVector * ray, CoinIndexedVector * spare, double changeCost)`

Checks if tentative optimal actually means unbounded in primal Returns -3 if not, 2 if is unbounded.

4.14.2.19 `void AbcSimplexPrimal::statusOfProblemInPrimal (int type)`

Refactorizes if necessary Checks if finished.

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

type - 0 initial so set up save arrays etc

- 1 normal -if good update save

2 restoring from saved

4.14.2.20 `void AbcSimplexPrimal::perturb (int type)`

Perturbs problem (method depends on [perturbation\(\)](#))

4.14.2.21 `bool AbcSimplexPrimal::unPerturb ()`

Take off effect of perturbation and say whether to try dual.

4.14.2.22 `int AbcSimplexPrimal::unflag ()`

Unflag all variables and return number unflagged.

4.14.2.23 `int AbcSimplexPrimal::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.

4.14.2.24 `void AbcSimplexPrimal::primalRay (CoinIndexedVector * rowArray)`

Create primal ray.

4.14.2.25 `void AbcSimplexPrimal::clearAll ()`

Clears all bits and clears rowArray[1] etc.

4.14.2.26 `int AbcSimplexPrimal::lexSolve ()`

Sort of lexicographic resolve.

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

- [src/AbcSimplexPrimal.hpp](#)

4.15 **AbcTolerancesEtc Class Reference**

```
#include <CoinAbcCommon.hpp>
```

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 251 of file CoinAbcCommon.hpp.

4.15.2 Constructor & Destructor Documentation

4.15.2.1 `AbcTolerancesEtc::AbcTolerancesEtc ()`

Default Constructor.

4.15.2.2 `AbcTolerancesEtc::AbcTolerancesEtc (const ClpSimplex * model)`

Useful Constructors.

4.15.2.3 `AbcTolerancesEtc::AbcTolerancesEtc (const AbcSimplex * model)`4.15.2.4 `AbcTolerancesEtc::AbcTolerancesEtc (const AbcTolerancesEtc &)`

Copy constructor.

4.15.2.5 **AbcTolerancesEtc::~~AbcTolerancesEtc ()**

Destructor.

4.15.3 Member Function Documentation

4.15.3.1 **AbcTolerancesEtc& AbcTolerancesEtc::operator= (const AbcTolerancesEtc & rhs)**

Assignment operator.

4.15.4 Member Data Documentation

4.15.4.1 **double AbcTolerancesEtc::zeroTolerance_**

Zero tolerance.

Definition at line 283 of file CoinAbcCommon.hpp.

4.15.4.2 **double AbcTolerancesEtc::primalToleranceToGetOptimal_**

Primal tolerance needed to make dual feasible (<largeTolerance)

Definition at line 285 of file CoinAbcCommon.hpp.

4.15.4.3 **double AbcTolerancesEtc::largeValue_**

Large bound value (for complementarity etc)

Definition at line 287 of file CoinAbcCommon.hpp.

4.15.4.4 **double AbcTolerancesEtc::alphaAccuracy_**

For computing whether to re-factorize.

Definition at line 289 of file CoinAbcCommon.hpp.

4.15.4.5 **double AbcTolerancesEtc::dualBound_**

Dual bound.

Definition at line 291 of file CoinAbcCommon.hpp.

4.15.4.6 **double AbcTolerancesEtc::dualTolerance_**

Current dual tolerance for algorithm.

Definition at line 293 of file CoinAbcCommon.hpp.

4.15.4.7 **double AbcTolerancesEtc::primalTolerance_**

Current primal tolerance for algorithm.

Definition at line 295 of file CoinAbcCommon.hpp.

4.15.4.8 **double AbcTolerancesEtc::infeasibilityCost_**

Weight assigned to being infeasible in primal.

Definition at line 297 of file CoinAbcCommon.hpp.

4.15.4.9 `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 although you may end up with an all slack basis.

Defaults are 1.0,10.0

Definition at line 307 of file CoinAbcCommon.hpp.

4.15.4.10 `double AbcTolerancesEtc::allowedInfeasibility_`

Definition at line 308 of file CoinAbcCommon.hpp.

4.15.4.11 `int AbcTolerancesEtc::baseIteration_`

Iteration when we entered dual or primal.

Definition at line 310 of file CoinAbcCommon.hpp.

4.15.4.12 `int AbcTolerancesEtc::numberRefinements_`

How many iterative refinements to do.

Definition at line 312 of file CoinAbcCommon.hpp.

4.15.4.13 `int AbcTolerancesEtc::forceFactorization_`

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

Definition at line 315 of file CoinAbcCommon.hpp.

4.15.4.14 `int AbcTolerancesEtc::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.

Definition at line 323 of file CoinAbcCommon.hpp.

4.15.4.15 `int AbcTolerancesEtc::dontFactorizePivots_`

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

Definition at line 325 of file CoinAbcCommon.hpp.

4.15.4.16 `int AbcTolerancesEtc::maximumPivots_`

For factorization Maximum number of pivots before factorization.

Definition at line 328 of file CoinAbcCommon.hpp.

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

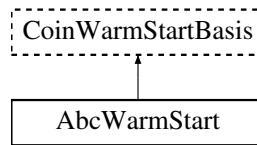
- [src/CoinAbcCommon.hpp](#)

4.16 `AbcWarmStart` Class Reference

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

```
#include <AbcWarmStart.hpp>
```

Inheritance diagram for AbcWarmStart:



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.

4.16.2.3 [AbcWarmStart::AbcWarmStart \(const \[AbcWarmStart\]\(#\) & *ws* \)](#)

Copy constructor.

4.16.2.4 **virtual AbcWarmStart::~AbcWarmStart ()** [virtual]

Destructor.

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.

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.)

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.

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.

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.

4.16.3.6 `void AbcWarmStart::setModel (AbcSimplex * model) [inline]`

Set model.

Definition at line 140 of file AbcWarmStart.hpp.

4.16.3.7 `AbcSimplex* AbcWarmStart::model () const [inline]`

Get model.

Definition at line 143 of file AbcWarmStart.hpp.

4.16.3.8 `void AbcWarmStart::createBasis0 (const AbcSimplex * model)`

Create Basis type 0.

4.16.3.9 `void AbcWarmStart::createBasis12 (const AbcSimplex * model)`

Create Basis type 12.

4.16.3.10 `void AbcWarmStart::createBasis34 (const AbcSimplex * model)`

Create Basis type 34.

4.16.3.11 `virtual CoinWarmStart* AbcWarmStart::clone () const [inline],[virtual]`

'Virtual constructor'

Definition at line 177 of file AbcWarmStart.hpp.

4.16.3.12 `virtual AbcWarmStart& AbcWarmStart::operator= (const AbcWarmStart & rhs) [virtual]`

Assignment.

4.16.3.13 `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[]`.

4.16.4 Member Data Documentation

4.16.4.1 `int AbcWarmStart::typeExtrInformation_ [protected]`

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

Definition at line 218 of file AbcWarmStart.hpp.

4.16.4.2 `int AbcWarmStart::lengthExtralInformation_ [protected]`

Length of extra information in bytes.

Definition at line 220 of file `AbcWarmStart.hpp`.

4.16.4.3 `char* AbcWarmStart::extralInformation_ [protected]`

The extra information.

Definition at line 222 of file `AbcWarmStart.hpp`.

4.16.4.4 `AbcSimplex* AbcWarmStart::model_ [protected]`

Pointer back to [AbcSimplex](#) (can only be applied to that)

Definition at line 224 of file `AbcWarmStart.hpp`.

4.16.4.5 `AbcWarmStartOrganizer* AbcWarmStart::organizer_ [protected]`

Pointer back to [AbcWarmStartOrganizer](#) for organization.

Definition at line 226 of file `AbcWarmStart.hpp`.

4.16.4.6 `AbcWarmStart* AbcWarmStart::previousBasis_ [protected]`

Pointer to previous basis.

Definition at line 228 of file `AbcWarmStart.hpp`.

4.16.4.7 `AbcWarmStart* AbcWarmStart::nextBasis_ [protected]`

Pointer to next basis.

Definition at line 230 of file `AbcWarmStart.hpp`.

4.16.4.8 `int AbcWarmStart::stamp_ [protected]`

Sequence stamp for deletion.

Definition at line 232 of file `AbcWarmStart.hpp`.

4.16.4.9 `int AbcWarmStart::numberValidRows_ [protected]`

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.

Definition at line 237 of file `AbcWarmStart.hpp`.

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

- `src/AbcWarmStart.hpp`

4.17 AbcWarmStartOrganizer Class Reference

```
#include <AbcWarmStart.hpp>
```

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

4.17.2.2 [AbcWarmStartOrganizer::AbcWarmStartOrganizer](#) (const [AbcWarmStartOrganizer](#) & ws)

Copy constructor.

4.17.2.3 virtual [AbcWarmStartOrganizer::~~AbcWarmStartOrganizer](#) () [virtual]

Destructor.

4.17.3 Member Function Documentation

4.17.3.1 void AbcWarmStartOrganizer::createBasis0 ()

Create Basis type 0.

4.17.3.2 void AbcWarmStartOrganizer::createBasis12 ()

Create Basis type 1,2.

4.17.3.3 void AbcWarmStartOrganizer::createBasis34 ()

Create Basis type 3,4.

4.17.3.4 void AbcWarmStartOrganizer::deleteBasis (AbcWarmStart * *basis*)

delete basis

4.17.3.5 virtual AbcWarmStartOrganizer& AbcWarmStartOrganizer::operator= (const AbcWarmStartOrganizer & *rhs*) [virtual]

Assignment.

4.17.4 Member Data Documentation

4.17.4.1 AbcSimplex* AbcWarmStartOrganizer::model_ [protected]

Pointer to [AbcSimplex](#) (can only be applied to that)

Definition at line 58 of file AbcWarmStart.hpp.

4.17.4.2 AbcWarmStart* AbcWarmStartOrganizer::firstBasis_ [protected]

Pointer to first basis.

Definition at line 60 of file AbcWarmStart.hpp.

4.17.4.3 AbcWarmStart* AbcWarmStartOrganizer::lastBasis_ [protected]

Pointer to last basis.

Definition at line 62 of file AbcWarmStart.hpp.

4.17.4.4 int AbcWarmStartOrganizer::numberBases_ [protected]

Number of bases.

Definition at line 64 of file AbcWarmStart.hpp.

4.17.4.5 int AbcWarmStartOrganizer::sizeBases_ [protected]

Size of bases (extra)

Definition at line 66 of file AbcWarmStart.hpp.

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

- [src/AbcWarmStart.hpp](#)

4.18 ampl_info Struct Reference

```
#include <Clp_ampl.h>
```

Public Attributes

- int [numberRows](#)
- int [numberColumns](#)
- int [numberBinary](#)
- int [numberIntegers](#)
- int [numberSos](#)
- int [numberElements](#)
- int [numberArguments](#)
- int [problemStatus](#)
- double [direction](#)
- double [offset](#)
- double [objValue](#)
- double * [objective](#)
- double * [rowLower](#)
- double * [rowUpper](#)
- double * [columnLower](#)
- double * [columnUpper](#)
- int * [starts](#)
- int * [rows](#)
- double * [elements](#)
- double * [primalSolution](#)
- double * [dualSolution](#)
- int * [columnStatus](#)
- int * [rowStatus](#)
- int * [priorities](#)
- int * [branchDirection](#)
- double * [pseudoDown](#)
- double * [pseudoUp](#)
- char * [sosType](#)
- int * [sosPriority](#)
- int * [sosStart](#)
- int * [sosIndices](#)
- double * [sosReference](#)
- int * [cut](#)
- int * [special](#)
- char ** [arguments](#)
- char [buffer](#) [300]
- int [logLevel](#)
- int [nonLinear](#)

4.18.1 Detailed Description

Definition at line 11 of file Clp_ampl.h.

4.18.2 Member Data Documentation

4.18.2.1 `int ampl_info::numberRows`

Definition at line 12 of file `Clp_ampl.h`.

4.18.2.2 `int ampl_info::numberColumns`

Definition at line 13 of file `Clp_ampl.h`.

4.18.2.3 `int ampl_info::numberBinary`

Definition at line 14 of file `Clp_ampl.h`.

4.18.2.4 `int ampl_info::numberIntegers`

Definition at line 15 of file `Clp_ampl.h`.

4.18.2.5 `int ampl_info::numberSos`

Definition at line 16 of file `Clp_ampl.h`.

4.18.2.6 `int ampl_info::numberElements`

Definition at line 17 of file `Clp_ampl.h`.

4.18.2.7 `int ampl_info::numberArguments`

Definition at line 18 of file `Clp_ampl.h`.

4.18.2.8 `int ampl_info::problemStatus`

Definition at line 19 of file `Clp_ampl.h`.

4.18.2.9 `double ampl_info::direction`

Definition at line 20 of file `Clp_ampl.h`.

4.18.2.10 `double ampl_info::offset`

Definition at line 21 of file `Clp_ampl.h`.

4.18.2.11 `double ampl_info::objValue`

Definition at line 22 of file `Clp_ampl.h`.

4.18.2.12 `double* ampl_info::objective`

Definition at line 23 of file `Clp_ampl.h`.

4.18.2.13 `double* ampl_info::rowLower`

Definition at line 24 of file `Clp_ampl.h`.

4.18.2.14 `double* ampl_info::rowUpper`

Definition at line 25 of file `Clp_ampl.h`.

4.18.2.15 double* ampl_info::columnLower

Definition at line 26 of file Clp_ampl.h.

4.18.2.16 double* ampl_info::columnUpper

Definition at line 27 of file Clp_ampl.h.

4.18.2.17 int* ampl_info::starts

Definition at line 28 of file Clp_ampl.h.

4.18.2.18 int* ampl_info::rows

Definition at line 29 of file Clp_ampl.h.

4.18.2.19 double* ampl_info::elements

Definition at line 30 of file Clp_ampl.h.

4.18.2.20 double* ampl_info::primalSolution

Definition at line 31 of file Clp_ampl.h.

4.18.2.21 double* ampl_info::dualSolution

Definition at line 32 of file Clp_ampl.h.

4.18.2.22 int* ampl_info::columnStatus

Definition at line 33 of file Clp_ampl.h.

4.18.2.23 int* ampl_info::rowStatus

Definition at line 34 of file Clp_ampl.h.

4.18.2.24 int* ampl_info::priorities

Definition at line 35 of file Clp_ampl.h.

4.18.2.25 int* ampl_info::branchDirection

Definition at line 36 of file Clp_ampl.h.

4.18.2.26 double* ampl_info::pseudoDown

Definition at line 37 of file Clp_ampl.h.

4.18.2.27 double* ampl_info::pseudoUp

Definition at line 38 of file Clp_ampl.h.

4.18.2.28 char* ampl_info::sosType

Definition at line 39 of file Clp_ampl.h.

4.18.2.29 int* ampl_info::sosPriority

Definition at line 40 of file Clp_ampl.h.

4.18.2.30 int* ampl_info::sosStart

Definition at line 41 of file Clp_ampl.h.

4.18.2.31 int* ampl_info::sosIndices

Definition at line 42 of file Clp_ampl.h.

4.18.2.32 double* ampl_info::sosReference

Definition at line 43 of file Clp_ampl.h.

4.18.2.33 int* ampl_info::cut

Definition at line 44 of file Clp_ampl.h.

4.18.2.34 int* ampl_info::special

Definition at line 45 of file Clp_ampl.h.

4.18.2.35 char ampl_info::arguments**

Definition at line 46 of file Clp_ampl.h.

4.18.2.36 char ampl_info::buffer[300]

Definition at line 47 of file Clp_ampl.h.

4.18.2.37 int ampl_info::logLevel

Definition at line 48 of file Clp_ampl.h.

4.18.2.38 int ampl_info::nonLinear

Definition at line 49 of file Clp_ampl.h.

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

- [src/Clp_ampl.h](#)

4.19 blockStruct Struct Reference

```
#include <ClpPackedMatrix.hpp>
```

Public Attributes

- CoinBigIndex [startElements_](#)
- int [startIndices_](#)
- int [numberInBlock_](#)
- int [numberPrice_](#)
- int [numberElements_](#)

4.19.1 Detailed Description

Definition at line 571 of file ClpPackedMatrix.hpp.

4.19.2 Member Data Documentation

4.19.2.1 CoinBigIndex blockStruct::startElements_

Definition at line 572 of file ClpPackedMatrix.hpp.

4.19.2.2 int blockStruct::startIndices_

Definition at line 573 of file ClpPackedMatrix.hpp.

4.19.2.3 int blockStruct::numberInBlock_

Definition at line 574 of file ClpPackedMatrix.hpp.

4.19.2.4 int blockStruct::numberPrice_

Definition at line 575 of file ClpPackedMatrix.hpp.

4.19.2.5 int blockStruct::numberElements_

Definition at line 576 of file ClpPackedMatrix.hpp.

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

- [src/ClpPackedMatrix.hpp](#)

4.20 blockStruct3 Struct Reference

```
#include <AbcMatrix.hpp>
```

Public Attributes

- CoinBigIndex [startElements_](#)
- int [startIndices_](#)
- int [numberInBlock_](#)
- int [numberPrice_](#)
- int [numberElements_](#)

4.20.1 Detailed Description

Definition at line 557 of file AbcMatrix.hpp.

4.20.2 Member Data Documentation

4.20.2.1 CoinBigIndex blockStruct3::startElements_

Definition at line 558 of file AbcMatrix.hpp.

4.20.2.2 int blockStruct3::startIndices_

Definition at line 559 of file AbcMatrix.hpp.

4.20.2.3 int blockStruct3::numberInBlock_

Definition at line 560 of file AbcMatrix.hpp.

4.20.2.4 int blockStruct3::numberPrice_

Definition at line 561 of file AbcMatrix.hpp.

4.20.2.5 int blockStruct3::numberElements_

Definition at line 562 of file AbcMatrix.hpp.

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

- [src/AbcMatrix.hpp](#)

4.21 ClpNode::branchState Struct Reference

```
#include <ClpNode.hpp>
```

Public Attributes

- unsigned int [firstBranch](#): 1
- unsigned int [branch](#): 2
- unsigned int [spare](#): 29

4.21.1 Detailed Description

Definition at line 121 of file ClpNode.hpp.

4.21.2 Member Data Documentation

4.21.2.1 unsigned int ClpNode::branchState::firstBranch

Definition at line 122 of file ClpNode.hpp.

4.21.2.2 unsigned int ClpNode::branchState::branch

Definition at line 123 of file ClpNode.hpp.

4.21.2.3 unsigned int ClpNode::branchState::spare

Definition at line 124 of file ClpNode.hpp.

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

- [src/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 printing.
- 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.
- 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.
- int [intValue](#) () const

- void `setDoubleValue` (double value)
Sets double value.
- 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 291 of file `CbcOrClpParam.hpp`.

4.22.2 Constructor & Destructor Documentation

4.22.2.1 `CbcOrClpParam::CbcOrClpParam ()`

Constructors.

4.22.2.2 `CbcOrClpParam::CbcOrClpParam (std::string name, std::string help, double lower, double upper, CbcOrClpParameterType type, int display = 2)`

4.22.2.3 `CbcOrClpParam::CbcOrClpParam (std::string name, std::string help, int lower, int upper, CbcOrClpParameterType type, int display = 2)`

4.22.2.4 `CbcOrClpParam::CbcOrClpParam (std::string name, std::string help, std::string firstValue, CbcOrClpParameterType type, int whereUsed = 7, int display = 2)`

4.22.2.5 `CbcOrClpParam::CbcOrClpParam (std::string name, std::string help, CbcOrClpParameterType type, int whereUsed = 7, int display = 2)`

4.22.2.6 CbcOrClpParam::CbcOrClpParam (const CbcOrClpParam &)

Copy constructor.

4.22.2.7 CbcOrClpParam::~~CbcOrClpParam ()

Destructor.

4.22.3 Member Function Documentation

4.22.3.1 CbcOrClpParam& CbcOrClpParam::operator= (const CbcOrClpParam & rhs)

Assignment operator. This copies the data.

4.22.3.2 void CbcOrClpParam::append (std::string *keyWord*)

Insert string (only valid for keywords)

4.22.3.3 void CbcOrClpParam::addHelp (std::string *keyWord*)

Adds one help line.

4.22.3.4 std::string CbcOrClpParam::name () const [inline]

Returns name.

Definition at line 322 of file CbcOrClpParam.hpp.

4.22.3.5 std::string CbcOrClpParam::shortHelp () const [inline]

Returns short help.

Definition at line 326 of file CbcOrClpParam.hpp.

4.22.3.6 int CbcOrClpParam::setDoubleParameter (CbcModel & *model*, double *value*)

Sets a double parameter (nonzero code if error)

4.22.3.7 const char* CbcOrClpParam::setDoubleParameterWithMessage (CbcModel & *model*, double *value*, int & *returnCode*)

Sets double parameter and returns printable string and error code.

4.22.3.8 double CbcOrClpParam::doubleParameter (CbcModel & *model*) const

Gets a double parameter.

4.22.3.9 int CbcOrClpParam::setIntParameter (CbcModel & *model*, int *value*)

Sets a int parameter (nonzero code if error)

4.22.3.10 const char* CbcOrClpParam::setIntParameterWithMessage (CbcModel & *model*, int *value*, int & *returnCode*)

Sets int parameter and returns printable string and error code.

4.22.3.11 int CbcOrClpParam::intParameter (CbcModel & *model*) const

Gets a int parameter.

4.22.3.12 `int CbcOrClpParam::setDoubleParameter (ClpSimplex * model, double value)`

Sets a double parameter (nonzero code if error)

4.22.3.13 `double CbcOrClpParam::doubleParameter (ClpSimplex * model) const`

Gets a double parameter.

4.22.3.14 `const char* CbcOrClpParam::setDoubleParameterWithMessage (ClpSimplex * model, double value, int & returnCode)`

Sets double parameter and returns printable string and error code.

4.22.3.15 `int CbcOrClpParam::setIntParameter (ClpSimplex * model, int value)`

Sets a int parameter (nonzero code if error)

4.22.3.16 `const char* CbcOrClpParam::setIntParameterWithMessage (ClpSimplex * model, int value, int & returnCode)`

Sets int parameter and returns printable string and error code.

4.22.3.17 `int CbcOrClpParam::intParameter (ClpSimplex * model) const`

Gets a int parameter.

4.22.3.18 `int CbcOrClpParam::setDoubleParameter (OsiSolverInterface * model, double value)`

Sets a double parameter (nonzero code if error)

4.22.3.19 `const char* CbcOrClpParam::setDoubleParameterWithMessage (OsiSolverInterface * model, double value, int & returnCode)`

Sets double parameter and returns printable string and error code.

4.22.3.20 `double CbcOrClpParam::doubleParameter (OsiSolverInterface * model) const`

Gets a double parameter.

4.22.3.21 `int CbcOrClpParam::setIntParameter (OsiSolverInterface * model, int value)`

Sets a int parameter (nonzero code if error)

4.22.3.22 `const char* CbcOrClpParam::setIntParameterWithMessage (OsiSolverInterface * model, int value, int & returnCode)`

Sets int parameter and returns printable string and error code.

4.22.3.23 `int CbcOrClpParam::intParameter (OsiSolverInterface * model) const`

Gets a int parameter.

4.22.3.24 `int CbcOrClpParam::checkDoubleParameter (double value) const`

Checks a double parameter (nonzero code if error)

4.22.3.25 `std::string CbcOrClpParam::matchName () const`

Returns name which could match.

4.22.3.26 `int CbcOrClpParam::lengthMatchName () const`

Returns length of name for printing.

4.22.3.27 `int CbcOrClpParam::parameterOption (std::string check) const`

Returns parameter option which matches (-1 if none)

4.22.3.28 `void CbcOrClpParam::printOptions () const`

Prints parameter options.

4.22.3.29 `std::string CbcOrClpParam::currentOption () const` `[inline]`

Returns current parameter option.

Definition at line 376 of file CbcOrClpParam.hpp.

4.22.3.30 `void CbcOrClpParam::setCurrentOption (int value, bool printIt = false)`

Sets current parameter option.

4.22.3.31 `const char* CbcOrClpParam::setCurrentOptionWithMessage (int value)`

Sets current parameter option and returns printable string.

4.22.3.32 `void CbcOrClpParam::setCurrentOption (const std::string value)`

Sets current parameter option using string.

4.22.3.33 `int CbcOrClpParam::currentOptionAsInteger () const`

Returns current parameter option position.

4.22.3.34 `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

4.22.3.35 `void CbcOrClpParam::setIntValue (int value)`

Sets int value.

4.22.3.36 `int CbcOrClpParam::intValue () const` `[inline]`

Definition at line 394 of file CbcOrClpParam.hpp.

4.22.3.37 `void CbcOrClpParam::setDoubleValue (double value)`

Sets double value.

4.22.3.38 `double CbcOrClpParam::doubleValue () const` `[inline]`

Definition at line 399 of file CbcOrClpParam.hpp.

4.22.3.39 `void CbcOrClpParam::setStringValue (std::string value)`

Sets string value.

4.22.3.40 `std::string CbcOrClpParam::stringValue () const [inline]`

Definition at line 404 of file CbcOrClpParam.hpp.

4.22.3.41 `int CbcOrClpParam::matches (std::string input) const`

Returns 1 if matches minimum, 2 if matches less, 0 if not matched.

4.22.3.42 `CbcOrClpParameterType CbcOrClpParam::type () const [inline]`

type

Definition at line 410 of file CbcOrClpParam.hpp.

4.22.3.43 `int CbcOrClpParam::displayThis () const [inline]`

whether to display

Definition at line 414 of file CbcOrClpParam.hpp.

4.22.3.44 `void CbcOrClpParam::setLonghelp (const std::string help) [inline]`

Set Long help.

Definition at line 418 of file CbcOrClpParam.hpp.

4.22.3.45 `void CbcOrClpParam::printLongHelp () const`

Print Long help.

4.22.3.46 `void CbcOrClpParam::printString () const`

Print action and string.

4.22.3.47 `int CbcOrClpParam::whereUsed () const [inline]`

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

Definition at line 430 of file CbcOrClpParam.hpp.

4.22.3.48 `int CbcOrClpParam::fakeKeyWord () const [inline]`

Gets value of fake keyword.

Definition at line 434 of file CbcOrClpParam.hpp.

4.22.3.49 `void CbcOrClpParam::setFakeKeyWord (int value, int fakeValue) [inline]`

Sets value of fake keyword.

Definition at line 437 of file CbcOrClpParam.hpp.

4.22.3.50 `void CbcOrClpParam::setFakeKeyWord (int fakeValue)`

Sets value of fake keyword to current size of keywords.

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

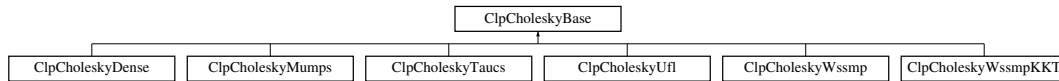
- [src/CbcOrClpParam.hpp](#)

4.23 ClpCholeskyBase Class Reference

Base class for Clp Cholesky factorization Will do better factorization.

```
#include <ClpCholeskyBase.hpp>
```

Inheritance diagram for ClpCholeskyBase:



Public Member Functions

Gets

- int [status](#) () const
status. Returns status
- int [numberOfRowsDropped](#) () const
numberOfRowsDropped. Number of rows gone
- void [resetRowsDropped](#) ()
reset numberOfRowsDropped 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 [numberOfRows](#) () 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.

4.23.2.2 virtual [ClpCholeskyBase::~~ClpCholeskyBase](#) () [virtual]

Destructor (has to be public)

4.23.2.3 [ClpCholeskyBase::ClpCholeskyBase](#) (const [ClpCholeskyBase](#) &)

Copy.

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](#), [ClpCholeskyWssmp-KKT](#), 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](#), [ClpCholeskyWssmp-KKT](#), 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::status () const` `[inline]`

status. Returns status

Definition at line 88 of file [ClpCholeskyBase.hpp](#).

4.23.3.7 `int ClpCholeskyBase::numberOfRowsDropped () const` `[inline]`

numberOfRowsDropped. Number of rows gone

Definition at line 92 of file [ClpCholeskyBase.hpp](#).

4.23.3.8 `void ClpCholeskyBase::resetRowsDropped ()`

reset numberOfRowsDropped and rowsDropped.

4.23.3.9 `char* ClpCholeskyBase::rowsDropped () const` `[inline]`

rowsDropped - which rows are gone

Definition at line 98 of file [ClpCholeskyBase.hpp](#).

4.23.3.10 `double ClpCholeskyBase::choleskyCondition () const` `[inline]`

choleskyCondition.

Definition at line 102 of file [ClpCholeskyBase.hpp](#).

4.23.3.11 `double ClpCholeskyBase::goDense () const` `[inline]`

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

Definition at line 106 of file [ClpCholeskyBase.hpp](#).

4.23.3.12 `void ClpCholeskyBase::setGoDense (double value) [inline]`

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

Definition at line 110 of file ClpCholeskyBase.hpp.

4.23.3.13 `int ClpCholeskyBase::rank () const [inline]`

rank. Returns rank

Definition at line 114 of file ClpCholeskyBase.hpp.

4.23.3.14 `int ClpCholeskyBase::numberOfRows () const [inline]`

Return number of rows.

Definition at line 118 of file ClpCholeskyBase.hpp.

4.23.3.15 `CoinBigIndex ClpCholeskyBase::size () const [inline]`

Return size.

Definition at line 122 of file ClpCholeskyBase.hpp.

4.23.3.16 `longDouble* ClpCholeskyBase::sparseFactor () const [inline]`

Return sparseFactor.

Definition at line 126 of file ClpCholeskyBase.hpp.

4.23.3.17 `longDouble* ClpCholeskyBase::diagonal () const [inline]`

Return diagonal.

Definition at line 130 of file ClpCholeskyBase.hpp.

4.23.3.18 `longDouble* ClpCholeskyBase::workDouble () const [inline]`

Return workDouble.

Definition at line 134 of file ClpCholeskyBase.hpp.

4.23.3.19 `bool ClpCholeskyBase::kkt () const [inline]`

If KKT on.

Definition at line 138 of file ClpCholeskyBase.hpp.

4.23.3.20 `void ClpCholeskyBase::setKKT (bool yesNo) [inline]`

Set KKT.

Definition at line 142 of file ClpCholeskyBase.hpp.

4.23.3.21 `void ClpCholeskyBase::setIntegerParameter (int i, int value) [inline]`

Set integer parameter.

Definition at line 146 of file ClpCholeskyBase.hpp.

4.23.3.22 `int ClpCholeskyBase::getIntegerParameter (int i) [inline]`

get integer parameter

Definition at line 150 of file ClpCholeskyBase.hpp.

4.23.3.23 `void ClpCholeskyBase::setDoubleParameter (int i, double value) [inline]`

Set double parameter.

Definition at line 154 of file ClpCholeskyBase.hpp.

4.23.3.24 `double ClpCholeskyBase::getDoubleParameter (int i) [inline]`

get double parameter

Definition at line 158 of file ClpCholeskyBase.hpp.

4.23.3.25 `ClpCholeskyBase& ClpCholeskyBase::operator= (const ClpCholeskyBase &)`

Assignment.

4.23.3.26 `virtual ClpCholeskyBase* ClpCholeskyBase::clone () const [virtual]`

Reimplemented in [ClpCholeskyDense](#), [ClpCholeskyTaucs](#), [ClpCholeskyUfl](#), [ClpCholeskyWssmpKKT](#), [ClpCholeskyMumps](#), and [ClpCholeskyWssmp](#).

4.23.3.27 `int ClpCholeskyBase::type () const [inline]`

Returns type.

Definition at line 185 of file ClpCholeskyBase.hpp.

4.23.3.28 `void ClpCholeskyBase::setType (int type) [inline],[protected]`

Sets type.

Definition at line 191 of file ClpCholeskyBase.hpp.

4.23.3.29 `void ClpCholeskyBase::setModel (ClpInterior * model) [inline],[protected]`

model.

Definition at line 195 of file ClpCholeskyBase.hpp.

4.23.3.30 `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

4.23.3.31 `void ClpCholeskyBase::symbolic2 (const CoinBigIndex * Astart, const int * Arow) [protected]`

Symbolic2 - Fills in indices Uses lower triangular so can do cliques etc.

4.23.3.32 `void ClpCholeskyBase::factorizePart2 (int * rowsDropped) [protected]`

Factorize - filling in rowsDropped and returning number dropped in integerParam.

4.23.3.33 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

4.23.3.34 int ClpCholeskyBase::preOrder (bool *lowerTriangular*, bool *includeDiagonal*, bool *doKKT*) [protected]

Forms ADAT - returns nonzero if not enough memory.

4.23.3.35 void ClpCholeskyBase::updateDense (longDouble * *d*, int * *first*) [protected]

Updates dense part (broken out for profiling)

4.23.4 Member Data Documentation

4.23.4.1 int ClpCholeskyBase::type_ [protected]

type (may be useful) if > 20 do KKT

Definition at line 230 of file ClpCholeskyBase.hpp.

4.23.4.2 bool ClpCholeskyBase::doKKT_ [protected]

Doing full KKT (only used if default symbolic and factorization)

Definition at line 232 of file ClpCholeskyBase.hpp.

4.23.4.3 double ClpCholeskyBase::goDense_ [protected]

Go dense at this fraction.

Definition at line 234 of file ClpCholeskyBase.hpp.

4.23.4.4 double ClpCholeskyBase::choleskyCondition_ [protected]

choleskyCondition.

Definition at line 236 of file ClpCholeskyBase.hpp.

4.23.4.5 ClpInterior* ClpCholeskyBase::model_ [protected]

model.

Definition at line 238 of file ClpCholeskyBase.hpp.

4.23.4.6 int ClpCholeskyBase::numberTrials_ [protected]

numberTrials. Number of trials before rejection

Definition at line 240 of file ClpCholeskyBase.hpp.

4.23.4.7 int ClpCholeskyBase::numberRows_ [protected]

numberRows. Number of Rows in factorization

Definition at line 242 of file ClpCholeskyBase.hpp.

4.23.4.8 `int ClpCholeskyBase::status_` [protected]

status. Status of factorization

Definition at line 244 of file ClpCholeskyBase.hpp.

4.23.4.9 `char* ClpCholeskyBase::rowsDropped_` [protected]

rowsDropped

Definition at line 246 of file ClpCholeskyBase.hpp.

4.23.4.10 `int* ClpCholeskyBase::permuteInverse_` [protected]

permute inverse.

Definition at line 248 of file ClpCholeskyBase.hpp.

4.23.4.11 `int* ClpCholeskyBase::permute_` [protected]

main permute.

Definition at line 250 of file ClpCholeskyBase.hpp.

4.23.4.12 `int ClpCholeskyBase::numberOfRowsDropped_` [protected]

numberOfRowsDropped. Number of rows gone

Definition at line 252 of file ClpCholeskyBase.hpp.

4.23.4.13 `longDouble* ClpCholeskyBase::sparseFactor_` [protected]

sparseFactor.

Definition at line 254 of file ClpCholeskyBase.hpp.

4.23.4.14 `CoinBigIndex* ClpCholeskyBase::choleskyStart_` [protected]

choleskyStart - element starts

Definition at line 256 of file ClpCholeskyBase.hpp.

4.23.4.15 `int* ClpCholeskyBase::choleskyRow_` [protected]

choleskyRow (can be shorter than sparsefactor)

Definition at line 258 of file ClpCholeskyBase.hpp.

4.23.4.16 `CoinBigIndex* ClpCholeskyBase::indexStart_` [protected]

Index starts.

Definition at line 260 of file ClpCholeskyBase.hpp.

4.23.4.17 `longDouble* ClpCholeskyBase::diagonal_` [protected]

Diagonal.

Definition at line 262 of file ClpCholeskyBase.hpp.

4.23.4.18 `longDouble* ClpCholeskyBase::workDouble_` [protected]

double work array

Definition at line 264 of file ClpCholeskyBase.hpp.

4.23.4.19 `int* ClpCholeskyBase::link_` [protected]

link array

Definition at line 266 of file ClpCholeskyBase.hpp.

4.23.4.20 `CoinBigIndex* ClpCholeskyBase::workInteger_` [protected]

Definition at line 268 of file ClpCholeskyBase.hpp.

4.23.4.21 `int* ClpCholeskyBase::clique_` [protected]

Definition at line 270 of file ClpCholeskyBase.hpp.

4.23.4.22 `CoinBigIndex ClpCholeskyBase::sizeFactor_` [protected]

sizeFactor.

Definition at line 272 of file ClpCholeskyBase.hpp.

4.23.4.23 `CoinBigIndex ClpCholeskyBase::sizeIndex_` [protected]

Size of index array.

Definition at line 274 of file ClpCholeskyBase.hpp.

4.23.4.24 `int ClpCholeskyBase::firstDense_` [protected]

First dense row.

Definition at line 276 of file ClpCholeskyBase.hpp.

4.23.4.25 `int ClpCholeskyBase::integerParameters_[64]` [protected]

integerParameters

Definition at line 278 of file ClpCholeskyBase.hpp.

4.23.4.26 `double ClpCholeskyBase::doubleParameters_[64]` [protected]

doubleParameters;

Definition at line 280 of file ClpCholeskyBase.hpp.

4.23.4.27 `ClpMatrixBase* ClpCholeskyBase::rowCopy_` [protected]

Row copy of matrix.

Definition at line 282 of file ClpCholeskyBase.hpp.

4.23.4.28 `char* ClpCholeskyBase::whichDense_` [protected]

Dense indicators.

Definition at line 284 of file ClpCholeskyBase.hpp.

4.23.4.29 `longDouble* ClpCholeskyBase::denseColumn_` [protected]

Dense columns (updated)

Definition at line 286 of file `ClpCholeskyBase.hpp`.

4.23.4.30 `ClpCholeskyDense* ClpCholeskyBase::dense_` [protected]

Dense cholesky.

Definition at line 288 of file `ClpCholeskyBase.hpp`.

4.23.4.31 `int ClpCholeskyBase::denseThreshold_` [protected]

Dense threshold (for taking out of Cholesky)

Definition at line 290 of file `ClpCholeskyBase.hpp`.

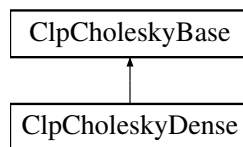
The documentation for this class was generated from the following file:

- [src/ClpCholeskyBase.hpp](#)

4.24 ClpCholeskyDense Class Reference

```
#include <ClpCholeskyDense.hpp>
```

Inheritance 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 [numberOfRows](#))
Reserves space.
- [CoinBigIndex](#) [space](#) (int [numberOfRows](#)) 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.

4.24.2.2 `virtual ClpCholeskyDense::~~ClpCholeskyDense ()` [virtual]

Destructor.

4.24.2.3 `ClpCholeskyDense::ClpCholeskyDense (const ClpCholeskyDense &)`

Copy.

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 numberOfRows)`

Reserves space.

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

4.24.3.6 `CoinBigIndex ClpCholeskyDense::space (int numberOfRows) const`

Returns space needed.

4.24.3.7 `void ClpCholeskyDense::factorizePart2 (int * rowsDropped)`

part 2 of Factorize - filling in rowsDropped

4.24.3.8 `void ClpCholeskyDense::factorizePart3 (int * rowsDropped)`

part 2 of Factorize - filling in rowsDropped - blocked

4.24.3.9 `void ClpCholeskyDense::solveF1 (longDouble * a, int n, CoinWorkDouble * region)`

Forward part of solve.

4.24.3.10 `void ClpCholeskyDense::solveF2 (longDouble * a, int n, CoinWorkDouble * region, CoinWorkDouble * region2)`

4.24.3.11 `void ClpCholeskyDense::solveB1 (longDouble * a, int n, CoinWorkDouble * region)`

Backward part of solve.

4.24.3.12 `void ClpCholeskyDense::solveB2 (longDouble * a, int n, CoinWorkDouble * region, CoinWorkDouble * region2)`

4.24.3.13 `int ClpCholeskyDense::bNumber (const longDouble * array, int &, int &)`

4.24.3.14 `longDouble* ClpCholeskyDense::aMatrix () const [inline]`

A.

Definition at line 54 of file ClpCholeskyDense.hpp.

4.24.3.15 `longDouble* ClpCholeskyDense::diagonal () const` `[inline]`

Diagonal.

Definition at line 58 of file `ClpCholeskyDense.hpp`.

4.24.3.16 `ClpCholeskyDense& ClpCholeskyDense::operator= (const ClpCholeskyDense &)`

Assignment.

4.24.3.17 `virtual ClpCholeskyBase* ClpCholeskyDense::clone () const` `[virtual]`

Clone.

Reimplemented from [ClpCholeskyBase](#).

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

- [src/ClpCholeskyDense.hpp](#)

4.25 ClpCholeskyDenseC Struct Reference

```
#include <ClpCholeskyDense.hpp>
```

Public Attributes

- [longDouble * diagonal_](#)
- [longDouble * a](#)
- [longDouble * work](#)
- [int * rowsDropped](#)
- [double doubleParameters_ \[1\]](#)
- [int integerParameters_ \[2\]](#)
- [int n](#)
- [int numberBlocks](#)

4.25.1 Detailed Description

Definition at line 88 of file `ClpCholeskyDense.hpp`.

4.25.2 Member Data Documentation

4.25.2.1 `longDouble* ClpCholeskyDenseC::diagonal_`

Definition at line 89 of file `ClpCholeskyDense.hpp`.

4.25.2.2 `longDouble* ClpCholeskyDenseC::a`

Definition at line 90 of file `ClpCholeskyDense.hpp`.

4.25.2.3 `longDouble* ClpCholeskyDenseC::work`

Definition at line 91 of file `ClpCholeskyDense.hpp`.

4.25.2.4 `int* ClpCholeskyDenseC::rowsDropped`

Definition at line 92 of file `ClpCholeskyDense.hpp`.

4.25.2.5 `double ClpCholeskyDenseC::doubleParameters_[1]`

Definition at line 93 of file `ClpCholeskyDense.hpp`.

4.25.2.6 `int ClpCholeskyDenseC::integerParameters_[2]`

Definition at line 94 of file `ClpCholeskyDense.hpp`.

4.25.2.7 `int ClpCholeskyDenseC::n`

Definition at line 95 of file `ClpCholeskyDense.hpp`.

4.25.2.8 `int ClpCholeskyDenseC::numberBlocks`

Definition at line 96 of file `ClpCholeskyDense.hpp`.

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

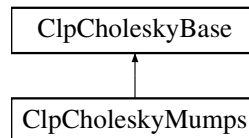
- [src/ClpCholeskyDense.hpp](#)

4.26 ClpCholeskyMumps Class Reference

Mumps class for Clp Cholesky factorization.

```
#include <ClpCholeskyMumps.hpp>
```

Inheritance 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.

4.26.2.2 virtual [ClpCholeskyMumps::~ClpCholeskyMumps](#) () [virtual]

Destructor.

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.

4.26.3.5 virtual [ClpCholeskyBase](#)* [ClpCholeskyMumps::clone](#) () const [virtual]

Clone.

Reimplemented from [ClpCholeskyBase](#).

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

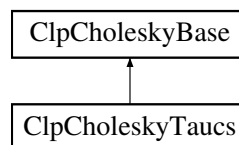
- [src/ClpCholeskyMumps.hpp](#)

4.27 ClpCholeskyTaucs Class Reference

Taucs class for Clp Cholesky factorization.

```
#include <ClpCholeskyTaucs.hpp>
```

Inheritance 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 dptf2.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.

4.27.2.2 virtual ClpCholeskyTaucs::~~ClpCholeskyTaucs () [virtual]

Destructor.

4.27.2.3 ClpCholeskyTaucs::ClpCholeskyTaucs (const ClpCholeskyTaucs &)

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::symbolic () [virtual]

Dummy.

Reimplemented from [ClpCholeskyBase](#).

4.27.3.3 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

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

Uses factorization to solve.

4.27.3.5 ClpCholeskyTaucs& ClpCholeskyTaucs::operator= (const ClpCholeskyTaucs &)

4.27.3.6 virtual ClpCholeskyBase* ClpCholeskyTaucs::clone () const [virtual]

Clone.

Reimplemented from [ClpCholeskyBase](#).

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

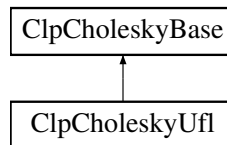
- [src/ClpCholeskyTaucs.hpp](#)

4.28 ClpCholeskyUfl Class Reference

Ufl class for Clp Cholesky factorization.

```
#include <ClpCholeskyUfl.hpp>
```

Inheritance 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.

4.28.2.2 virtual ClpCholeskyUfl::~~ClpCholeskyUfl () [virtual]

Destructor.

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).

4.28.3.5 virtual ClpCholeskyBase* ClpCholeskyUfl::clone () const [virtual]

Clone.

Reimplemented from [ClpCholeskyBase](#).

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

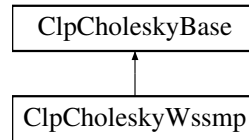
- [src/ClpCholeskyUfl.hpp](#)

4.29 ClpCholeskyWssmp Class Reference

Wssmp class for Clp Cholesky factorization.

```
#include <ClpCholeskyWssmp.hpp>
```

Inheritance 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.

4.29.2.2 `virtual ClpCholeskyWssmp::~~ClpCholeskyWssmp () [virtual]`

Destructor.

4.29.2.3 `ClpCholeskyWssmp::ClpCholeskyWssmp (const ClpCholeskyWssmp &)`

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.

4.29.3.5 `ClpCholeskyWssmp& ClpCholeskyWssmp::operator= (const ClpCholeskyWssmp &)`

4.29.3.6 `virtual ClpCholeskyBase* ClpCholeskyWssmp::clone () const [virtual]`

Clone.

Reimplemented from [ClpCholeskyBase](#).

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

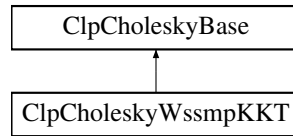
- [src/ClpCholeskyWssmp.hpp](#)

4.30 ClpCholeskyWssmpKKT Class Reference

WssmpKKT class for Clp Cholesky factorization.

```
#include <ClpCholeskyWssmpKKT.hpp>
```

Inheritance 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.

4.30.2.2 `virtual ClpCholeskyWssmpKKT::~~ClpCholeskyWssmpKKT ()` [virtual]

Destructor.

4.30.2.3 ClpCholeskyWssmpKKT::ClpCholeskyWssmpKKT (const ClpCholeskyWssmpKKT &)

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

4.30.3.6 ClpCholeskyWssmpKKT& ClpCholeskyWssmpKKT::operator= (const ClpCholeskyWssmpKKT &)

4.30.3.7 virtual ClpCholeskyBase* ClpCholeskyWssmpKKT::clone () const [virtual]

Clone.

Reimplemented from [ClpCholeskyBase](#).

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

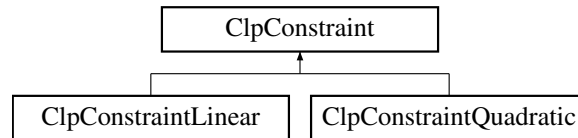
- [src/ClpCholeskyWssmpKKT.hpp](#)

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 Constructor & Destructor Documentation

4.31.2.1 ClpConstraint::ClpConstraint ()

Default Constructor.

4.31.2.2 ClpConstraint::ClpConstraint (const ClpConstraint &)

Copy constructor.

4.31.2.3 virtual ClpConstraint::~~ClpConstraint () [virtual]

Destructor.

4.31.3 Member Function Documentation

4.31.3.1 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](#).

4.31.3.2 virtual double ClpConstraint::functionValue (const ClpSimplex * *model*, const double * *solution*, bool *useScaling* = false, bool *refresh* = true) const [virtual]

Constraint function value.

4.31.3.3 `virtual void ClpConstraint::resize (int newNumberColumns) [pure virtual]`

Resize constraint.

Implemented in [ClpConstraintLinear](#), and [ClpConstraintQuadratic](#).

4.31.3.4 `virtual void ClpConstraint::deleteSome (int numberToDelete, const int * which) [pure virtual]`

Delete columns in constraint.

Implemented in [ClpConstraintLinear](#), and [ClpConstraintQuadratic](#).

4.31.3.5 `virtual void ClpConstraint::reallyScale (const double * columnScale) [pure virtual]`

Scale constraint.

Implemented in [ClpConstraintLinear](#), and [ClpConstraintQuadratic](#).

4.31.3.6 `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](#).

4.31.3.7 `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](#).

4.31.3.8 `ClpConstraint& ClpConstraint::operator= (const ClpConstraint & rhs)`

Assignment operator.

4.31.3.9 `virtual ClpConstraint* ClpConstraint::clone () const [pure virtual]`

Clone.

Implemented in [ClpConstraintQuadratic](#), and [ClpConstraintLinear](#).

4.31.3.10 `int ClpConstraint::type () [inline]`

Returns type, 0 linear, 1 nonlinear.

Definition at line 83 of file [ClpConstraint.hpp](#).

4.31.3.11 `int ClpConstraint::rowNumber () const [inline]`

Row number (-1 is objective)

Definition at line 87 of file [ClpConstraint.hpp](#).

4.31.3.12 `virtual int ClpConstraint::numberCoefficients () const [pure virtual]`

Number of possible coefficients in gradient.

Implemented in [ClpConstraintQuadratic](#), and [ClpConstraintLinear](#).

4.31.3.13 `double ClpConstraint::functionValue () const [inline]`

Stored constraint function value.

Definition at line 95 of file `ClpConstraint.hpp`.

4.31.3.14 `double ClpConstraint::offset () const [inline]`

Constraint offset.

Definition at line 100 of file `ClpConstraint.hpp`.

4.31.3.15 `virtual void ClpConstraint::newXValues () [inline],[virtual]`

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

Definition at line 104 of file `ClpConstraint.hpp`.

4.31.4 Member Data Documentation

4.31.4.1 `double* ClpConstraint::lastGradient_ [mutable],[protected]`

Gradient at last evaluation.

Definition at line 113 of file `ClpConstraint.hpp`.

4.31.4.2 `double ClpConstraint::functionValue_ [mutable],[protected]`

Value of non-linear part of constraint.

Definition at line 115 of file `ClpConstraint.hpp`.

4.31.4.3 `double ClpConstraint::offset_ [mutable],[protected]`

Value of offset for constraint.

Definition at line 117 of file `ClpConstraint.hpp`.

4.31.4.4 `int ClpConstraint::type_ [protected]`

Type of constraint - linear is 1.

Definition at line 119 of file `ClpConstraint.hpp`.

4.31.4.5 `int ClpConstraint::rowNumber_ [protected]`

Row number (-1 is objective)

Definition at line 121 of file `ClpConstraint.hpp`.

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

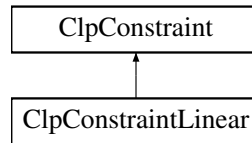
- [src/ClpConstraint.hpp](#)

4.32 ClpConstraintLinear Class Reference

Linear Constraint Class.

```
#include <ClpConstraintLinear.hpp>
```

Inheritance 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 Constructor & Destructor Documentation

4.32.2.1 ClpConstraintLinear::ClpConstraintLinear ()

Default Constructor.

4.32.2.2 ClpConstraintLinear::ClpConstraintLinear (int row, int *numberCoefficients*, int *numberColumns*, const int * *column*, const double * *element*)

Constructor from constraint.

4.32.2.3 ClpConstraintLinear::ClpConstraintLinear (const ClpConstraintLinear & *rhs*)

Copy constructor .

4.32.2.4 virtual ClpConstraintLinear::~~ClpConstraintLinear () [virtual]

Destructor.

4.32.3 Member Function Documentation

4.32.3.1 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 undefined at current solution

Implements [ClpConstraint](#).

4.32.3.2 virtual void ClpConstraintLinear::resize (int *newNumberColumns*) [virtual]

Resize constraint.

Implements [ClpConstraint](#).

4.32.3.3 virtual void ClpConstraintLinear::deleteSome (int *numberToDelete*, const int * *which*) [virtual]

Delete columns in constraint.

Implements [ClpConstraint](#).

4.32.3.4 virtual void ClpConstraintLinear::reallyScale (const double * *columnScale*) [virtual]

Scale constraint.

Implements [ClpConstraint](#).

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

Given a zeroed array sets nonlinear columns to 1.

Returns number of nonlinear columns

Implements [ClpConstraint](#).

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

Given a zeroed array sets possible nonzero coefficients to 1.

Returns number of nonzeros

Implements [ClpConstraint](#).

4.32.3.7 `ClpConstraintLinear& ClpConstraintLinear::operator= (const ClpConstraintLinear & rhs)`

Assignment operator.

4.32.3.8 `virtual ClpConstraint* ClpConstraintLinear::clone () const [virtual]`

Clone.

Implements [ClpConstraint](#).

4.32.3.9 `virtual int ClpConstraintLinear::numberCoefficients () const [virtual]`

Number of coefficients.

Implements [ClpConstraint](#).

4.32.3.10 `int ClpConstraintLinear::numberColumns () const [inline]`

Number of columns in linear constraint.

Definition at line 82 of file `ClpConstraintLinear.hpp`.

4.32.3.11 `const int* ClpConstraintLinear::column () const [inline]`

Columns.

Definition at line 86 of file `ClpConstraintLinear.hpp`.

4.32.3.12 `const double* ClpConstraintLinear::coefficient () const [inline]`

Coefficients.

Definition at line 90 of file `ClpConstraintLinear.hpp`.

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

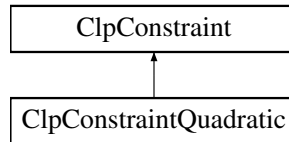
- [src/ClpConstraintLinear.hpp](#)

4.33 ClpConstraintQuadratic Class Reference

Quadratic Constraint Class.

```
#include <ClpConstraintQuadratic.hpp>
```

Inheritance 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 Constructor & Destructor Documentation

4.33.2.1 ClpConstraintQuadratic::ClpConstraintQuadratic ()

Default Constructor.

4.33.2.2 ClpConstraintQuadratic::ClpConstraintQuadratic (int *row*, int *numberQuadraticColumns*, int *numberColumns*, const CoinBigIndex * *start*, const int * *column*, const double * *element*)

Constructor from quadratic.

4.33.2.3 ClpConstraintQuadratic::ClpConstraintQuadratic (const ClpConstraintQuadratic & *rhs*)

Copy constructor .

4.33.2.4 virtual ClpConstraintQuadratic::~~ClpConstraintQuadratic () [virtual]

Destructor.

4.33.3 Member Function Documentation

4.33.3.1 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 undefined at current solution

Implements [ClpConstraint](#).

4.33.3.2 virtual void ClpConstraintQuadratic::resize (int *newNumberColumns*) [virtual]

Resize constraint.

Implements [ClpConstraint](#).

4.33.3.3 virtual void ClpConstraintQuadratic::deleteSome (int *numberToDelete*, const int * *which*) [virtual]

Delete columns in constraint.

Implements [ClpConstraint](#).

4.33.3.4 virtual void ClpConstraintQuadratic::reallyScale (const double * *columnScale*) [virtual]

Scale constraint.

Implements [ClpConstraint](#).

4.33.3.5 `virtual int ClpConstraintQuadratic::markNonlinear (char * which) const` [virtual]

Given a zeroed array sets nonquadratic columns to 1.

Returns number of nonquadratic columns

Implements [ClpConstraint](#).

4.33.3.6 `virtual int ClpConstraintQuadratic::markNonzero (char * which) const` [virtual]

Given a zeroed array sets possible nonzero coefficients to 1.

Returns number of nonzeros

Implements [ClpConstraint](#).

4.33.3.7 `ClpConstraintQuadratic& ClpConstraintQuadratic::operator= (const ClpConstraintQuadratic & rhs)`

Assignment operator.

4.33.3.8 `virtual ClpConstraint* ClpConstraintQuadratic::clone () const` [virtual]

Clone.

Implements [ClpConstraint](#).

4.33.3.9 `virtual int ClpConstraintQuadratic::numberCoefficients () const` [virtual]

Number of coefficients.

Implements [ClpConstraint](#).

4.33.3.10 `int ClpConstraintQuadratic::numberColumns () const` [inline]

Number of columns in constraint.

Definition at line 83 of file `ClpConstraintQuadratic.hpp`.

4.33.3.11 `CoinBigIndex* ClpConstraintQuadratic::start () const` [inline]

Column starts.

Definition at line 87 of file `ClpConstraintQuadratic.hpp`.

4.33.3.12 `const int* ClpConstraintQuadratic::column () const` [inline]

Columns.

Definition at line 91 of file `ClpConstraintQuadratic.hpp`.

4.33.3.13 `const double* ClpConstraintQuadratic::coefficient () const` [inline]

Coefficients.

Definition at line 95 of file `ClpConstraintQuadratic.hpp`.

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

- [src/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 1267 of file ClpModel.hpp.

4.34.2 Constructor & Destructor Documentation

4.34.2.1 [ClpDataSave::ClpDataSave](#) ()

Default constructor.

4.34.2.2 [ClpDataSave::ClpDataSave](#) (const [ClpDataSave](#) &)

Copy constructor.

4.34.2.3 [ClpDataSave::~~ClpDataSave](#) ()

Destructor.

4.34.3 Member Function Documentation

4.34.3.1 `ClpDataSave& ClpDataSave::operator= (const ClpDataSave & rhs)`

Assignment operator. This copies the data.

4.34.4 Member Data Documentation

4.34.4.1 `double ClpDataSave::dualBound_`

Definition at line 1290 of file `ClpModel.hpp`.

4.34.4.2 `double ClpDataSave::infeasibilityCost_`

Definition at line 1291 of file `ClpModel.hpp`.

4.34.4.3 `double ClpDataSave::pivotTolerance_`

Definition at line 1292 of file `ClpModel.hpp`.

4.34.4.4 `double ClpDataSave::zeroFactorizationTolerance_`

Definition at line 1293 of file `ClpModel.hpp`.

4.34.4.5 `double ClpDataSave::zeroSimplexTolerance_`

Definition at line 1294 of file `ClpModel.hpp`.

4.34.4.6 `double ClpDataSave::acceptablePivot_`

Definition at line 1295 of file `ClpModel.hpp`.

4.34.4.7 `double ClpDataSave::objectiveScale_`

Definition at line 1296 of file `ClpModel.hpp`.

4.34.4.8 `int ClpDataSave::sparseThreshold_`

Definition at line 1297 of file `ClpModel.hpp`.

4.34.4.9 `int ClpDataSave::perturbation_`

Definition at line 1298 of file `ClpModel.hpp`.

4.34.4.10 `int ClpDataSave::forceFactorization_`

Definition at line 1299 of file `ClpModel.hpp`.

4.34.4.11 `int ClpDataSave::scalingFlag_`

Definition at line 1300 of file `ClpModel.hpp`.

4.34.4.12 `unsigned int ClpDataSave::specialOptions_`

Definition at line 1301 of file `ClpModel.hpp`.

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

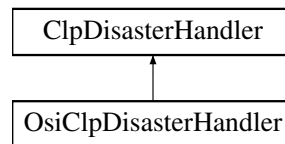
- [src/ClpModel.hpp](#)

4.35 ClpDisasterHandler Class Reference

Base class for Clp disaster handling.

```
#include <ClpEventHandler.hpp>
```

Inheritance 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 133 of file ClpEventHandler.hpp.

4.35.2 Constructor & Destructor Documentation

4.35.2.1 ClpDisasterHandler::ClpDisasterHandler (ClpSimplex * *model* = NULL)

Default constructor.

4.35.2.2 virtual ClpDisasterHandler::~ClpDisasterHandler () [virtual]

Destructor.

4.35.2.3 ClpDisasterHandler::ClpDisasterHandler (const ClpDisasterHandler &)

4.35.3 Member Function Documentation

4.35.3.1 virtual void ClpDisasterHandler::intoSimplex () [pure virtual]

Into simplex.

Implemented in [OsiClpDisasterHandler](#).

4.35.3.2 virtual bool ClpDisasterHandler::check () const [pure virtual]

Checks if disaster.

Implemented in [OsiClpDisasterHandler](#).

4.35.3.3 virtual void ClpDisasterHandler::saveInfo () [pure virtual]

saves information for next attempt

Implemented in [OsiClpDisasterHandler](#).

4.35.3.4 virtual int ClpDisasterHandler::typeOfDisaster () [virtual]

Type of disaster 0 can fix, 1 abort.

Reimplemented in [OsiClpDisasterHandler](#).

4.35.3.5 ClpDisasterHandler& ClpDisasterHandler::operator= (const ClpDisasterHandler &)

4.35.3.6 virtual ClpDisasterHandler* ClpDisasterHandler::clone () const [pure virtual]

Clone.

Implemented in [OsiClpDisasterHandler](#).

4.35.3.7 void ClpDisasterHandler::setSimplex (ClpSimplex * *model*)

set model.

4.35.3.8 ClpSimplex* ClpDisasterHandler::simplex () const [inline]

Get model.

Definition at line 172 of file ClpEventHandler.hpp.

4.35.4 Member Data Documentation

4.35.4.1 ClpSimplex* ClpDisasterHandler::model_ [protected]

Pointer to simplex.

Definition at line 183 of file ClpEventHandler.hpp.

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

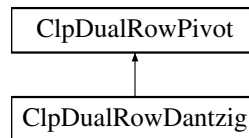
- [src/ClpEventHandler.hpp](#)

4.36 ClpDualRowDantzig Class Reference

Dual Row Pivot Dantzig Algorithm Class.

```
#include <ClpDualRowDantzig.hpp>
```

Inheritance 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 Constructor & Destructor Documentation

4.36.2.1 [ClpDualRowDantzig::ClpDualRowDantzig](#) ()

Default Constructor.

4.36.2.2 [ClpDualRowDantzig::ClpDualRowDantzig](#) (const [ClpDualRowDantzig](#) &)

Copy constructor.

4.36.2.3 virtual [ClpDualRowDantzig::~~ClpDualRowDantzig](#) () [virtual]

Destructor.

4.36.3 Member Function Documentation

4.36.3.1 virtual int [ClpDualRowDantzig::pivotRow](#) () [virtual]

Returns pivot row, -1 if none.

Implements [ClpDualRowPivot](#).

4.36.3.2 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](#).

4.36.3.3 virtual void [ClpDualRowDantzig::updatePrimalSolution](#) ([CoinIndexedVector](#) * *input*, double *theta*, double & *changeInObjective*) [virtual]

Updates primal solution (and maybe list of candidates) Uses input vector which it deletes Computes change in objective function.

Implements [ClpDualRowPivot](#).

4.36.3.4 [ClpDualRowDantzig& ClpDualRowDantzig::operator=](#) (const [ClpDualRowDantzig](#) & *rhs*)

Assignment operator.

4.36.3.5 `virtual ClpDualRowPivot* ClpDualRowDantzig::clone (bool copyData = true) const` [virtual]

Clone.

Implements [ClpDualRowPivot](#).

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

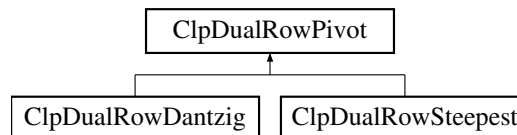
- [src/ClpDualRowDantzig.hpp](#)

4.37 ClpDualRowPivot Class Reference

Dual Row Pivot Abstract Base Class.

```
#include <ClpDualRowPivot.hpp>
```

Inheritance 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 Constructor & Destructor Documentation

4.37.2.1 `ClpDualRowPivot::ClpDualRowPivot ()`

Default Constructor.

4.37.2.2 `ClpDualRowPivot::ClpDualRowPivot (const ClpDualRowPivot &)`

Copy constructor.

4.37.2.3 `virtual ClpDualRowPivot::~~ClpDualRowPivot ()` [virtual]

Destructor.

4.37.3 Member Function Documentation

4.37.3.1 `virtual int ClpDualRowPivot::pivotRow () [pure virtual]`

Returns pivot row, -1 if none.

Implemented in [ClpDualRowSteepest](#), and [ClpDualRowDantzig](#).

4.37.3.2 `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](#).

4.37.3.3 `virtual void ClpDualRowPivot::updatePrimalSolution (CoinIndexedVector * input, double theta, double & changeInObjective) [pure virtual]`

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.

Implemented in [ClpDualRowSteepest](#), and [ClpDualRowDantzig](#).

4.37.3.4 `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 , infeasibilities

Reimplemented in [ClpDualRowSteepest](#).

4.37.3.5 `virtual void ClpDualRowPivot::checkAccuracy () [virtual]`

checks accuracy and may re-initialize (may be empty)

4.37.3.6 `virtual void ClpDualRowPivot::unrollWeights () [virtual]`

Gets rid of last update (may be empty)

Reimplemented in [ClpDualRowSteepest](#).

4.37.3.7 `virtual void ClpDualRowPivot::clearArrays () [virtual]`

Gets rid of all arrays (may be empty)

Reimplemented in [ClpDualRowSteepest](#).

4.37.3.8 `virtual bool ClpDualRowPivot::looksOptimal () const [inline],[virtual]`

Returns true if would not find any row.

Reimplemented in [ClpDualRowSteepest](#).

Definition at line 67 of file [ClpDualRowPivot.hpp](#).

4.37.3.9 `virtual void ClpDualRowPivot::maximumPivotsChanged () [inline],[virtual]`

Called when maximum pivots changes.

Reimplemented in [ClpDualRowSteepest](#).

Definition at line 71 of file ClpDualRowPivot.hpp.

4.37.3.10 `ClpDualRowPivot& ClpDualRowPivot::operator= (const ClpDualRowPivot & rhs)`

Assignment operator.

4.37.3.11 `virtual ClpDualRowPivot* ClpDualRowPivot::clone (bool copyData = true) const` [pure virtual]

Clone.

Implemented in [ClpDualRowSteepest](#), and [ClpDualRowDantzig](#).

4.37.3.12 `ClpSimplex* ClpDualRowPivot::model ()` [inline]

Returns model.

Definition at line 97 of file ClpDualRowPivot.hpp.

4.37.3.13 `void ClpDualRowPivot::setModel (ClpSimplex * newmodel)` [inline]

Sets model (normally to NULL)

Definition at line 102 of file ClpDualRowPivot.hpp.

4.37.3.14 `int ClpDualRowPivot::type ()` [inline]

Returns type (above 63 is extra information)

Definition at line 107 of file ClpDualRowPivot.hpp.

4.37.4 Member Data Documentation

4.37.4.1 `ClpSimplex* ClpDualRowPivot::model_` [protected]

Pointer to model.

Definition at line 119 of file ClpDualRowPivot.hpp.

4.37.4.2 `int ClpDualRowPivot::type_` [protected]

Type of row pivot algorithm.

Definition at line 121 of file ClpDualRowPivot.hpp.

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

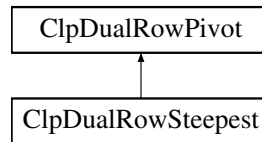
- [src/ClpDualRowPivot.hpp](#)

4.38 ClpDualRowSteepest Class Reference

Dual Row Pivot Steepest Edge Algorithm Class.

```
#include <ClpDualRowSteepest.hpp>
```

Inheritance diagram for ClpDualRowSteepest:



Public Types

- enum [Persistence](#) { [normal](#) = 0x00, [keep](#) = 0x01 }
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.
- 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 `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 Member Enumeration Documentation

4.38.2.1 enum `ClpDualRowSteepest::Persistence`

enums for persistence

Enumerator

normal

keep

Definition at line 69 of file `ClpDualRowSteepest.hpp`.

4.38.3 Constructor & Destructor Documentation

4.38.3.1 `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.

4.38.3.2 `ClpDualRowSteepest::ClpDualRowSteepest (const ClpDualRowSteepest &)`

Copy constructor.

4.38.3.3 `virtual ClpDualRowSteepest::~ClpDualRowSteepest () [virtual]`

Destructor.

4.38.4 Member Function Documentation

4.38.4.1 `virtual int ClpDualRowSteepest::pivotRow () [virtual]`

Returns pivot row, -1 if none.

Implements `ClpDualRowPivot`.

4.38.4.2 `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](#).

4.38.4.3 `virtual void ClpDualRowSteepest::updatePrimalSolution (CoinIndexedVector * input, double theta, double & changeInObjective) [virtual]`

Updates primal solution (and maybe list of candidates) Uses input vector which it deletes Computes change in objective function.

Implements [ClpDualRowPivot](#).

4.38.4.4 `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](#).

4.38.4.5 `virtual void ClpDualRowSteepest::unrollWeights () [virtual]`

Gets rid of last update.

Reimplemented from [ClpDualRowPivot](#).

4.38.4.6 `virtual void ClpDualRowSteepest::clearArrays () [virtual]`

Gets rid of all arrays.

Reimplemented from [ClpDualRowPivot](#).

4.38.4.7 `virtual bool ClpDualRowSteepest::looksOptimal () const [virtual]`

Returns true if would not find any row.

Reimplemented from [ClpDualRowPivot](#).

4.38.4.8 `virtual void ClpDualRowSteepest::maximumPivotsChanged () [virtual]`

Called when maximum pivots changes.

Reimplemented from [ClpDualRowPivot](#).

4.38.4.9 `ClpDualRowSteepest& ClpDualRowSteepest::operator= (const ClpDualRowSteepest & rhs)`

Assignment operator.

4.38.4.10 `void ClpDualRowSteepest::fill (const ClpDualRowSteepest & rhs)`

Fill most values.

4.38.4.11 `virtual ClpDualRowPivot* ClpDualRowSteepest::clone (bool copyData =true) const` [virtual]

Clone.

Implements [ClpDualRowPivot](#).

4.38.4.12 `int ClpDualRowSteepest::mode () const` [inline]

Mode.

Definition at line 104 of file `ClpDualRowSteepest.hpp`.

4.38.4.13 `void ClpDualRowSteepest::setPersistence (Persistence life)` [inline]

Set/ get persistence.

Definition at line 108 of file `ClpDualRowSteepest.hpp`.

4.38.4.14 `Persistence ClpDualRowSteepest::persistence () const` [inline]

Definition at line 111 of file `ClpDualRowSteepest.hpp`.

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

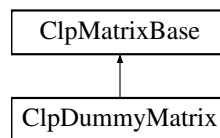
- [src/ClpDualRowSteepest.hpp](#)

4.39 ClpDummyMatrix Class Reference

This implements a dummy matrix as derived from [ClpMatrixBase](#).

```
#include <ClpDummyMatrix.hpp>
```

Inheritance 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 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 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 *columnScale) 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, 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 .*

Constructors, destructor

- [ClpDummyMatrix](#) ()
Default constructor.
- [ClpDummyMatrix](#) (int numberColumns, int numberOfRows, 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.

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 \(int *numberColumns*, int *numberRows*, int *numberElements* \)](#)

Constructor with data.

4.39.2.3 [virtual ClpDummyMatrix::~~ClpDummyMatrix \(\)](#) [virtual]

Destructor.

4.39.2.4 [ClpDummyMatrix::ClpDummyMatrix \(const \[ClpDummyMatrix\]\(#\) & \)](#)

The copy constructor.

4.39.2.5 [ClpDummyMatrix::ClpDummyMatrix \(const \[CoinPackedMatrix\]\(#\) & \)](#)

The copy constructor from an [CoinDummyMatrix](#).

4.39.3 Member Function Documentation

4.39.3.1 `virtual CoinPackedMatrix* ClpDummyMatrix::getPackedMatrix () const [virtual]`

Return a complete CoinPackedMatrix.

Implements [ClpMatrixBase](#).

4.39.3.2 `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.

4.39.3.3 `virtual CoinBigIndex ClpDummyMatrix::getNumElements () const [inline],[virtual]`

Number of entries in the packed matrix.

Implements [ClpMatrixBase](#).

Definition at line 32 of file ClpDummyMatrix.hpp.

4.39.3.4 `virtual int ClpDummyMatrix::getNumCols () const [inline],[virtual]`

Number of columns.

Implements [ClpMatrixBase](#).

Definition at line 36 of file ClpDummyMatrix.hpp.

4.39.3.5 `virtual int ClpDummyMatrix::getNumRows () const [inline],[virtual]`

Number of rows.

Implements [ClpMatrixBase](#).

Definition at line 40 of file ClpDummyMatrix.hpp.

4.39.3.6 `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](#).

4.39.3.7 `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](#).

4.39.3.8 `virtual const CoinBigIndex* ClpDummyMatrix::getVectorStarts () const [virtual]`

Implements [ClpMatrixBase](#).

4.39.3.9 `virtual const int* ClpDummyMatrix::getVectorLengths () const [virtual]`

The lengths of the major-dimension vectors.

Implements [ClpMatrixBase](#).

4.39.3.10 `virtual void ClpDummyMatrix::deleteCols (const int numDel, const int * indDel) [virtual]`

Delete the columns whose indices are listed in `indDel`.

Implements [ClpMatrixBase](#).

4.39.3.11 `virtual void ClpDummyMatrix::deleteRows (const int numDel, const int * indDel) [virtual]`

Delete the rows whose indices are listed in `indDel`.

Implements [ClpMatrixBase](#).

4.39.3.12 `virtual ClpMatrixBase* ClpDummyMatrix::reverseOrderedCopy () const [virtual]`

Returns a new matrix in reverse order without gaps.

Reimplemented from [ClpMatrixBase](#).

4.39.3.13 `virtual CoinBigIndex ClpDummyMatrix::countBasis (const int * whichColumn, int & numberColumnBasic) [virtual]`

Returns number of elements in column part of basis.

Implements [ClpMatrixBase](#).

4.39.3.14 `virtual void ClpDummyMatrix::fillBasis (ClpSimplex * model, const int * whichColumn, int & numberColumnBasic, int * row, int * start, int * rowCount, int * columnCount, CoinFactorizationDouble * element) [virtual]`

Fills in column part of basis.

Implements [ClpMatrixBase](#).

4.39.3.15 `virtual void ClpDummyMatrix::unpack (const ClpSimplex * model, CoinIndexedVector * rowArray, int column) const [virtual]`

Unpacks a column into an `CoinIndexedvector`.

Implements [ClpMatrixBase](#).

4.39.3.16 `virtual void ClpDummyMatrix::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](#).

4.39.3.17 `virtual void ClpDummyMatrix::add (const ClpSimplex * model, CoinIndexedVector * rowArray, int column, double multiplier) const [virtual]`

Adds multiple of a column into an `CoinIndexedvector`. You can use `quickAdd` to add to vector.

Implements [ClpMatrixBase](#).

4.39.3.18 `virtual void ClpDummyMatrix::add (const ClpSimplex * model, double * array, int column, double multiplier) const` [virtual]

Adds multiple of a column into an array.

Implements [ClpMatrixBase](#).

4.39.3.19 `virtual void ClpDummyMatrix::releasePackedMatrix () const` [inline],[virtual]

Allow any parts of a created CoinMatrix to be deleted Allow any parts of a created CoinPackedMatrix to be deleted.

Implements [ClpMatrixBase](#).

Definition at line 96 of file ClpDummyMatrix.hpp.

4.39.3.20 `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()`

Implements [ClpMatrixBase](#).

4.39.3.21 `virtual void ClpDummyMatrix::times (double scalar, const double * x, double * y, const double * rowScale, const double * columnScale) const` [virtual]

And for scaling.

Reimplemented from [ClpMatrixBase](#).

4.39.3.22 `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()`

Implements [ClpMatrixBase](#).

4.39.3.23 `virtual void ClpDummyMatrix::transposeTimes (double scalar, const double * x, double * y, const double * rowScale, const double * columnScale) const` [virtual]

And for scaling.

4.39.3.24 `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](#).

4.39.3.25 `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](#).

4.39.3.26 `ClpDummyMatrix& ClpDummyMatrix::operator= (const ClpDummyMatrix &)`

4.39.3.27 `virtual ClpMatrixBase* ClpDummyMatrix::clone () const` [virtual]

Clone.

Implements [ClpMatrixBase](#).

4.39.4 Member Data Documentation

4.39.4.1 `int ClpDummyMatrix::numberRows_` [protected]

Number of rows.

Definition at line 174 of file `ClpDummyMatrix.hpp`.

4.39.4.2 `int ClpDummyMatrix::numberColumns_` [protected]

Number of columns.

Definition at line 176 of file `ClpDummyMatrix.hpp`.

4.39.4.3 `int ClpDummyMatrix::numberElements_` [protected]

Number of elements.

Definition at line 178 of file `ClpDummyMatrix.hpp`.

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

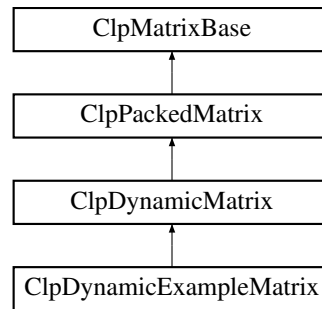
- [src/ClpDummyMatrix.hpp](#)

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`:



Public Member Functions

Main functions provided

- virtual void [partialPricing](#) ([ClpSimplex](#) *model, double start, double end, int &bestSequence, int &numberWanted)
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 meant 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 [ClpDynamicMatrix](#) 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 idea 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 virtual ClpDynamicExampleMatrix::~~ClpDynamicExampleMatrix () [virtual]

Destructor.

4.40.2.4 ClpDynamicExampleMatrix::ClpDynamicExampleMatrix (const ClpDynamicExampleMatrix &)

The copy constructor.

4.40.3 Member Function Documentation

```
4.40.3.1 virtual void ClpDynamicExampleMatrix::partialPricing ( ClpSimplex * model, double start, double end, int &
bestSequence, int & numberWanted ) [virtual]
```

Partial pricing.

Reimplemented from [ClpDynamicMatrix](#).

```
4.40.3.2 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](#).

```
4.40.3.3 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](#).

4.40.3.4 **ClpDynamicExampleMatrix& ClpDynamicExampleMatrix::operator= (const ClpDynamicExampleMatrix &)**

4.40.3.5 **virtual ClpMatrixBase* ClpDynamicExampleMatrix::clone () const** [virtual]

Clone.

Reimplemented from [ClpDynamicMatrix](#).

4.40.3.6 **CoinBigIndex* ClpDynamicExampleMatrix::startColumnGen () const** [inline]

Starts of each column.

Definition at line 101 of file ClpDynamicExampleMatrix.hpp.

4.40.3.7 **int* ClpDynamicExampleMatrix::rowGen () const** [inline]

rows

Definition at line 105 of file ClpDynamicExampleMatrix.hpp.

4.40.3.8 **double* ClpDynamicExampleMatrix::elementGen () const** [inline]

elements

Definition at line 109 of file ClpDynamicExampleMatrix.hpp.

4.40.3.9 **double* ClpDynamicExampleMatrix::costGen () const** [inline]

costs

Definition at line 113 of file ClpDynamicExampleMatrix.hpp.

4.40.3.10 **int* ClpDynamicExampleMatrix::fullStartGen () const** [inline]

full starts

Definition at line 117 of file ClpDynamicExampleMatrix.hpp.

4.40.3.11 **int* ClpDynamicExampleMatrix::idGen () const** [inline]

ids in next level matrix

Definition at line 121 of file ClpDynamicExampleMatrix.hpp.

4.40.3.12 **double* ClpDynamicExampleMatrix::columnLowerGen () const** [inline]

Optional lower bounds on columns.

Definition at line 125 of file ClpDynamicExampleMatrix.hpp.

4.40.3.13 **double* ClpDynamicExampleMatrix::columnUpperGen () const** [inline]

Optional upper bounds on columns.

Definition at line 129 of file ClpDynamicExampleMatrix.hpp.

4.40.3.14 **int ClpDynamicExampleMatrix::numberColumns () const** [inline]

size

Definition at line 133 of file ClpDynamicExampleMatrix.hpp.

4.40.3.15 void ClpDynamicExampleMatrix::setDynamicStatusGen (int *sequence*, DynamicStatus *status*) [inline]

Definition at line 136 of file ClpDynamicExampleMatrix.hpp.

4.40.3.16 DynamicStatus ClpDynamicExampleMatrix::getDynamicStatusGen (int *sequence*) const [inline]

Definition at line 141 of file ClpDynamicExampleMatrix.hpp.

4.40.3.17 bool ClpDynamicExampleMatrix::flaggedGen (int *i*) const [inline]

Whether flagged.

Definition at line 145 of file ClpDynamicExampleMatrix.hpp.

4.40.3.18 void ClpDynamicExampleMatrix::setFlaggedGen (int *i*) [inline]

Definition at line 148 of file ClpDynamicExampleMatrix.hpp.

4.40.3.19 void ClpDynamicExampleMatrix::unsetFlagged (int *i*) [inline]

Definition at line 151 of file ClpDynamicExampleMatrix.hpp.

4.40.4 Member Data Documentation

4.40.4.1 int ClpDynamicExampleMatrix::numberColumns_ [protected]

size

Definition at line 162 of file ClpDynamicExampleMatrix.hpp.

4.40.4.2 CoinBigIndex* ClpDynamicExampleMatrix::startColumnGen_ [protected]

Starts of each column.

Definition at line 164 of file ClpDynamicExampleMatrix.hpp.

4.40.4.3 int* ClpDynamicExampleMatrix::rowGen_ [protected]

rows

Definition at line 166 of file ClpDynamicExampleMatrix.hpp.

4.40.4.4 double* ClpDynamicExampleMatrix::elementGen_ [protected]

elements

Definition at line 168 of file ClpDynamicExampleMatrix.hpp.

4.40.4.5 double* ClpDynamicExampleMatrix::costGen_ [protected]

costs

Definition at line 170 of file ClpDynamicExampleMatrix.hpp.

4.40.4.6 int* ClpDynamicExampleMatrix::fullStartGen_ [protected]

start of each set

Definition at line 172 of file ClpDynamicExampleMatrix.hpp.

4.40.4.7 `unsigned char* ClpDynamicExampleMatrix::dynamicStatusGen_` [protected]

for status and which bound

Definition at line 174 of file `ClpDynamicExampleMatrix.hpp`.

4.40.4.8 `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`.

4.40.4.9 `double* ClpDynamicExampleMatrix::columnLowerGen_` [protected]

Optional lower bounds on columns.

Definition at line 180 of file `ClpDynamicExampleMatrix.hpp`.

4.40.4.10 `double* ClpDynamicExampleMatrix::columnUpperGen_` [protected]

Optional upper bounds on columns.

Definition at line 182 of file `ClpDynamicExampleMatrix.hpp`.

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

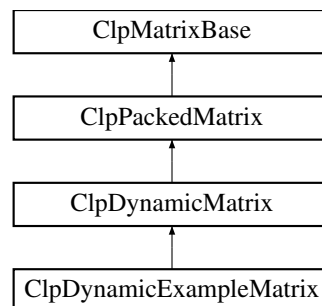
- [src/ClpDynamicExampleMatrix.hpp](#)

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`:



Public Types

- enum `DynamicStatus` { `soloKey` = 0x00, `inSmall` = 0x01, `atUpperBound` = 0x02, `atLowerBound` = 0x03 }
- enums for status of various sorts*

Public Member Functions

Main functions provided

- virtual void **partialPricing** (ClpSimplex *model, double start, double end, int &bestSequence, int &numberWanted)
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 primal.
- 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 Member Enumeration Documentation

4.41.2.1 enum `ClpDynamicMatrix::DynamicStatus`

enums for status of various sorts

Enumerator

soloKey

inSmall

atUpperBound

atLowerBound

Definition at line 24 of file `ClpDynamicMatrix.hpp`.

4.41.3 Constructor & Destructor Documentation

4.41.3.1 `ClpDynamicMatrix::ClpDynamicMatrix ()`

Default constructor.

4.41.3.2 `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)

4.41.3.3 `virtual ClpDynamicMatrix::~ClpDynamicMatrix () [virtual]`

Destructor.

4.41.3.4 `ClpDynamicMatrix::ClpDynamicMatrix (const ClpDynamicMatrix &)`

The copy constructor.

4.41.3.5 `ClpDynamicMatrix::ClpDynamicMatrix (const CoinPackedMatrix &)`

The copy constructor from an `CoinPackedMatrix`.

4.41.4 Member Function Documentation

4.41.4.1 `virtual void ClpDynamicMatrix::partialPricing (ClpSimplex * model, double start, double end, int & bestSequence, int & numberWanted) [virtual]`

Partial pricing.

Reimplemented from [ClpPackedMatrix](#).

Reimplemented in [ClpDynamicExampleMatrix](#).

4.41.4.2 `virtual int ClpDynamicMatrix::updatePivot (ClpSimplex * model, double oldInValue, double oldOutValue) [virtual]`

update information for a pivot (and effective rhs)

Reimplemented from [ClpMatrixBase](#).

4.41.4.3 `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](#).

4.41.4.4 `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](#).

4.41.4.5 `void ClpDynamicMatrix::modifyOffset (int sequence, double amount)`

Modifies rhs offset.

4.41.4.6 `double ClpDynamicMatrix::keyValue (int iSet) const`

Gets key value when none in small.

4.41.4.7 `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](#).

4.41.4.8 `virtual int ClpDynamicMatrix::generalExpanded (ClpSimplex * model, int mode, int & number) [virtual]`

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

Reimplemented from [ClpMatrixBase](#).

4.41.4.9 `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](#).

4.41.4.10 `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](#).

4.41.4.11 `virtual double ClpDynamicMatrix::reducedCost (ClpSimplex * model, int sequence) const [virtual]`

Returns reduced cost of a variable.

4.41.4.12 `void ClpDynamicMatrix::gubCrash ()`

Does gub crash.

4.41.4.13 `void ClpDynamicMatrix::writeMps (const char * name)`

Writes out model (without names)

4.41.4.14 `void ClpDynamicMatrix::initialProblem ()`

Populates initial matrix from dynamic status.

4.41.4.15 `int ClpDynamicMatrix::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.

4.41.4.16 `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](#).

4.41.4.17 `double ClpDynamicMatrix::columnLower (int sequence) const` `[inline]`

Gets lower bound (to simplify coding)

Definition at line 111 of file `ClpDynamicMatrix.hpp`.

4.41.4.18 `double ClpDynamicMatrix::columnUpper (int sequence) const` `[inline]`

Gets upper bound (to simplify coding)

Definition at line 116 of file `ClpDynamicMatrix.hpp`.

4.41.4.19 `ClpDynamicMatrix& ClpDynamicMatrix::operator= (const ClpDynamicMatrix &)`

4.41.4.20 `virtual ClpMatrixBase* ClpDynamicMatrix::clone () const` `[virtual]`

Clone.

Reimplemented from [ClpPackedMatrix](#).

Reimplemented in [ClpDynamicExampleMatrix](#).

4.41.4.21 `ClpSimplex::Status ClpDynamicMatrix::getStatus (int sequence) const` `[inline]`

Status of row slacks.

Definition at line 162 of file `ClpDynamicMatrix.hpp`.

4.41.4.22 `void ClpDynamicMatrix::setStatus (int sequence, ClpSimplex::Status status)` `[inline]`

Definition at line 165 of file `ClpDynamicMatrix.hpp`.

4.41.4.23 `bool ClpDynamicMatrix::flaggedSlack (int i) const` `[inline]`

Whether flagged slack.

Definition at line 171 of file `ClpDynamicMatrix.hpp`.

4.41.4.24 `void ClpDynamicMatrix::setFlaggedSlack (int i)` `[inline]`

Definition at line 174 of file `ClpDynamicMatrix.hpp`.

4.41.4.25 `void ClpDynamicMatrix::unsetFlaggedSlack (int i)` `[inline]`

Definition at line 177 of file `ClpDynamicMatrix.hpp`.

4.41.4.26 `int ClpDynamicMatrix::numberSets () const` `[inline]`

Number of sets (dynamic rows)

Definition at line 181 of file `ClpDynamicMatrix.hpp`.

4.41.4.27 `int ClpDynamicMatrix::numberGubEntries () const` `[inline]`

Number of possible gub variables.

Definition at line 185 of file `ClpDynamicMatrix.hpp`.

4.41.4.28 `int* ClpDynamicMatrix::startSets () const` `[inline]`

Sets.

Definition at line 188 of file ClpDynamicMatrix.hpp.

4.41.4.29 `bool ClpDynamicMatrix::flagged (int i) const` `[inline]`

Whether flagged.

Definition at line 191 of file ClpDynamicMatrix.hpp.

4.41.4.30 `void ClpDynamicMatrix::setFlagged (int i)` `[inline]`

Definition at line 194 of file ClpDynamicMatrix.hpp.

4.41.4.31 `void ClpDynamicMatrix::unsetFlagged (int i)` `[inline]`

Definition at line 197 of file ClpDynamicMatrix.hpp.

4.41.4.32 `void ClpDynamicMatrix::setDynamicStatus (int sequence, DynamicStatus status)` `[inline]`

Definition at line 200 of file ClpDynamicMatrix.hpp.

4.41.4.33 `DynamicStatus ClpDynamicMatrix::getDynamicStatus (int sequence) const` `[inline]`

Definition at line 205 of file ClpDynamicMatrix.hpp.

4.41.4.34 `double ClpDynamicMatrix::objectiveOffset () const` `[inline]`

Saved value of objective offset.

Definition at line 209 of file ClpDynamicMatrix.hpp.

4.41.4.35 `CoinBigIndex* ClpDynamicMatrix::startColumn () const` `[inline]`

Starts of each column.

Definition at line 213 of file ClpDynamicMatrix.hpp.

4.41.4.36 `int* ClpDynamicMatrix::row () const` `[inline]`

rows

Definition at line 217 of file ClpDynamicMatrix.hpp.

4.41.4.37 `double* ClpDynamicMatrix::element () const` `[inline]`

elements

Definition at line 221 of file ClpDynamicMatrix.hpp.

4.41.4.38 `double* ClpDynamicMatrix::cost () const` `[inline]`

costs

Definition at line 225 of file ClpDynamicMatrix.hpp.

4.41.4.39 `int* ClpDynamicMatrix::id () const` `[inline]`

ids of active columns (just index here)

Definition at line 229 of file ClpDynamicMatrix.hpp.

4.41.4.40 `double* ClpDynamicMatrix::columnLower () const` `[inline]`

Optional lower bounds on columns.

Definition at line 233 of file ClpDynamicMatrix.hpp.

4.41.4.41 `double* ClpDynamicMatrix::columnUpper () const` `[inline]`

Optional upper bounds on columns.

Definition at line 237 of file ClpDynamicMatrix.hpp.

4.41.4.42 `double* ClpDynamicMatrix::lowerSet () const` `[inline]`

Lower bounds on sets.

Definition at line 241 of file ClpDynamicMatrix.hpp.

4.41.4.43 `double* ClpDynamicMatrix::upperSet () const` `[inline]`

Upper bounds on sets.

Definition at line 245 of file ClpDynamicMatrix.hpp.

4.41.4.44 `int ClpDynamicMatrix::numberGubColumns () const` `[inline]`

size

Definition at line 249 of file ClpDynamicMatrix.hpp.

4.41.4.45 `int ClpDynamicMatrix::firstAvailable () const` `[inline]`

first free

Definition at line 253 of file ClpDynamicMatrix.hpp.

4.41.4.46 `int ClpDynamicMatrix::firstDynamic () const` `[inline]`

first dynamic

Definition at line 257 of file ClpDynamicMatrix.hpp.

4.41.4.47 `int ClpDynamicMatrix::lastDynamic () const` `[inline]`

number of columns in dynamic model

Definition at line 261 of file ClpDynamicMatrix.hpp.

4.41.4.48 `int ClpDynamicMatrix::numberStaticRows () const` `[inline]`

number of rows in original model

Definition at line 265 of file ClpDynamicMatrix.hpp.

4.41.4.49 `int ClpDynamicMatrix::numberElements () const` `[inline]`

size of working matrix (max)

Definition at line 269 of file ClpDynamicMatrix.hpp.

4.41.4.50 `int* ClpDynamicMatrix::keyVariable () const [inline]`

Definition at line 272 of file ClpDynamicMatrix.hpp.

4.41.4.51 `void ClpDynamicMatrix::switchOffCheck ()`

Switches off dj checking each factorization (for BIG models)

4.41.4.52 `unsigned char* ClpDynamicMatrix::gubRowStatus () const [inline]`

Status region for gub slacks.

Definition at line 278 of file ClpDynamicMatrix.hpp.

4.41.4.53 `unsigned char* ClpDynamicMatrix::dynamicStatus () const [inline]`

Status region for gub variables.

Definition at line 282 of file ClpDynamicMatrix.hpp.

4.41.4.54 `int ClpDynamicMatrix::whichSet (int sequence) const`

Returns which set a variable is in.

4.41.5 Member Data Documentation

4.41.5.1 `double ClpDynamicMatrix::sumDualInfeasibilities_ [protected]`

Sum of dual infeasibilities.

Definition at line 295 of file ClpDynamicMatrix.hpp.

4.41.5.2 `double ClpDynamicMatrix::sumPrimalInfeasibilities_ [protected]`

Sum of primal infeasibilities.

Definition at line 297 of file ClpDynamicMatrix.hpp.

4.41.5.3 `double ClpDynamicMatrix::sumOfRelaxedDualInfeasibilities_ [protected]`

Sum of Dual infeasibilities using tolerance based on error in duals.

Definition at line 299 of file ClpDynamicMatrix.hpp.

4.41.5.4 `double ClpDynamicMatrix::sumOfRelaxedPrimalInfeasibilities_ [protected]`

Sum of Primal infeasibilities using tolerance based on error in primals.

Definition at line 301 of file ClpDynamicMatrix.hpp.

4.41.5.5 `double ClpDynamicMatrix::savedBestGubDual_ [protected]`

Saved best dual on gub row in pricing.

Definition at line 303 of file ClpDynamicMatrix.hpp.

4.41.5.6 `int ClpDynamicMatrix::savedBestSet_ [protected]`

Saved best set in pricing.

Definition at line 305 of file ClpDynamicMatrix.hpp.

4.41.5.7 `int* ClpDynamicMatrix::backToPivotRow_` `[protected]`

Backward pointer to pivot row !!!

Definition at line 307 of file ClpDynamicMatrix.hpp.

4.41.5.8 `int* ClpDynamicMatrix::keyVariable_` `[mutable],[protected]`

Key variable of set (only accurate if none in small problem)

Definition at line 309 of file ClpDynamicMatrix.hpp.

4.41.5.9 `int* ClpDynamicMatrix::toIndex_` `[protected]`

Backward pointer to extra row.

Definition at line 311 of file ClpDynamicMatrix.hpp.

4.41.5.10 `int* ClpDynamicMatrix::fromIndex_` `[protected]`

Definition at line 313 of file ClpDynamicMatrix.hpp.

4.41.5.11 `int ClpDynamicMatrix::numberSets_` `[protected]`

Number of sets (dynamic rows)

Definition at line 315 of file ClpDynamicMatrix.hpp.

4.41.5.12 `int ClpDynamicMatrix::numberActiveSets_` `[protected]`

Number of active sets.

Definition at line 317 of file ClpDynamicMatrix.hpp.

4.41.5.13 `double ClpDynamicMatrix::objectiveOffset_` `[protected]`

Saved value of objective offset.

Definition at line 319 of file ClpDynamicMatrix.hpp.

4.41.5.14 `double* ClpDynamicMatrix::lowerSet_` `[protected]`

Lower bounds on sets.

Definition at line 321 of file ClpDynamicMatrix.hpp.

4.41.5.15 `double* ClpDynamicMatrix::upperSet_` `[protected]`

Upper bounds on sets.

Definition at line 323 of file ClpDynamicMatrix.hpp.

4.41.5.16 `unsigned char* ClpDynamicMatrix::status_` `[protected]`

Status of slack on set.

Definition at line 325 of file ClpDynamicMatrix.hpp.

4.41.5.17 ClpSimplex* ClpDynamicMatrix::model_ [protected]

Pointer back to model.

Definition at line 327 of file ClpDynamicMatrix.hpp.

4.41.5.18 int ClpDynamicMatrix::firstAvailable_ [protected]

first free

Definition at line 329 of file ClpDynamicMatrix.hpp.

4.41.5.19 int ClpDynamicMatrix::firstAvailableBefore_ [protected]

first free when iteration started

Definition at line 331 of file ClpDynamicMatrix.hpp.

4.41.5.20 int ClpDynamicMatrix::firstDynamic_ [protected]

first dynamic

Definition at line 333 of file ClpDynamicMatrix.hpp.

4.41.5.21 int ClpDynamicMatrix::lastDynamic_ [protected]

number of columns in dynamic model

Definition at line 335 of file ClpDynamicMatrix.hpp.

4.41.5.22 int ClpDynamicMatrix::numberStaticRows_ [protected]

number of rows in original model

Definition at line 337 of file ClpDynamicMatrix.hpp.

4.41.5.23 int ClpDynamicMatrix::numberElements_ [protected]

size of working matrix (max)

Definition at line 339 of file ClpDynamicMatrix.hpp.

4.41.5.24 int ClpDynamicMatrix::numberDualInfeasibilities_ [protected]

Number of dual infeasibilities.

Definition at line 341 of file ClpDynamicMatrix.hpp.

4.41.5.25 int ClpDynamicMatrix::numberPrimalInfeasibilities_ [protected]

Number of primal infeasibilities.

Definition at line 343 of file ClpDynamicMatrix.hpp.

4.41.5.26 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.

4.41.5.27 double ClpDynamicMatrix::infeasibilityWeight_ [protected]

Infeasibility weight when last full pass done.

Definition at line 351 of file ClpDynamicMatrix.hpp.

4.41.5.28 int ClpDynamicMatrix::numberGubColumns_ [protected]

size

Definition at line 353 of file ClpDynamicMatrix.hpp.

4.41.5.29 int ClpDynamicMatrix::maximumGubColumns_ [protected]

current maximum number of columns (then compress)

Definition at line 355 of file ClpDynamicMatrix.hpp.

4.41.5.30 int ClpDynamicMatrix::maximumElements_ [protected]

current maximum number of elements (then compress)

Definition at line 357 of file ClpDynamicMatrix.hpp.

4.41.5.31 int* ClpDynamicMatrix::startSet_ [protected]

Start of each set.

Definition at line 359 of file ClpDynamicMatrix.hpp.

4.41.5.32 int* ClpDynamicMatrix::next_ [protected]

next in chain

Definition at line 361 of file ClpDynamicMatrix.hpp.

4.41.5.33 CoinBigIndex* ClpDynamicMatrix::startColumn_ [protected]

Starts of each column.

Definition at line 363 of file ClpDynamicMatrix.hpp.

4.41.5.34 int* ClpDynamicMatrix::row_ [protected]

rows

Definition at line 365 of file ClpDynamicMatrix.hpp.

4.41.5.35 double* ClpDynamicMatrix::element_ [protected]

elements

Definition at line 367 of file ClpDynamicMatrix.hpp.

4.41.5.36 double* ClpDynamicMatrix::cost_ [protected]

costs

Definition at line 369 of file ClpDynamicMatrix.hpp.

4.41.5.37 `int* ClpDynamicMatrix::id_` [protected]

ids of active columns (just index here)

Definition at line 371 of file `ClpDynamicMatrix.hpp`.

4.41.5.38 `unsigned char* ClpDynamicMatrix::dynamicStatus_` [protected]

for status and which bound

Definition at line 373 of file `ClpDynamicMatrix.hpp`.

4.41.5.39 `double* ClpDynamicMatrix::columnLower_` [protected]

Optional lower bounds on columns.

Definition at line 375 of file `ClpDynamicMatrix.hpp`.

4.41.5.40 `double* ClpDynamicMatrix::columnUpper_` [protected]

Optional upper bounds on columns.

Definition at line 377 of file `ClpDynamicMatrix.hpp`.

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

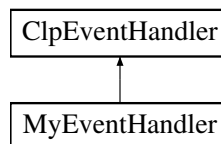
- [src/ClpDynamicMatrix.hpp](#)

4.42 ClpEventHandler Class Reference

Base class for Clp event handling.

```
#include <ClpEventHandler.hpp>
```

Inheritance diagram for ClpEventHandler:



Public Types

- enum `Event` {
`endOfIteration` = 100, `endOfFactorization`, `endOfValuesPass`, `node`,
`treeStatus`, `solution`, `theta`, `pivotRow`,
`presolveStart`, `presolveSize`, `presolveInfeasible`, `presolveBeforeSolve`,
`presolveAfterFirstSolve`, `presolveAfterSolve`, `presolveEnd`, `goodFactorization`,
`complicatedPivotIn`, `noCandidateInPrimal`, `looksEndInPrimal`, `endInPrimal`,
`beforeStatusOfProblemInPrimal`, `startOfStatusOfProblemInPrimal`, `complicatedPivotOut`, `noCandidateInDual`,
`looksEndInDual`, `endInDual`, `beforeStatusOfProblemInDual`, `startOfStatusOfProblemInDual`,
`startOfIterationInDual`, `updateDualsInDual`, `endOfCreateRim`, `slightlyInfeasible`,
`modifyMatrixInMiniPresolve`, `moreMiniPresolve`, `modifyMatrixInMiniPostsolve`, `noTheta` }

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

Enumerator

endOfIteration
endOfFactorization
endOfValuesPass
node
treeStatus
solution
theta
pivotRow
presolveStart
presolveSize
presolveInfeasible
presolveBeforeSolve
presolveAfterFirstSolve
presolveAfterSolve
presolveEnd
goodFactorization
complicatedPivotIn
noCandidateInPrimal
looksEndInPrimal
endInPrimal
beforeStatusOfProblemInPrimal
startOfStatusOfProblemInPrimal
complicatedPivotOut
noCandidateInDual
looksEndInDual
endInDual
beforeStatusOfProblemInDual
startOfStatusOfProblemInDual
startOfIterationInDual
updateDualsInDual
endOfCreateRim
slightlyInfeasible
modifyMatrixInMiniPresolve
moreMiniPresolve
modifyMatrixInMiniPostsolve
noTheta

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.

4.42.3.2 virtual ClpEventHandler::~~ClpEventHandler () [virtual]

Destructor.

4.42.3.3 ClpEventHandler::ClpEventHandler (const ClpEventHandler &)

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 ClpEventHandler& ClpEventHandler::operator= (const ClpEventHandler &)

4.42.4.4 virtual ClpEventHandler* ClpEventHandler::clone () const [virtual]

Clone.

Reimplemented in [MyEventHandler](#).

4.42.4.5 void ClpEventHandler::setSimplex (ClpSimplex * *model*)

set model.

4.42.4.6 ClpSimplex* ClpEventHandler::simplex () const [inline]

Get model.

Definition at line 112 of file [ClpEventHandler.hpp](#).

4.42.5 Member Data Documentation

4.42.5.1 ClpSimplex* ClpEventHandler::model_ [protected]

Pointer to simplex.

Definition at line 123 of file [ClpEventHandler.hpp](#).

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

- [src/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 denselfSmaller=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 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.

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 noPermute=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 32 of file ClpFactorization.hpp.

4.43.2 Constructor & Destructor Documentation

4.43.2.1 `ClpFactorization::ClpFactorization ()`

Default constructor.

4.43.2.2 `ClpFactorization::~~ClpFactorization ()`

Destructor.

4.43.2.3 `ClpFactorization::ClpFactorization (const CoinFactorization &)`

The copy constructor from an CoinFactorization.

4.43.2.4 `ClpFactorization::ClpFactorization (const ClpFactorization & , int denselfSmaller = 0)`

The copy constructor.

4.43.2.5 `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 `ClpFactorization& ClpFactorization::operator= (const ClpFactorization &)`

4.43.3.3 `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

4.43.3.4 `int ClpFactorization::updateColumnFT (CoinIndexedVector * regionSparse, CoinIndexedVector * regionSparse2)`

Updates one column (FTAN) from region2 Tries to do FT update number returned is negative if no room region1 starts as zero and is zero at end.

4.43.3.5 `int ClpFactorization::updateColumn (CoinIndexedVector * regionSparse, CoinIndexedVector * regionSparse2, bool noPermute = false) const`

Updates one column (FTRAN) from region2 region1 starts as zero and is zero at end.

4.43.3.6 `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

4.43.3.7 `int ClpFactorization::updateColumnForDebug (CoinIndexedVector * regionSparse, CoinIndexedVector * regionSparse2, bool noPermute = false) const`

For debug (no statistics update)

4.43.3.8 `int ClpFactorization::updateColumnTranspose (CoinIndexedVector * regionSparse, CoinIndexedVector * regionSparse2) const`

Updates one column (BTRAN) from region2 region1 starts as zero and is zero at end.

4.43.3.9 `int ClpFactorization::numberElements () const [inline]`

Total number of elements in factorization.

Definition at line 133 of file ClpFactorization.hpp.

4.43.3.10 `int* ClpFactorization::permute () const [inline]`

Returns address of permute region.

Definition at line 138 of file ClpFactorization.hpp.

4.43.3.11 `int* ClpFactorization::pivotColumn () const [inline]`

Returns address of pivotColumn region (also used for permuting)

Definition at line 143 of file ClpFactorization.hpp.

4.43.3.12 `int ClpFactorization::maximumPivots () const [inline]`

Maximum number of pivots between factorizations.

Definition at line 148 of file ClpFactorization.hpp.

4.43.3.13 `void ClpFactorization::maximumPivots (int value) [inline]`

Set maximum number of pivots between factorizations.

Definition at line 153 of file ClpFactorization.hpp.

4.43.3.14 `int ClpFactorization::pivots () const [inline]`

Returns number of pivots since factorization.

Definition at line 158 of file ClpFactorization.hpp.

4.43.3.15 `double ClpFactorization::areaFactor () const [inline]`

Whether larger areas needed.

Definition at line 163 of file ClpFactorization.hpp.

4.43.3.16 `void ClpFactorization::areaFactor (double value) [inline]`

Set whether larger areas needed.

Definition at line 168 of file ClpFactorization.hpp.

4.43.3.17 `double ClpFactorization::zeroTolerance () const [inline]`

Zero tolerance.

Definition at line 172 of file ClpFactorization.hpp.

4.43.3.18 `void ClpFactorization::zeroTolerance (double value) [inline]`

Set zero tolerance.

Definition at line 177 of file ClpFactorization.hpp.

4.43.3.19 `void ClpFactorization::saferTolerances (double zeroTolerance, double pivotTolerance)`

Set tolerances to safer of existing and given.

4.43.3.20 `int ClpFactorization::sparseThreshold () const [inline]`

get sparse threshold

Definition at line 184 of file ClpFactorization.hpp.

4.43.3.21 `void ClpFactorization::sparseThreshold (int value) [inline]`

Set sparse threshold.

Definition at line 189 of file ClpFactorization.hpp.

4.43.3.22 `int ClpFactorization::status () const [inline]`

Returns status.

Definition at line 193 of file ClpFactorization.hpp.

4.43.3.23 `void ClpFactorization::setStatus (int value) [inline]`

Sets status.

Definition at line 198 of file ClpFactorization.hpp.

4.43.3.24 `int ClpFactorization::numberDense () const [inline]`

Returns number of dense rows.

Definition at line 203 of file ClpFactorization.hpp.

4.43.3.25 `CoinBigIndex ClpFactorization::numberElementsU () const [inline]`

Returns number in U area.

Definition at line 209 of file ClpFactorization.hpp.

4.43.3.26 `CoinBigIndex ClpFactorization::numberElementsL () const [inline]`

Returns number in L area.

Definition at line 214 of file ClpFactorization.hpp.

4.43.3.27 `CoinBigIndex ClpFactorization::numberElementsR () const [inline]`

Returns number in R area.

Definition at line 219 of file ClpFactorization.hpp.

4.43.3.28 `bool ClpFactorization::timeToRefactorize () const [inline]`

Definition at line 224 of file ClpFactorization.hpp.

4.43.3.29 `int ClpFactorization::messageLevel () const [inline]`

Level of detail of messages.

Definition at line 235 of file ClpFactorization.hpp.

4.43.3.30 `void ClpFactorization::messageLevel (int value) [inline]`

Set level of detail of messages.

Definition at line 240 of file ClpFactorization.hpp.

4.43.3.31 `void ClpFactorization::clearArrays () [inline]`

Get rid of all memory.

Definition at line 244 of file ClpFactorization.hpp.

4.43.3.32 `int ClpFactorization::numberRows () const [inline]`

Number of Rows after factorization.

Definition at line 251 of file ClpFactorization.hpp.

4.43.3.33 `int ClpFactorization::denseThreshold () const [inline]`

Gets dense threshold.

Definition at line 256 of file ClpFactorization.hpp.

4.43.3.34 `void ClpFactorization::setDenseThreshold (int value) [inline]`

Sets dense threshold.

Definition at line 261 of file ClpFactorization.hpp.

4.43.3.35 `double ClpFactorization::pivotTolerance () const [inline]`

Pivot tolerance.

Definition at line 265 of file ClpFactorization.hpp.

4.43.3.36 `void ClpFactorization::pivotTolerance (double value) [inline]`

Set pivot tolerance.

Definition at line 271 of file ClpFactorization.hpp.

4.43.3.37 `void ClpFactorization::relaxAccuracyCheck (double value) [inline]`

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

Definition at line 276 of file ClpFactorization.hpp.

4.43.3.38 `int ClpFactorization::persistenceFlag () const [inline]`

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.

Definition at line 284 of file ClpFactorization.hpp.

4.43.3.39 `void ClpFactorization::setPersistenceFlag (int value) [inline]`

Definition at line 288 of file ClpFactorization.hpp.

4.43.3.40 `void ClpFactorization::almostDestructor () [inline]`

Delete all stuff (leaves as after CoinFactorization())

Definition at line 292 of file ClpFactorization.hpp.

4.43.3.41 `double ClpFactorization::adjustedAreaFactor () const [inline]`

Returns areaFactor but adjusted for dense.

Definition at line 299 of file ClpFactorization.hpp.

4.43.3.42 `void ClpFactorization::setBiasLU (int value) [inline]`

Definition at line 303 of file ClpFactorization.hpp.

4.43.3.43 `void ClpFactorization::setForrestTomlin (bool value) [inline]`

true if Forrest Tomlin update, false if PFI

Definition at line 307 of file ClpFactorization.hpp.

4.43.3.44 `void ClpFactorization::setDefaultValues () [inline]`

Sets default values.

Definition at line 311 of file ClpFactorization.hpp.

4.43.3.45 `void ClpFactorization::forceOtherFactorization (int which)`

If nonzero force use of 1,dense 2,small 3,osl.

4.43.3.46 `int ClpFactorization::goOslThreshold () const [inline]`

Get switch to osl if number rows <= this.

Definition at line 323 of file ClpFactorization.hpp.

4.43.3.47 `void ClpFactorization::setGoOslThreshold (int value) [inline]`

Set switch to osl if number rows <= this.

Definition at line 327 of file ClpFactorization.hpp.

4.43.3.48 `int ClpFactorization::goDenseThreshold () const [inline]`

Get switch to dense if number rows \leq this.

Definition at line 331 of file ClpFactorization.hpp.

4.43.3.49 `void ClpFactorization::setGoDenseThreshold (int value) [inline]`

Set switch to dense if number rows \leq this.

Definition at line 335 of file ClpFactorization.hpp.

4.43.3.50 `int ClpFactorization::goSmallThreshold () const [inline]`

Get switch to small if number rows \leq this.

Definition at line 339 of file ClpFactorization.hpp.

4.43.3.51 `void ClpFactorization::setGoSmallThreshold (int value) [inline]`

Set switch to small if number rows \leq this.

Definition at line 343 of file ClpFactorization.hpp.

4.43.3.52 `void ClpFactorization::goDenseOrSmall (int numberOfRows)`

Go over to dense or small code if small enough.

4.43.3.53 `void ClpFactorization::setFactorization (ClpFactorization & factorization)`

Sets factorization.

4.43.3.54 `int ClpFactorization::isDenseOrSmall () const [inline]`

Return 1 if dense code.

Definition at line 351 of file ClpFactorization.hpp.

4.43.3.55 `void ClpFactorization::goSparse ()`

makes a row copy of L for speed and to allow very sparse problems

4.43.3.56 `void ClpFactorization::cleanUp ()`

Cleans up i.e. gets rid of network basis.

4.43.3.57 `bool ClpFactorization::needToReorder () const`

Says whether to redo pivot order.

4.43.3.58 `bool ClpFactorization::networkBasis () const [inline]`

Says if a network basis.

Definition at line 383 of file ClpFactorization.hpp.

4.43.3.59 void ClpFactorization::getWeights (int * *weights*) const

Fills weighted row list.

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

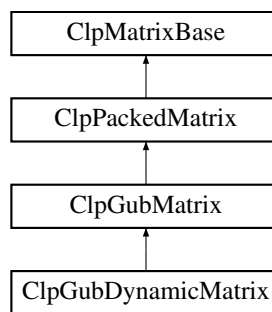
- [src/ClpFactorization.hpp](#)

4.44 ClpGubDynamicMatrix Class Reference

This implements Gub rows plus a [ClpPackedMatrix](#).

```
#include <ClpGubDynamicMatrix.hpp>
```

Inheritance diagram for ClpGubDynamicMatrix:



Public Member Functions

Main functions provided

- virtual void [partialPricing](#) ([ClpSimplex](#) *model, double [start](#), double [end](#), int &bestSequence, int &numberWanted)
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` { `inSmall` = 0x01, `atUpperBound` = 0x02, `atLowerBound` = 0x03 }
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 Member Enumeration Documentation

4.44.2.1 enum ClpGubDynamicMatrix::DynamicStatus

enums for status of various sorts

Enumerator

inSmall
atUpperBound
atLowerBound

Definition at line 100 of file ClpGubDynamicMatrix.hpp.

4.44.3 Constructor & Destructor Documentation

4.44.3.1 ClpGubDynamicMatrix::ClpGubDynamicMatrix ()

Default constructor.

4.44.3.2 virtual ClpGubDynamicMatrix::~~ClpGubDynamicMatrix () [virtual]

Destructor.

4.44.3.3 ClpGubDynamicMatrix::ClpGubDynamicMatrix (const ClpGubDynamicMatrix &)

The copy constructor.

4.44.3.4 `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 !!!!

4.44.4 Member Function Documentation

4.44.4.1 `virtual void ClpGubDynamicMatrix::partialPricing (ClpSimplex * model, double start, double end, int & bestSequence, int & numberWanted) [virtual]`

Partial pricing.

Reimplemented from [ClpGubMatrix](#).

4.44.4.2 `virtual int ClpGubDynamicMatrix::synchronize (ClpSimplex * model, int mode) [virtual]`

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.

Reimplemented from [ClpGubMatrix](#).

4.44.4.3 `virtual void ClpGubDynamicMatrix::useEffectiveRhs (ClpSimplex * model, bool cheapest = true) [virtual]`

Sets up an effective RHS and does gub crash if needed.

Reimplemented from [ClpGubMatrix](#).

4.44.4.4 `virtual int ClpGubDynamicMatrix::updatePivot (ClpSimplex * model, double oldInValue, double oldOutValue) [virtual]`

update information for a pivot (and effective rhs)

Reimplemented from [ClpGubMatrix](#).

4.44.4.5 `void ClpGubDynamicMatrix::insertNonBasic (int sequence, int iSet)`

Add a new variable to a set.

4.44.4.6 `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](#).

4.44.4.7 `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](#).

4.44.4.8 `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](#).

4.44.4.9 `void ClpGubDynamicMatrix::cleanData (ClpSimplex * model)`

Cleans data after setWarmStart.

4.44.4.10 `ClpGubDynamicMatrix& ClpGubDynamicMatrix::operator= (const ClpGubDynamicMatrix &)`

4.44.4.11 `virtual ClpMatrixBase* ClpGubDynamicMatrix::clone () const` [virtual]

Clone.

Reimplemented from [ClpGubMatrix](#).

4.44.4.12 `bool ClpGubDynamicMatrix::flagged (int i) const` [inline]

Whether flagged.

Definition at line 106 of file `ClpGubDynamicMatrix.hpp`.

4.44.4.13 `void ClpGubDynamicMatrix::setFlagged (int i)` [inline]

Definition at line 109 of file `ClpGubDynamicMatrix.hpp`.

4.44.4.14 `void ClpGubDynamicMatrix::unsetFlagged (int i)` [inline]

Definition at line 112 of file `ClpGubDynamicMatrix.hpp`.

4.44.4.15 `void ClpGubDynamicMatrix::setDynamicStatus (int sequence, DynamicStatus status)` [inline]

Definition at line 115 of file `ClpGubDynamicMatrix.hpp`.

4.44.4.16 `DynamicStatus ClpGubDynamicMatrix::getDynamicStatus (int sequence) const` [inline]

Definition at line 120 of file `ClpGubDynamicMatrix.hpp`.

4.44.4.17 `double ClpGubDynamicMatrix::objectiveOffset () const` [inline]

Saved value of objective offset.

Definition at line 124 of file `ClpGubDynamicMatrix.hpp`.

4.44.4.18 `CoinBigIndex* ClpGubDynamicMatrix::startColumn () const` [inline]

Starts of each column.

Definition at line 128 of file `ClpGubDynamicMatrix.hpp`.

4.44.4.19 `int* ClpGubDynamicMatrix::row () const [inline]`

rows

Definition at line 132 of file ClpGubDynamicMatrix.hpp.

4.44.4.20 `double* ClpGubDynamicMatrix::element () const [inline]`

elements

Definition at line 136 of file ClpGubDynamicMatrix.hpp.

4.44.4.21 `double* ClpGubDynamicMatrix::cost () const [inline]`

costs

Definition at line 140 of file ClpGubDynamicMatrix.hpp.

4.44.4.22 `int* ClpGubDynamicMatrix::fullStart () const [inline]`

full starts

Definition at line 144 of file ClpGubDynamicMatrix.hpp.

4.44.4.23 `int* ClpGubDynamicMatrix::id () const [inline]`

ids of active columns (just index here)

Definition at line 148 of file ClpGubDynamicMatrix.hpp.

4.44.4.24 `double* ClpGubDynamicMatrix::lowerColumn () const [inline]`

Optional lower bounds on columns.

Definition at line 152 of file ClpGubDynamicMatrix.hpp.

4.44.4.25 `double* ClpGubDynamicMatrix::upperColumn () const [inline]`

Optional upper bounds on columns.

Definition at line 156 of file ClpGubDynamicMatrix.hpp.

4.44.4.26 `double* ClpGubDynamicMatrix::lowerSet () const [inline]`

Optional true lower bounds on sets.

Definition at line 160 of file ClpGubDynamicMatrix.hpp.

4.44.4.27 `double* ClpGubDynamicMatrix::upperSet () const [inline]`

Optional true upper bounds on sets.

Definition at line 164 of file ClpGubDynamicMatrix.hpp.

4.44.4.28 `int ClpGubDynamicMatrix::numberGubColumns () const [inline]`

size

Definition at line 168 of file ClpGubDynamicMatrix.hpp.

4.44.4.29 `int ClpGubDynamicMatrix::firstAvailable () const` `[inline]`

first free

Definition at line 172 of file ClpGubDynamicMatrix.hpp.

4.44.4.30 `void ClpGubDynamicMatrix::setFirstAvailable (int value)` `[inline]`

set first free

Definition at line 176 of file ClpGubDynamicMatrix.hpp.

4.44.4.31 `int ClpGubDynamicMatrix::firstDynamic () const` `[inline]`

first dynamic

Definition at line 180 of file ClpGubDynamicMatrix.hpp.

4.44.4.32 `int ClpGubDynamicMatrix::lastDynamic () const` `[inline]`

number of columns in dynamic model

Definition at line 184 of file ClpGubDynamicMatrix.hpp.

4.44.4.33 `int ClpGubDynamicMatrix::numberElements () const` `[inline]`

size of working matrix (max)

Definition at line 188 of file ClpGubDynamicMatrix.hpp.

4.44.4.34 `unsigned char* ClpGubDynamicMatrix::gubRowStatus () const` `[inline]`

Status region for gub slacks.

Definition at line 192 of file ClpGubDynamicMatrix.hpp.

4.44.4.35 `unsigned char* ClpGubDynamicMatrix::dynamicStatus () const` `[inline]`

Status region for gub variables.

Definition at line 196 of file ClpGubDynamicMatrix.hpp.

4.44.4.36 `int ClpGubDynamicMatrix::whichSet (int sequence) const`

Returns which set a variable is in.

4.44.5 Member Data Documentation

4.44.5.1 `double ClpGubDynamicMatrix::objectiveOffset_` `[protected]`

Saved value of objective offset.

Definition at line 209 of file ClpGubDynamicMatrix.hpp.

4.44.5.2 `CoinBigIndex* ClpGubDynamicMatrix::startColumn_` `[protected]`

Starts of each column.

Definition at line 211 of file ClpGubDynamicMatrix.hpp.

4.44.5.3 `int* ClpGubDynamicMatrix::row_` [protected]

rows

Definition at line 213 of file `ClpGubDynamicMatrix.hpp`.

4.44.5.4 `double* ClpGubDynamicMatrix::element_` [protected]

elements

Definition at line 215 of file `ClpGubDynamicMatrix.hpp`.

4.44.5.5 `double* ClpGubDynamicMatrix::cost_` [protected]

costs

Definition at line 217 of file `ClpGubDynamicMatrix.hpp`.

4.44.5.6 `int* ClpGubDynamicMatrix::fullStart_` [protected]

full starts

Definition at line 219 of file `ClpGubDynamicMatrix.hpp`.

4.44.5.7 `int* ClpGubDynamicMatrix::id_` [protected]

ids of active columns (just index here)

Definition at line 221 of file `ClpGubDynamicMatrix.hpp`.

4.44.5.8 `unsigned char* ClpGubDynamicMatrix::dynamicStatus_` [protected]

for status and which bound

Definition at line 223 of file `ClpGubDynamicMatrix.hpp`.

4.44.5.9 `double* ClpGubDynamicMatrix::lowerColumn_` [protected]

Optional lower bounds on columns.

Definition at line 225 of file `ClpGubDynamicMatrix.hpp`.

4.44.5.10 `double* ClpGubDynamicMatrix::upperColumn_` [protected]

Optional upper bounds on columns.

Definition at line 227 of file `ClpGubDynamicMatrix.hpp`.

4.44.5.11 `double* ClpGubDynamicMatrix::lowerSet_` [protected]

Optional true lower bounds on sets.

Definition at line 229 of file `ClpGubDynamicMatrix.hpp`.

4.44.5.12 `double* ClpGubDynamicMatrix::upperSet_` [protected]

Optional true upper bounds on sets.

Definition at line 231 of file `ClpGubDynamicMatrix.hpp`.

4.44.5.13 `int ClpGubDynamicMatrix::numberGubColumns_` [protected]

size

Definition at line 233 of file `ClpGubDynamicMatrix.hpp`.

4.44.5.14 `int ClpGubDynamicMatrix::firstAvailable_` [protected]

first free

Definition at line 235 of file `ClpGubDynamicMatrix.hpp`.

4.44.5.15 `int ClpGubDynamicMatrix::savedFirstAvailable_` [protected]

saved first free

Definition at line 237 of file `ClpGubDynamicMatrix.hpp`.

4.44.5.16 `int ClpGubDynamicMatrix::firstDynamic_` [protected]

first dynamic

Definition at line 239 of file `ClpGubDynamicMatrix.hpp`.

4.44.5.17 `int ClpGubDynamicMatrix::lastDynamic_` [protected]

number of columns in dynamic model

Definition at line 241 of file `ClpGubDynamicMatrix.hpp`.

4.44.5.18 `int ClpGubDynamicMatrix::numberElements_` [protected]

size of working matrix (max)

Definition at line 243 of file `ClpGubDynamicMatrix.hpp`.

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

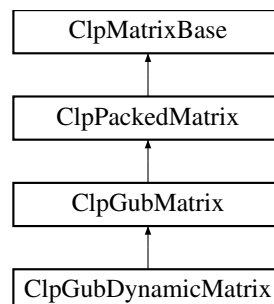
- [src/ClpGubDynamicMatrix.hpp](#)

4.45 ClpGubMatrix Class Reference

This implements Gub rows plus a [ClpPackedMatrix](#).

```
#include <ClpGubMatrix.hpp>
```

Inheritance 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 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 `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 numberOfRows, const int *whichRows, int numberColumns, const int *whichColumns)
Subset constructor (without gaps).
- `ClpGubMatrix` (const `CoinPackedMatrix` &wholeModel, int numberOfRows, 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 numberOfRows, 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 virtual ClpGubMatrix::~ClpGubMatrix () [virtual]

Destructor.

4.45.2.3 ClpGubMatrix::ClpGubMatrix (const ClpGubMatrix &)

The copy constructor.

4.45.2.4 ClpGubMatrix::ClpGubMatrix (const CoinPackedMatrix &)

The copy constructor from an CoinPackedMatrix.

4.45.2.5 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

4.45.2.6 ClpGubMatrix::ClpGubMatrix (const CoinPackedMatrix & *wholeModel*, int *numberRows*, const int * *whichRows*, int *numberColumns*, const int * *whichColumns*)

4.45.2.7 ClpGubMatrix::ClpGubMatrix (CoinPackedMatrix * *matrix*)

This takes over ownership (for space reasons)

4.45.2.8 ClpGubMatrix::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.

4.45.3 Member Function Documentation

4.45.3.1 virtual ClpMatrixBase* ClpGubMatrix::reverseOrderedCopy () const [virtual]

Returns a new matrix in reverse order without gaps (GUB wants NULL)

Reimplemented from [ClpPackedMatrix](#).

4.45.3.2 virtual CoinBigIndex ClpGubMatrix::countBasis (const int * *whichColumn*, int & *numberColumnBasic*) [virtual]

Returns number of elements in column part of basis.

Reimplemented from [ClpPackedMatrix](#).

4.45.3.3 virtual void ClpGubMatrix::fillBasis (ClpSimplex * *model*, const int * *whichColumn*, int & *numberColumnBasic*, int * *row*, int * *start*, int * *rowCount*, int * *columnCount*, CoinFactorizationDouble * *element*) [virtual]

Fills in column part of basis.

Reimplemented from [ClpPackedMatrix](#).

4.45.3.4 virtual void ClpGubMatrix::unpack (const ClpSimplex * *model*, CoinIndexedVector * *rowArray*, int *column*) const [virtual]

Unpacks a column into an CoinIndexedvector.

Reimplemented from [ClpPackedMatrix](#).

4.45.3.5 `virtual void ClpGubMatrix::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)

Reimplemented from [ClpPackedMatrix](#).

4.45.3.6 `virtual void ClpGubMatrix::add (const ClpSimplex * model, CoinIndexedVector * rowArray, int column, double multiplier) const`
`[virtual]`

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

Reimplemented from [ClpPackedMatrix](#).

4.45.3.7 `virtual void ClpGubMatrix::add (const ClpSimplex * model, double * array, int column, double multiplier) const`
`[virtual]`

Adds multiple of a column into an array.

Reimplemented from [ClpPackedMatrix](#).

4.45.3.8 `virtual void ClpGubMatrix::partialPricing (ClpSimplex * model, double start, double end, int & bestSequence, int & numberWanted)`
`[virtual]`

Partial pricing.

Reimplemented from [ClpPackedMatrix](#).

Reimplemented in [ClpGubDynamicMatrix](#).

4.45.3.9 `virtual int ClpGubMatrix::hiddenRows () const` `[virtual]`

Returns number of hidden rows e.g. gub.

Reimplemented from [ClpMatrixBase](#).

4.45.3.10 `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](#).

4.45.3.11 `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](#).

4.45.3.12 `virtual void ClpGubMatrix::subsetTransposeTimes (const ClpSimplex * model, const CoinIndexedVector * x, const CoinIndexedVector * y, CoinIndexedVector * z) const` `[virtual]`

Return `<code>x *A</code>` in `<code>z</code>` but

just for indices in y.

Note - z always packed mode

Reimplemented from [ClpPackedMatrix](#).

4.45.3.13 `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](#).

4.45.3.14 `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](#).

4.45.3.15 `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](#).

4.45.3.16 `virtual int ClpGubMatrix::generalExpanded (ClpSimplex * model, int mode, int & number)` [virtual]

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

Reimplemented from [ClpMatrixBase](#).

4.45.3.17 `virtual int ClpGubMatrix::updatePivot (ClpSimplex * model, double oldInValue, double oldOutValue)` [virtual]

update information for a pivot (and effective rhs)

Reimplemented from [ClpMatrixBase](#).

Reimplemented in [ClpGubDynamicMatrix](#).

4.45.3.18 `virtual void ClpGubMatrix::useEffectiveRhs (ClpSimplex * model, bool cheapest = true)` [virtual]

Sets up an effective RHS and does gub crash if needed.

Reimplemented in [ClpGubDynamicMatrix](#).

4.45.3.19 `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](#).

4.45.3.20 `virtual int ClpGubMatrix::synchronize (ClpSimplex * model, int mode) [virtual]`

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.

Reimplemented in [ClpGubDynamicMatrix](#).

4.45.3.21 `virtual void ClpGubMatrix::correctSequence (const ClpSimplex * model, int & sequenceIn, int & sequenceOut) [virtual]`

Correct sequence in and out to give true value.

Reimplemented from [ClpPackedMatrix](#).

4.45.3.22 `ClpGubMatrix& ClpGubMatrix::operator= (const ClpGubMatrix &)`

4.45.3.23 `virtual ClpMatrixBase* ClpGubMatrix::clone () const [virtual]`

Clone.

Reimplemented from [ClpPackedMatrix](#).

Reimplemented in [ClpGubDynamicMatrix](#).

4.45.3.24 `virtual ClpMatrixBase* ClpGubMatrix::subsetClone (int numberOfRows, 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](#).

4.45.3.25 `void ClpGubMatrix::redoSet (ClpSimplex * model, int newKey, int oldKey, int iSet)`

redoes next_ for a set.

4.45.3.26 `ClpSimplex::Status ClpGubMatrix::getStatus (int sequence) const [inline]`

Status.

Definition at line 210 of file [ClpGubMatrix.hpp](#).

4.45.3.27 `void ClpGubMatrix::setStatus (int sequence, ClpSimplex::Status status)` [inline]

Definition at line 213 of file ClpGubMatrix.hpp.

4.45.3.28 `void ClpGubMatrix::setFlagged (int sequence)` [inline]

To flag a variable.

Definition at line 219 of file ClpGubMatrix.hpp.

4.45.3.29 `void ClpGubMatrix::clearFlagged (int sequence)` [inline]

Definition at line 222 of file ClpGubMatrix.hpp.

4.45.3.30 `bool ClpGubMatrix::flagged (int sequence) const` [inline]

Definition at line 225 of file ClpGubMatrix.hpp.

4.45.3.31 `void ClpGubMatrix::setAbove (int sequence)` [inline]

To say key is above ub.

Definition at line 229 of file ClpGubMatrix.hpp.

4.45.3.32 `void ClpGubMatrix::setFeasible (int sequence)` [inline]

To say key is feasible.

Definition at line 235 of file ClpGubMatrix.hpp.

4.45.3.33 `void ClpGubMatrix::setBelow (int sequence)` [inline]

To say key is below lb.

Definition at line 241 of file ClpGubMatrix.hpp.

4.45.3.34 `double ClpGubMatrix::weight (int sequence) const` [inline]

Definition at line 246 of file ClpGubMatrix.hpp.

4.45.3.35 `int* ClpGubMatrix::start () const` [inline]

Starts.

Definition at line 252 of file ClpGubMatrix.hpp.

4.45.3.36 `int* ClpGubMatrix::end () const` [inline]

End.

Definition at line 256 of file ClpGubMatrix.hpp.

4.45.3.37 `double* ClpGubMatrix::lower () const` [inline]

Lower bounds on sets.

Definition at line 260 of file ClpGubMatrix.hpp.

4.45.3.38 `double* ClpGubMatrix::upper () const` [inline]

Upper bounds on sets.

Definition at line 264 of file ClpGubMatrix.hpp.

4.45.3.39 `int* ClpGubMatrix::keyVariable () const [inline]`

Key variable of set.

Definition at line 268 of file ClpGubMatrix.hpp.

4.45.3.40 `int* ClpGubMatrix::backward () const [inline]`

Backward pointer to set number.

Definition at line 272 of file ClpGubMatrix.hpp.

4.45.3.41 `int ClpGubMatrix::numberSets () const [inline]`

Number of sets (gub rows)

Definition at line 276 of file ClpGubMatrix.hpp.

4.45.3.42 `void ClpGubMatrix::switchOffCheck ()`

Switches off dj checking each factorization (for BIG models)

4.45.4 Member Data Documentation

4.45.4.1 `double ClpGubMatrix::sumDualInfeasibilities_ [protected]`

Sum of dual infeasibilities.

Definition at line 289 of file ClpGubMatrix.hpp.

4.45.4.2 `double ClpGubMatrix::sumPrimalInfeasibilities_ [protected]`

Sum of primal infeasibilities.

Definition at line 291 of file ClpGubMatrix.hpp.

4.45.4.3 `double ClpGubMatrix::sumOfRelaxedDualInfeasibilities_ [protected]`

Sum of Dual infeasibilities using tolerance based on error in duals.

Definition at line 293 of file ClpGubMatrix.hpp.

4.45.4.4 `double ClpGubMatrix::sumOfRelaxedPrimalInfeasibilities_ [protected]`

Sum of Primal infeasibilities using tolerance based on error in primals.

Definition at line 295 of file ClpGubMatrix.hpp.

4.45.4.5 `double ClpGubMatrix::infeasibilityWeight_ [protected]`

Infeasibility weight when last full pass done.

Definition at line 297 of file ClpGubMatrix.hpp.

4.45.4.6 `int* ClpGubMatrix::start_ [protected]`

Starts.

Definition at line 299 of file ClpGubMatrix.hpp.

4.45.4.7 `int* ClpGubMatrix::end_` [protected]

End.

Definition at line 301 of file ClpGubMatrix.hpp.

4.45.4.8 `double* ClpGubMatrix::lower_` [protected]

Lower bounds on sets.

Definition at line 303 of file ClpGubMatrix.hpp.

4.45.4.9 `double* ClpGubMatrix::upper_` [protected]

Upper bounds on sets.

Definition at line 305 of file ClpGubMatrix.hpp.

4.45.4.10 `unsigned char* ClpGubMatrix::status_` [mutable], [protected]

Status of slacks.

Definition at line 307 of file ClpGubMatrix.hpp.

4.45.4.11 `unsigned char* ClpGubMatrix::saveStatus_` [protected]

Saved status of slacks.

Definition at line 309 of file ClpGubMatrix.hpp.

4.45.4.12 `int* ClpGubMatrix::savedKeyVariable_` [protected]

Saved key variables.

Definition at line 311 of file ClpGubMatrix.hpp.

4.45.4.13 `int* ClpGubMatrix::backward_` [protected]

Backward pointer to set number.

Definition at line 313 of file ClpGubMatrix.hpp.

4.45.4.14 `int* ClpGubMatrix::backToPivotRow_` [protected]

Backward pointer to pivot row !!!

Definition at line 315 of file ClpGubMatrix.hpp.

4.45.4.15 `double* ClpGubMatrix::changeCost_` [protected]

Change in costs for keys.

Definition at line 317 of file ClpGubMatrix.hpp.

4.45.4.16 `int* ClpGubMatrix::keyVariable_` [mutable], [protected]

Key variable of set.

Definition at line 319 of file ClpGubMatrix.hpp.

4.45.4.17 `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.

4.45.4.18 `int* ClpGubMatrix::toIndex_ [protected]`

Backward pointer to index in CoinIndexedVector.

Definition at line 325 of file ClpGubMatrix.hpp.

4.45.4.19 `int* ClpGubMatrix::fromIndex_ [protected]`

Definition at line 327 of file ClpGubMatrix.hpp.

4.45.4.20 `ClpSimplex* ClpGubMatrix::model_ [protected]`

Pointer back to model.

Definition at line 329 of file ClpGubMatrix.hpp.

4.45.4.21 `int ClpGubMatrix::numberDualInfeasibilities_ [protected]`

Number of dual infeasibilities.

Definition at line 331 of file ClpGubMatrix.hpp.

4.45.4.22 `int ClpGubMatrix::numberPrimalInfeasibilities_ [protected]`

Number of primal infeasibilities.

Definition at line 333 of file ClpGubMatrix.hpp.

4.45.4.23 `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.

4.45.4.24 `int ClpGubMatrix::numberSets_ [protected]`

Number of sets (gub rows)

Definition at line 341 of file ClpGubMatrix.hpp.

4.45.4.25 `int ClpGubMatrix::saveNumber_ [protected]`

Number in vector without gub extension.

Definition at line 343 of file ClpGubMatrix.hpp.

4.45.4.26 `int ClpGubMatrix::possiblePivotKey_ [protected]`

Pivot row of possible next key.

Definition at line 345 of file ClpGubMatrix.hpp.

4.45.4.27 int ClpGubMatrix::gubSlackIn_ [protected]

Gub slack in (set number or -1)

Definition at line 347 of file ClpGubMatrix.hpp.

4.45.4.28 int ClpGubMatrix::firstGub_ [protected]

First gub variables (same as start_[0] at present)

Definition at line 349 of file ClpGubMatrix.hpp.

4.45.4.29 int ClpGubMatrix::lastGub_ [protected]

last gub variable (same as end_[numberSets_-1] at present)

Definition at line 351 of file ClpGubMatrix.hpp.

4.45.4.30 int ClpGubMatrix::gubType_ [protected]

type of gub - 0 not contiguous, 1 contiguous add 8 bit to say no ubs on individual variables

Definition at line 354 of file ClpGubMatrix.hpp.

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

- [src/ClpGubMatrix.hpp](#)

4.46 ClpHashValue Class Reference

```
#include <ClpNode.hpp>
```

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 [virtual \[ClpHashValue::~~ClpHashValue \\(\\)\]\(#\) \[virtual\]](#)

Destructor.

4.46.2.4 [ClpHashValue::ClpHashValue \(const \[ClpHashValue\]\(#\) & \)](#)

The copy constructor.

4.46.3 Member Function Documentation

4.46.3.1 [int \[ClpHashValue::index \\(double *value* \\) const\]\(#\)](#)

Return index or -1 if not found.

4.46.3.2 `int ClpHashValue::addValue (double value)`

Add value to list and return index.

4.46.3.3 `int ClpHashValue::numberEntries () const [inline]`

Number of different entries.

Definition at line 298 of file `ClpNode.hpp`.

4.46.3.4 `ClpHashValue& ClpHashValue::operator= (const ClpHashValue &)`

=

4.46.4 Member Data Documentation

4.46.4.1 `CoinHashLink* ClpHashValue::hash_ [mutable],[protected]`

Hash table.

Definition at line 340 of file `ClpNode.hpp`.

4.46.4.2 `int ClpHashValue::numberHash_ [protected]`

Number of entries in hash table.

Definition at line 342 of file `ClpNode.hpp`.

4.46.4.3 `int ClpHashValue::maxHash_ [protected]`

Maximum number of entries in hash table i.e. size.

Definition at line 344 of file `ClpNode.hpp`.

4.46.4.4 `int ClpHashValue::lastUsed_ [protected]`

Last used space.

Definition at line 346 of file `ClpNode.hpp`.

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

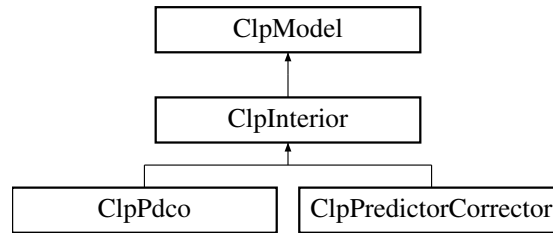
- [src/ClpNode.hpp](#)

4.47 ClpInterior Class Reference

This solves LPs using interior point methods.

```
#include <ClpInterior.hpp>
```

Inheritance 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, CoinWorkDouble 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 [maximumRHSCheck_](#)
maximumRHSCheck
- 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 \(\)](#)

Default constructor.

4.47.2.2 [ClpInterior::ClpInterior \(const \[ClpInterior\]\(#\) & \)](#)

Copy constructor.

4.47.2.3 [ClpInterior::ClpInterior \(const \[ClpModel\]\(#\) & \)](#)

Copy constructor from model.

4.47.2.4 [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

4.47.2.5 [ClpInterior::~~ClpInterior \(\)](#)

Destructor.

4.47.3 Member Function Documentation

4.47.3.1 `ClpInterior& ClpInterior::operator= (const ClpInterior & rhs)`

Assignment operator. This copies the data.

4.47.3.2 `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

4.47.3.3 `void ClpInterior::loadProblem (const CoinPackedMatrix & matrix, const double * collb, const double * colub, const double * obj, const double * rowlb, const double * rowub, const double * rowObjective = NULL)`

4.47.3.4 `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).

4.47.3.5 `void ClpInterior::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.

4.47.3.6 `int ClpInterior::readMps (const char * filename, bool keepNames = false, bool ignoreErrors = false)`

Read an mps file from the given filename.

4.47.3.7 `void ClpInterior::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. This is same as [ClpModel](#) one.

4.47.3.8 `void ClpInterior::returnModel (ClpModel & otherModel)`

Return model - updates any scalars.

4.47.3.9 `int ClpInterior::pdco ()`

Pdco algorithm - see [ClpPdco.hpp](#) for method.

4.47.3.10 `int ClpInterior::pdco (ClpPdcoBase * stuff, Options & options, Info & info, Outfo & outfo)`

4.47.3.11 `int ClpInterior::primalDual ()`

Primal-Dual Predictor-Corrector barrier.

4.47.3.12 `bool ClpInterior::primalFeasible () const [inline]`

If problem is primal feasible.

Definition at line 165 of file ClpInterior.hpp.

4.47.3.13 `bool ClpInterior::dualFeasible () const [inline]`

If problem is dual feasible.

Definition at line 169 of file ClpInterior.hpp.

4.47.3.14 `int ClpInterior::algorithm () const [inline]`

Current (or last) algorithm.

Definition at line 173 of file ClpInterior.hpp.

4.47.3.15 `void ClpInterior::setAlgorithm (int value) [inline]`

Set algorithm.

Definition at line 177 of file ClpInterior.hpp.

4.47.3.16 `CoinWorkDouble ClpInterior::sumDualInfeasibilities () const [inline]`

Sum of dual infeasibilities.

Definition at line 181 of file ClpInterior.hpp.

4.47.3.17 `CoinWorkDouble ClpInterior::sumPrimalInfeasibilities () const [inline]`

Sum of primal infeasibilities.

Definition at line 185 of file ClpInterior.hpp.

4.47.3.18 `CoinWorkDouble ClpInterior::dualObjective () const [inline]`

dualObjective.

Definition at line 189 of file ClpInterior.hpp.

4.47.3.19 `CoinWorkDouble ClpInterior::primalObjective () const [inline]`

primalObjective.

Definition at line 193 of file ClpInterior.hpp.

4.47.3.20 `CoinWorkDouble ClpInterior::diagonalNorm () const [inline]`

diagonalNorm

Definition at line 197 of file ClpInterior.hpp.

4.47.3.21 `CoinWorkDouble ClpInterior::linearPerturbation () const [inline]`

`linearPerturbation`

Definition at line 201 of file `ClpInterior.hpp`.

4.47.3.22 `void ClpInterior::setLinearPerturbation (CoinWorkDouble value) [inline]`

Definition at line 204 of file `ClpInterior.hpp`.

4.47.3.23 `CoinWorkDouble ClpInterior::projectionTolerance () const [inline]`

`projectionTolerance`

Definition at line 208 of file `ClpInterior.hpp`.

4.47.3.24 `void ClpInterior::setProjectionTolerance (CoinWorkDouble value) [inline]`

Definition at line 211 of file `ClpInterior.hpp`.

4.47.3.25 `CoinWorkDouble ClpInterior::diagonalPerturbation () const [inline]`

`diagonalPerturbation`

Definition at line 215 of file `ClpInterior.hpp`.

4.47.3.26 `void ClpInterior::setDiagonalPerturbation (CoinWorkDouble value) [inline]`

Definition at line 218 of file `ClpInterior.hpp`.

4.47.3.27 `CoinWorkDouble ClpInterior::gamma () const [inline]`

`gamma`

Definition at line 222 of file `ClpInterior.hpp`.

4.47.3.28 `void ClpInterior::setGamma (CoinWorkDouble value) [inline]`

Definition at line 225 of file `ClpInterior.hpp`.

4.47.3.29 `CoinWorkDouble ClpInterior::delta () const [inline]`

`delta`

Definition at line 229 of file `ClpInterior.hpp`.

4.47.3.30 `void ClpInterior::setDelta (CoinWorkDouble value) [inline]`

Definition at line 232 of file `ClpInterior.hpp`.

4.47.3.31 `CoinWorkDouble ClpInterior::complementarityGap () const [inline]`

`ComplementarityGap`.

Definition at line 236 of file `ClpInterior.hpp`.

4.47.3.32 `CoinWorkDouble ClpInterior::largestPrimalError () const [inline]`

`Largest error on Ax=b`.

Definition at line 244 of file `ClpInterior.hpp`.

4.47.3.33 `CoinWorkDouble ClpInterior::largestDualError () const [inline]`

Largest error on basic duals.

Definition at line 248 of file ClpInterior.hpp.

4.47.3.34 `int ClpInterior::maximumBarrierIterations () const [inline]`

Maximum iterations.

Definition at line 252 of file ClpInterior.hpp.

4.47.3.35 `void ClpInterior::setMaximumBarrierIterations (int value) [inline]`

Definition at line 255 of file ClpInterior.hpp.

4.47.3.36 `void ClpInterior::setCholesky (ClpCholeskyBase * cholesky)`

Set cholesky (and delete present one)

4.47.3.37 `int ClpInterior::numberFixed () const`

Return number fixed to see if worth presolving.

4.47.3.38 `void ClpInterior::fixFixed (bool reallyFix = true)`

fix variables interior says should be.

If reallyFix false then just set values to exact bounds

4.47.3.39 `CoinWorkDouble* ClpInterior::primalR () const [inline]`

Primal erturbation vector.

Definition at line 266 of file ClpInterior.hpp.

4.47.3.40 `CoinWorkDouble* ClpInterior::dualR () const [inline]`

Dual erturbation vector.

Definition at line 270 of file ClpInterior.hpp.

4.47.3.41 `void ClpInterior::gutsOfDelete () [protected]`

Does most of deletion.

4.47.3.42 `void ClpInterior::gutsOfCopy (const ClpInterior & rhs) [protected]`

Does most of copying.

4.47.3.43 `bool ClpInterior::createWorkingData () [protected]`

Returns true if data looks okay, false if not.

4.47.3.44 `void ClpInterior::deleteWorkingData () [protected]`

4.47.3.45 `bool ClpInterior::sanityCheck () [protected]`

Sanity check on input rim data.

4.47.3.46 `int ClpInterior::housekeeping () [protected]`

This does housekeeping.

4.47.3.47 `CoinWorkDouble ClpInterior::rawObjectiveValue () const [inline]`

Raw objective value (so always minimize)

Definition at line 294 of file `ClpInterior.hpp`.

4.47.3.48 `int ClpInterior::isColumn (int sequence) const [inline]`

Returns 1 if *sequence* indicates column.

Definition at line 298 of file `ClpInterior.hpp`.

4.47.3.49 `int ClpInterior::sequenceWithin (int sequence) const [inline]`

Returns sequence number within section.

Definition at line 302 of file `ClpInterior.hpp`.

4.47.3.50 `void ClpInterior::checkSolution ()`

Checks solution.

4.47.3.51 `CoinWorkDouble ClpInterior::quadraticDjs (CoinWorkDouble * djRegion, const CoinWorkDouble * solution, CoinWorkDouble scaleFactor)`

Modifies djs to allow for quadratic.

returns quadratic offset

4.47.3.52 `void ClpInterior::setFixed (int sequence) [inline]`

To say a variable is fixed.

Definition at line 313 of file `ClpInterior.hpp`.

4.47.3.53 `void ClpInterior::clearFixed (int sequence) [inline]`

Definition at line 316 of file `ClpInterior.hpp`.

4.47.3.54 `bool ClpInterior::fixed (int sequence) const [inline]`

Definition at line 319 of file `ClpInterior.hpp`.

4.47.3.55 `void ClpInterior::setFlagged (int sequence) [inline]`

To flag a variable.

Definition at line 324 of file `ClpInterior.hpp`.

4.47.3.56 `void ClpInterior::clearFlagged (int sequence) [inline]`

Definition at line 327 of file `ClpInterior.hpp`.

4.47.3.57 `bool ClpInterior::flagged (int sequence) const [inline]`

Definition at line 330 of file `ClpInterior.hpp`.

4.47.3.58 void ClpInterior::setFixedOrFree (int *sequence*) [inline]

To say a variable is fixed OR free.

Definition at line 335 of file ClpInterior.hpp.

4.47.3.59 void ClpInterior::clearFixedOrFree (int *sequence*) [inline]

Definition at line 338 of file ClpInterior.hpp.

4.47.3.60 bool ClpInterior::fixedOrFree (int *sequence*) const [inline]

Definition at line 341 of file ClpInterior.hpp.

4.47.3.61 void ClpInterior::setLowerBound (int *sequence*) [inline]

To say a variable has lower bound.

Definition at line 346 of file ClpInterior.hpp.

4.47.3.62 void ClpInterior::clearLowerBound (int *sequence*) [inline]

Definition at line 349 of file ClpInterior.hpp.

4.47.3.63 bool ClpInterior::lowerBound (int *sequence*) const [inline]

Definition at line 352 of file ClpInterior.hpp.

4.47.3.64 void ClpInterior::setUpperBound (int *sequence*) [inline]

To say a variable has upper bound.

Definition at line 357 of file ClpInterior.hpp.

4.47.3.65 void ClpInterior::clearUpperBound (int *sequence*) [inline]

Definition at line 360 of file ClpInterior.hpp.

4.47.3.66 bool ClpInterior::upperBound (int *sequence*) const [inline]

Definition at line 363 of file ClpInterior.hpp.

4.47.3.67 void ClpInterior::setFakeLower (int *sequence*) [inline]

To say a variable has fake lower bound.

Definition at line 368 of file ClpInterior.hpp.

4.47.3.68 void ClpInterior::clearFakeLower (int *sequence*) [inline]

Definition at line 371 of file ClpInterior.hpp.

4.47.3.69 bool ClpInterior::fakeLower (int *sequence*) const [inline]

Definition at line 374 of file ClpInterior.hpp.

4.47.3.70 void ClpInterior::setFakeUpper (int *sequence*) [inline]

To say a variable has fake upper bound.

Definition at line 379 of file ClpInterior.hpp.

4.47.3.71 void ClpInterior::clearFakeUpper (int *sequence*) [inline]

Definition at line 382 of file ClpInterior.hpp.

4.47.3.72 bool ClpInterior::fakeUpper (int *sequence*) const [inline]

Definition at line 385 of file ClpInterior.hpp.

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::largestPrimalError_ [protected]

Largest error on Ax=b.

Definition at line 400 of file ClpInterior.hpp.

4.47.5.2 CoinWorkDouble ClpInterior::largestDualError_ [protected]

Largest error on basic duals.

Definition at line 402 of file ClpInterior.hpp.

4.47.5.3 CoinWorkDouble ClpInterior::sumDualInfeasibilities_ [protected]

Sum of dual infeasibilities.

Definition at line 404 of file ClpInterior.hpp.

4.47.5.4 CoinWorkDouble ClpInterior::sumPrimalInfeasibilities_ [protected]

Sum of primal infeasibilities.

Definition at line 406 of file ClpInterior.hpp.

4.47.5.5 CoinWorkDouble ClpInterior::worstComplementarity_ [protected]

Worst complementarity.

Definition at line 408 of file ClpInterior.hpp.

4.47.5.6 CoinWorkDouble ClpInterior::xsize_

Definition at line 411 of file ClpInterior.hpp.

4.47.5.7 CoinWorkDouble ClpInterior::zsize_

Definition at line 412 of file ClpInterior.hpp.

4.47.5.8 CoinWorkDouble* ClpInterior::lower_ [protected]

Working copy of lower bounds (Owner of arrays below)

Definition at line 415 of file ClpInterior.hpp.

4.47.5.9 CoinWorkDouble* ClpInterior::rowLowerWork_ [protected]

Row lower bounds - working copy.

Definition at line 417 of file ClpInterior.hpp.

4.47.5.10 CoinWorkDouble* ClpInterior::columnLowerWork_ [protected]

Column lower bounds - working copy.

Definition at line 419 of file ClpInterior.hpp.

4.47.5.11 CoinWorkDouble* ClpInterior::upper_ [protected]

Working copy of upper bounds (Owner of arrays below)

Definition at line 421 of file ClpInterior.hpp.

4.47.5.12 CoinWorkDouble* ClpInterior::rowUpperWork_ [protected]

Row upper bounds - working copy.

Definition at line 423 of file ClpInterior.hpp.

4.47.5.13 CoinWorkDouble* ClpInterior::columnUpperWork_ [protected]

Column upper bounds - working copy.

Definition at line 425 of file ClpInterior.hpp.

4.47.5.14 CoinWorkDouble* ClpInterior::cost_ [protected]

Working copy of objective.

Definition at line 427 of file ClpInterior.hpp.

4.47.5.15 CoinWorkDouble* ClpInterior::rhs_

Rhs.

Definition at line 430 of file ClpInterior.hpp.

4.47.5.16 CoinWorkDouble* ClpInterior::x_

Definition at line 431 of file ClpInterior.hpp.

4.47.5.17 CoinWorkDouble* ClpInterior::y_

Definition at line 432 of file ClpInterior.hpp.

4.47.5.18 CoinWorkDouble* ClpInterior::dj_

Definition at line 433 of file ClpInterior.hpp.

4.47.5.19 ClpLsqr* ClpInterior::lsqrObject_ [protected]

Pointer to Lsqr object.

Definition at line 436 of file ClpInterior.hpp.

4.47.5.20 ClpPdcoBase* ClpInterior::pdcoStuff_ [protected]

Pointer to stuff.

Definition at line 438 of file ClpInterior.hpp.

4.47.5.21 CoinWorkDouble ClpInterior::mu_ [protected]

Below here is standard barrier stuff mu.

Definition at line 441 of file ClpInterior.hpp.

4.47.5.22 CoinWorkDouble ClpInterior::objectiveNorm_ [protected]

objectiveNorm.

Definition at line 443 of file ClpInterior.hpp.

4.47.5.23 CoinWorkDouble ClpInterior::rhsNorm_ [protected]

rhsNorm.

Definition at line 445 of file ClpInterior.hpp.

4.47.5.24 CoinWorkDouble ClpInterior::solutionNorm_ [protected]

solutionNorm.

Definition at line 447 of file ClpInterior.hpp.

4.47.5.25 CoinWorkDouble ClpInterior::dualObjective_ [protected]

dualObjective.

Definition at line 449 of file ClpInterior.hpp.

4.47.5.26 CoinWorkDouble ClpInterior::primalObjective_ [protected]

primalObjective.

Definition at line 451 of file ClpInterior.hpp.

4.47.5.27 CoinWorkDouble ClpInterior::diagonalNorm_ [protected]

diagonalNorm.

Definition at line 453 of file ClpInterior.hpp.

4.47.5.28 CoinWorkDouble ClpInterior::stepLength_ [protected]

stepLength

Definition at line 455 of file ClpInterior.hpp.

4.47.5.29 `CoinWorkDouble ClpInterior::linearPerturbation_` `[protected]`

linearPerturbation

Definition at line 457 of file ClpInterior.hpp.

4.47.5.30 `CoinWorkDouble ClpInterior::diagonalPerturbation_` `[protected]`

diagonalPerturbation

Definition at line 459 of file ClpInterior.hpp.

4.47.5.31 `CoinWorkDouble ClpInterior::gamma_` `[protected]`

Definition at line 461 of file ClpInterior.hpp.

4.47.5.32 `CoinWorkDouble ClpInterior::delta_` `[protected]`

Definition at line 463 of file ClpInterior.hpp.

4.47.5.33 `CoinWorkDouble ClpInterior::targetGap_` `[protected]`

targetGap

Definition at line 465 of file ClpInterior.hpp.

4.47.5.34 `CoinWorkDouble ClpInterior::projectionTolerance_` `[protected]`

projectionTolerance

Definition at line 467 of file ClpInterior.hpp.

4.47.5.35 `CoinWorkDouble ClpInterior::maximumRHSError_` `[protected]`

maximumRHSError. maximum Ax

Definition at line 469 of file ClpInterior.hpp.

4.47.5.36 `CoinWorkDouble ClpInterior::maximumBoundInfeasibility_` `[protected]`

maximumBoundInfeasibility.

Definition at line 471 of file ClpInterior.hpp.

4.47.5.37 `CoinWorkDouble ClpInterior::maximumDualError_` `[protected]`

maximumDualError.

Definition at line 473 of file ClpInterior.hpp.

4.47.5.38 `CoinWorkDouble ClpInterior::diagonalScaleFactor_` `[protected]`

diagonalScaleFactor.

Definition at line 475 of file ClpInterior.hpp.

4.47.5.39 `CoinWorkDouble ClpInterior::scaleFactor_` `[protected]`

scaleFactor. For scaling objective

Definition at line 477 of file ClpInterior.hpp.

4.47.5.40 `CoinWorkDouble ClpInterior::actualPrimalStep_` [protected]

actualPrimalStep

Definition at line 479 of file ClpInterior.hpp.

4.47.5.41 `CoinWorkDouble ClpInterior::actualDualStep_` [protected]

actualDualStep

Definition at line 481 of file ClpInterior.hpp.

4.47.5.42 `CoinWorkDouble ClpInterior::smallestInfeasibility_` [protected]

smallestInfeasibility

Definition at line 483 of file ClpInterior.hpp.

4.47.5.43 `CoinWorkDouble ClpInterior::historyInfeasibility_[LENGTH_HISTORY]` [protected]

Definition at line 486 of file ClpInterior.hpp.

4.47.5.44 `CoinWorkDouble ClpInterior::complementarityGap_` [protected]

complementarityGap.

Definition at line 488 of file ClpInterior.hpp.

4.47.5.45 `CoinWorkDouble ClpInterior::baseObjectiveNorm_` [protected]

baseObjectiveNorm

Definition at line 490 of file ClpInterior.hpp.

4.47.5.46 `CoinWorkDouble ClpInterior::worstDirectionAccuracy_` [protected]

worstDirectionAccuracy

Definition at line 492 of file ClpInterior.hpp.

4.47.5.47 `CoinWorkDouble ClpInterior::maximumRHSChange_` [protected]

maximumRHSChange

Definition at line 494 of file ClpInterior.hpp.

4.47.5.48 `CoinWorkDouble* ClpInterior::errorRegion_` [protected]

errorRegion. i.e. Ax

Definition at line 496 of file ClpInterior.hpp.

4.47.5.49 `CoinWorkDouble* ClpInterior::rhsFixRegion_` [protected]

rhsFixRegion.

Definition at line 498 of file ClpInterior.hpp.

4.47.5.50 `CoinWorkDouble* ClpInterior::upperSlack_` [protected]

upperSlack

Definition at line 500 of file ClpInterior.hpp.

4.47.5.51 `CoinWorkDouble* ClpInterior::lowerSlack_` [protected]

lowerSlack

Definition at line 502 of file ClpInterior.hpp.

4.47.5.52 `CoinWorkDouble* ClpInterior::diagonal_` [protected]

diagonal

Definition at line 504 of file ClpInterior.hpp.

4.47.5.53 `CoinWorkDouble* ClpInterior::solution_` [protected]

solution

Definition at line 506 of file ClpInterior.hpp.

4.47.5.54 `CoinWorkDouble* ClpInterior::workArray_` [protected]

work array

Definition at line 508 of file ClpInterior.hpp.

4.47.5.55 `CoinWorkDouble* ClpInterior::deltaX_` [protected]

delta X

Definition at line 510 of file ClpInterior.hpp.

4.47.5.56 `CoinWorkDouble* ClpInterior::deltaY_` [protected]

delta Y

Definition at line 512 of file ClpInterior.hpp.

4.47.5.57 `CoinWorkDouble* ClpInterior::deltaZ_` [protected]

deltaZ.

Definition at line 514 of file ClpInterior.hpp.

4.47.5.58 `CoinWorkDouble* ClpInterior::deltaW_` [protected]

deltaW.

Definition at line 516 of file ClpInterior.hpp.

4.47.5.59 `CoinWorkDouble* ClpInterior::deltaSU_` [protected]

deltaS.

Definition at line 518 of file ClpInterior.hpp.

4.47.5.60 `CoinWorkDouble* ClpInterior::deltaSL_` [protected]

Definition at line 519 of file `ClpInterior.hpp`.

4.47.5.61 `CoinWorkDouble* ClpInterior::primalR_` [protected]

Primal regularization array.

Definition at line 521 of file `ClpInterior.hpp`.

4.47.5.62 `CoinWorkDouble* ClpInterior::dualR_` [protected]

Dual regularization array.

Definition at line 523 of file `ClpInterior.hpp`.

4.47.5.63 `CoinWorkDouble* ClpInterior::rhsB_` [protected]

rhs B

Definition at line 525 of file `ClpInterior.hpp`.

4.47.5.64 `CoinWorkDouble* ClpInterior::rhsU_` [protected]

rhsU.

Definition at line 527 of file `ClpInterior.hpp`.

4.47.5.65 `CoinWorkDouble* ClpInterior::rhsL_` [protected]

rhsL.

Definition at line 529 of file `ClpInterior.hpp`.

4.47.5.66 `CoinWorkDouble* ClpInterior::rhsZ_` [protected]

rhsZ.

Definition at line 531 of file `ClpInterior.hpp`.

4.47.5.67 `CoinWorkDouble* ClpInterior::rhsW_` [protected]

rhsW.

Definition at line 533 of file `ClpInterior.hpp`.

4.47.5.68 `CoinWorkDouble* ClpInterior::rhsC_` [protected]

rhs C

Definition at line 535 of file `ClpInterior.hpp`.

4.47.5.69 `CoinWorkDouble* ClpInterior::zVec_` [protected]

zVec

Definition at line 537 of file `ClpInterior.hpp`.

4.47.5.70 `CoinWorkDouble* ClpInterior::wVec_` [protected]

wVec

Definition at line 539 of file ClpInterior.hpp.

4.47.5.71 `ClpCholeskyBase* ClpInterior::cholesky_` `[protected]`

cholesky.

Definition at line 541 of file ClpInterior.hpp.

4.47.5.72 `int ClpInterior::numberComplementarityPairs_` `[protected]`

numberComplementarityPairs i.e. ones with lower and/or upper bounds (not fixed)

Definition at line 543 of file ClpInterior.hpp.

4.47.5.73 `int ClpInterior::numberComplementarityItems_` `[protected]`

numberComplementarityItems_ i.e. number of active bounds

Definition at line 545 of file ClpInterior.hpp.

4.47.5.74 `int ClpInterior::maximumBarrierIterations_` `[protected]`

Maximum iterations.

Definition at line 547 of file ClpInterior.hpp.

4.47.5.75 `bool ClpInterior::gonePrimalFeasible_` `[protected]`

gonePrimalFeasible.

Definition at line 549 of file ClpInterior.hpp.

4.47.5.76 `bool ClpInterior::goneDualFeasible_` `[protected]`

goneDualFeasible.

Definition at line 551 of file ClpInterior.hpp.

4.47.5.77 `int ClpInterior::algorithm_` `[protected]`

Which algorithm being used.

Definition at line 553 of file ClpInterior.hpp.

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

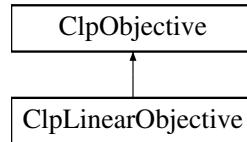
- [src/ClpInterior.hpp](#)

4.48 ClpLinearObjective Class Reference

Linear Objective Class.

```
#include <ClpLinearObjective.hpp>
```

Inheritance 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 ¤tObj, 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 ()

Default Constructor.

4.48.2.2 ClpLinearObjective::ClpLinearObjective (const double * *objective*, int *numberColumns*)

Constructor from objective.

4.48.2.3 ClpLinearObjective::ClpLinearObjective (const ClpLinearObjective &)

Copy constructor.

4.48.2.4 ClpLinearObjective::ClpLinearObjective (const ClpLinearObjective & *rhs*, int *numberColumns*, const int * *whichColumns*)

Subset constructor.

Duplicates are allowed and order is as given.

4.48.2.5 virtual ClpLinearObjective::~~ClpLinearObjective () [virtual]

Destructor.

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::reducedGradient (ClpSimplex * *model*, double * *region*, bool *useFeasibleCosts*) [virtual]

Returns reduced gradient. Returns an offset (to be added to current one).

Implements [ClpObjective](#).

4.48.3.3 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](#).

4.48.3.4 virtual double ClpLinearObjective::objectiveValue (const ClpSimplex * *model*, const double * *solution*) const [virtual]

Return objective value (without any [ClpModel](#) offset) (model may be NULL)

Implements [ClpObjective](#).

4.48.3.5 `virtual void ClpLinearObjective::resize (int newNumberColumns) [virtual]`

Resize objective.

Implements [ClpObjective](#).

4.48.3.6 `virtual void ClpLinearObjective::deleteSome (int numberToDelete, const int * which) [virtual]`

Delete columns in objective.

Implements [ClpObjective](#).

4.48.3.7 `virtual void ClpLinearObjective::reallyScale (const double * columnScale) [virtual]`

Scale objective.

Implements [ClpObjective](#).

4.48.3.8 `ClpLinearObjective& ClpLinearObjective::operator= (const ClpLinearObjective & rhs)`

Assignment operator.

4.48.3.9 `virtual ClpObjective* ClpLinearObjective::clone () const [virtual]`

Clone.

Implements [ClpObjective](#).

4.48.3.10 `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:

- [src/ClpLinearObjective.hpp](#)

4.49 ClpLsqqr Class Reference

This class implements LSQR.

```
#include <ClpLsqqr.hpp>
```

Public Member Functions

Constructors and destructors

- [ClpLsqqr](#) ()
Default constructor.
- [ClpLsqqr](#) ([ClpInterior](#) *model)
Constructor for use with Pdco model (note modified for pdco!!!!)
- [ClpLsqqr](#) (const [ClpLsqqr](#) &)
Copy constructor.
- [ClpLsqqr](#) & `operator=` (const [ClpLsqqr](#) &rhs)

- Assignment operator. This copies the data.
- `~ClpLsqqr ()`
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 Lsqqr 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 $\text{cond}(A)$ and the size of damp.)

conlim is also a stopping tolerance. lsqr terminates if an estimate of $\text{cond}(A)$ 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 $\text{atol} = \text{btol} = \text{conlim} = \text{zero}$, 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.
= 1 means x is an approximate solution to $Ax = b$.
= 2 means x approximately solves the least-squares problem.

rnorm = $\text{norm}(r)$ if damp = 0, where $r = b - Ax$,
= $\sqrt{\text{norm}(r)^2 + \text{damp}^2 * \text{norm}(x)^2}$ otherwise.

xnorm = $\text{norm}(x)$.

var estimates $\text{diag}(\text{inv}(A'A))$. 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.

4.49.2 Constructor & Destructor Documentation

4.49.2.1 ClpLsqr::ClpLsqr ()

Default constructor.

4.49.2.2 ClpLsqr::ClpLsqr (ClpInterior * model)

Constructor for use with Pdco model (note modified for pdco!!!!)

4.49.2.3 ClpLsqr::ClpLsqr (const ClpLsqr &)

Copy constructor.

4.49.2.4 ClpLsqr::~~ClpLsqr ()

Destructor.

4.49.3 Member Function Documentation

4.49.3.1 ClpLsqr& ClpLsqr::operator= (const ClpLsqr & rhs)

Assignment operator. This copies the data.

4.49.3.2 bool ClpLsqr::setParam (char * *parmName*, int *parmValue*)

Set an int parameter.

4.49.3.3 void ClpLsqr::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.

4.49.3.4 void ClpLsqr::matVecMult (int , CoinDenseVector< double > * , CoinDenseVector< double > *)

Matrix-vector multiply - implemented by user.

4.49.3.5 void ClpLsqr::matVecMult (int , CoinDenseVector< double > & , CoinDenseVector< double > &)**4.49.3.6 void ClpLsqr::borrowDiag1 (double * *array*) [inline]**

diag1 - we just borrow as it is part of a CoinDenseVector<double>

Definition at line 125 of file ClpLsqr.hpp.

4.49.4 Member Data Documentation**4.49.4.1 int ClpLsqr::rows_**

Row dimension of matrix.

Definition at line 86 of file ClpLsqr.hpp.

4.49.4.2 int ClpLsqr::ncols_

Column dimension of matrix.

Definition at line 88 of file ClpLsqr.hpp.

4.49.4.3 ClpInterior* ClpLsqr::model_

Pointer to Model object for this instance.

Definition at line 90 of file ClpLsqr.hpp.

4.49.4.4 double* ClpLsqr::diag1_

Diagonal array 1.

Definition at line 92 of file ClpLsqr.hpp.

4.49.4.5 double ClpLsqr::diag2_

Constant diagonal 2.

Definition at line 94 of file ClpLsqr.hpp.

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

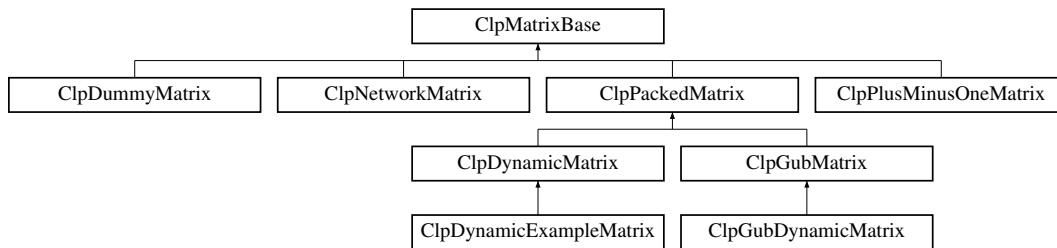
- [src/ClpLsqr.hpp](#)

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

Really 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 "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)
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 *x, double *y) const =0
*Return $y + A * x * \text{scalar}$ in y .*
- virtual void **times** (double scalar, const double *x, double *y, const double *rowScale, const double *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 *x, double *y) const =0
*Return $y + x * \text{scalar} * A$ 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 - default aborts for when scaling not supported (unless pointers NULL when as normal)
- virtual void **transposeTimes** (const **ClpSimplex** *model, double scalar, const CoinIndexedVector *x, CoinIndexedVector *y, CoinIndexedVector *z) const =0
*Return $x * \text{scalar} * A + y$ in z .*
- virtual void **subsetTransposeTimes** (const **ClpSimplex** *model, const CoinIndexedVector *x, const CoinIndexedVector *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 numberOfRows, 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 30 of file ClpMatrixBase.hpp.

4.50.2 Constructor & Destructor Documentation

4.50.2.1 ClpMatrixBase::ClpMatrixBase () [protected]

Default constructor.

4.50.2.2 virtual ClpMatrixBase::~~ClpMatrixBase () [virtual]

Destructor (has to be public)

4.50.2.3 ClpMatrixBase::ClpMatrixBase (const ClpMatrixBase &) [protected]

4.50.3 Member Function Documentation

4.50.3.1 virtual CoinPackedMatrix* ClpMatrixBase::getPackedMatrix () const [pure virtual]

Return a complete CoinPackedMatrix.

Implemented in [ClpPackedMatrix](#), [ClpDummyMatrix](#), [ClpNetworkMatrix](#), and [ClpPlusMinusOneMatrix](#).

4.50.3.2 virtual bool ClpMatrixBase::isColOrdered () const [pure virtual]

Whether the packed matrix is column major ordered or not.

Implemented in [ClpPackedMatrix](#), [ClpDummyMatrix](#), [ClpNetworkMatrix](#), and [ClpPlusMinusOneMatrix](#).

4.50.3.3 virtual CoinBigIndex ClpMatrixBase::getNumElements () const [pure virtual]

Number of entries in the packed matrix.

Implemented in [ClpPackedMatrix](#), [ClpDummyMatrix](#), [ClpNetworkMatrix](#), and [ClpPlusMinusOneMatrix](#).

4.50.3.4 virtual int ClpMatrixBase::getNumCols () const [pure virtual]

Number of columns.

Implemented in [ClpPackedMatrix](#), [ClpDummyMatrix](#), [ClpNetworkMatrix](#), and [ClpPlusMinusOneMatrix](#).

4.50.3.5 virtual int ClpMatrixBase::getNumRows () const [pure virtual]

Number of rows.

Implemented in [ClpPackedMatrix](#), [ClpDummyMatrix](#), [ClpNetworkMatrix](#), and [ClpPlusMinusOneMatrix](#).

4.50.3.6 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](#).

4.50.3.7 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](#).

4.50.3.8 `virtual const CoinBigIndex* ClpMatrixBase::getVectorStarts () const` [pure virtual]

Implemented in [ClpPackedMatrix](#), [ClpNetworkMatrix](#), [ClpDummyMatrix](#), and [ClpPlusMinusOneMatrix](#).

4.50.3.9 `virtual const int* ClpMatrixBase::getVectorLengths () const` [pure virtual]

The lengths of the major-dimension vectors.

Implemented in [ClpPackedMatrix](#), [ClpNetworkMatrix](#), [ClpDummyMatrix](#), and [ClpPlusMinusOneMatrix](#).

4.50.3.10 `virtual int ClpMatrixBase::getVectorLength (int index) const` [virtual]

The length of a single major-dimension vector.

Reimplemented in [ClpPackedMatrix](#).

4.50.3.11 `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](#).

4.50.3.12 `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](#).

4.50.3.13 `virtual void ClpMatrixBase::appendCols (int number, const CoinPackedVectorBase *const * columns)` [virtual]

Append Columns.

Reimplemented in [ClpPackedMatrix](#), [ClpNetworkMatrix](#), and [ClpPlusMinusOneMatrix](#).

4.50.3.14 `virtual void ClpMatrixBase::appendRows (int number, const CoinPackedVectorBase *const * rows)` [virtual]

Append Rows.

Reimplemented in [ClpPackedMatrix](#), [ClpNetworkMatrix](#), and [ClpPlusMinusOneMatrix](#).

4.50.3.15 `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` is true.

Reimplemented in [ClpPackedMatrix](#).

4.50.3.16 `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](#).

4.50.3.17 `virtual ClpMatrixBase* ClpMatrixBase::reverseOrderedCopy () const [inline],[virtual]`

Returns a new matrix in reverse order without gaps Is allowed to return NULL if doesn't want to have row copy.

Reimplemented in [ClpPackedMatrix](#), [ClpNetworkMatrix](#), [ClpPlusMinusOneMatrix](#), [ClpDummyMatrix](#), and [ClpGubMatrix](#).

Definition at line 88 of file `ClpMatrixBase.hpp`.

4.50.3.18 `virtual CoinBigIndex ClpMatrixBase::countBasis (const int * whichColumn, int & numberColumnBasic) [pure virtual]`

Returns number of elements in column part of basis.

Implemented in [ClpPackedMatrix](#), [ClpNetworkMatrix](#), [ClpPlusMinusOneMatrix](#), [ClpDummyMatrix](#), and [ClpGubMatrix](#).

4.50.3.19 `virtual void ClpMatrixBase::fillBasis (ClpSimplex * model, const int * whichColumn, int & numberColumnBasic, int * row, int * start, int * rowCount, int * columnCount, CoinFactorizationDouble * element) [pure virtual]`

Fills in column part of basis.

Implemented in [ClpPackedMatrix](#), [ClpNetworkMatrix](#), [ClpPlusMinusOneMatrix](#), [ClpDummyMatrix](#), and [ClpGubMatrix](#).

4.50.3.20 `virtual int ClpMatrixBase::scale (ClpModel *, const ClpSimplex * =NULL) const [inline],[virtual]`

Creates scales for column copy (rowCopy in model may be modified) default does not allow scaling returns non-zero if no scaling done.

Reimplemented in [ClpPackedMatrix](#).

Definition at line 105 of file `ClpMatrixBase.hpp`.

4.50.3.21 `virtual void ClpMatrixBase::scaleRowCopy (ClpModel *) const [inline],[virtual]`

Scales rowCopy if column copy scaled Only called if scales already exist.

Reimplemented in [ClpPackedMatrix](#).

Definition at line 110 of file `ClpMatrixBase.hpp`.

4.50.3.22 `virtual bool ClpMatrixBase::canGetRowCopy () const [inline],[virtual]`

Returns true if can create row copy.

Definition at line 112 of file `ClpMatrixBase.hpp`.

4.50.3.23 `virtual ClpMatrixBase* ClpMatrixBase::scaledColumnCopy (ClpModel *) const [inline],[virtual]`

Really really scales column copy Only called if scales already exist.

Up to user to delete

Reimplemented in [ClpPackedMatrix](#).

Definition at line 118 of file `ClpMatrixBase.hpp`.

4.50.3.24 `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 132 of file ClpMatrixBase.hpp.

4.50.3.25 `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](#).

4.50.3.26 `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](#).

4.50.3.27 `virtual void ClpMatrixBase::unpack (const ClpSimplex * model, CoinIndexedVector * rowArray, int column) const [pure virtual]`

Unpacks a column into an CoinIndexedvector.

Implemented in [ClpPackedMatrix](#), [ClpNetworkMatrix](#), [ClpPlusMinusOneMatrix](#), [ClpDummyMatrix](#), and [ClpGubMatrix](#).

4.50.3.28 `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](#).

4.50.3.29 `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 164 of file ClpMatrixBase.hpp.

4.50.3.30 `virtual void ClpMatrixBase::reallyScale (const double * rowScale, const double * columnScale) [virtual]`

Reimplemented in [ClpPackedMatrix](#).

4.50.3.31 `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](#).

4.50.3.32 `virtual void ClpMatrixBase::add (const ClpSimplex * model, CoinIndexedVector * rowArray, int column, double multiplier) const [pure virtual]`

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

Implemented in [ClpPackedMatrix](#), [ClpNetworkMatrix](#), [ClpPlusMinusOneMatrix](#), [ClpDummyMatrix](#), and [ClpGubMatrix](#).

4.50.3.33 `virtual void ClpMatrixBase::add (const ClpSimplex * model, double * array, int column, double multiplier) const [pure virtual]`

Adds multiple of a column into an array.

Implemented in [ClpPackedMatrix](#), [ClpNetworkMatrix](#), [ClpPlusMinusOneMatrix](#), [ClpDummyMatrix](#), and [ClpGubMatrix](#).

4.50.3.34 `virtual void ClpMatrixBase::releasePackedMatrix () const [pure virtual]`

Allow any parts of a created CoinPackedMatrix to be deleted.

Implemented in [ClpPackedMatrix](#), [ClpNetworkMatrix](#), [ClpPlusMinusOneMatrix](#), and [ClpDummyMatrix](#).

4.50.3.35 `virtual bool ClpMatrixBase::canDoPartialPricing () const [virtual]`

Says whether it can do partial pricing.

Reimplemented in [ClpPlusMinusOneMatrix](#), [ClpPackedMatrix](#), and [ClpNetworkMatrix](#).

4.50.3.36 `virtual int ClpMatrixBase::hiddenRows () const [virtual]`

Returns number of hidden rows e.g. gub.

Reimplemented in [ClpGubMatrix](#).

4.50.3.37 `virtual void ClpMatrixBase::partialPricing (ClpSimplex * model, double start, double end, int & bestSequence, int & numberWanted) [virtual]`

Partial pricing.

Reimplemented in [ClpPlusMinusOneMatrix](#), [ClpPackedMatrix](#), [ClpNetworkMatrix](#), [ClpGubMatrix](#), [ClpDynamicExampleMatrix](#), [ClpDynamicMatrix](#), and [ClpGubDynamicMatrix](#).

4.50.3.38 `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](#).

4.50.3.39 `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](#).

4.50.3.40 `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 "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 in [ClpGubMatrix](#), and [ClpDynamicMatrix](#).

4.50.3.41 `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](#).

4.50.3.42 `virtual int ClpMatrixBase::updatePivot (ClpSimplex * model, double oldInValue, double oldOutValue) [virtual]`

update information for a pivot (and effective rhs)

Reimplemented in [ClpGubMatrix](#), [ClpGubDynamicMatrix](#), and [ClpDynamicMatrix](#).

4.50.3.43 `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](#).

4.50.3.44 `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](#).

4.50.3.45 `double ClpMatrixBase::reducedCost (ClpSimplex * model, int sequence) const`

Returns reduced cost of a variable.

4.50.3.46 `virtual void ClpMatrixBase::correctSequence (const ClpSimplex * model, int & sequenceIn, int & sequenceOut) [virtual]`

Correct sequence in and out to give true value (if both -1 maybe do whole matrix)

Reimplemented in [ClpPackedMatrix](#), and [ClpGubMatrix](#).

4.50.3.47 `virtual void ClpMatrixBase::times (double scalar, const double * x, double * y) const` [pure virtual]

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

Precondition

x must be of size `numColumns()`

y must be of size `numRows()`

Implemented in [ClpPackedMatrix](#), [ClpPlusMinusOneMatrix](#), [ClpNetworkMatrix](#), [ClpDummyMatrix](#), [ClpGubDynamicMatrix](#), and [ClpDynamicMatrix](#).

4.50.3.48 `virtual void ClpMatrixBase::times (double scalar, const double * x, double * y, const double * rowScale, const double * columnScale) const` [virtual]

And for scaling - default aborts for when scaling not supported (unless pointers NULL when as normal)

Reimplemented in [ClpPackedMatrix](#), [ClpPlusMinusOneMatrix](#), [ClpNetworkMatrix](#), and [ClpDummyMatrix](#).

4.50.3.49 `virtual void ClpMatrixBase::transposeTimes (double scalar, const double * x, double * y) const` [pure virtual]

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

Precondition

x must be of size `numRows()`

y must be of size `numColumns()`

Implemented in [ClpPackedMatrix](#), [ClpPlusMinusOneMatrix](#), [ClpNetworkMatrix](#), and [ClpDummyMatrix](#).

4.50.3.50 `virtual void ClpMatrixBase::transposeTimes (double scalar, const double * x, double * y, const double * rowScale, const double * columnScale, double * sparse = NULL) const` [virtual]

And for scaling - default aborts for when scaling not supported (unless pointers NULL when as normal)

Reimplemented in [ClpPackedMatrix](#), [ClpPlusMinusOneMatrix](#), and [ClpNetworkMatrix](#).

4.50.3.51 `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](#).

4.50.3.52 `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](#).

4.50.3.53 `virtual bool ClpMatrixBase::canCombine (const ClpSimplex * , const CoinIndexedVector *) const` [inline], [virtual]

Returns true if can combine `transposeTimes` and `subsetTransposeTimes` and if it would be faster.

Reimplemented in [ClpPackedMatrix](#), and [ClpPlusMinusOneMatrix](#).

Definition at line 312 of file `ClpMatrixBase.hpp`.

```
4.50.3.54 virtual void ClpMatrixBase::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 ) [virtual]
```

Updates two arrays for steepest and does devex weights (need not be coded)

Reimplemented in [ClpPackedMatrix](#), and [ClpPlusMinusOneMatrix](#).

```
4.50.3.55 virtual void ClpMatrixBase::subsetTimes2 ( const ClpSimplex * model, CoinIndexedVector * dj1, const
CoinIndexedVector * pi2, CoinIndexedVector * dj2, double referenceIn, double devex, unsigned int * reference, double
* weights, double scaleFactor ) [virtual]
```

Updates second array for steepest and does devex weights (need not be coded)

Reimplemented in [ClpPackedMatrix](#), and [ClpPlusMinusOneMatrix](#).

```
4.50.3.56 virtual void ClpMatrixBase::listTransposeTimes ( const ClpSimplex * model, double * x, int * y, int number, double *
z ) const [virtual]
```

Return $x \cdot A$ in z but just for number indices in y .

Default cheats with fake `CoinIndexedVector` and then calls `subsetTransposeTimes`

```
4.50.3.57 virtual ClpMatrixBase* ClpMatrixBase::clone ( ) const [pure virtual]
```

Implemented in [ClpPackedMatrix](#), [ClpPlusMinusOneMatrix](#), [ClpGubMatrix](#), [ClpNetworkMatrix](#), [ClpDummyMatrix](#), [ClpDynamicMatrix](#), [ClpDynamicExampleMatrix](#), and [ClpGubDynamicMatrix](#).

```
4.50.3.58 virtual ClpMatrixBase* ClpMatrixBase::subsetClone ( int numberOfRows, 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](#).

```
4.50.3.59 virtual void ClpMatrixBase::backToBasics ( ) [inline],[virtual]
```

Gets rid of any mutable by products.

Definition at line 355 of file `ClpMatrixBase.hpp`.

```
4.50.3.60 int ClpMatrixBase::type ( ) const [inline]
```

Returns type.

The types which code may need to know about are: 1 - [ClpPackedMatrix](#) 11 - [ClpNetworkMatrix](#) 12 - [ClpPlusMinusOneMatrix](#)

Definition at line 362 of file `ClpMatrixBase.hpp`.

```
4.50.3.61 void ClpMatrixBase::setType ( int newtype ) [inline]
```

Sets type.

Definition at line 366 of file `ClpMatrixBase.hpp`.

4.50.3.62 void ClpMatrixBase::useEffectiveRhs (ClpSimplex * *model*)

Sets up an effective RHS.

4.50.3.63 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](#).

4.50.3.64 int ClpMatrixBase::lastRefresh () const [inline]

If rhsOffset used this is iteration last refreshed.

Definition at line 377 of file ClpMatrixBase.hpp.

4.50.3.65 int ClpMatrixBase::refreshFrequency () const [inline]

If rhsOffset used this is refresh frequency (0==off)

Definition at line 381 of file ClpMatrixBase.hpp.

4.50.3.66 void ClpMatrixBase::setRefreshFrequency (int *value*) [inline]

Definition at line 384 of file ClpMatrixBase.hpp.

4.50.3.67 bool ClpMatrixBase::skipDualCheck () const [inline]

whether to skip dual checks most of time

Definition at line 388 of file ClpMatrixBase.hpp.

4.50.3.68 void ClpMatrixBase::setSkipDualCheck (bool *yes*) [inline]

Definition at line 391 of file ClpMatrixBase.hpp.

4.50.3.69 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 396 of file ClpMatrixBase.hpp.

4.50.3.70 void ClpMatrixBase::setMinimumObjectsScan (int *value*) [inline]

Definition at line 399 of file ClpMatrixBase.hpp.

4.50.3.71 int ClpMatrixBase::minimumGoodReducedCosts () const [inline]

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

Definition at line 403 of file ClpMatrixBase.hpp.

4.50.3.72 void ClpMatrixBase::setMinimumGoodReducedCosts (int *value*) [inline]

Definition at line 406 of file ClpMatrixBase.hpp.

4.50.3.73 `double ClpMatrixBase::startFraction () const [inline]`

Current start of search space in matrix (as fraction)

Definition at line 410 of file ClpMatrixBase.hpp.

4.50.3.74 `void ClpMatrixBase::setStartFraction (double value) [inline]`

Definition at line 413 of file ClpMatrixBase.hpp.

4.50.3.75 `double ClpMatrixBase::endFraction () const [inline]`

Current end of search space in matrix (as fraction)

Definition at line 417 of file ClpMatrixBase.hpp.

4.50.3.76 `void ClpMatrixBase::setEndFraction (double value) [inline]`

Definition at line 420 of file ClpMatrixBase.hpp.

4.50.3.77 `double ClpMatrixBase::savedBestDj () const [inline]`

Current best reduced cost.

Definition at line 424 of file ClpMatrixBase.hpp.

4.50.3.78 `void ClpMatrixBase::setSavedBestDj (double value) [inline]`

Definition at line 427 of file ClpMatrixBase.hpp.

4.50.3.79 `int ClpMatrixBase::originalWanted () const [inline]`

Initial number of negative reduced costs wanted.

Definition at line 431 of file ClpMatrixBase.hpp.

4.50.3.80 `void ClpMatrixBase::setOriginalWanted (int value) [inline]`

Definition at line 434 of file ClpMatrixBase.hpp.

4.50.3.81 `int ClpMatrixBase::currentWanted () const [inline]`

Current number of negative reduced costs which we still need.

Definition at line 438 of file ClpMatrixBase.hpp.

4.50.3.82 `void ClpMatrixBase::setCurrentWanted (int value) [inline]`

Definition at line 441 of file ClpMatrixBase.hpp.

4.50.3.83 `int ClpMatrixBase::savedBestSequence () const [inline]`

Current best sequence.

Definition at line 445 of file ClpMatrixBase.hpp.

4.50.3.84 `void ClpMatrixBase::setSavedBestSequence (int value) [inline]`

Definition at line 448 of file ClpMatrixBase.hpp.

4.50.3.85 **ClpMatrixBase& ClpMatrixBase::operator= (const ClpMatrixBase &)** [protected]

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 480 of file ClpMatrixBase.hpp.

4.50.4.2 **double ClpMatrixBase::startFraction_** [protected]

Current start of search space in matrix (as fraction)

Definition at line 482 of file ClpMatrixBase.hpp.

4.50.4.3 **double ClpMatrixBase::endFraction_** [protected]

Current end of search space in matrix (as fraction)

Definition at line 484 of file ClpMatrixBase.hpp.

4.50.4.4 **double ClpMatrixBase::savedBestDj_** [protected]

Best reduced cost so far.

Definition at line 486 of file ClpMatrixBase.hpp.

4.50.4.5 **int ClpMatrixBase::originalWanted_** [protected]

Initial number of negative reduced costs wanted.

Definition at line 488 of file ClpMatrixBase.hpp.

4.50.4.6 **int ClpMatrixBase::currentWanted_** [protected]

Current number of negative reduced costs which we still need.

Definition at line 490 of file ClpMatrixBase.hpp.

4.50.4.7 **int ClpMatrixBase::savedBestSequence_** [protected]

Saved best sequence in pricing.

Definition at line 492 of file ClpMatrixBase.hpp.

4.50.4.8 **int ClpMatrixBase::type_** [protected]

type (may be useful)

Definition at line 494 of file ClpMatrixBase.hpp.

4.50.4.9 **int ClpMatrixBase::lastRefresh_** [protected]

If rhsOffset used this is iteration last refreshed.

Definition at line 496 of file ClpMatrixBase.hpp.

4.50.4.10 `int ClpMatrixBase::refreshFrequency_` [protected]

If rhsOffset used this is refresh frequency (0==off)

Definition at line 498 of file ClpMatrixBase.hpp.

4.50.4.11 `int ClpMatrixBase::minimumObjectsScan_` [protected]

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

Definition at line 500 of file ClpMatrixBase.hpp.

4.50.4.12 `int ClpMatrixBase::minimumGoodReducedCosts_` [protected]

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

Definition at line 502 of file ClpMatrixBase.hpp.

4.50.4.13 `int ClpMatrixBase::trueSequenceIn_` [protected]

True sequence in (i.e. from larger problem)

Definition at line 504 of file ClpMatrixBase.hpp.

4.50.4.14 `int ClpMatrixBase::trueSequenceOut_` [protected]

True sequence out (i.e. from larger problem)

Definition at line 506 of file ClpMatrixBase.hpp.

4.50.4.15 `bool ClpMatrixBase::skipDualCheck_` [protected]

whether to skip dual checks most of time

Definition at line 508 of file ClpMatrixBase.hpp.

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

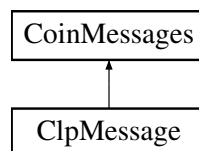
- [src/ClpMatrixBase.hpp](#)

4.51 ClpMessage Class Reference

This deals with Clp messages (as against Osi messages etc)

```
#include <ClpMessage.hpp>
```

Inheritance 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.

4.51.2 Constructor & Destructor Documentation

4.51.2.1 ClpMessage::ClpMessage (Language *language* = us_en)

Constructor.

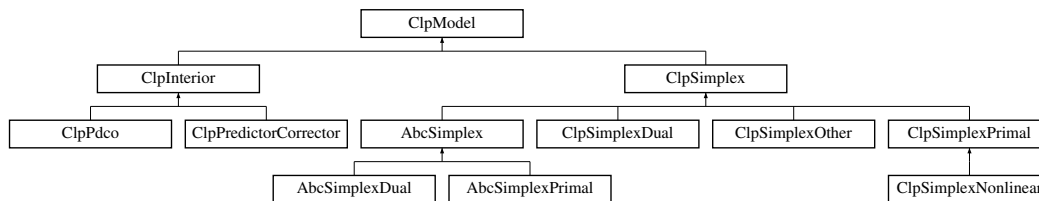
The documentation for this class was generated from the following file:

- [src/ClpMessage.hpp](#)

4.52 ClpModel Class Reference

```
#include <ClpModel.hpp>
```

Inheritance diagram for ClpModel:



Public Member Functions

- const double * [rowScale](#) () const
Scaling.
- const double * [columnScale](#) () const
- const double * [inverseRowScale](#) () const
- const double * [inverseColumnScale](#) () const
- double * [mutableRowScale](#) () const
- double * [mutableColumnScale](#) () const
- double * [mutableInverseRowScale](#) () const
- double * [mutableInverseColumnScale](#) () const
- double * [swapRowScale](#) (double *newScale)
- void [setRowScale](#) (double *scale)
- void [setColumnScale](#) (double *scale)
- double [objectiveScale](#) () const
Scaling of objective.
- void [setObjectiveScale](#) (double value)
- double [rhsScale](#) () const
Scaling of rhs and bounds.

- void [setRhsScale](#) (double value)
- 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 * [objective](#) (const double *solution, double &offset, bool refresh=true) const
- const double * [getObjCoefficients](#) () const
- double * [rowObjective](#) () const
Row Objective.
- const double * [getRowObjCoefficients](#) () const
- double * [columnLower](#) () const
Column Lower.
- const double * [getColLower](#) () const
- double * [columnUpper](#) () const
Column Upper.
- const double * [getColUpper](#) () const
- 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.
- void [setSmallElementValue](#) (double value)
- [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.
- void [setObjectiveValue](#) (double value)
- double [getObjValue](#) () const
- 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 * [unboundedRay](#) () const
- 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 * [getUserPointer](#) () const
- void [setTrustedUserPointer](#) (ClpTrustedData *pointer)
 - Trusted user pointer.*
- ClpTrustedData * [getTrustedUserPointer](#) () const
- int [whatsChanged](#) () const
 - What has changed in model (only for masochistic users)*
- void [setWhatsChanged](#) (int value)
- int [numberThreads](#) () const
 - Number of threads (not really being used)*
- void [setNumberThreads](#) (int value)

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_DBL_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 *whichColumns)
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 > &[columnNames](#))
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)
- 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 infeasibility proven?
- bool [isProvenDualInfeasible](#) () const
Is dual infeasibility 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 names)
 - void `setLengthNames` (int value)
length of names (0 means no names)
 - 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 Rnnnnnnnn.
 - 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 Cnnnnnnnn.
 - `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 infeasibilities.

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 $y + A * x * \text{scalar}$ in y.*
- void `transposeTimes` (double scalar, const double *x, double *y) const
*Return $y + x * \text{scalar} * A$ 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 [setDbiParam](#) ([ClpDbiParam](#) 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 [getDbiParam](#) ([ClpDbiParam](#) 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_](#) [[ClpLastDbiParam](#)]
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 (bool *emptyMessages* = false)

Default constructor.

4.52.2.2 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

4.52.2.3 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

4.52.2.4 ClpModel::~~ClpModel ()

Destructor.

4.52.3 Member Function Documentation

4.52.3.1 ClpModel& ClpModel::operator= (const ClpModel & *rhs*)

Assignment operator. This copies the data.

4.52.3.2 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

4.52.3.3 `void ClpModel::loadProblem (const CoinPackedMatrix & matrix, const double * collb, const double * colub, const double * obj, const double * rowlb, const double * rowub, const double * rowObjective = NULL)`

4.52.3.4 `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).

4.52.3.5 `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

4.52.3.6 `void ClpModel::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.

4.52.3.7 `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`

4.52.3.8 `void ClpModel::loadQuadraticObjective (const CoinPackedMatrix & matrix)`

4.52.3.9 `void ClpModel::deleteQuadraticObjective ()`

Get rid of quadratic objective.

4.52.3.10 `void ClpModel::setRowObjective (const double * rowObjective)`

This just loads up a row objective.

4.52.3.11 `int ClpModel::readMps (const char * filename, bool keepNames = false, bool ignoreErrors = false)`

Read an mps file from the given filename.

4.52.3.12 `int ClpModel::readGMPL (const char * filename, const char * dataName, bool keepNames = false)`

Read GMPL files from the given filenames.

4.52.3.13 `void ClpModel::copyInIntegerInformation (const char * information)`

Copy in integer informations.

4.52.3.14 `void ClpModel::deleteIntegerInformation ()`

Drop integer informations.

4.52.3.15 void ClpModel::setContinuous (int *index*)

Set the index-th variable to be a continuous variable.

4.52.3.16 void ClpModel::setInteger (int *index*)

Set the index-th variable to be an integer variable.

4.52.3.17 bool ClpModel::isInteger (int *index*) const

Return true if the index-th variable is an integer variable.

4.52.3.18 void ClpModel::resize (int *newNumberRows*, int *newNumberColumns*)

Resizes rim part of model.

4.52.3.19 void ClpModel::deleteRows (int *number*, const int * *which*)

Deletes rows.

4.52.3.20 void ClpModel::addRow (int *numberInRow*, const int * *columns*, const double * *elements*, double *rowLower* = -COIN_DBL_MAX, double *rowUpper* = COIN_DBL_MAX)

Add one row.

4.52.3.21 void ClpModel::addRows (int *number*, const double * *rowLower*, const double * *rowUpper*, const CoinBigIndex * *rowStarts*, const int * *columns*, const double * *elements*)

Add rows.

4.52.3.22 void ClpModel::addRows (int *number*, const double * *rowLower*, const double * *rowUpper*, const CoinBigIndex * *rowStarts*, const int * *rowLengths*, const int * *columns*, const double * *elements*)

Add rows.

4.52.3.23 void ClpModel::addRows (int *number*, const double * *rowLower*, const double * *rowUpper*, const CoinPackedVectorBase *const * *rows*)

4.52.3.24 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

4.52.3.25 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.

4.52.3.26 void ClpModel::deleteColumns (int *number*, const int * *which*)

Deletes columns.

4.52.3.27 void ClpModel::deleteRowsAndColumns (int *numberOfRows*, const int * *whichRows*, int *numberOfColumns*, const int * *whichColumns*)

Deletes rows AND columns (keeps old sizes)

4.52.3.28 void ClpModel::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.

4.52.3.29 void ClpModel::addColumnns (int *number*, const double * *columnLower*, const double * *columnUpper*, const double * *objective*, const CoinBigIndex * *columnStarts*, const int * *rows*, const double * *elements*)

Add columns.

4.52.3.30 void ClpModel::addColumnns (int *number*, const double * *columnLower*, const double * *columnUpper*, const double * *objective*, const CoinBigIndex * *columnStarts*, const int * *columnLengths*, const int * *rows*, const double * *elements*)

4.52.3.31 void ClpModel::addColumnns (int *number*, const double * *columnLower*, const double * *columnUpper*, const double * *objective*, const CoinPackedVectorBase *const * *columns*)

4.52.3.32 int ClpModel::addColumnns (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

4.52.3.33 int ClpModel::addColumnns (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.

4.52.3.34 void ClpModel::modifyCoefficient (int *row*, int *column*, double *newElement*, bool *keepZero* = false) [inline]

Modify one element of a matrix.

Definition at line 231 of file ClpModel.hpp.

4.52.3.35 void ClpModel::chgRowLower (const double * *rowLower*)

Change row lower bounds.

4.52.3.36 void ClpModel::chgRowUpper (const double * *rowUpper*)

Change row upper bounds.

4.52.3.37 void ClpModel::chgColumnLower (const double * *columnLower*)

Change column lower bounds.

4.52.3.38 void ClpModel::chgColumnUpper (const double * *columnUpper*)

Change column upper bounds.

4.52.3.39 void ClpModel::chgObjCoefficients (const double * *objIn*)

Change objective coefficients.

4.52.3.40 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

4.52.3.41 void ClpModel::returnModel (ClpModel & *otherModel*)

Return model - nulls all arrays so can be deleted safely also updates any scalars.

4.52.3.42 void ClpModel::createEmptyMatrix ()

Create empty [ClpPackedMatrix](#).

4.52.3.43 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](#)

4.52.3.44 void ClpModel::copy (const ClpMatrixBase * *from*, ClpMatrixBase *& *to*)

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

4.52.3.45 void ClpModel::dropNames ()

Drops names - makes *lengthnames* 0 and names empty.

4.52.3.46 void ClpModel::copyNames (const std::vector< std::string > & *rowNames*, const std::vector< std::string > & *columnNames*)

Copies in names.

4.52.3.47 void ClpModel::copyRowNames (const std::vector< std::string > & *rowNames*, int *first*, int *last*)

Copies in Row names - modifies names first .. last-1.

4.52.3.48 void ClpModel::copyColumnNames (const std::vector< std::string > & *columnNames*, int *first*, int *last*)

Copies in Column names - modifies names first .. last-1.

4.52.3.49 void ClpModel::copyRowNames (const char *const * *rowNames*, int *first*, int *last*)

Copies in Row names - modifies names first .. last-1.

4.52.3.50 void ClpModel::copyColumnNames (const char *const * *columnNames*, int *first*, int *last*)

Copies in Column names - modifies names first .. last-1.

4.52.3.51 void ClpModel::setRowName (int *rowIndex*, std::string & *name*)

Set name of row.

4.52.3.52 void ClpModel::setColumnName (int *colIndex*, std::string & *name*)

Set name of col.

4.52.3.53 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

4.52.3.54 CoinModel* ClpModel::createCoinModel () const

This creates a coinModel object.

4.52.3.55 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

4.52.3.56 int ClpModel::numberRows () const [inline]

Number of rows.

Definition at line 315 of file ClpModel.hpp.

4.52.3.57 int ClpModel::getNumRows () const [inline]

Definition at line 318 of file ClpModel.hpp.

4.52.3.58 int ClpModel::getNumCols () const [inline]

Number of columns.

Definition at line 322 of file ClpModel.hpp.

4.52.3.59 int ClpModel::numberColumns () const [inline]

Definition at line 325 of file ClpModel.hpp.

4.52.3.60 double ClpModel::primalTolerance () const [inline]

Primal tolerance to use.

Definition at line 329 of file ClpModel.hpp.

4.52.3.61 void ClpModel::setPrimalTolerance (double *value*)

4.52.3.62 double ClpModel::dualTolerance () const [inline]

Dual tolerance to use.

Definition at line 334 of file ClpModel.hpp.

4.52.3.63 void ClpModel::setDualTolerance (double *value*)

4.52.3.64 double ClpModel::primalObjectiveLimit () const [inline]

Primal objective limit.

Definition at line 339 of file ClpModel.hpp.

4.52.3.65 void ClpModel::setPrimalObjectiveLimit (double *value*)

4.52.3.66 double ClpModel::dualObjectiveLimit () const [inline]

Dual objective limit.

Definition at line 344 of file ClpModel.hpp.

4.52.3.67 void ClpModel::setDualObjectiveLimit (double *value*)

4.52.3.68 double ClpModel::objectiveOffset () const [inline]

Objective offset.

Definition at line 349 of file ClpModel.hpp.

4.52.3.69 void ClpModel::setObjectiveOffset (double *value*)

4.52.3.70 double ClpModel::presolveTolerance () const [inline]

Presolve tolerance to use.

Definition at line 354 of file ClpModel.hpp.

4.52.3.71 const std::string& ClpModel::problemName () const [inline]

Definition at line 358 of file ClpModel.hpp.

4.52.3.72 int ClpModel::numberIterations () const [inline]

Number of iterations.

Definition at line 363 of file ClpModel.hpp.

4.52.3.73 int ClpModel::getIterationCount () const [inline]

Definition at line 366 of file ClpModel.hpp.

4.52.3.74 void ClpModel::setNumberIterations (int *numberIterationsNew*) [inline]

Definition at line 369 of file ClpModel.hpp.

4.52.3.75 `int ClpModel::solveType () const [inline]`

Solve type - 1 simplex, 2 simplex interface, 3 Interior.

Definition at line 373 of file ClpModel.hpp.

4.52.3.76 `void ClpModel::setSolveType (int type) [inline]`

Definition at line 376 of file ClpModel.hpp.

4.52.3.77 `int ClpModel::maximumIterations () const [inline]`

Maximum number of iterations.

Definition at line 380 of file ClpModel.hpp.

4.52.3.78 `void ClpModel::setMaximumIterations (int value)`

4.52.3.79 `double ClpModel::maximumSeconds () const [inline]`

Maximum time in seconds (from when set called)

Definition at line 385 of file ClpModel.hpp.

4.52.3.80 `void ClpModel::setMaximumSeconds (double value)`

4.52.3.81 `bool ClpModel::hitMaximumIterations () const`

Returns true if hit maximum iterations (or time)

4.52.3.82 `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 400 of file ClpModel.hpp.

4.52.3.83 `int ClpModel::problemStatus () const [inline]`

Definition at line 403 of file ClpModel.hpp.

4.52.3.84 `void ClpModel::setProblemStatus (int problemStatusNew) [inline]`

Set problem status.

Definition at line 407 of file ClpModel.hpp.

4.52.3.85 `int ClpModel::secondaryStatus () const [inline]`

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](#).

Definition at line 425 of file ClpModel.hpp.

4.52.3.86 void ClpModel::setSecondaryStatus (int *newstatus*) [inline]

Definition at line 428 of file ClpModel.hpp.

4.52.3.87 bool ClpModel::isAbandoned () const [inline]

Are there a numerical difficulties?

Definition at line 432 of file ClpModel.hpp.

4.52.3.88 bool ClpModel::isProvenOptimal () const [inline]

Is optimality proven?

Definition at line 436 of file ClpModel.hpp.

4.52.3.89 bool ClpModel::isProvenPrimalInfeasible () const [inline]

Is primal infeasibility proven?

Definition at line 440 of file ClpModel.hpp.

4.52.3.90 bool ClpModel::isProvenDualInfeasible () const [inline]

Is dual infeasibility proven?

Definition at line 444 of file ClpModel.hpp.

4.52.3.91 bool ClpModel::isPrimalObjectiveLimitReached () const

Is the given primal objective limit reached?

4.52.3.92 bool ClpModel::isDualObjectiveLimitReached () const

Is the given dual objective limit reached?

4.52.3.93 bool ClpModel::isIterationLimitReached () const [inline]

Iteration limit reached?

Definition at line 452 of file ClpModel.hpp.

4.52.3.94 double ClpModel::optimizationDirection () const [inline]

Direction of optimization (1 - minimize, -1 - maximize, 0 - ignore.

Definition at line 456 of file ClpModel.hpp.

4.52.3.95 double ClpModel::getObjSense () const [inline]

Definition at line 459 of file ClpModel.hpp.

4.52.3.96 void ClpModel::setOptimizationDirection (double *value*)

4.52.3.97 double* ClpModel::primalRowSolution () const [inline]

Primal row solution.

Definition at line 464 of file ClpModel.hpp.

4.52.3.98 `const double* ClpModel::getRowActivity () const` `[inline]`

Definition at line 467 of file ClpModel.hpp.

4.52.3.99 `double* ClpModel::primalColumnSolution () const` `[inline]`

Primal column solution.

Definition at line 471 of file ClpModel.hpp.

4.52.3.100 `const double* ClpModel::getColSolution () const` `[inline]`

Definition at line 474 of file ClpModel.hpp.

4.52.3.101 `void ClpModel::setColSolution (const double * input)` `[inline]`

Definition at line 477 of file ClpModel.hpp.

4.52.3.102 `double* ClpModel::dualRowSolution () const` `[inline]`

Dual row solution.

Definition at line 481 of file ClpModel.hpp.

4.52.3.103 `const double* ClpModel::getRowPrice () const` `[inline]`

Definition at line 484 of file ClpModel.hpp.

4.52.3.104 `double* ClpModel::dualColumnSolution () const` `[inline]`

Reduced costs.

Definition at line 488 of file ClpModel.hpp.

4.52.3.105 `const double* ClpModel::getReducedCost () const` `[inline]`

Definition at line 491 of file ClpModel.hpp.

4.52.3.106 `double* ClpModel::rowLower () const` `[inline]`

Row lower.

Definition at line 495 of file ClpModel.hpp.

4.52.3.107 `const double* ClpModel::getRowLower () const` `[inline]`

Definition at line 498 of file ClpModel.hpp.

4.52.3.108 `double* ClpModel::rowUpper () const` `[inline]`

Row upper.

Definition at line 502 of file ClpModel.hpp.

4.52.3.109 `const double* ClpModel::getRowUpper () const` `[inline]`

Definition at line 505 of file ClpModel.hpp.

4.52.3.110 `void ClpModel::setObjectiveCoefficient (int elementIndex, double elementValue)`

Set an objective function coefficient.

4.52.3.111 `void ClpModel::setObjCoeff (int elementIndex, double elementValue)` `[inline]`

Set an objective function coefficient.

Definition at line 514 of file ClpModel.hpp.

4.52.3.112 `void ClpModel::setColumnLower (int elementIndex, double elementValue)`

Set a single column lower bound

Use -DBL_MAX for -infinity.

4.52.3.113 `void ClpModel::setColumnUpper (int elementIndex, double elementValue)`

Set a single column upper bound

Use DBL_MAX for infinity.

4.52.3.114 `void ClpModel::setColumnBounds (int elementIndex, double lower, double upper)`

Set a single column lower and upper bound.

4.52.3.115 `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

<i>indexFirst, index-Last</i>	pointers to the beginning and after the end of the array of the indices of the variables whose <i>either</i> bound changes
<i>boundList</i>	the new lower/upper bound pairs for the variables

4.52.3.116 `void ClpModel::setColLower (int elementIndex, double elementValue)` `[inline]`

Set a single column lower bound

Use -DBL_MAX for -infinity.

Definition at line 544 of file ClpModel.hpp.

4.52.3.117 `void ClpModel::setColUpper (int elementIndex, double elementValue)` `[inline]`

Set a single column upper bound

Use DBL_MAX for infinity.

Definition at line 549 of file ClpModel.hpp.

4.52.3.118 `void ClpModel::setColBounds (int elementIndex, double lower, double upper)` `[inline]`

Set a single column lower and upper bound.

Definition at line 554 of file ClpModel.hpp.

4.52.3.119 `void ClpModel::setColSetBounds (const int * indexFirst, const int * indexLast, const double * boundList)`
`[inline]`

Set the bounds on a number of columns simultaneously

Parameters

<i>indexFirst, index-Last</i>	pointers to the beginning and after the end of the array of the indices of the variables whose <i>either</i> bound changes
<i>boundList</i>	the new lower/upper bound pairs for the variables

Definition at line 565 of file ClpModel.hpp.

4.52.3.120 void ClpModel::setRowLower (int *elementIndex*, double *elementValue*)

Set a single row lower bound

Use -DBL_MAX for -infinity.

4.52.3.121 void ClpModel::setRowUpper (int *elementIndex*, double *elementValue*)

Set a single row upper bound

Use DBL_MAX for infinity.

4.52.3.122 void ClpModel::setRowBounds (int *elementIndex*, double *lower*, double *upper*)

Set a single row lower and upper bound.

4.52.3.123 void ClpModel::setRowSetBounds (const int * *indexFirst*, const int * *indexLast*, const double * *boundList*)

Set the bounds on a number of rows simultaneously

Parameters

<i>indexFirst, index-Last</i>	pointers to the beginning and after the end of the array of the indices of the constraints whose <i>either</i> bound changes
<i>boundList</i>	the new lower/upper bound pairs for the constraints

4.52.3.124 const double* ClpModel::rowScale () const [inline]

Scaling.

Definition at line 595 of file ClpModel.hpp.

4.52.3.125 const double* ClpModel::columnScale () const [inline]

Definition at line 598 of file ClpModel.hpp.

4.52.3.126 const double* ClpModel::inverseRowScale () const [inline]

Definition at line 601 of file ClpModel.hpp.

4.52.3.127 const double* ClpModel::inverseColumnScale () const [inline]

Definition at line 604 of file ClpModel.hpp.

4.52.3.128 double* ClpModel::mutableRowScale () const [inline]

Definition at line 607 of file ClpModel.hpp.

4.52.3.129 double* ClpModel::mutableColumnScale () const [inline]

Definition at line 610 of file ClpModel.hpp.

4.52.3.130 `double* ClpModel::mutableInverseRowScale () const [inline]`

Definition at line 613 of file ClpModel.hpp.

4.52.3.131 `double* ClpModel::mutableInverseColumnScale () const [inline]`

Definition at line 616 of file ClpModel.hpp.

4.52.3.132 `double* ClpModel::swapRowScale (double * newScale) [inline]`

Definition at line 619 of file ClpModel.hpp.

4.52.3.133 `void ClpModel::setRowScale (double * scale)`

4.52.3.134 `void ClpModel::setColumnScale (double * scale)`

4.52.3.135 `double ClpModel::objectiveScale () const [inline]`

Scaling of objective.

Definition at line 627 of file ClpModel.hpp.

4.52.3.136 `void ClpModel::setObjectiveScale (double value) [inline]`

Definition at line 630 of file ClpModel.hpp.

4.52.3.137 `double ClpModel::rhsScale () const [inline]`

Scaling of rhs and bounds.

Definition at line 634 of file ClpModel.hpp.

4.52.3.138 `void ClpModel::setRhsScale (double value) [inline]`

Definition at line 637 of file ClpModel.hpp.

4.52.3.139 `void ClpModel::scaling (int mode = 1)`

Sets or unsets scaling, 0 -off, 1 equilibrium, 2 geometric, 3 auto, 4 auto-but-as-initialSolve-in-bab.

4.52.3.140 `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

4.52.3.141 `int ClpModel::scalingFlag () const [inline]`

Gets scalingFlag.

Definition at line 646 of file ClpModel.hpp.

4.52.3.142 `double* ClpModel::objective () const [inline]`

Objective.

Definition at line 650 of file ClpModel.hpp.

4.52.3.143 `double* ClpModel::objective (const double * solution, double & offset, bool refresh = true) const` [inline]

Definition at line 658 of file ClpModel.hpp.

4.52.3.144 `const double* ClpModel::getObjCoefficients () const` [inline]

Definition at line 666 of file ClpModel.hpp.

4.52.3.145 `double* ClpModel::rowObjective () const` [inline]

Row Objective.

Definition at line 675 of file ClpModel.hpp.

4.52.3.146 `const double* ClpModel::getRowObjCoefficients () const` [inline]

Definition at line 678 of file ClpModel.hpp.

4.52.3.147 `double* ClpModel::columnLower () const` [inline]

Column Lower.

Definition at line 682 of file ClpModel.hpp.

4.52.3.148 `const double* ClpModel::getColLower () const` [inline]

Definition at line 685 of file ClpModel.hpp.

4.52.3.149 `double* ClpModel::columnUpper () const` [inline]

Column Upper.

Definition at line 689 of file ClpModel.hpp.

4.52.3.150 `const double* ClpModel::getColUpper () const` [inline]

Definition at line 692 of file ClpModel.hpp.

4.52.3.151 `CoinPackedMatrix* ClpModel::matrix () const` [inline]

Matrix (if not ClpPackedmatrix be careful about memory leak.

Definition at line 696 of file ClpModel.hpp.

4.52.3.152 `int ClpModel::getNumElements () const` [inline]

Number of elements in matrix.

Definition at line 701 of file ClpModel.hpp.

4.52.3.153 `double ClpModel::getSmallElementValue () const` [inline]

Small element value - elements less than this set to zero, default is 1.0e-20.

Definition at line 706 of file ClpModel.hpp.

4.52.3.154 `void ClpModel::setSmallElementValue (double value)` [inline]

Definition at line 709 of file ClpModel.hpp.

4.52.3.155 `ClpMatrixBase* ClpModel::rowCopy () const` [inline]

Row Matrix.

Definition at line 713 of file ClpModel.hpp.

4.52.3.156 `void ClpModel::setNewRowCopy (ClpMatrixBase * newCopy)`

Set new row matrix.

4.52.3.157 `ClpMatrixBase* ClpModel::clpMatrix () const` [inline]

Clp Matrix.

Definition at line 719 of file ClpModel.hpp.

4.52.3.158 `ClpPackedMatrix* ClpModel::clpScaledMatrix () const` [inline]

Scaled [ClpPackedMatrix](#).

Definition at line 723 of file ClpModel.hpp.

4.52.3.159 `void ClpModel::setClpScaledMatrix (ClpPackedMatrix * scaledMatrix)` [inline]

Sets pointer to scaled [ClpPackedMatrix](#).

Definition at line 727 of file ClpModel.hpp.

4.52.3.160 `ClpPackedMatrix* ClpModel::swapScaledMatrix (ClpPackedMatrix * scaledMatrix)` [inline]

Swaps pointer to scaled [ClpPackedMatrix](#).

Definition at line 732 of file ClpModel.hpp.

4.52.3.161 `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.

4.52.3.162 `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](#). [ClpModel](#) takes ownership.

Definition at line 748 of file ClpModel.hpp.

4.52.3.163 `double ClpModel::objectiveValue () const` [inline]

Objective value.

Definition at line 753 of file ClpModel.hpp.

4.52.3.164 `void ClpModel::setObjectiveValue (double value)` [inline]

Definition at line 756 of file ClpModel.hpp.

4.52.3.165 `double ClpModel::getObjValue () const` [inline]

Definition at line 759 of file ClpModel.hpp.

4.52.3.166 `char* ClpModel::integerInformation () const [inline]`

Integer information.

Definition at line 763 of file ClpModel.hpp.

4.52.3.167 `double* ClpModel::infeasibilityRay (bool fullRay = false) const`

Infeasibility/unbounded ray (NULL returned if none/wrong) Up to user to use delete [] on these arrays.

4.52.3.168 `double* ClpModel::unboundedRay () const`

4.52.3.169 `double* ClpModel::ray () const [inline]`

For advanced users - no need to delete - sign not changed.

Definition at line 771 of file ClpModel.hpp.

4.52.3.170 `bool ClpModel::rayExists () const [inline]`

just test if infeasibility or unbounded Ray exists

Definition at line 774 of file ClpModel.hpp.

4.52.3.171 `void ClpModel::deleteRay () [inline]`

just delete ray if exists

Definition at line 778 of file ClpModel.hpp.

4.52.3.172 `const double* ClpModel::internalRay () const [inline]`

Access internal ray storage. Users should call [infeasibilityRay\(\)](#) or [unboundedRay\(\)](#) instead.

Definition at line 783 of file ClpModel.hpp.

4.52.3.173 `bool ClpModel::statusExists () const [inline]`

See if status (i.e. basis) array exists (partly for OsiClp)

Definition at line 787 of file ClpModel.hpp.

4.52.3.174 `unsigned char* ClpModel::statusArray () const [inline]`

Return address of status (i.e. basis) array (char[numberRows+numberColumns])

Definition at line 791 of file ClpModel.hpp.

4.52.3.175 `unsigned char* ClpModel::statusCopy () const`

Return copy of status (i.e.

basis) array (char[numberRows+numberColumns]), use delete []

4.52.3.176 `void ClpModel::copyinStatus (const unsigned char * statusArray)`

Copy in status (basis) vector.

4.52.3.177 `void ClpModel::setUserPointer (void * pointer) [inline]`

User pointer for whatever reason.

Definition at line 801 of file ClpModel.hpp.

4.52.3.178 `void* ClpModel::getUserPointer () const [inline]`

Definition at line 804 of file ClpModel.hpp.

4.52.3.179 `void ClpModel::setTrustedUserPointer (ClpTrustedData * pointer) [inline]`

Trusted user pointer.

Definition at line 808 of file ClpModel.hpp.

4.52.3.180 `ClpTrustedData* ClpModel::getTrustedUserPointer () const [inline]`

Definition at line 811 of file ClpModel.hpp.

4.52.3.181 `int ClpModel::whatsChanged () const [inline]`

What has changed in model (only for masochistic users)

Definition at line 815 of file ClpModel.hpp.

4.52.3.182 `void ClpModel::setWhatsChanged (int value) [inline]`

Definition at line 818 of file ClpModel.hpp.

4.52.3.183 `int ClpModel::numberThreads () const [inline]`

Number of threads (not really being used)

Definition at line 822 of file ClpModel.hpp.

4.52.3.184 `void ClpModel::setNumberThreads (int value) [inline]`

Definition at line 825 of file ClpModel.hpp.

4.52.3.185 `void ClpModel::passInMessageHandler (CoinMessageHandler * handler)`

Pass in Message handler (not deleted at end)

4.52.3.186 `CoinMessageHandler* ClpModel::pushMessageHandler (CoinMessageHandler * handler, bool & oldDefault)`

Pass in Message handler (not deleted at end) and return current.

4.52.3.187 `void ClpModel::popMessageHandler (CoinMessageHandler * oldHandler, bool oldDefault)`

back to previous message handler

4.52.3.188 `void ClpModel::newLanguage (CoinMessages::Language language)`

Set language.

4.52.3.189 `void ClpModel::setLanguage (CoinMessages::Language language) [inline]`

Definition at line 840 of file ClpModel.hpp.

4.52.3.190 `void ClpModel::setDefaultMessageHandler ()`

Overrides message handler with a default one.

4.52.3.191 `CoinMessageHandler* ClpModel::messageHandler () const` `[inline]`

Return handler.

Definition at line 846 of file ClpModel.hpp.

4.52.3.192 `CoinMessages ClpModel::messages () const` `[inline]`

Return messages.

Definition at line 850 of file ClpModel.hpp.

4.52.3.193 `CoinMessages* ClpModel::messagesPointer ()` `[inline]`

Return pointer to messages.

Definition at line 854 of file ClpModel.hpp.

4.52.3.194 `CoinMessages ClpModel::coinMessages () const` `[inline]`

Return Coin messages.

Definition at line 858 of file ClpModel.hpp.

4.52.3.195 `CoinMessages* ClpModel::coinMessagesPointer ()` `[inline]`

Return pointer to Coin messages.

Definition at line 862 of file ClpModel.hpp.

4.52.3.196 `void ClpModel::setLogLevel(int value)` `[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.

Definition at line 873 of file ClpModel.hpp.

4.52.3.197 `int ClpModel::logLevel () const` `[inline]`

Definition at line 876 of file ClpModel.hpp.

4.52.3.198 `bool ClpModel::defaultHandler () const` `[inline]`

Return true if default handler.

Definition at line 880 of file ClpModel.hpp.

4.52.3.199 `void ClpModel::passInEventHandler (const ClpEventHandler* eventHandler)`

Pass in Event handler (cloned and deleted at end)

4.52.3.200 `ClpEventHandler* ClpModel::eventHandler () const` `[inline]`

Event handler.

Definition at line 886 of file ClpModel.hpp.

4.52.3.201 `CoinThreadRandom* ClpModel::randomNumberGenerator ()` `[inline]`

Thread specific random number generator.

Definition at line 890 of file ClpModel.hpp.

4.52.3.202 `CoinThreadRandom& ClpModel::mutableRandomNumberGenerator () [inline]`

Thread specific random number generator.

Definition at line 894 of file ClpModel.hpp.

4.52.3.203 `void ClpModel::setRandomSeed (int value) [inline]`

Set seed for thread specific random number generator.

Definition at line 898 of file ClpModel.hpp.

4.52.3.204 `int ClpModel::lengthNames () const [inline]`

length of names (0 means no names)

Definition at line 902 of file ClpModel.hpp.

4.52.3.205 `void ClpModel::setLengthNames (int value) [inline]`

length of names (0 means no names)

Definition at line 907 of file ClpModel.hpp.

4.52.3.206 `const std::vector<std::string>* ClpModel::rowNames () const [inline]`

Row names.

Definition at line 911 of file ClpModel.hpp.

4.52.3.207 `const std::string& ClpModel::rowName (int iRow) const [inline]`

Definition at line 914 of file ClpModel.hpp.

4.52.3.208 `std::string ClpModel::getRowName (int iRow) const`

Return name or Rnnnnnnnn.

4.52.3.209 `const std::vector<std::string>* ClpModel::columnNames () const [inline]`

Column names.

Definition at line 920 of file ClpModel.hpp.

4.52.3.210 `const std::string& ClpModel::columnName (int iColumn) const [inline]`

Definition at line 923 of file ClpModel.hpp.

4.52.3.211 `std::string ClpModel::getColumnName (int iColumn) const`

Return name or Cnnnnnnnn.

4.52.3.212 `ClpObjective* ClpModel::objectiveAsObject () const [inline]`

Objective methods.

Definition at line 930 of file ClpModel.hpp.

4.52.3.213 `void ClpModel::setObjective (ClpObjective * objective)`

4.52.3.214 `void ClpModel::setObjectivePointer (ClpObjective * newobjective) [inline]`

Definition at line 934 of file ClpModel.hpp.

4.52.3.215 `int ClpModel::emptyProblem (int * infeasNumber = NULL, double * infeasSum = NULL, bool printMessage = true)`

Solve a problem with no elements - return status and dual and primal infeasibilities.

4.52.3.216 `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()`

4.52.3.217 `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()`

4.52.3.218 `bool ClpModel::setIntParam (ClpIntParam key, int value)`

Set an integer parameter.

4.52.3.219 `bool ClpModel::setDbiParam (ClpDbiParam key, double value)`

Set an double parameter.

4.52.3.220 `bool ClpModel::setStrParam (ClpStrParam key, const std::string & value)`

Set an string parameter.

4.52.3.221 `bool ClpModel::getIntParam (ClpIntParam key, int & value) const [inline]`

Definition at line 988 of file ClpModel.hpp.

4.52.3.222 `bool ClpModel::getDbiParam (ClpDbiParam key, double & value) const [inline]`

Definition at line 997 of file ClpModel.hpp.

4.52.3.223 `bool ClpModel::getStrParam (ClpStrParam key, std::string & value) const [inline]`

Definition at line 1007 of file ClpModel.hpp.

4.52.3.224 `void ClpModel::generateCpp (FILE * fp)`

Create C++ lines to get to current state.

4.52.3.225 `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 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 1052 of file ClpModel.hpp.

4.52.3.226 void ClpModel::setSpecialOptions (unsigned int *value*)

4.52.3.227 bool ClpModel::inCbcBranchAndBound () const [inline]

Definition at line 1057 of file ClpModel.hpp.

4.52.3.228 void ClpModel::gutsOfDelete (int *type*) [protected]

Does most of deletion (0 = all, 1 = most)

4.52.3.229 void ClpModel::gutsOfCopy (const ClpModel & *rhs*, int *trueCopy* = 1) [protected]

Does most of copying If trueCopy 0 then just points to arrays If -1 leaves as much as possible.

4.52.3.230 void ClpModel::getRowBound (int *iRow*, double & *lower*, double & *upper*) const [protected]

gets lower and upper bounds on rows

4.52.3.231 void ClpModel::gutsOfLoadModel (int *numberRows*, int *numberColumns*, const double * *collb*, const double * *colub*, const double * *obj*, const double * *rowlb*, const double * *rowub*, const double * *rowObjective* = NULL) [protected]

puts in format I like - 4 array matrix - may make row copy

4.52.3.232 void ClpModel::gutsOfScaling () [protected]

Does much of scaling.

4.52.3.233 double ClpModel::rawObjectiveValue () const [inline], [protected]

Objective value - always minimize.

Definition at line 1082 of file ClpModel.hpp.

4.52.3.234 bool ClpModel::permanentArrays () const [inline], [protected]

If we are using maximumRows_ and Columns_.

Definition at line 1086 of file ClpModel.hpp.

4.52.3.235 void ClpModel::startPermanentArrays () [protected]

Start using maximumRows_ and Columns_.

4.52.3.236 void ClpModel::stopPermanentArrays () [protected]

Stop using maximumRows_ and Columns_.

4.52.3.237 const char* const* ClpModel::rowNamesAsChar () const [protected]

Create row names as char **.

4.52.3.238 const char* const* ClpModel::columnNamesAsChar () const [protected]

Create column names as char **.

4.52.3.239 void ClpModel::deleteNamesAsChar (const char *const * *names*, int *number*) const [protected]

Delete char * version of names.

4.52.3.240 void ClpModel::onStopped () [protected]

On stopped - sets secondary status.

4.52.4 Member Data Documentation

4.52.4.1 double ClpModel::optimizationDirection_ [protected]

Direction of optimization (1 - minimize, -1 - maximize, 0 - ignore.

Definition at line 1110 of file ClpModel.hpp.

4.52.4.2 double ClpModel::dbiParam_[ClpLastDbiParam] [protected]

Array of double parameters.

Definition at line 1112 of file ClpModel.hpp.

4.52.4.3 double ClpModel::objectiveValue_ [protected]

Objective value.

Definition at line 1114 of file ClpModel.hpp.

4.52.4.4 double ClpModel::smallElement_ [protected]

Small element value.

Definition at line 1116 of file ClpModel.hpp.

4.52.4.5 double ClpModel::objectiveScale_ [protected]

Scaling of objective.

Definition at line 1118 of file ClpModel.hpp.

4.52.4.6 double ClpModel::rhsScale_ [protected]

Scaling of rhs and bounds.

Definition at line 1120 of file ClpModel.hpp.

4.52.4.7 `int ClpModel::numberOfRows_` [protected]

Number of rows.

Definition at line 1122 of file ClpModel.hpp.

4.52.4.8 `int ClpModel::numberOfColumns_` [protected]

Number of columns.

Definition at line 1124 of file ClpModel.hpp.

4.52.4.9 `double* ClpModel::rowActivity_` [protected]

Row activities.

Definition at line 1126 of file ClpModel.hpp.

4.52.4.10 `double* ClpModel::columnActivity_` [protected]

Column activities.

Definition at line 1128 of file ClpModel.hpp.

4.52.4.11 `double* ClpModel::dual_` [protected]

Duals.

Definition at line 1130 of file ClpModel.hpp.

4.52.4.12 `double* ClpModel::reducedCost_` [protected]

Reduced costs.

Definition at line 1132 of file ClpModel.hpp.

4.52.4.13 `double* ClpModel::rowLower_` [protected]

Row lower.

Definition at line 1134 of file ClpModel.hpp.

4.52.4.14 `double* ClpModel::rowUpper_` [protected]

Row upper.

Definition at line 1136 of file ClpModel.hpp.

4.52.4.15 `ClpObjective* ClpModel::objective_` [protected]

Objective.

Definition at line 1138 of file ClpModel.hpp.

4.52.4.16 `double* ClpModel::rowObjective_` [protected]

Row Objective (? sign) - may be NULL.

Definition at line 1140 of file ClpModel.hpp.

4.52.4.17 `double* ClpModel::columnLower_` [protected]

Column Lower.

Definition at line 1142 of file ClpModel.hpp.

4.52.4.18 `double* ClpModel::columnUpper_` [protected]

Column Upper.

Definition at line 1144 of file ClpModel.hpp.

4.52.4.19 `ClpMatrixBase* ClpModel::matrix_` [protected]

Packed matrix.

Definition at line 1146 of file ClpModel.hpp.

4.52.4.20 `ClpMatrixBase* ClpModel::rowCopy_` [protected]

Row copy if wanted.

Definition at line 1148 of file ClpModel.hpp.

4.52.4.21 `ClpPackedMatrix* ClpModel::scaledMatrix_` [protected]

Scaled packed matrix.

Definition at line 1150 of file ClpModel.hpp.

4.52.4.22 `double* ClpModel::ray_` [protected]

Infeasible/unbounded ray.

Definition at line 1152 of file ClpModel.hpp.

4.52.4.23 `double* ClpModel::rowScale_` [protected]

Row scale factors for matrix.

Definition at line 1154 of file ClpModel.hpp.

4.52.4.24 `double* ClpModel::columnScale_` [protected]

Column scale factors.

Definition at line 1156 of file ClpModel.hpp.

4.52.4.25 `double* ClpModel::inverseRowScale_` [protected]

Inverse row scale factors for matrix (end of rowScale_)

Definition at line 1158 of file ClpModel.hpp.

4.52.4.26 `double* ClpModel::inverseColumnScale_` [protected]

Inverse column scale factors for matrix (end of columnScale_)

Definition at line 1160 of file ClpModel.hpp.

4.52.4.27 int ClpModel::scalingFlag_ [protected]

Scale flag, 0 none, 1 equilibrium, 2 geometric, 3, auto, 4 dynamic, 5 geometric on rows.

Definition at line 1163 of file ClpModel.hpp.

4.52.4.28 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 1171 of file ClpModel.hpp.

4.52.4.29 char* ClpModel::integerType_ [protected]

Integer information.

Definition at line 1173 of file ClpModel.hpp.

4.52.4.30 void* ClpModel::userPointer_ [protected]

User pointer for whatever reason.

Definition at line 1175 of file ClpModel.hpp.

4.52.4.31 ClpTrustedData* ClpModel::trustedUserPointer_ [protected]

Trusted user pointer e.g. for heuristics.

Definition at line 1177 of file ClpModel.hpp.

4.52.4.32 int ClpModel::intParam_[ClpLastIntParam] [protected]

Array of integer parameters.

Definition at line 1179 of file ClpModel.hpp.

4.52.4.33 int ClpModel::numberIterations_ [protected]

Number of iterations.

Definition at line 1181 of file ClpModel.hpp.

4.52.4.34 int ClpModel::solveType_ [protected]

Solve type - 1 simplex, 2 simplex interface, 3 Interior.

Definition at line 1183 of file ClpModel.hpp.

4.52.4.35 unsigned int ClpModel::whatsChanged_ [protected]

Definition at line 1212 of file ClpModel.hpp.

4.52.4.36 int ClpModel::problemStatus_ [protected]

Status of problem.

Definition at line 1214 of file ClpModel.hpp.

4.52.4.37 `int ClpModel::secondaryStatus_` [protected]

Secondary status of problem.

Definition at line 1216 of file ClpModel.hpp.

4.52.4.38 `int ClpModel::lengthNames_` [protected]

length of names (0 means no names)

Definition at line 1218 of file ClpModel.hpp.

4.52.4.39 `int ClpModel::numberThreads_` [protected]

Number of threads (not very operational)

Definition at line 1220 of file ClpModel.hpp.

4.52.4.40 `unsigned int ClpModel::specialOptions_` [protected]

For advanced options See get and set for meaning.

Definition at line 1224 of file ClpModel.hpp.

4.52.4.41 `CoinMessageHandler* ClpModel::handler_` [protected]

Message handler.

Definition at line 1226 of file ClpModel.hpp.

4.52.4.42 `bool ClpModel::defaultHandler_` [protected]

Flag to say if default handler (so delete)

Definition at line 1228 of file ClpModel.hpp.

4.52.4.43 `CoinThreadRandom ClpModel::randomNumberGenerator_` [protected]

Thread specific random number generator.

Definition at line 1230 of file ClpModel.hpp.

4.52.4.44 `ClpEventHandler* ClpModel::eventHandler_` [protected]

Event handler.

Definition at line 1232 of file ClpModel.hpp.

4.52.4.45 `std::vector<std::string> ClpModel::rowNames_` [protected]

Row names.

Definition at line 1235 of file ClpModel.hpp.

4.52.4.46 `std::vector<std::string> ClpModel::columnNames_` [protected]

Column names.

Definition at line 1237 of file ClpModel.hpp.

4.52.4.47 CoinMessages ClpModel::messages_ [protected]

Messages.

Definition at line 1240 of file ClpModel.hpp.

4.52.4.48 CoinMessages ClpModel::coinMessages_ [protected]

Coin messages.

Definition at line 1242 of file ClpModel.hpp.

4.52.4.49 int ClpModel::maximumColumns_ [protected]

Maximum number of columns in model.

Definition at line 1244 of file ClpModel.hpp.

4.52.4.50 int ClpModel::maximumRows_ [protected]

Maximum number of rows in model.

Definition at line 1246 of file ClpModel.hpp.

4.52.4.51 int ClpModel::maximumInternalColumns_ [protected]

Maximum number of columns (internal arrays) in model.

Definition at line 1248 of file ClpModel.hpp.

4.52.4.52 int ClpModel::maximumInternalRows_ [protected]

Maximum number of rows (internal arrays) in model.

Definition at line 1250 of file ClpModel.hpp.

4.52.4.53 CoinPackedMatrix ClpModel::baseMatrix_ [protected]

Base packed matrix.

Definition at line 1252 of file ClpModel.hpp.

4.52.4.54 CoinPackedMatrix ClpModel::baseRowCopy_ [protected]

Base row copy.

Definition at line 1254 of file ClpModel.hpp.

4.52.4.55 double* ClpModel::savedRowScale_ [protected]

Saved row scale factors for matrix.

Definition at line 1256 of file ClpModel.hpp.

4.52.4.56 double* ClpModel::savedColumnScale_ [protected]

Saved column scale factors.

Definition at line 1258 of file ClpModel.hpp.

4.52.4.57 `std::string ClpModel::strParam_[ClpLastStrParam]` [protected]

Array of string parameters.

Definition at line 1261 of file ClpModel.hpp.

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

- [src/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 numberOfRows, 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 rowsBasic[], int columnsBasic[])
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 pivotRow)
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 Constructor & Destructor Documentation

4.53.2.1 ClpNetworkBasis::ClpNetworkBasis ()

Default constructor.

4.53.2.2 ClpNetworkBasis::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.

4.53.2.3 ClpNetworkBasis::ClpNetworkBasis (const ClpNetworkBasis & *other*)

Copy constructor.

4.53.2.4 ClpNetworkBasis::~~ClpNetworkBasis ()

Destructor.

4.53.3 Member Function Documentation

4.53.3.1 ClpNetworkBasis& ClpNetworkBasis::operator= (const ClpNetworkBasis & *other*)

= copy

4.53.3.2 int ClpNetworkBasis::factorize (const ClpMatrixBase * *matrix*, int *rowsBasic*[], int *columnsBasic*[])

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

4.53.3.3 int ClpNetworkBasis::replaceColumn (CoinIndexedVector * *column*, int *pivotRow*)

Replaces one Column to basis, returns 0=OK, 1=Probably OK, 2=singular!!

4.53.3.4 double ClpNetworkBasis::updateColumn (CoinIndexedVector * *regionSparse*, CoinIndexedVector * *regionSparse2*, int *pivotRow*)

Updates one column (FTRAN) from region, Returns pivot value if "pivotRow" >=0.

4.53.3.5 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)

4.53.3.6 `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

4.53.3.7 `int ClpNetworkBasis::updateColumnTranspose (CoinIndexedVector * regionSparse, CoinIndexedVector * regionSparse2) const`

Updates one column (BTRAN) from region2.

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

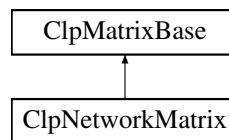
- [src/ClpNetworkBasis.hpp](#)

4.54 ClpNetworkMatrix Class Reference

This implements a simple network matrix as derived from [ClpMatrixBase](#).

```
#include <ClpNetworkMatrix.hpp>
```

Inheritance 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 &numberWanted)
- 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 *columnScale) 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 `<code>x *A</code> in <code>z</code> 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.

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 (int numberColumns, const int * head, const int * tail)`

Constructor from two arrays.

4.54.2.3 `virtual ClpNetworkMatrix::~ClpNetworkMatrix () [virtual]`

Destructor.

4.54.2.4 `ClpNetworkMatrix::ClpNetworkMatrix (const ClpNetworkMatrix &)`

The copy constructor.

4.54.2.5 `ClpNetworkMatrix::ClpNetworkMatrix (const CoinPackedMatrix &)`

The copy constructor from an CoinNetworkMatrix.

4.54.2.6 `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 CoinPackedMatrix* ClpNetworkMatrix::getPackedMatrix () const [virtual]`

Return a complete CoinPackedMatrix.

Implements [ClpMatrixBase](#).

4.54.3.2 `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.

4.54.3.3 `virtual CoinBigIndex ClpNetworkMatrix::getNumElements () const [inline],[virtual]`

Number of entries in the packed matrix.

Implements [ClpMatrixBase](#).

Definition at line 31 of file ClpNetworkMatrix.hpp.

4.54.3.4 `virtual int ClpNetworkMatrix::getNumCols () const [inline],[virtual]`

Number of columns.

Implements [ClpMatrixBase](#).

Definition at line 35 of file ClpNetworkMatrix.hpp.

4.54.3.5 `virtual int ClpNetworkMatrix::getNumRows () const [inline],[virtual]`

Number of rows.

Implements [ClpMatrixBase](#).

Definition at line 39 of file `ClpNetworkMatrix.hpp`.

4.54.3.6 `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](#).

4.54.3.7 `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`.

4.54.3.8 `virtual const CoinBigIndex* ClpNetworkMatrix::getVectorStarts () const [virtual]`

Implements [ClpMatrixBase](#).

4.54.3.9 `virtual const int* ClpNetworkMatrix::getVectorLengths () const [virtual]`

The lengths of the major-dimension vectors.

Implements [ClpMatrixBase](#).

4.54.3.10 `virtual void ClpNetworkMatrix::deleteCols (const int numDel, const int * indDel) [virtual]`

Delete the columns whose indices are listed in `indDel`.

Implements [ClpMatrixBase](#).

4.54.3.11 `virtual void ClpNetworkMatrix::deleteRows (const int numDel, const int * indDel) [virtual]`

Delete the rows whose indices are listed in `indDel`.

Implements [ClpMatrixBase](#).

4.54.3.12 `virtual void ClpNetworkMatrix::appendCols (int number, const CoinPackedVectorBase *const * columns) [virtual]`

Append Columns.

Reimplemented from [ClpMatrixBase](#).

4.54.3.13 `virtual void ClpNetworkMatrix::appendRows (int number, const CoinPackedVectorBase *const * rows) [virtual]`

Append Rows.

Reimplemented from [ClpMatrixBase](#).

4.54.3.14 `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](#).

4.54.3.15 `virtual ClpMatrixBase* ClpNetworkMatrix::reverseOrderedCopy () const [virtual]`

Returns a new matrix in reverse order without gaps.

Reimplemented from [ClpMatrixBase](#).

4.54.3.16 `virtual CoinBigIndex ClpNetworkMatrix::countBasis (const int * whichColumn, int & numberColumnBasic) [virtual]`

Returns number of elements in column part of basis.

Implements [ClpMatrixBase](#).

4.54.3.17 `virtual void ClpNetworkMatrix::fillBasis (ClpSimplex * model, const int * whichColumn, int & numberColumnBasic, int * row, int * start, int * rowCount, int * columnCount, CoinFactorizationDouble * element) [virtual]`

Fills in column part of basis.

Implements [ClpMatrixBase](#).

4.54.3.18 `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](#).

4.54.3.19 `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](#).

4.54.3.20 `virtual void ClpNetworkMatrix::unpack (const ClpSimplex * model, CoinIndexedVector * rowArray, int column) const [virtual]`

Unpacks a column into an CoinIndexedvector.

Implements [ClpMatrixBase](#).

4.54.3.21 `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](#).

4.54.3.22 `virtual void ClpNetworkMatrix::add (const ClpSimplex * model, CoinIndexedVector * rowArray, int column, double multiplier) const [virtual]`

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

Implements [ClpMatrixBase](#).

4.54.3.23 `virtual void ClpNetworkMatrix::add (const ClpSimplex * model, double * array, int column, double multiplier) const [virtual]`

Adds multiple of a column into an array.

Implements [ClpMatrixBase](#).

4.54.3.24 `virtual void ClpNetworkMatrix::releasePackedMatrix () const [virtual]`

Allow any parts of a created CoinMatrix to be deleted.

Implements [ClpMatrixBase](#).

4.54.3.25 `virtual bool ClpNetworkMatrix::canDoPartialPricing () const [virtual]`

Says whether it can do partial pricing.

Reimplemented from [ClpMatrixBase](#).

4.54.3.26 `virtual void ClpNetworkMatrix::partialPricing (ClpSimplex * model, double start, double end, int & bestSequence, int & numberWanted) [virtual]`

Partial pricing.

Reimplemented from [ClpMatrixBase](#).

4.54.3.27 `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()`

Implements [ClpMatrixBase](#).

4.54.3.28 `virtual void ClpNetworkMatrix::times (double scalar, const double * x, double * y, const double * rowScale, const double * columnScale) const [virtual]`

And for scaling.

Reimplemented from [ClpMatrixBase](#).

4.54.3.29 `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()`

Implements [ClpMatrixBase](#).

4.54.3.30 `virtual void ClpNetworkMatrix::transposeTimes (double scalar, const double * x, double * y, const double * rowScale, const double * columnScale, double * sparse = NULL) const` [virtual]

And for scaling.

Reimplemented from [ClpMatrixBase](#).

4.54.3.31 `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](#).

4.54.3.32 `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](#).

4.54.3.33 `bool ClpNetworkMatrix::trueNetwork () const` [inline]

Return true if really network, false if has slacks.

Definition at line 170 of file `ClpNetworkMatrix.hpp`.

4.54.3.34 `ClpNetworkMatrix& ClpNetworkMatrix::operator= (const ClpNetworkMatrix &)`

4.54.3.35 `virtual ClpMatrixBase* ClpNetworkMatrix::clone () const` [virtual]

Clone.

Implements [ClpMatrixBase](#).

4.54.3.36 `virtual ClpMatrixBase* ClpNetworkMatrix::subsetClone (int numberOfRows, const int * whichRows, int numberOfColumns, const int * whichColumns) const` [virtual]

Subset clone (without gaps).

Duplicates are allowed and order is as given

Reimplemented from [ClpMatrixBase](#).

4.54.4 Member Data Documentation

4.54.4.1 `CoinPackedMatrix* ClpNetworkMatrix::matrix_` [mutable], [protected]

For fake `CoinPackedMatrix`.

Definition at line 215 of file `ClpNetworkMatrix.hpp`.

4.54.4.2 `int* ClpNetworkMatrix::lengths_` [mutable], [protected]

Definition at line 216 of file `ClpNetworkMatrix.hpp`.

4.54.4.3 `int* ClpNetworkMatrix::indices_` [protected]

Data -1, then +1 rows in pairs (row==-1 if one entry)

Definition at line 218 of file `ClpNetworkMatrix.hpp`.

4.54.4.4 `int ClpNetworkMatrix::numberOfRows_` [protected]

Number of rows.

Definition at line 220 of file `ClpNetworkMatrix.hpp`.

4.54.4.5 `int ClpNetworkMatrix::numberOfColumns_` [protected]

Number of columns.

Definition at line 222 of file `ClpNetworkMatrix.hpp`.

4.54.4.6 `bool ClpNetworkMatrix::trueNetwork_` [protected]

True if all entries have two elements.

Definition at line 224 of file `ClpNetworkMatrix.hpp`.

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

- [src/ClpNetworkMatrix.hpp](#)

4.55 ClpNode Class Reference

```
#include <ClpNode.hpp>
```

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 `numberOfInfeasibilities` () 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 (ClpSimplex * model, const ClpNodeStuff * stuff, int depth)`

Constructor from model.

4.55.2.3 `virtual ClpNode::~~ClpNode () [virtual]`

Destructor.

4.55.2.4 `ClpNode::ClpNode (const ClpNode &)`

The copy constructor.

4.55.3 Member Function Documentation

4.55.3.1 `void ClpNode::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.

4.55.3.2 `void ClpNode::chooseVariable (ClpSimplex * model, ClpNodeStuff * info)`

Choose a new variable.

4.55.3.3 `int ClpNode::fixOnReducedCosts (ClpSimplex * model)`

Fix on reduced costs.

4.55.3.4 `void ClpNode::createArrays (ClpSimplex * model)`

Create odd arrays.

4.55.3.5 `void ClpNode::cleanUpForCrunch ()`

Clean up as crunch is different model.

4.55.3.6 `double ClpNode::objectiveValue () const [inline]`

Objective value.

Definition at line 43 of file `ClpNode.hpp`.

4.55.3.7 `void ClpNode::setObjectiveValue (double value) [inline]`

Set objective value.

Definition at line 47 of file `ClpNode.hpp`.

4.55.3.8 `const double* ClpNode::primalSolution () const [inline]`

Primal solution.

Definition at line 51 of file `ClpNode.hpp`.

4.55.3.9 `const double* ClpNode::dualSolution () const` `[inline]`

Dual solution.

Definition at line 55 of file ClpNode.hpp.

4.55.3.10 `double ClpNode::branchingValue () const` `[inline]`

Initial value of integer variable.

Definition at line 59 of file ClpNode.hpp.

4.55.3.11 `double ClpNode::sumInfeasibilities () const` `[inline]`

Sum infeasibilities.

Definition at line 63 of file ClpNode.hpp.

4.55.3.12 `int ClpNode::numberInfeasibilities () const` `[inline]`

Number infeasibilities.

Definition at line 67 of file ClpNode.hpp.

4.55.3.13 `int ClpNode::depth () const` `[inline]`

Relative depth.

Definition at line 71 of file ClpNode.hpp.

4.55.3.14 `double ClpNode::estimatedSolution () const` `[inline]`

Estimated solution value.

Definition at line 75 of file ClpNode.hpp.

4.55.3.15 `int ClpNode::way () const`

Way for integer variable -1 down , +1 up.

4.55.3.16 `bool ClpNode::fathomed () const`

Return true if branch exhausted.

4.55.3.17 `void ClpNode::changeState ()`

Change state of variable i.e. go other way.

4.55.3.18 `int ClpNode::sequence () const` `[inline]`

Sequence number of integer variable (-1 if none)

Definition at line 85 of file ClpNode.hpp.

4.55.3.19 `bool ClpNode::oddArraysExist () const` `[inline]`

If odd arrays exist.

Definition at line 89 of file ClpNode.hpp.

4.55.3.20 `const unsigned char* ClpNode::statusArray () const [inline]`

Status array.

Definition at line 93 of file ClpNode.hpp.

4.55.3.21 `void ClpNode::gutsOfConstructor (ClpSimplex * model, const ClpNodeStuff * stuff, int arraysExist, int depth)`

Does work of constructor (partly so gdb will work)

4.55.3.22 `ClpNode& ClpNode::operator= (const ClpNode &)`

Operator =.

4.55.4 Member Data Documentation

4.55.4.1 `double ClpNode::branchingValue_ [protected]`

Initial value of integer variable.

Definition at line 129 of file ClpNode.hpp.

4.55.4.2 `double ClpNode::objectiveValue_ [protected]`

Value of objective.

Definition at line 131 of file ClpNode.hpp.

4.55.4.3 `double ClpNode::sumInfeasibilities_ [protected]`

Sum of infeasibilities.

Definition at line 133 of file ClpNode.hpp.

4.55.4.4 `double ClpNode::estimatedSolution_ [protected]`

Estimated solution value.

Definition at line 135 of file ClpNode.hpp.

4.55.4.5 `ClpFactorization* ClpNode::factorization_ [protected]`

Factorization.

Definition at line 137 of file ClpNode.hpp.

4.55.4.6 `ClpDualRowSteepest* ClpNode::weights_ [protected]`

Steepest edge weights.

Definition at line 139 of file ClpNode.hpp.

4.55.4.7 `unsigned char* ClpNode::status_ [protected]`

Status vector.

Definition at line 141 of file ClpNode.hpp.

4.55.4.8 `double* ClpNode::primalSolution_ [protected]`

Primal solution.

Definition at line 143 of file ClpNode.hpp.

4.55.4.9 `double* ClpNode::dualSolution_ [protected]`

Dual solution.

Definition at line 145 of file ClpNode.hpp.

4.55.4.10 `int* ClpNode::lower_ [protected]`

Integer lower bounds (only used in fathomMany)

Definition at line 147 of file ClpNode.hpp.

4.55.4.11 `int* ClpNode::upper_ [protected]`

Integer upper bounds (only used in fathomMany)

Definition at line 149 of file ClpNode.hpp.

4.55.4.12 `int* ClpNode::pivotVariables_ [protected]`

Pivot variables for factorization.

Definition at line 151 of file ClpNode.hpp.

4.55.4.13 `int* ClpNode::fixed_ [protected]`

Variables fixed by reduced costs (at end of branch) 0x10000000 added if fixed to UB.

Definition at line 153 of file ClpNode.hpp.

4.55.4.14 `branchState ClpNode::branchState_ [protected]`

State of branch.

Definition at line 155 of file ClpNode.hpp.

4.55.4.15 `int ClpNode::sequence_ [protected]`

Sequence number of integer variable (-1 if none)

Definition at line 157 of file ClpNode.hpp.

4.55.4.16 `int ClpNode::numberInfeasibilities_ [protected]`

Number of infeasibilities.

Definition at line 159 of file ClpNode.hpp.

4.55.4.17 `int ClpNode::depth_ [protected]`

Relative depth.

Definition at line 161 of file ClpNode.hpp.

4.55.4.18 int ClpNode::numberFixed_ [protected]

Number fixed by reduced cost.

Definition at line 163 of file ClpNode.hpp.

4.55.4.19 int ClpNode::flags_ [protected]

Flags - 1 duals scaled.

Definition at line 165 of file ClpNode.hpp.

4.55.4.20 int ClpNode::maximumFixed_ [protected]

Maximum number fixed by reduced cost.

Definition at line 167 of file ClpNode.hpp.

4.55.4.21 int ClpNode::maximumRows_ [protected]

Maximum rows so far.

Definition at line 169 of file ClpNode.hpp.

4.55.4.22 int ClpNode::maximumColumns_ [protected]

Maximum columns so far.

Definition at line 171 of file ClpNode.hpp.

4.55.4.23 int ClpNode::maximumIntegers_ [protected]

Maximum Integers so far.

Definition at line 173 of file ClpNode.hpp.

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

- [src/ClpNode.hpp](#)

4.56 ClpNodeStuff Class Reference

```
#include <ClpNode.hpp>
```

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 virtual ClpNodeStuff::~~ClpNodeStuff () [virtual]

Destructor.

4.56.2.3 ClpNodeStuff::ClpNodeStuff (const ClpNodeStuff &)

The copy constructor.

4.56.3 Member Function Documentation

4.56.3.1 ClpNodeStuff& ClpNodeStuff::operator= (const ClpNodeStuff &)

Operator =.

4.56.3.2 void ClpNodeStuff::zap (int type)

Zaps stuff 1 - arrays, 2 ints, 3 both.

4.56.3.3 void ClpNodeStuff::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.

4.56.3.4 void ClpNodeStuff::update (int *way*, int *sequence*, double *change*, bool *feasible*)

Update pseudo costs.

4.56.3.5 int ClpNodeStuff::maximumNodes () const

Return maximum number of nodes.

4.56.3.6 int ClpNodeStuff::maximumSpace () const

Return maximum space for nodes.

4.56.4 Member Data Documentation

4.56.4.1 double ClpNodeStuff::integerTolerance_

Integer tolerance.

Definition at line 218 of file ClpNode.hpp.

4.56.4.2 double ClpNodeStuff::integerIncrement_

Integer increment.

Definition at line 220 of file ClpNode.hpp.

4.56.4.3 double ClpNodeStuff::smallChange_

Small change in branch.

Definition at line 222 of file ClpNode.hpp.

4.56.4.4 double* ClpNodeStuff::downPseudo_

Down pseudo costs.

Definition at line 224 of file ClpNode.hpp.

4.56.4.5 double* ClpNodeStuff::upPseudo_

Up pseudo costs.

Definition at line 226 of file ClpNode.hpp.

4.56.4.6 int* ClpNodeStuff::priority_

Priority.

Definition at line 228 of file ClpNode.hpp.

4.56.4.7 int* ClpNodeStuff::numberDown_

Number of times down.

Definition at line 230 of file ClpNode.hpp.

4.56.4.8 int* ClpNodeStuff::numberUp_

Number of times up.

Definition at line 232 of file ClpNode.hpp.

4.56.4.9 int* ClpNodeStuff::numberDownInfeasible_

Number of times down infeasible.

Definition at line 234 of file ClpNode.hpp.

4.56.4.10 int* ClpNodeStuff::numberUpInfeasible_

Number of times up infeasible.

Definition at line 236 of file ClpNode.hpp.

4.56.4.11 double* ClpNodeStuff::saveCosts_

Copy of costs (local)

Definition at line 238 of file ClpNode.hpp.

4.56.4.12 ClpNode ClpNodeStuff::nodeInfo_**

Array of ClpNodes.

Definition at line 240 of file ClpNode.hpp.

4.56.4.13 ClpSimplex* ClpNodeStuff::large_

Large model if crunched.

Definition at line 242 of file ClpNode.hpp.

4.56.4.14 int* ClpNodeStuff::whichRow_

Which rows in large model.

Definition at line 244 of file ClpNode.hpp.

4.56.4.15 int* ClpNodeStuff::whichColumn_

Which columns in large model.

Definition at line 246 of file ClpNode.hpp.

4.56.4.16 CoinMessageHandler* ClpNodeStuff::handler_

Cbc's message handler.

Definition at line 249 of file ClpNode.hpp.

4.56.4.17 int ClpNodeStuff::nBound_

Number bounds in large model.

Definition at line 252 of file ClpNode.hpp.

4.56.4.18 int ClpNodeStuff::saveOptions_

Save of specialOptions_ (local)

Definition at line 254 of file ClpNode.hpp.

4.56.4.19 int ClpNodeStuff::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.

Definition at line 263 of file ClpNode.hpp.

4.56.4.20 int ClpNodeStuff::maximumNodes_

Maximum number of nodes to do.

Definition at line 265 of file ClpNode.hpp.

4.56.4.21 int ClpNodeStuff::numberBeforeTrust_

Number before trust from CbcModel.

Definition at line 267 of file ClpNode.hpp.

4.56.4.22 int ClpNodeStuff::stateOfSearch_

State of search from CbcModel.

Definition at line 269 of file ClpNode.hpp.

4.56.4.23 int ClpNodeStuff::nDepth_

Number deep.

Definition at line 271 of file ClpNode.hpp.

4.56.4.24 int ClpNodeStuff::nNodes_

Number nodes returned (-1 if fathom aborted)

Definition at line 273 of file ClpNode.hpp.

4.56.4.25 int ClpNodeStuff::numberNodesExplored_

Number of nodes explored.

Definition at line 275 of file ClpNode.hpp.

4.56.4.26 int ClpNodeStuff::numberIterations_

Number of iterations.

Definition at line 277 of file ClpNode.hpp.

4.56.4.27 int ClpNodeStuff::presolveType_

Type of presolve - 0 none, 1 crunch.

Definition at line 279 of file ClpNode.hpp.

4.56.4.28 int ClpNodeStuff::startingDepth_

Depth passed in.

Definition at line 282 of file ClpNode.hpp.

4.56.4.29 int ClpNodeStuff::nodeCalled_

Node at which called.

Definition at line 284 of file ClpNode.hpp.

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

- [src/ClpNode.hpp](#)

4.57 ClpNonLinearCost Class Reference

```
#include <ClpNonLinearCost.hpp>
```

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 costValue=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 ()

Default constructor.

4.57.2.2 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 .

4.57.2.3 ClpNonLinearCost::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 \leq upper (if $=$ then cost ignored) This could obviously be changed to make more user friendly.

4.57.2.4 ClpNonLinearCost::~~ClpNonLinearCost ()

Destructor.

4.57.2.5 ClpNonLinearCost::ClpNonLinearCost (const ClpNonLinearCost &)

4.57.3 Member Function Documentation

4.57.3.1 ClpNonLinearCost& ClpNonLinearCost::operator= (const ClpNonLinearCost &)

4.57.3.2 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 \leq oldTolerance away

4.57.3.3 void ClpNonLinearCost::checkInfeasibilities (int *numberInArray*, const int * *index*)

Changes infeasible costs for each variable The indices are row indices and need converting to sequences.

4.57.3.4 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

4.57.3.5 void ClpNonLinearCost::goThru (int *numberInArray*, double *multiplier*, const int * *index*, const double * *work*, double * *rhs*)

Goes through one bound for each variable.

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

4.57.3.6 void ClpNonLinearCost::goBack (int *numberInArray*, const int * *index*, double * *rhs*)

Takes off last iteration (i.e.
offsets closer to 0)

4.57.3.7 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

4.57.3.8 void ClpNonLinearCost::zapCosts ()

Temporary zeroing of feasible costs.

4.57.3.9 void ClpNonLinearCost::refreshCosts (const double * *columnCosts*)

Refreshes costs always makes row costs zero.

4.57.3.10 void ClpNonLinearCost::feasibleBounds ()

Puts feasible bounds into lower and upper.

4.57.3.11 void ClpNonLinearCost::refresh ()

Refresh - assuming regions OK.

4.57.3.12 double ClpNonLinearCost::setOne (int *sequence*, double *solutionValue*)

Sets bounds and cost for one variable Returns change in cost May need to be inline for speed.

4.57.3.13 void ClpNonLinearCost::setOne (int *sequence*, double *solutionValue*, double *lowerValue*, double *upperValue*, double *costValue* = 0.0)

Sets bounds and infeasible cost and true cost for one variable This is for gub and column generation etc.

4.57.3.14 int ClpNonLinearCost::setOneOutgoing (int *sequence*, double & *solutionValue*)

Sets bounds and cost for outgoing variable may change value Returns direction.

4.57.3.15 double ClpNonLinearCost::nearest (int *sequence*, double *solutionValue*)

Returns nearest bound.

4.57.3.16 double ClpNonLinearCost::changeInCost (int *sequence*, double *alpha*) const [inline]

Returns change in cost - one down if $\alpha > 0.0$, up if $\alpha < 0.0$ Value is current - new.

Definition at line 171 of file ClpNonLinearCost.hpp.

4.57.3.17 double ClpNonLinearCost::changeUpInCost (int *sequence*) const [inline]

Definition at line 185 of file ClpNonLinearCost.hpp.

4.57.3.18 double ClpNonLinearCost::changeDownInCost (int *sequence*) const [inline]

Definition at line 199 of file ClpNonLinearCost.hpp.

4.57.3.19 `double ClpNonLinearCost::changeInCost (int sequence, double alpha, double & rhs)` `[inline]`

This also updates next bound.

Definition at line 214 of file ClpNonLinearCost.hpp.

4.57.3.20 `double ClpNonLinearCost::lower (int sequence) const` `[inline]`

Returns current lower bound.

Definition at line 273 of file ClpNonLinearCost.hpp.

4.57.3.21 `double ClpNonLinearCost::upper (int sequence) const` `[inline]`

Returns current upper bound.

Definition at line 277 of file ClpNonLinearCost.hpp.

4.57.3.22 `double ClpNonLinearCost::cost (int sequence) const` `[inline]`

Returns current cost.

Definition at line 281 of file ClpNonLinearCost.hpp.

4.57.3.23 `int ClpNonLinearCost::numberInfeasibilities () const` `[inline]`

Number of infeasibilities.

Definition at line 290 of file ClpNonLinearCost.hpp.

4.57.3.24 `double ClpNonLinearCost::changeInCost () const` `[inline]`

Change in cost.

Definition at line 294 of file ClpNonLinearCost.hpp.

4.57.3.25 `double ClpNonLinearCost::feasibleCost () const` `[inline]`

Feasible cost.

Definition at line 298 of file ClpNonLinearCost.hpp.

4.57.3.26 `double ClpNonLinearCost::feasibleReportCost () const`

Feasible cost with offset and direction (i.e. for reporting)

4.57.3.27 `double ClpNonLinearCost::sumInfeasibilities () const` `[inline]`

Sum of infeasibilities.

Definition at line 304 of file ClpNonLinearCost.hpp.

4.57.3.28 `double ClpNonLinearCost::largestInfeasibility () const` `[inline]`

Largest infeasibility.

Definition at line 308 of file ClpNonLinearCost.hpp.

4.57.3.29 `double ClpNonLinearCost::averageTheta () const` `[inline]`

Average theta.

Definition at line 312 of file ClpNonLinearCost.hpp.

4.57.3.30 `void ClpNonLinearCost::setAverageTheta (double value) [inline]`

Definition at line 315 of file ClpNonLinearCost.hpp.

4.57.3.31 `void ClpNonLinearCost::setChangeInCost (double value) [inline]`

Definition at line 318 of file ClpNonLinearCost.hpp.

4.57.3.32 `void ClpNonLinearCost::setMethod (int value) [inline]`

Definition at line 321 of file ClpNonLinearCost.hpp.

4.57.3.33 `bool ClpNonLinearCost::lookBothWays () const [inline]`

See if may want to look both ways.

Definition at line 325 of file ClpNonLinearCost.hpp.

4.57.3.34 `bool ClpNonLinearCost::infeasible (int i) const [inline]`

Definition at line 330 of file ClpNonLinearCost.hpp.

4.57.3.35 `void ClpNonLinearCost::setInfeasible (int i, bool trueFalse) [inline]`

Definition at line 333 of file ClpNonLinearCost.hpp.

4.57.3.36 `unsigned char* ClpNonLinearCost::statusArray () const [inline]`

Definition at line 341 of file ClpNonLinearCost.hpp.

4.57.3.37 `void ClpNonLinearCost::validate ()`

For debug.

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

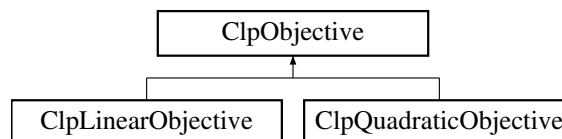
- [src/ClpNonLinearCost.hpp](#)

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 ¤tObj, 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 Constructor & Destructor Documentation

4.58.2.1 ClpObjective::ClpObjective ()

Default Constructor.

4.58.2.2 ClpObjective::ClpObjective (const ClpObjective &)

Copy constructor.

4.58.2.3 virtual ClpObjective::~~ClpObjective () [virtual]

Destructor.

4.58.3 Member Function Documentation

4.58.3.1 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](#).

4.58.3.2 virtual double ClpObjective::reducedGradient (ClpSimplex * model, double * region, bool useFeasibleCosts) [pure virtual]

Returns reduced gradient. Returns an offset (to be added to current one).

Implemented in [ClpQuadraticObjective](#), and [ClpLinearObjective](#).

4.58.3.3 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](#).

4.58.3.4 `virtual double ClpObjective::objectiveValue (const ClpSimplex * model, const double * solution) const` `[pure virtual]`

Return objective value (without any [ClpModel](#) offset) (model may be NULL)

Implemented in [ClpQuadraticObjective](#), and [ClpLinearObjective](#).

4.58.3.5 `virtual void ClpObjective::resize (int newNumberColumns)` `[pure virtual]`

Resize objective.

Implemented in [ClpQuadraticObjective](#), and [ClpLinearObjective](#).

4.58.3.6 `virtual void ClpObjective::deleteSome (int numberToDelete, const int * which)` `[pure virtual]`

Delete columns in objective.

Implemented in [ClpQuadraticObjective](#), and [ClpLinearObjective](#).

4.58.3.7 `virtual void ClpObjective::reallyScale (const double * columnScale)` `[pure virtual]`

Scale objective.

Implemented in [ClpQuadraticObjective](#), and [ClpLinearObjective](#).

4.58.3.8 `virtual int ClpObjective::markNonlinear (char * which)` `[virtual]`

Given a zeroed array sets nonlinear columns to 1.

Returns number of nonlinear columns

Reimplemented in [ClpQuadraticObjective](#).

4.58.3.9 `virtual void ClpObjective::newXValues ()` `[inline],[virtual]`

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

Definition at line 66 of file `ClpObjective.hpp`.

4.58.3.10 `ClpObjective& ClpObjective::operator= (const ClpObjective & rhs)`

Assignment operator.

4.58.3.11 `virtual ClpObjective* ClpObjective::clone () const` `[pure virtual]`

Clone.

Implemented in [ClpQuadraticObjective](#), and [ClpLinearObjective](#).

4.58.3.12 `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](#).

4.58.3.13 `int ClpObjective::type () const` `[inline]`

Returns type (above 63 is extra information)

Definition at line 98 of file ClpObjective.hpp.

4.58.3.14 `void ClpObjective::setType (int value)` `[inline]`

Sets type (above 63 is extra information)

Definition at line 102 of file ClpObjective.hpp.

4.58.3.15 `int ClpObjective::activated () const` `[inline]`

Whether activated.

Definition at line 106 of file ClpObjective.hpp.

4.58.3.16 `void ClpObjective::setActivated (int value)` `[inline]`

Set whether activated.

Definition at line 110 of file ClpObjective.hpp.

4.58.3.17 `double ClpObjective::nonlinearOffset () const` `[inline]`

Objective offset.

Definition at line 115 of file ClpObjective.hpp.

4.58.4 Member Data Documentation

4.58.4.1 `double ClpObjective::offset_` `[protected]`

Value of non-linear part of objective.

Definition at line 126 of file ClpObjective.hpp.

4.58.4.2 `int ClpObjective::type_` `[protected]`

Type of objective - linear is 1.

Definition at line 128 of file ClpObjective.hpp.

4.58.4.3 `int ClpObjective::activated_` `[protected]`

Whether activated.

Definition at line 130 of file ClpObjective.hpp.

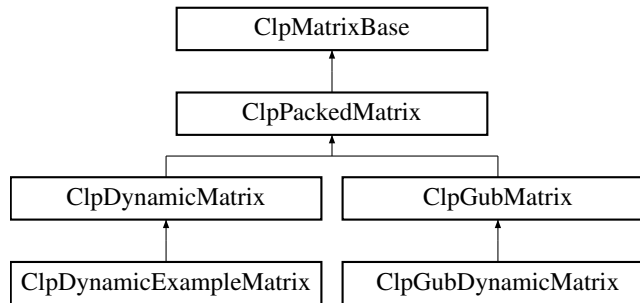
The documentation for this class was generated from the following file:

- [src/ClpObjective.hpp](#)

4.59 ClpPackedMatrix Class Reference

```
#include <ClpPackedMatrix.hpp>
```

Inheritance 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
- Really 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 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 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 &numberWanted)
- 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 *columnScale) 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 `<code>x * A</code>` in `<code>z</code>` 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 numberOfRows, const int *whichRows, int numberColumns, const int *whichColumns)
Subset constructor (without gaps).
- [ClpPackedMatrix](#) (const [CoinPackedMatrix](#) &wholeModel, int numberOfRows, 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 numberOfRows, 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 30 of file ClpPackedMatrix.hpp.

4.59.2 Constructor & Destructor Documentation

4.59.2.1 ClpPackedMatrix::ClpPackedMatrix ()

Default constructor.

4.59.2.2 virtual ClpPackedMatrix::~~ClpPackedMatrix () [virtual]

Destructor.

4.59.2.3 ClpPackedMatrix::ClpPackedMatrix (const ClpPackedMatrix &)

The copy constructor.

4.59.2.4 ClpPackedMatrix::ClpPackedMatrix (const CoinPackedMatrix &)

The copy constructor from an CoinPackedMatrix.

4.59.2.5 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

4.59.2.6 ClpPackedMatrix::ClpPackedMatrix (const CoinPackedMatrix & *wholeModel*, int *numberRows*, const int * *whichRows*, int *numberColumns*, const int * *whichColumns*)

4.59.2.7 ClpPackedMatrix::ClpPackedMatrix (CoinPackedMatrix * *matrix*)

This takes over ownership (for space reasons)

4.59.3 Member Function Documentation

4.59.3.1 virtual CoinPackedMatrix* ClpPackedMatrix::getPackedMatrix () const [inline],[virtual]

Return a complete CoinPackedMatrix.

Implements [ClpMatrixBase](#).

Definition at line 36 of file ClpPackedMatrix.hpp.

4.59.3.2 virtual bool ClpPackedMatrix::isColOrdered () const [inline],[virtual]

Whether the packed matrix is column major ordered or not.

Implements [ClpMatrixBase](#).

Definition at line 40 of file ClpPackedMatrix.hpp.

4.59.3.3 virtual CoinBigIndex ClpPackedMatrix::getNumElements () const [inline],[virtual]

Number of entries in the packed matrix.

Implements [ClpMatrixBase](#).

Definition at line 44 of file ClpPackedMatrix.hpp.

4.59.3.4 `virtual int ClpPackedMatrix::getNumCols () const [inline],[virtual]`

Number of columns.

Implements [ClpMatrixBase](#).

Definition at line 48 of file ClpPackedMatrix.hpp.

4.59.3.5 `virtual int ClpPackedMatrix::getNumRows () const [inline],[virtual]`

Number of rows.

Implements [ClpMatrixBase](#).

Definition at line 52 of file ClpPackedMatrix.hpp.

4.59.3.6 `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 60 of file ClpPackedMatrix.hpp.

4.59.3.7 `double* ClpPackedMatrix::getMutableElements () const [inline]`

Mutable elements.

Definition at line 64 of file ClpPackedMatrix.hpp.

4.59.3.8 `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 72 of file ClpPackedMatrix.hpp.

4.59.3.9 `virtual const CoinBigIndex* ClpPackedMatrix::getVectorStarts () const [inline],[virtual]`

Implements [ClpMatrixBase](#).

Definition at line 76 of file ClpPackedMatrix.hpp.

4.59.3.10 `virtual const int* ClpPackedMatrix::getVectorLengths () const [inline],[virtual]`

The lengths of the major-dimension vectors.

Implements [ClpMatrixBase](#).

Definition at line 80 of file ClpPackedMatrix.hpp.

4.59.3.11 `virtual int ClpPackedMatrix::getVectorLength (int index) const [inline],[virtual]`

The length of a single major-dimension vector.

Reimplemented from [ClpMatrixBase](#).

Definition at line 84 of file `ClpPackedMatrix.hpp`.

4.59.3.12 `virtual void ClpPackedMatrix::deleteCols (const int numDel, const int * indDel) [virtual]`

Delete the columns whose indices are listed in `indDel`.

Implements [ClpMatrixBase](#).

4.59.3.13 `virtual void ClpPackedMatrix::deleteRows (const int numDel, const int * indDel) [virtual]`

Delete the rows whose indices are listed in `indDel`.

Implements [ClpMatrixBase](#).

4.59.3.14 `virtual void ClpPackedMatrix::appendCols (int number, const CoinPackedVectorBase *const * columns) [virtual]`

Append Columns.

Reimplemented from [ClpMatrixBase](#).

4.59.3.15 `virtual void ClpPackedMatrix::appendRows (int number, const CoinPackedVectorBase *const * rows) [virtual]`

Append Rows.

Reimplemented from [ClpMatrixBase](#).

4.59.3.16 `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](#).

4.59.3.17 `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 109 of file `ClpPackedMatrix.hpp`.

4.59.3.18 `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 116 of file ClpPackedMatrix.hpp.

4.59.3.19 `virtual ClpMatrixBase* ClpPackedMatrix::reverseOrderedCopy () const [virtual]`

Returns a new matrix in reverse order without gaps.

Reimplemented from [ClpMatrixBase](#).

Reimplemented in [ClpGubMatrix](#).

4.59.3.20 `virtual CoinBigIndex ClpPackedMatrix::countBasis (const int * whichColumn, int & numberColumnBasic) [virtual]`

Returns number of elements in column part of basis.

Implements [ClpMatrixBase](#).

Reimplemented in [ClpGubMatrix](#).

4.59.3.21 `virtual void ClpPackedMatrix::fillBasis (ClpSimplex * model, const int * whichColumn, int & numberColumnBasic, int * row, int * start, int * rowCount, int * columnCount, CoinFactorizationDouble * element) [virtual]`

Fills in column part of basis.

Implements [ClpMatrixBase](#).

Reimplemented in [ClpGubMatrix](#).

4.59.3.22 `virtual int ClpPackedMatrix::scale (ClpModel * model, const ClpSimplex * baseModel = NULL) const [virtual]`

Creates scales for column copy (rowCopy in model may be modified) returns non-zero if no scaling done.

Reimplemented from [ClpMatrixBase](#).

4.59.3.23 `virtual void ClpPackedMatrix::scaleRowCopy (ClpModel * model) const [virtual]`

Scales rowCopy if column copy scaled Only called if scales already exist.

Reimplemented from [ClpMatrixBase](#).

4.59.3.24 `void ClpPackedMatrix::createScaledMatrix (ClpSimplex * model) const`

Creates scaled column copy if scales exist.

4.59.3.25 `virtual ClpMatrixBase* ClpPackedMatrix::scaledColumnCopy (ClpModel * model) const [virtual]`

Really really scales column copy Only called if scales already exist.

Up to user to delete

Reimplemented from [ClpMatrixBase](#).

4.59.3.26 `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](#).

4.59.3.27 `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](#).

4.59.3.28 `virtual void ClpPackedMatrix::unpack (const ClpSimplex * model, CoinIndexedVector * rowArray, int column) const [virtual]`

Unpacks a column into an CoinIndexedvector.

Implements [ClpMatrixBase](#).

Reimplemented in [ClpGubMatrix](#).

4.59.3.29 `virtual void ClpPackedMatrix::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](#).

Reimplemented in [ClpGubMatrix](#).

4.59.3.30 `virtual void ClpPackedMatrix::add (const ClpSimplex * model, CoinIndexedVector * rowArray, int column, double multiplier) const [virtual]`

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

Implements [ClpMatrixBase](#).

Reimplemented in [ClpGubMatrix](#).

4.59.3.31 `virtual void ClpPackedMatrix::add (const ClpSimplex * model, double * array, int column, double multiplier) const [virtual]`

Adds multiple of a column into an array.

Implements [ClpMatrixBase](#).

Reimplemented in [ClpGubMatrix](#).

4.59.3.32 `virtual void ClpPackedMatrix::releasePackedMatrix () const [inline],[virtual]`

Allow any parts of a created CoinPackedMatrix to be deleted.

Implements [ClpMatrixBase](#).

Definition at line 182 of file ClpPackedMatrix.hpp.

4.59.3.33 `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](#).

4.59.3.34 `virtual bool ClpPackedMatrix::canDoPartialPricing () const [virtual]`

Says whether it can do partial pricing.

Reimplemented from [ClpMatrixBase](#).

4.59.3.35 `virtual void ClpPackedMatrix::partialPricing (ClpSimplex * model, double start, double end, int & bestSequence, int & numberWanted) [virtual]`

Partial pricing.

Reimplemented from [ClpMatrixBase](#).

Reimplemented in [ClpGubMatrix](#), [ClpDynamicExampleMatrix](#), [ClpDynamicMatrix](#), and [ClpGubDynamicMatrix](#).

4.59.3.36 `virtual int ClpPackedMatrix::refresh (ClpSimplex * model) [virtual]`

makes sure active columns correct

Reimplemented from [ClpMatrixBase](#).

Reimplemented in [ClpDynamicMatrix](#).

4.59.3.37 `virtual void ClpPackedMatrix::reallyScale (const double * rowScale, const double * columnScale) [virtual]`

Reimplemented from [ClpMatrixBase](#).

4.59.3.38 `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](#).

4.59.3.39 `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()`

Implements [ClpMatrixBase](#).

Reimplemented in [ClpGubDynamicMatrix](#), and [ClpDynamicMatrix](#).

4.59.3.40 `virtual void ClpPackedMatrix::times (double scalar, const double * x, double * y, const double * rowScale, const double * columnScale) const [virtual]`

And for scaling.

Reimplemented from [ClpMatrixBase](#).

4.59.3.41 `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()`

Implements [ClpMatrixBase](#).

4.59.3.42 `virtual void ClpPackedMatrix::transposeTimes (double scalar, const double * x, double * y, const double * rowScale, const double * columnScale, double * spare = NULL) const [virtual]`

And for scaling.

Reimplemented from [ClpMatrixBase](#).

4.59.3.43 `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 .

@pre `pi` must be of size `numRows()`
 @pre `y` must be of size `numColumns()`

This just does subset (but puts in correct place in y)

4.59.3.44 `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](#).

4.59.3.45 `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](#)

4.59.3.46 `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](#).

4.59.3.47 `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](#).

```
4.59.3.48 virtual bool ClpPackedMatrix::canCombine ( const ClpSimplex * model, const CoinIndexedVector * pi ) const
    [virtual]
```

Returns true if can combine transposeTimes and subsetTransposeTimes and if it would be faster.

Reimplemented from [ClpMatrixBase](#).

```
4.59.3.49 virtual void ClpPackedMatrix::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 ) [virtual]
```

Updates two arrays for steepest.

Reimplemented from [ClpMatrixBase](#).

```
4.59.3.50 virtual void ClpPackedMatrix::subsetTimes2 ( const ClpSimplex * model, CoinIndexedVector * dj1, const
    CoinIndexedVector * pi2, CoinIndexedVector * dj2, double referenceIn, double devex, unsigned int * reference, double
    * weights, double scaleFactor ) [virtual]
```

Updates second array for steepest and does devex weights.

Reimplemented from [ClpMatrixBase](#).

```
4.59.3.51 void ClpPackedMatrix::useEffectiveRhs ( ClpSimplex * model )
```

Sets up an effective RHS.

```
4.59.3.52 CoinPackedMatrix* ClpPackedMatrix::matrix ( ) const [inline]
```

Returns CoinPackedMatrix (non const)

Definition at line 305 of file ClpPackedMatrix.hpp.

```
4.59.3.53 void ClpPackedMatrix::setMatrixNull ( ) [inline]
```

Just sets matrix_ to NULL so it can be used elsewhere.

used in GUB

Definition at line 311 of file ClpPackedMatrix.hpp.

```
4.59.3.54 void ClpPackedMatrix::makeSpecialColumnCopy ( ) [inline]
```

Say we want special column copy.

Definition at line 315 of file ClpPackedMatrix.hpp.

```
4.59.3.55 void ClpPackedMatrix::releaseSpecialColumnCopy ( )
```

Say we don't want special column copy.

```
4.59.3.56 bool ClpPackedMatrix::zeros ( ) const [inline]
```

Are there zeros?

Definition at line 321 of file ClpPackedMatrix.hpp.

4.59.3.57 `bool ClpPackedMatrix::wantsSpecialColumnCopy () const [inline]`

Do we want special column copy.

Definition at line 325 of file `ClpPackedMatrix.hpp`.

4.59.3.58 `int ClpPackedMatrix::flags () const [inline]`

Flags.

Definition at line 329 of file `ClpPackedMatrix.hpp`.

4.59.3.59 `void ClpPackedMatrix::checkGaps () [inline]`

Sets `flags_` correctly.

Definition at line 333 of file `ClpPackedMatrix.hpp`.

4.59.3.60 `int ClpPackedMatrix::numberActiveColumns () const [inline]`

number of active columns (normally same as number of columns)

Definition at line 337 of file `ClpPackedMatrix.hpp`.

4.59.3.61 `void ClpPackedMatrix::setNumberActiveColumns (int value) [inline]`

Set number of active columns (normally same as number of columns)

Definition at line 340 of file `ClpPackedMatrix.hpp`.

4.59.3.62 `ClpPackedMatrix& ClpPackedMatrix::operator= (const ClpPackedMatrix &)`

4.59.3.63 `virtual ClpMatrixBase* ClpPackedMatrix::clone () const [virtual]`

Clone.

Implements [ClpMatrixBase](#).

Reimplemented in [ClpGubMatrix](#), [ClpDynamicMatrix](#), [ClpDynamicExampleMatrix](#), and [ClpGubDynamicMatrix](#).

4.59.3.64 `virtual void ClpPackedMatrix::copy (const ClpPackedMatrix * from) [virtual]`

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

4.59.3.65 `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](#).

4.59.3.66 `void ClpPackedMatrix::specialRowCopy (ClpSimplex * model, const ClpMatrixBase * rowCopy)`

make special row copy

4.59.3.67 `void ClpPackedMatrix::specialColumnCopy (ClpSimplex * model)`

make special column copy

4.59.3.68 `virtual void ClpPackedMatrix::correctSequence (const ClpSimplex * model, int & sequenceIn, int & sequenceOut)`
`[virtual]`

Correct sequence in and out to give true value.

Reimplemented from [ClpMatrixBase](#).

Reimplemented in [ClpGubMatrix](#).

4.59.3.69 `void ClpPackedMatrix::checkFlags (int type) const` `[protected]`

Check validity.

4.59.4 Member Data Documentation

4.59.4.1 `CoinPackedMatrix* ClpPackedMatrix::matrix_` `[protected]`

Data.

Definition at line 467 of file `ClpPackedMatrix.hpp`.

4.59.4.2 `int ClpPackedMatrix::numberActiveColumns_` `[protected]`

number of active columns (normally same as number of columns)

Definition at line 469 of file `ClpPackedMatrix.hpp`.

4.59.4.3 `int ClpPackedMatrix::flags_` `[mutable], [protected]`

Flags - 1 - has zero elements 2 - has gaps 4 - has special row copy 8 - has special column copy 16 - wants special column copy.

Definition at line 477 of file `ClpPackedMatrix.hpp`.

4.59.4.4 `ClpPackedMatrix2* ClpPackedMatrix::rowCopy_` `[protected]`

Special row copy.

Definition at line 479 of file `ClpPackedMatrix.hpp`.

4.59.4.5 `ClpPackedMatrix3* ClpPackedMatrix::columnCopy_` `[protected]`

Special column copy.

Definition at line 481 of file `ClpPackedMatrix.hpp`.

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

- [src/ClpPackedMatrix.hpp](#)

4.60 ClpPackedMatrix2 Class Reference

```
#include <ClpPackedMatrix.hpp>
```

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 [usefullInfo](#) () 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 509 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 virtual ClpPackedMatrix2::~~ClpPackedMatrix2 () [virtual]

Destructor.

4.60.2.4 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 * *sparseArray*, 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](#)

4.60.3.2 bool ClpPackedMatrix2::usefulInfo () const [inline]

Returns true if copy has useful information.

Definition at line 523 of file ClpPackedMatrix.hpp.

4.60.3.3 ClpPackedMatrix2& ClpPackedMatrix2::operator= (const ClpPackedMatrix2 &)

4.60.4 Member Data Documentation

4.60.4.1 int ClpPackedMatrix2::numberBlocks_ [protected]

Number of blocks.

Definition at line 552 of file ClpPackedMatrix.hpp.

4.60.4.2 int ClpPackedMatrix2::numberRows_ [protected]

Number of rows.

Definition at line 554 of file ClpPackedMatrix.hpp.

4.60.4.3 int* ClpPackedMatrix2::offset_ [protected]

Column offset for each block (plus one at end)

Definition at line 556 of file ClpPackedMatrix.hpp.

4.60.4.4 unsigned short* ClpPackedMatrix2::count_ [mutable], [protected]

Counts of elements in each part of row.

Definition at line 558 of file ClpPackedMatrix.hpp.

4.60.4.5 CoinBigIndex* ClpPackedMatrix2::rowStart_ [mutable], [protected]

Row starts.

Definition at line 560 of file ClpPackedMatrix.hpp.

4.60.4.6 unsigned short* ClpPackedMatrix2::column_ [protected]

columns within block

Definition at line 562 of file ClpPackedMatrix.hpp.

4.60.4.7 double* ClpPackedMatrix2::work_ [protected]

work arrays

Definition at line 564 of file ClpPackedMatrix.hpp.

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

- [src/ClpPackedMatrix.hpp](#)

4.61 ClpPackedMatrix3 Class Reference

```
#include <ClpPackedMatrix.hpp>
```

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 578 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 virtual ClpPackedMatrix3::~ClpPackedMatrix3 () [virtual]

Destructor.

4.61.2.4 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](#)

4.61.3.2 `void ClpPackedMatrix3::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.

4.61.3.3 `ClpPackedMatrix3& ClpPackedMatrix3::operator= (const ClpPackedMatrix3 &)`

4.61.3.4 `void ClpPackedMatrix3::sortBlocks (const ClpSimplex * model)`

Sort blocks.

4.61.3.5 `void ClpPackedMatrix3::swapOne (const ClpSimplex * model, const ClpPackedMatrix * matrix, int iColumn)`

Swap one variable.

4.61.4 Member Data Documentation

4.61.4.1 `int ClpPackedMatrix3::numberBlocks_ [protected]`

Number of blocks.

Definition at line 631 of file `ClpPackedMatrix.hpp`.

4.61.4.2 `int ClpPackedMatrix3::numberColumns_ [protected]`

Number of columns.

Definition at line 633 of file `ClpPackedMatrix.hpp`.

4.61.4.3 `int* ClpPackedMatrix3::column_ [protected]`

Column indices and reverse lookup (within block)

Definition at line 635 of file `ClpPackedMatrix.hpp`.

4.61.4.4 `CoinBigIndex* ClpPackedMatrix3::start_ [protected]`

Starts for odd/long vectors.

Definition at line 637 of file `ClpPackedMatrix.hpp`.

4.61.4.5 `int* ClpPackedMatrix3::row_ [protected]`

Rows.

Definition at line 639 of file `ClpPackedMatrix.hpp`.

4.61.4.6 `double* ClpPackedMatrix3::element_ [protected]`

Elements.

Definition at line 641 of file `ClpPackedMatrix.hpp`.

4.61.4.7 `blockStruct* ClpPackedMatrix3::block_ [protected]`

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

Definition at line 643 of file `ClpPackedMatrix.hpp`.

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

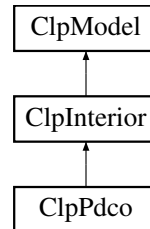
- [src/ClpPackedMatrix.hpp](#)

4.62 ClpPdco Class Reference

This solves problems in Primal Dual Convex Optimization.

```
#include <ClpPdco.hpp>
```

Inheritance 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

4.62.2.2 `int ClpPdco::pdco (ClpPdcoBase * stuff, Options & options, Info & info, Outfo & outfo)`

4.62.2.3 `void ClpPdco::lsqr ()`

LSQR.

4.62.2.4 `void ClpPdco::matVecMult (int , double * , double *)`

4.62.2.5 `void ClpPdco::matVecMult (int , CoinDenseVector< double > & , double *)`

4.62.2.6 `void ClpPdco::matVecMult (int , CoinDenseVector< double > & , CoinDenseVector< double > &)`

4.62.2.7 `void ClpPdco::matVecMult (int , CoinDenseVector< double > * , CoinDenseVector< double > *)`

4.62.2.8 `void ClpPdco::getBoundTypes (int * , int * , int * , int **)`

4.62.2.9 `void ClpPdco::getGrad (CoinDenseVector< double > & x, CoinDenseVector< double > & grad)`

4.62.2.10 `void ClpPdco::getHessian (CoinDenseVector< double > & x, CoinDenseVector< double > & H)`

4.62.2.11 `double ClpPdco::getObj (CoinDenseVector< double > & x)`

4.62.2.12 `void ClpPdco::matPrecon (double , double * , double *)`

4.62.2.13 `void ClpPdco::matPrecon (double , CoinDenseVector< double > & , double *)`

4.62.2.14 `void ClpPdco::matPrecon (double , CoinDenseVector< double > & , CoinDenseVector< double > &)`

4.62.2.15 `void ClpPdco::matPrecon (double , CoinDenseVector< double > * , CoinDenseVector< double > *)`

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

- [src/ClpPdco.hpp](#)

4.63 ClpPdcoBase Class Reference

Abstract base class for tailoring everything for Pcdco.

```
#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 Pcdco.

Since this class is abstract, no object of this type can be created.

If a derived class provides all methods then all ClpPcdco 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.

4.63.2.2 virtual ClpPdcoBase::~~ClpPdcoBase () [virtual]

Destructor (has to be public)

4.63.2.3 ClpPdcoBase::ClpPdcoBase (const ClpPdcoBase &) [protected]

4.63.3 Member Function Documentation

4.63.3.1 virtual void ClpPdcoBase::matVecMult (ClpInterior * *model*, int *mode*, double * *x*, double * *y*) const [pure virtual]

4.63.3.2 virtual void ClpPdcoBase::getGrad (ClpInterior * *model*, CoinDenseVector< double > & *x*, CoinDenseVector< double > & *grad*) const [pure virtual]

4.63.3.3 virtual void ClpPdcoBase::getHessian (ClpInterior * *model*, CoinDenseVector< double > & *x*, CoinDenseVector< double > & *H*) const [pure virtual]

4.63.3.4 virtual double ClpPdcoBase::getObj (ClpInterior * *model*, CoinDenseVector< double > & *x*) const [pure virtual]

4.63.3.5 virtual void ClpPdcoBase::matPrecon (ClpInterior * *model*, double *delta*, double * *x*, double * *y*) const [pure virtual]

4.63.3.6 virtual ClpPdcoBase* ClpPdcoBase::clone () const [pure virtual]

4.63.3.7 int ClpPdcoBase::type () const [inline]

Returns type.

Definition at line 46 of file ClpPdcoBase.hpp.

4.63.3.8 void ClpPdcoBase::setType (int *type*) [inline]

Sets type.

Definition at line 50 of file ClpPdcoBase.hpp.

4.63.3.9 int ClpPdcoBase::sizeD1 () const [inline]

Returns size of d1.

Definition at line 54 of file ClpPdcoBase.hpp.

4.63.3.10 double ClpPdcoBase::getD1 () const [inline]

Returns d1 as scalar.

Definition at line 58 of file ClpPdcoBase.hpp.

4.63.3.11 int ClpPdcoBase::sizeD2 () const [inline]

Returns size of d2.

Definition at line 62 of file ClpPdcoBase.hpp.

4.63.3.12 `double ClpPdcoBase::getD2 () const [inline]`

Returns d2 as scalar.

Definition at line 66 of file ClpPdcoBase.hpp.

4.63.3.13 `ClpPdcoBase& ClpPdcoBase::operator= (const ClpPdcoBase &) [protected]`

4.63.4 Member Data Documentation

4.63.4.1 `double ClpPdcoBase::d1_ [protected]`

Should be dense vectors.

Definition at line 96 of file ClpPdcoBase.hpp.

4.63.4.2 `double ClpPdcoBase::d2_ [protected]`

Definition at line 97 of file ClpPdcoBase.hpp.

4.63.4.3 `int ClpPdcoBase::type_ [protected]`

type (may be useful)

Definition at line 99 of file ClpPdcoBase.hpp.

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

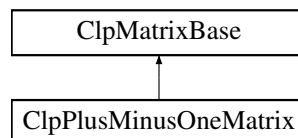
- [src/ClpPdcoBase.hpp](#)

4.64 ClpPlusMinusOneMatrix Class Reference

This implements a simple +- one matrix as derived from [ClpMatrixBase](#).

```
#include <ClpPlusMinusOneMatrix.hpp>
```

Inheritance 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 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 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 *columnScale) 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 [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.

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 numberOfRows, int numberOfColumns, bool columnOrdered, const int *indices, const CoinBigIndex *startPositive, const CoinBigIndex *startNegative)
Constructor from arrays.
- [ClpPlusMinusOneMatrix](#) (const [ClpPlusMinusOneMatrix](#) &wholeModel, int numberOfRows, const int *whichRows, int numberOfColumns, const int *whichColumns)

- Subset constructor (without gaps).*
- [ClpPlusMinusOneMatrix](#) & [operator=](#) (const [ClpPlusMinusOneMatrix](#) &)
- virtual [ClpMatrixBase](#) * [clone](#) () const
- Clone.*
- virtual [ClpMatrixBase](#) * [subsetClone](#) (int numberOfRows, const int *whichRows, int numberColumns, const int *whichColumns) const
- Subset clone (without gaps).*
- void [passInCopy](#) (int numberOfRows, 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 &numberWanted)
- 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 * [startPositive_](#)
- Start of +1's for each.*
- CoinBigIndex * [startNegative_](#)
- Start of -1's for each.*
- int * [indices_](#)
- Data -1, then +1 rows in pairs (row==-1 if one entry)*
- int [numberOfRows_](#)
- Number of rows.*
- int [numberColumns_](#)
- Number of columns.*
- bool [columnOrdered_](#)
- True if column ordered.*

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 [virtual ClpPlusMinusOneMatrix::~ClpPlusMinusOneMatrix](#) () [virtual]

Destructor.

4.64.2.3 ClpPlusMinusOneMatrix::ClpPlusMinusOneMatrix (const ClpPlusMinusOneMatrix &)

The copy constructor.

4.64.2.4 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,

4.64.2.5 ClpPlusMinusOneMatrix::ClpPlusMinusOneMatrix (int *numberRows*, int *numberColumns*, bool *columnOrdered*, const int * *indices*, const CoinBigIndex * *startPositive*, const CoinBigIndex * *startNegative*)

Constructor from arrays.

4.64.2.6 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 CoinPackedMatrix* ClpPlusMinusOneMatrix::getPackedMatrix () const [virtual]

Return a complete CoinPackedMatrix.

Implements [ClpMatrixBase](#).

4.64.3.2 virtual bool ClpPlusMinusOneMatrix::isColOrdered () const [virtual]

Whether the packed matrix is column major ordered or not.

Implements [ClpMatrixBase](#).

4.64.3.3 virtual CoinBigIndex ClpPlusMinusOneMatrix::getNumElements () const [virtual]

Number of entries in the packed matrix.

Implements [ClpMatrixBase](#).

4.64.3.4 virtual int ClpPlusMinusOneMatrix::getNumCols () const [inline],[virtual]

Number of columns.

Implements [ClpMatrixBase](#).

Definition at line 30 of file ClpPlusMinusOneMatrix.hpp.

4.64.3.5 virtual int ClpPlusMinusOneMatrix::getNumRows () const [inline],[virtual]

Number of rows.

Implements [ClpMatrixBase](#).

Definition at line 34 of file ClpPlusMinusOneMatrix.hpp.

4.64.3.6 `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](#).

4.64.3.7 `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`.

4.64.3.8 `int* ClpPlusMinusOneMatrix::getMutableIndices () const` [inline]

Definition at line 52 of file `ClpPlusMinusOneMatrix.hpp`.

4.64.3.9 `virtual const CoinBigIndex* ClpPlusMinusOneMatrix::getVectorStarts () const` [virtual]

Implements [ClpMatrixBase](#).

4.64.3.10 `virtual const int* ClpPlusMinusOneMatrix::getVectorLengths () const` [virtual]

The lengths of the major-dimension vectors.

Implements [ClpMatrixBase](#).

4.64.3.11 `virtual void ClpPlusMinusOneMatrix::deleteCols (const int numDel, const int * indDel)` [virtual]

Delete the columns whose indices are listed in `indDel`.

Implements [ClpMatrixBase](#).

4.64.3.12 `virtual void ClpPlusMinusOneMatrix::deleteRows (const int numDel, const int * indDel)` [virtual]

Delete the rows whose indices are listed in `indDel`.

Implements [ClpMatrixBase](#).

4.64.3.13 `virtual void ClpPlusMinusOneMatrix::appendCols (int number, const CoinPackedVectorBase *const * columns)`
[virtual]

Append Columns.

Reimplemented from [ClpMatrixBase](#).

4.64.3.14 `virtual void ClpPlusMinusOneMatrix::appendRows (int number, const CoinPackedVectorBase *const * rows)`
[virtual]

Append Rows.

Reimplemented from [ClpMatrixBase](#).

4.64.3.15 `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](#).

4.64.3.16 `virtual ClpMatrixBase* ClpPlusMinusOneMatrix::reverseOrderedCopy () const [virtual]`

Returns a new matrix in reverse order without gaps.

Reimplemented from [ClpMatrixBase](#).

4.64.3.17 `virtual CoinBigIndex ClpPlusMinusOneMatrix::countBasis (const int * whichColumn, int & numberColumnBasic) [virtual]`

Returns number of elements in column part of basis.

Implements [ClpMatrixBase](#).

4.64.3.18 `virtual void ClpPlusMinusOneMatrix::fillBasis (ClpSimplex * model, const int * whichColumn, int & numberColumnBasic, int * row, int * start, int * rowCount, int * columnCount, CoinFactorizationDouble * element) [virtual]`

Fills in column part of basis.

Implements [ClpMatrixBase](#).

4.64.3.19 `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](#).

4.64.3.20 `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](#).

4.64.3.21 `virtual void ClpPlusMinusOneMatrix::unpack (const ClpSimplex * model, CoinIndexedVector * rowArray, int column) const [virtual]`

Unpacks a column into an CoinIndexedvector.

Implements [ClpMatrixBase](#).

4.64.3.22 `virtual void ClpPlusMinusOneMatrix::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](#).

4.64.3.23 `virtual void ClpPlusMinusOneMatrix::add (const ClpSimplex * model, CoinIndexedVector * rowArray, int column, double multiplier) const [virtual]`

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

Implements [ClpMatrixBase](#).

4.64.3.24 `virtual void ClpPlusMinusOneMatrix::add (const ClpSimplex * model, double * array, int column, double multiplier) const [virtual]`

Adds multiple of a column into an array.

Implements [ClpMatrixBase](#).

4.64.3.25 `virtual void ClpPlusMinusOneMatrix::releasePackedMatrix () const [virtual]`

Allow any parts of a created CoinMatrix to be deleted.

Implements [ClpMatrixBase](#).

4.64.3.26 `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](#).

4.64.3.27 `void ClpPlusMinusOneMatrix::checkValid (bool detail) const`

Just checks matrix valid - will say if dimensions not quite right if detail.

4.64.3.28 `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()`

Implements [ClpMatrixBase](#).

4.64.3.29 `virtual void ClpPlusMinusOneMatrix::times (double scalar, const double * x, double * y, const double * rowScale, const double * columnScale) const [virtual]`

And for scaling.

Reimplemented from [ClpMatrixBase](#).

4.64.3.30 `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()`

Implements [ClpMatrixBase](#).

4.64.3.31 `virtual void ClpPlusMinusOneMatrix::transposeTimes (double scalar, const double * x, double * y, const double * rowScale, const double * columnScale, double * spare = NULL) const` [virtual]

And for scaling.

Reimplemented from [ClpMatrixBase](#).

4.64.3.32 `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](#).

4.64.3.33 `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

4.64.3.34 `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](#).

4.64.3.35 `virtual bool ClpPlusMinusOneMatrix::canCombine (const ClpSimplex * model, const CoinIndexedVector * pi) const` [virtual]

Returns true if can combine `transposeTimes` and `subsetTransposeTimes` and if it would be faster.

Reimplemented from [ClpMatrixBase](#).

4.64.3.36 `virtual void ClpPlusMinusOneMatrix::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)` [virtual]

Updates two arrays for steepest.

Reimplemented from [ClpMatrixBase](#).

4.64.3.37 `virtual void ClpPlusMinusOneMatrix::subsetTimes2 (const ClpSimplex * model, CoinIndexedVector * dj1, const CoinIndexedVector * pi2, CoinIndexedVector * dj2, double referenceIn, double devex, unsigned int * reference, double * weights, double scaleFactor) [virtual]`

Updates second array for steepest and does devex weights.

Reimplemented from [ClpMatrixBase](#).

4.64.3.38 `CoinBigIndex* ClpPlusMinusOneMatrix::startPositive () const [inline]`

Return starts of +1s.

Definition at line 202 of file `ClpPlusMinusOneMatrix.hpp`.

4.64.3.39 `CoinBigIndex* ClpPlusMinusOneMatrix::startNegative () const [inline]`

Return starts of -1s.

Definition at line 206 of file `ClpPlusMinusOneMatrix.hpp`.

4.64.3.40 `ClpPlusMinusOneMatrix& ClpPlusMinusOneMatrix::operator= (const ClpPlusMinusOneMatrix &)`

4.64.3.41 `virtual ClpMatrixBase* ClpPlusMinusOneMatrix::clone () const [virtual]`

Clone.

Implements [ClpMatrixBase](#).

4.64.3.42 `virtual ClpMatrixBase* ClpPlusMinusOneMatrix::subsetClone (int numberOfRows, 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](#).

4.64.3.43 `void ClpPlusMinusOneMatrix::passInCopy (int numberOfRows, int numberColumns, bool columnOrdered, int * indices, CoinBigIndex * startPositive, CoinBigIndex * startNegative)`

pass in copy (object takes ownership)

4.64.3.44 `virtual bool ClpPlusMinusOneMatrix::canDoPartialPricing () const [virtual]`

Says whether it can do partial pricing.

Reimplemented from [ClpMatrixBase](#).

4.64.3.45 `virtual void ClpPlusMinusOneMatrix::partialPricing (ClpSimplex * model, double start, double end, int & bestSequence, int & numberWanted) [virtual]`

Partial pricing.

Reimplemented from [ClpMatrixBase](#).

4.64.4 Member Data Documentation

4.64.4.1 `CoinPackedMatrix* ClpPlusMinusOneMatrix::matrix_ [mutable], [protected]`

For fake `CoinPackedMatrix`.

Definition at line 266 of file ClpPlusMinusOneMatrix.hpp.

4.64.4.2 `int* ClpPlusMinusOneMatrix::lengths_` `[mutable], [protected]`

Definition at line 267 of file ClpPlusMinusOneMatrix.hpp.

4.64.4.3 `CoinBigIndex* ClpPlusMinusOneMatrix::startPositive_` `[protected]`

Start of +1's for each.

Definition at line 269 of file ClpPlusMinusOneMatrix.hpp.

4.64.4.4 `CoinBigIndex* ClpPlusMinusOneMatrix::startNegative_` `[protected]`

Start of -1's for each.

Definition at line 271 of file ClpPlusMinusOneMatrix.hpp.

4.64.4.5 `int* ClpPlusMinusOneMatrix::indices_` `[protected]`

Data -1, then +1 rows in pairs (row==-1 if one entry)

Definition at line 273 of file ClpPlusMinusOneMatrix.hpp.

4.64.4.6 `int ClpPlusMinusOneMatrix::numberRows_` `[protected]`

Number of rows.

Definition at line 275 of file ClpPlusMinusOneMatrix.hpp.

4.64.4.7 `int ClpPlusMinusOneMatrix::numberColumns_` `[protected]`

Number of columns.

Definition at line 277 of file ClpPlusMinusOneMatrix.hpp.

4.64.4.8 `bool ClpPlusMinusOneMatrix::columnOrdered_` `[protected]`

True if column ordered.

Definition at line 279 of file ClpPlusMinusOneMatrix.hpp.

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

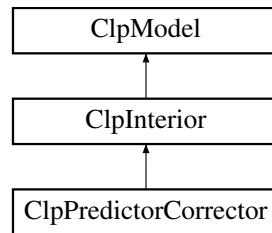
- [src/ClpPlusMinusOneMatrix.hpp](#)

4.65 ClpPredictorCorrector Class Reference

This solves LPs using the predictor-corrector method due to Mehrotra.

```
#include <ClpPredictorCorrector.hpp>
```

Inheritance diagram for ClpPredictorCorrector:



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 $\delta T * \delta W$ and $\delta S * \delta Z$.*
- 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 centrality 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 CoinWorkDouble ClpPredictorCorrector::findStepLength (int *phase*)

findStepLength.

4.65.2.3 CoinWorkDouble ClpPredictorCorrector::findDirectionVector (const int *phase*)

findDirectionVector.

4.65.2.4 int ClpPredictorCorrector::createSolution ()

createSolution. Creates solution from scratch (- code if no memory)

4.65.2.5 CoinWorkDouble ClpPredictorCorrector::complementarityGap (int & *numberComplementarityPairs*, int & *numberComplementarityItems*, const int *phase*)

complementarityGap. Computes gap

4.65.2.6 void ClpPredictorCorrector::setupForSolve (const int *phase*)

setupForSolve.

4.65.2.7 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)

4.65.2.8 bool ClpPredictorCorrector::checkGoodMove (const bool *doCorrector*, CoinWorkDouble & *bestNextGap*, bool *allowIncreasingGap*)

sees if looks plausible change in complementarity

4.65.2.9 bool ClpPredictorCorrector::checkGoodMove2 (CoinWorkDouble *move*, CoinWorkDouble & *bestNextGap*, bool *allowIncreasingGap*)

: checks for one step size

4.65.2.10 int ClpPredictorCorrector::updateSolution (CoinWorkDouble nextGap)

updateSolution. Updates solution at end of iteration

4.65.2.11 CoinWorkDouble ClpPredictorCorrector::affineProduct ()

Save info on products of affine $\Delta T * \Delta W$ and $\Delta S * \Delta Z$.

4.65.2.12 void ClpPredictorCorrector::debugMove (int phase, CoinWorkDouble primalStep, CoinWorkDouble dualStep)

See exactly what would happen given current deltas.

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

- [src/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 [doTripletion](#) () const
Whether we want to do tripletion part of presolve.
- void [setDoTripletion](#) (bool [doTripletion](#))
- 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 [doSingletonColumn](#))
- 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 [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 keepIntegers, 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 Constructor & Destructor Documentation**4.66.2.1 ClpPresolve::ClpPresolve ()**

Default constructor.

4.66.2.2 virtual ClpPresolve::~~ClpPresolve () [virtual]

Virtual destructor.

4.66.3 Member Function Documentation**4.66.3.1 ClpSimplex* ClpPresolve::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)****4.66.3.2 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

4.66.3.3 ClpSimplex* ClpPresolve::model () const

Return pointer to presolved model, Up to user to destroy.

4.66.3.4 ClpSimplex* ClpPresolve::originalModel () const

Return pointer to original model.

4.66.3.5 void ClpPresolve::setOriginalModel (ClpSimplex * *model*)

Set pointer to original model.

4.66.3.6 const int* ClpPresolve::originalColumns () const

return pointer to original columns

4.66.3.7 const int* ClpPresolve::originalRows () const

return pointer to original rows

4.66.3.8 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.

4.66.3.9 double ClpPresolve::nonLinearValue () const [inline]

Definition at line 79 of file ClpPresolve.hpp.

4.66.3.10 bool ClpPresolve::doDual () const [inline]

Whether we want to do dual part of presolve.

Definition at line 83 of file ClpPresolve.hpp.

4.66.3.11 void ClpPresolve::setDoDual (bool *doDual*) [inline]

Definition at line 86 of file ClpPresolve.hpp.

4.66.3.12 bool ClpPresolve::doSingleton () const [inline]

Whether we want to do singleton part of presolve.

Definition at line 91 of file ClpPresolve.hpp.

4.66.3.13 void ClpPresolve::setDoSingleton (bool *doSingleton*) [inline]

Definition at line 94 of file ClpPresolve.hpp.

4.66.3.14 bool ClpPresolve::doDoubleton () const [inline]

Whether we want to do doubleton part of presolve.

Definition at line 99 of file ClpPresolve.hpp.

4.66.3.15 `void ClpPresolve::setDoDoubleton (bool doDoubleton) [inline]`

Definition at line 102 of file ClpPresolve.hpp.

4.66.3.16 `bool ClpPresolve::doTripletion () const [inline]`

Whether we want to do tripletion part of presolve.

Definition at line 107 of file ClpPresolve.hpp.

4.66.3.17 `void ClpPresolve::setDoTripletion (bool doTripletion) [inline]`

Definition at line 110 of file ClpPresolve.hpp.

4.66.3.18 `bool ClpPresolve::doTighten () const [inline]`

Whether we want to do tighten part of presolve.

Definition at line 115 of file ClpPresolve.hpp.

4.66.3.19 `void ClpPresolve::setDoTighten (bool doTighten) [inline]`

Definition at line 118 of file ClpPresolve.hpp.

4.66.3.20 `bool ClpPresolve::doForcing () const [inline]`

Whether we want to do forcing part of presolve.

Definition at line 123 of file ClpPresolve.hpp.

4.66.3.21 `void ClpPresolve::setDoForcing (bool doForcing) [inline]`

Definition at line 126 of file ClpPresolve.hpp.

4.66.3.22 `bool ClpPresolve::doImpliedFree () const [inline]`

Whether we want to do impliedfree part of presolve.

Definition at line 131 of file ClpPresolve.hpp.

4.66.3.23 `void ClpPresolve::setDoImpliedFree (bool doImpliedfree) [inline]`

Definition at line 134 of file ClpPresolve.hpp.

4.66.3.24 `bool ClpPresolve::doDupcol () const [inline]`

Whether we want to do dupcol part of presolve.

Definition at line 139 of file ClpPresolve.hpp.

4.66.3.25 `void ClpPresolve::setDoDupcol (bool doDupcol) [inline]`

Definition at line 142 of file ClpPresolve.hpp.

4.66.3.26 `bool ClpPresolve::doDuprow () const [inline]`

Whether we want to do duprow part of presolve.

Definition at line 147 of file ClpPresolve.hpp.

4.66.3.27 void ClpPresolve::setDoDuprow (bool *doDuprow*) [inline]

Definition at line 150 of file ClpPresolve.hpp.

4.66.3.28 bool ClpPresolve::doSingletonColumn () const [inline]

Whether we want to do singleton column part of presolve.

Definition at line 155 of file ClpPresolve.hpp.

4.66.3.29 void ClpPresolve::setDoSingletonColumn (bool *doSingleton*) [inline]

Definition at line 158 of file ClpPresolve.hpp.

4.66.3.30 bool ClpPresolve::doGubrow () const [inline]

Whether we want to do gubrow part of presolve.

Definition at line 163 of file ClpPresolve.hpp.

4.66.3.31 void ClpPresolve::setDoGubrow (bool *doGubrow*) [inline]

Definition at line 166 of file ClpPresolve.hpp.

4.66.3.32 bool ClpPresolve::doTwoxTwo () const [inline]

Whether we want to do twoxtwo part of presolve.

Definition at line 171 of file ClpPresolve.hpp.

4.66.3.33 void ClpPresolve::setDoTwoxtwo (bool *doTwoxTwo*) [inline]

Definition at line 174 of file ClpPresolve.hpp.

4.66.3.34 bool ClpPresolve::doIntersection () const [inline]

Whether we want to allow duplicate intersections.

Definition at line 179 of file ClpPresolve.hpp.

4.66.3.35 void ClpPresolve::setDoIntersection (bool *doIntersection*) [inline]

Definition at line 182 of file ClpPresolve.hpp.

4.66.3.36 int ClpPresolve::presolveActions () const [inline]

Set whole group.

Definition at line 187 of file ClpPresolve.hpp.

4.66.3.37 void ClpPresolve::setPresolveActions (int *action*) [inline]

Definition at line 190 of file ClpPresolve.hpp.

4.66.3.38 void ClpPresolve::setSubstitution (int *value*) [inline]

Substitution level.

Definition at line 194 of file ClpPresolve.hpp.

4.66.3.39 `void ClpPresolve::statistics () [inline]`

Asks for statistics.

Definition at line 198 of file `ClpPresolve.hpp`.

4.66.3.40 `int ClpPresolve::presolveStatus () const`

Return presolve status (0,1,2)

4.66.3.41 `virtual void ClpPresolve::postsolve (bool updateStatus = true) [virtual]`

4.66.3.42 `void ClpPresolve::destroyPresolve ()`

Gets rid of presolve actions (e.g.when infeasible)

4.66.3.43 `virtual const CoinPresolveAction* ClpPresolve::presolve (CoinPresolveMatrix * prob) [protected],
[virtual]`

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.

4.66.3.44 `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.

4.66.3.45 `virtual ClpSimplex* ClpPresolve::gutsOfPresolvedModel (ClpSimplex * originalModel, double feasibilityTolerance,
bool keepIntegers, int numberPasses, bool dropNames, bool doRowObjective, const char * prohibitedRows = NULL,
const char * prohibitedColumns = NULL) [protected],[virtual]`

This is main part of Presolve.

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

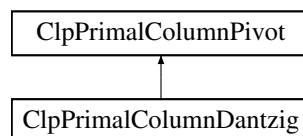
- [src/ClpPresolve.hpp](#)

4.67 ClpPrimalColumnDantzig Class Reference

Primal Column Pivot Dantzig Algorithm Class.

```
#include <ClpPrimalColumnDantzig.hpp>
```

Inheritance 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 Constructor & Destructor Documentation

4.67.2.1 [ClpPrimalColumnDantzig::ClpPrimalColumnDantzig](#) ()

Default Constructor.

4.67.2.2 [ClpPrimalColumnDantzig::ClpPrimalColumnDantzig](#) (const [ClpPrimalColumnDantzig](#) &)

Copy constructor.

4.67.2.3 [virtual ClpPrimalColumnDantzig::~~ClpPrimalColumnDantzig](#) () [virtual]

Destructor.

4.67.3 Member Function Documentation

4.67.3.1 [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](#).

4.67.3.2 `virtual void ClpPrimalColumnDantzig::saveWeights (ClpSimplex * model, int) [inline],[virtual]`

Just sets model.

Implements [ClpPrimalColumnPivot](#).

Definition at line 40 of file `ClpPrimalColumnDantzig.hpp`.

4.67.3.3 `ClpPrimalColumnDantzig& ClpPrimalColumnDantzig::operator= (const ClpPrimalColumnDantzig & rhs)`

Assignment operator.

4.67.3.4 `virtual ClpPrimalColumnPivot* ClpPrimalColumnDantzig::clone (bool copyData = true) const [virtual]`

Clone.

Implements [ClpPrimalColumnPivot](#).

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

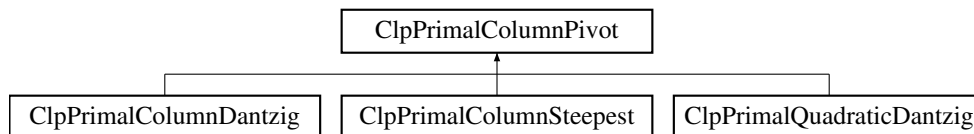
- [src/ClpPrimalColumnDantzig.hpp](#)

4.68 ClpPrimalColumnPivot Class Reference

Primal Column Pivot Abstract Base Class.

`#include <ClpPrimalColumnPivot.hpp>`

Inheritance 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 Constructor & Destructor Documentation

4.68.2.1 `ClpPrimalColumnPivot::ClpPrimalColumnPivot ()`

Default Constructor.

4.68.2.2 `ClpPrimalColumnPivot::ClpPrimalColumnPivot (const ClpPrimalColumnPivot &)`

Copy constructor.

4.68.2.3 `virtual ClpPrimalColumnPivot::~~ClpPrimalColumnPivot () [virtual]`

Destructor.

4.68.3 Member Function Documentation

4.68.3.1 `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](#).

4.68.3.2 `virtual void ClpPrimalColumnPivot::updateWeights (CoinIndexedVector * input) [virtual]`

Updates weights - part 1 (may be empty)

Reimplemented in [ClpPrimalColumnSteepest](#).

4.68.3.3 `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](#).

4.68.3.4 `virtual int ClpPrimalColumnPivot::pivotRow (double & way) [inline], [virtual]`

Signals pivot row choice: -2 (default) - use normal pivot row choice -1 to numberOfRows-1 - use this (will be checked) way should be -1 to go to lower bound, +1 to upper bound.

Definition at line 76 of file `ClpPrimalColumnPivot.hpp`.

4.68.3.5 `virtual void ClpPrimalColumnPivot::clearArrays () [virtual]`

Gets rid of all arrays (may be empty)

Reimplemented in [ClpPrimalColumnSteepest](#).

4.68.3.6 `virtual bool ClpPrimalColumnPivot::looksOptimal () const [inline],[virtual]`

Returns true if would not find any column.

Reimplemented in [ClpPrimalColumnSteepest](#).

Definition at line 83 of file `ClpPrimalColumnPivot.hpp`.

4.68.3.7 `virtual void ClpPrimalColumnPivot::setLooksOptimal (bool flag) [inline],[virtual]`

Sets optimality flag (for advanced use)

Definition at line 87 of file `ClpPrimalColumnPivot.hpp`.

4.68.3.8 `ClpPrimalColumnPivot& ClpPrimalColumnPivot::operator= (const ClpPrimalColumnPivot & rhs)`

Assignment operator.

4.68.3.9 `virtual ClpPrimalColumnPivot* ClpPrimalColumnPivot::clone (bool copyData = true) const [pure virtual]`

Clone.

Implemented in [ClpPrimalColumnSteepest](#), [ClpPrimalQuadraticDantzig](#), and [ClpPrimalColumnDantzig](#).

4.68.3.10 `ClpSimplex* ClpPrimalColumnPivot::model () [inline]`

Returns model.

Definition at line 115 of file `ClpPrimalColumnPivot.hpp`.

4.68.3.11 `void ClpPrimalColumnPivot::setModel (ClpSimplex * newmodel) [inline]`

Sets model.

Definition at line 119 of file `ClpPrimalColumnPivot.hpp`.

4.68.3.12 `int ClpPrimalColumnPivot::type () [inline]`

Returns type (above 63 is extra information)

Definition at line 124 of file `ClpPrimalColumnPivot.hpp`.

4.68.3.13 `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](#).

4.68.3.14 `virtual void ClpPrimalColumnPivot::switchOffSprint () [virtual]`

Switch off sprint idea.

Reimplemented in [ClpPrimalColumnSteepest](#).

4.68.3.15 `virtual void ClpPrimalColumnPivot::maximumPivotsChanged () [inline],[virtual]`

Called when maximum pivots changes.

Reimplemented in [ClpPrimalColumnSteepest](#).

Definition at line 135 of file `ClpPrimalColumnPivot.hpp`.

4.68.4 Member Data Documentation

4.68.4.1 `ClpSimplex* ClpPrimalColumnPivot::model_ [protected]`

Pointer to model.

Definition at line 145 of file `ClpPrimalColumnPivot.hpp`.

4.68.4.2 `int ClpPrimalColumnPivot::type_ [protected]`

Type of column pivot algorithm.

Definition at line 147 of file `ClpPrimalColumnPivot.hpp`.

4.68.4.3 `bool ClpPrimalColumnPivot::looksOptimal_ [protected]`

Says if looks optimal (normally computed)

Definition at line 149 of file `ClpPrimalColumnPivot.hpp`.

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

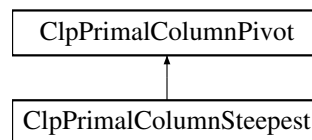
- [src/ClpPrimalColumnPivot.hpp](#)

4.69 ClpPrimalColumnSteepest Class Reference

Primal Column Pivot Steepest Edge Algorithm Class.

`#include <ClpPrimalColumnSteepest.hpp>`

Inheritance diagram for `ClpPrimalColumnSteepest`:



Public Types

- enum [Persistence](#) { `normal` = 0x00, `keep` = 0x01 }
- 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 Member Enumeration Documentation

4.69.2.1 enum `ClpPrimalColumnSteepest::Persistence`

enums for persistence

Enumerator

normal

keep

Definition at line 140 of file `ClpPrimalColumnSteepest.hpp`.

4.69.3 Constructor & Destructor Documentation

4.69.3.1 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.

4.69.3.2 ClpPrimalColumnSteepest::ClpPrimalColumnSteepest (const ClpPrimalColumnSteepest & *rhs*)

Copy constructor.

4.69.3.3 virtual ClpPrimalColumnSteepest::~~ClpPrimalColumnSteepest () [virtual]

Destructor.

4.69.4 Member Function Documentation

4.69.4.1 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](#).

4.69.4.2 int ClpPrimalColumnSteepest::pivotColumnOldMethod (CoinIndexedVector * *updates*, CoinIndexedVector * *spareRow1*, CoinIndexedVector * *spareRow2*, CoinIndexedVector * *spareColumn1*, CoinIndexedVector * *spareColumn2*)

For quadratic or funny nonlinearities.

4.69.4.3 void ClpPrimalColumnSteepest::justDjs (CoinIndexedVector * *updates*, CoinIndexedVector * *spareRow2*, CoinIndexedVector * *spareColumn1*, CoinIndexedVector * *spareColumn2*)

Just update djs.

4.69.4.4 int ClpPrimalColumnSteepest::partialPricing (CoinIndexedVector * *updates*, CoinIndexedVector * *spareRow2*, int *numberWanted*, int *numberLook*)

Update djs doing partial pricing (dantzig)

4.69.4.5 void ClpPrimalColumnSteepest::djsAndDevex (CoinIndexedVector * *updates*, CoinIndexedVector * *spareRow2*, CoinIndexedVector * *spareColumn1*, CoinIndexedVector * *spareColumn2*)

Update djs, weights for Devex using djs.

4.69.4.6 void ClpPrimalColumnSteepest::djsAndSteepest (CoinIndexedVector * *updates*, CoinIndexedVector * *spareRow2*, CoinIndexedVector * *spareColumn1*, CoinIndexedVector * *spareColumn2*)

Update djs, weights for Steepest using djs.

4.69.4.7 void ClpPrimalColumnSteepest::djsAndDevex2 (CoinIndexedVector * *updates*, CoinIndexedVector * *spareRow2*,
CoinIndexedVector * *spareColumn1*, CoinIndexedVector * *spareColumn2*)

Update djs, weights for Devex using pivot row.

4.69.4.8 void ClpPrimalColumnSteepest::djsAndSteepest2 (CoinIndexedVector * *updates*, CoinIndexedVector * *spareRow2*,
CoinIndexedVector * *spareColumn1*, CoinIndexedVector * *spareColumn2*)

Update djs, weights for Steepest using pivot row.

4.69.4.9 void ClpPrimalColumnSteepest::justDevex (CoinIndexedVector * *updates*, CoinIndexedVector * *spareRow2*,
CoinIndexedVector * *spareColumn1*, CoinIndexedVector * *spareColumn2*)

Update weights for Devex.

4.69.4.10 void ClpPrimalColumnSteepest::justSteepest (CoinIndexedVector * *updates*, CoinIndexedVector * *spareRow2*,
CoinIndexedVector * *spareColumn1*, CoinIndexedVector * *spareColumn2*)

Update weights for Steepest.

4.69.4.11 void ClpPrimalColumnSteepest::transposeTimes2 (const CoinIndexedVector * *pi1*, CoinIndexedVector * *dj1*, const
CoinIndexedVector * *pi2*, CoinIndexedVector * *dj2*, CoinIndexedVector * *spare*, double *scaleFactor*)

Updates two arrays for steepest.

4.69.4.12 virtual void ClpPrimalColumnSteepest::updateWeights (CoinIndexedVector * *input*) [virtual]

Updates weights - part 1 - also checks accuracy.

Reimplemented from [ClpPrimalColumnPivot](#).

4.69.4.13 void ClpPrimalColumnSteepest::checkAccuracy (int *sequence*, double *relativeTolerance*, CoinIndexedVector *
rowArray1, CoinIndexedVector * *rowArray2*)

Checks accuracy - just for debug.

4.69.4.14 void ClpPrimalColumnSteepest::initializeWeights ()

Initialize weights.

4.69.4.15 virtual void ClpPrimalColumnSteepest::saveWeights (ClpSimplex * *model*, int *mode*) [virtual]

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)

Implements [ClpPrimalColumnPivot](#).

4.69.4.16 virtual void ClpPrimalColumnSteepest::unrollWeights () [virtual]

Gets rid of last update.

4.69.4.17 virtual void ClpPrimalColumnSteepest::clearArrays () [virtual]

Gets rid of all arrays.

Reimplemented from [ClpPrimalColumnPivot](#).

4.69.4.18 `virtual bool ClpPrimalColumnSteepest::looksOptimal () const [virtual]`

Returns true if would not find any column.

Reimplemented from [ClpPrimalColumnPivot](#).

4.69.4.19 `virtual void ClpPrimalColumnSteepest::maximumPivotsChanged () [virtual]`

Called when maximum pivots changes.

Reimplemented from [ClpPrimalColumnPivot](#).

4.69.4.20 `int ClpPrimalColumnSteepest::mode () const [inline]`

Mode.

Definition at line 126 of file `ClpPrimalColumnSteepest.hpp`.

4.69.4.21 `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](#).

4.69.4.22 `virtual void ClpPrimalColumnSteepest::switchOffSprint () [virtual]`

Switch off sprint idea.

Reimplemented from [ClpPrimalColumnPivot](#).

4.69.4.23 `ClpPrimalColumnSteepest& ClpPrimalColumnSteepest::operator= (const ClpPrimalColumnSteepest & rhs)`

Assignment operator.

4.69.4.24 `virtual ClpPrimalColumnPivot* ClpPrimalColumnSteepest::clone (bool copyData = true) const [virtual]`

Clone.

Implements [ClpPrimalColumnPivot](#).

4.69.4.25 `bool ClpPrimalColumnSteepest::reference (int i) const [inline]`

reference would be faster using [ClpSimplex](#)'s `status_`, but I prefer to keep modularity.

Definition at line 175 of file `ClpPrimalColumnSteepest.hpp`.

4.69.4.26 `void ClpPrimalColumnSteepest::setReference (int i, bool trueFalse) [inline]`

Definition at line 178 of file `ClpPrimalColumnSteepest.hpp`.

4.69.4.27 `void ClpPrimalColumnSteepest::setPersistence (Persistence life) [inline]`

Set/ get persistence.

Definition at line 187 of file `ClpPrimalColumnSteepest.hpp`.

4.69.4.28 `Persistence ClpPrimalColumnSteepest::persistence () const [inline]`

Definition at line 190 of file `ClpPrimalColumnSteepest.hpp`.

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

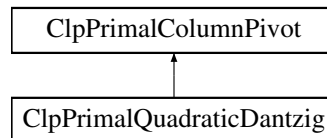
- [src/ClpPrimalColumnSteepest.hpp](#)

4.70 ClpPrimalQuadraticDantzig Class Reference

Primal Column Pivot Dantzig Algorithm Class.

```
#include <ClpPrimalQuadraticDantzig.hpp>
```

Inheritance 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 Constructor & Destructor Documentation

4.70.2.1 ClpPrimalQuadraticDantzig::ClpPrimalQuadraticDantzig ()

Default Constructor.

4.70.2.2 ClpPrimalQuadraticDantzig::ClpPrimalQuadraticDantzig (const ClpPrimalQuadraticDantzig &)

Copy constructor.

4.70.2.3 ClpPrimalQuadraticDantzig::ClpPrimalQuadraticDantzig (ClpSimplexPrimalQuadratic * *model*, ClpQuadraticInfo * *info*)

Constructor from model.

4.70.2.4 virtual ClpPrimalQuadraticDantzig::~~ClpPrimalQuadraticDantzig () [virtual]

Destructor.

4.70.3 Member Function Documentation

4.70.3.1 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](#).

4.70.3.2 virtual void ClpPrimalQuadraticDantzig::saveWeights (ClpSimplex * *model*, int *mode*) [inline], [virtual]

Just sets model.

Implements [ClpPrimalColumnPivot](#).

Definition at line 39 of file ClpPrimalQuadraticDantzig.hpp.

4.70.3.3 ClpPrimalQuadraticDantzig& ClpPrimalQuadraticDantzig::operator= (const ClpPrimalQuadraticDantzig & *rhs*)

Assignment operator.

4.70.3.4 virtual ClpPrimalColumnPivot* ClpPrimalQuadraticDantzig::clone (bool *copyData* = true) const [virtual]

Clone.

Implements [ClpPrimalColumnPivot](#).

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

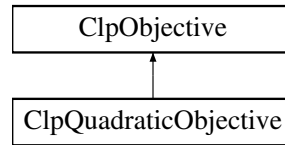
- [src/ClpPrimalQuadraticDantzig.hpp](#)

4.71 ClpQuadraticObjective Class Reference

Quadratic Objective Class.

```
#include <ClpQuadraticObjective.hpp>
```

Inheritance 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 ¤tObj, 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 ()

Default Constructor.

4.71.2.2 ClpQuadraticObjective::ClpQuadraticObjective (const double * *linearObjective*, int *numberColumns*, const CoinBigIndex * *start*, const int * *column*, const double * *element*, int *numberExtendedColumns_* = -1)

Constructor from objective.

4.71.2.3 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

4.71.2.4 ClpQuadraticObjective::ClpQuadraticObjective (const ClpQuadraticObjective & *rhs*, int *numberColumns*, const int * *whichColumns*)

Subset constructor.

Duplicates are allowed and order is as given.

4.71.2.5 virtual ClpQuadraticObjective::~~ClpQuadraticObjective () [virtual]

Destructor.

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 double ClpQuadraticObjective::objectiveValue (const ClpSimplex * model, const double * solution) const` [virtual]

Return objective value (without any [ClpModel](#) offset) (model may be NULL)

Implements [ClpObjective](#).

4.71.3.5 `virtual void ClpQuadraticObjective::resize (int newNumberColumns)` [virtual]

Resize objective.

Implements [ClpObjective](#).

4.71.3.6 `virtual void ClpQuadraticObjective::deleteSome (int numberToDelete, const int * which)` [virtual]

Delete columns in objective.

Implements [ClpObjective](#).

4.71.3.7 `virtual void ClpQuadraticObjective::reallyScale (const double * columnScale)` [virtual]

Scale objective.

Implements [ClpObjective](#).

4.71.3.8 `virtual int ClpQuadraticObjective::markNonlinear (char * which)` [virtual]

Given a zeroed array sets nonlinear columns to 1.

Returns number of nonlinear columns

Reimplemented from [ClpObjective](#).

4.71.3.9 ClpQuadraticObjective& ClpQuadraticObjective::operator= (const ClpQuadraticObjective & rhs)

Assignment operator.

4.71.3.10 virtual ClpObjective* ClpQuadraticObjective::clone () const [virtual]

Clone.

Implements [ClpObjective](#).

4.71.3.11 virtual ClpObjective* ClpQuadraticObjective::subsetClone (int *numberOfColumns*, const int * *whichColumns*) const [virtual]

Subset clone.

Duplicates are allowed and order is as given.

Reimplemented from [ClpObjective](#).

4.71.3.12 void ClpQuadraticObjective::loadQuadraticObjective (const int *numberOfColumns*, const CoinBigIndex * *start*, const int * *column*, const double * *element*, int *numberOfExtendedColumns* = -1)

Load up quadratic objective.

This is stored as a CoinPackedMatrix

4.71.3.13 void ClpQuadraticObjective::loadQuadraticObjective (const CoinPackedMatrix & *matrix*)**4.71.3.14 void ClpQuadraticObjective::deleteQuadraticObjective ()**

Get rid of quadratic objective.

4.71.3.15 CoinPackedMatrix* ClpQuadraticObjective::quadraticObjective () const [inline]

Quadratic objective.

Definition at line 115 of file [ClpQuadraticObjective.hpp](#).

4.71.3.16 double* ClpQuadraticObjective::linearObjective () const [inline]

Linear objective.

Definition at line 119 of file [ClpQuadraticObjective.hpp](#).

4.71.3.17 int ClpQuadraticObjective::numberOfExtendedColumns () const [inline]

Length of linear objective which could be bigger.

Definition at line 123 of file [ClpQuadraticObjective.hpp](#).

4.71.3.18 int ClpQuadraticObjective::numberOfColumns () const [inline]

Number of columns in quadratic objective.

Definition at line 127 of file [ClpQuadraticObjective.hpp](#).

4.71.3.19 bool ClpQuadraticObjective::fullMatrix () const [inline]

If a full or half matrix.

Definition at line 131 of file [ClpQuadraticObjective.hpp](#).

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

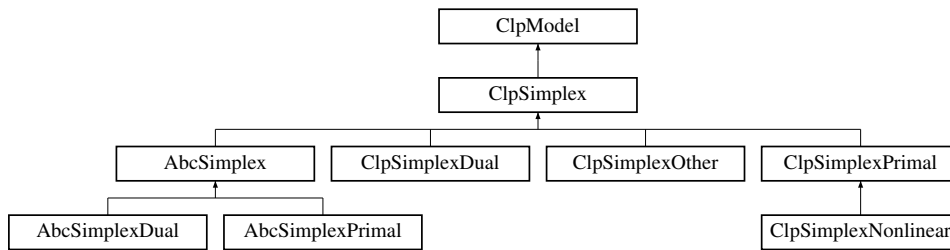
- [src/ClpQuadraticObjective.hpp](#)

4.72 ClpSimplex Class Reference

This solves LPs using the simplex method.

```
#include <ClpSimplex.hpp>
```

Inheritance diagram for ClpSimplex:



Public Types

- enum `Status` {
`isFree` = 0x00, `basic` = 0x01, `atUpperBound` = 0x02, `atLowerBound` = 0x03,
`superBasic` = 0x04, `isFixed` = 0x05 }
enums for status of various sorts.
- enum `FakeBound` { `noFake` = 0x00, `lowerFake` = 0x01, `upperFake` = 0x02, `bothFake` = 0x03 }

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 deltaTolerance)
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 numberOfRows and numberOfColumns 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 [scaleRealObjective](#) (double multiplier)
Scale real objective.
- void [scaleRealRhs](#) (double maxValue, double killIfSmaller)
Scale so no RHS (abs not infinite) > value.

- 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.
- 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 incoming 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 debug 32768 bit - do dual in netlibd 65536 (*3) initial stateDualColumn 262144 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 debug 32768 bit - do dual in netlibd 65536 (*3) initial stateDualColumn 262144 bit - stop when primal feasible.*

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 `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 `getBlInvARow` (int row, double *z, double *slack=NULL)
Get a row of the tableau (slack part in slack if not NULL)
- void `getBlInvRow` (int row, double *z)
Get a row of the basis inverse.
- void `getBlInvACol` (int col, double *vec)
Get a column of the tableau.

- void [getBlvCol](#) (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 returned by [getBlvACol\(\)](#) and [getBlvCol\(\)](#)).

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 *columnActivities=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)
Solve using Dantzig-Wolfe decomposition and maybe in parallel.
- int `solveBenders` (CoinStructuredModel *model)
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 ($< \text{largeTolerance}$)
- 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 primal.*

 - double [acceptablePivot_](#)
- Acceptable pivot value just after factorization.*

 - 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 [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.

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 55 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

Enumerator

isFree
basic
atUpperBound
atLowerBound
superBasic
isFixed

Definition at line 63 of file ClpSimplex.hpp.

4.72.2.2 enum ClpSimplex::FakeBound

Enumerator

noFake
lowerFake
upperFake
bothFake

Definition at line 72 of file ClpSimplex.hpp.

4.72.3 Constructor & Destructor Documentation

4.72.3.1 ClpSimplex::ClpSimplex (bool *emptyMessages* = false)

Default constructor.

4.72.3.2 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)

4.72.3.3 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)

4.72.3.4 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)

4.72.3.5 `ClpSimplex::ClpSimplex (const ClpSimplex * wholeModel, int numberOfRows, const int * whichRows, int numberOfColumns, 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`)

4.72.3.6 `ClpSimplex::ClpSimplex (ClpSimplex * wholeModel, int numberOfColumns, 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`

4.72.3.7 `ClpSimplex::~ClpSimplex ()`

Destructor.

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 `int ClpSimplex::abcState () const` `[inline]`

Definition at line 124 of file `ClpSimplex.hpp`.

4.72.4.3 `void ClpSimplex::setAbcState (int state)` `[inline]`

Definition at line 126 of file `ClpSimplex.hpp`.

4.72.4.4 `void ClpSimplex::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.

4.72.4.5 `void ClpSimplex::makeBaseModel ()`

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

4.72.4.6 `void ClpSimplex::deleteBaseModel ()`

Switch off base model.

4.72.4.7 `ClpSimplex* ClpSimplex::baseModel () const` `[inline]`

See if we have base model.

Definition at line 149 of file `ClpSimplex.hpp`.

4.72.4.8 `void ClpSimplex::setToBaseModel (ClpSimplex * model = NULL)`

Reset to base model (just size and arrays needed) If model NULL use internal copy.

4.72.4.9 `ClpSimplex& ClpSimplex::operator= (const ClpSimplex & rhs)`

Assignment operator. This copies the data.

4.72.4.10 `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

4.72.4.11 `void ClpSimplex::loadProblem (const CoinPackedMatrix & matrix, const double * collb, const double * colub, const double * obj, const double * rowlb, const double * rowub, const double * rowObjective = NULL)`

4.72.4.12 `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).

4.72.4.13 `void ClpSimplex::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.

4.72.4.14 `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

4.72.4.15 `int ClpSimplex::readMps (const char * filename, bool keepNames = false, bool ignoreErrors = false)`

Read an mps file from the given filename.

4.72.4.16 `int ClpSimplex::readGMPL (const char * filename, const char * dataName, bool keepNames = false)`

Read GMPL files from the given filenames.

4.72.4.17 `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.

4.72.4.18 `void ClpSimplex::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. This is same as [ClpModel](#) one, but sets scaling on etc.

4.72.4.19 void `ClpSimplex::borrowModel (ClpSimplex & otherModel)`

4.72.4.20 void `ClpSimplex::passInEventHandler (const ClpEventHandler * eventHandler)`

Pass in Event handler (cloned and deleted at end)

4.72.4.21 void `ClpSimplex::getbackSolution (const ClpSimplex & smallModel, const int * whichRow, const int * whichColumn)`

Puts solution back into small model.

4.72.4.22 int `ClpSimplex::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.

4.72.4.23 int `ClpSimplex::initialSolve (ClpSolve & options)`

General solve algorithm which can do presolve.

See [ClpSolve.hpp](#) for options

4.72.4.24 int `ClpSimplex::initialSolve ()`

Default initial solve.

4.72.4.25 int `ClpSimplex::initialDualSolve ()`

Dual initial solve.

4.72.4.26 int `ClpSimplex::initialPrimalSolve ()`

Primal initial solve.

4.72.4.27 int `ClpSimplex::initialBarrierSolve ()`

Barrier initial solve.

4.72.4.28 int `ClpSimplex::initialBarrierNoCrossSolve ()`

Barrier initial solve, not to be followed by crossover.

4.72.4.29 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

4.72.4.30 int `ClpSimplex::dualDebug (int ifValuesPass = 0, int startFinishOptions = 0)`

4.72.4.31 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

4.72.4.32 `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

4.72.4.33 `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

4.72.4.34 `int ClpSimplex::barrier (bool crossover = true)`

Solves using barrier (assumes you have good cholesky factor code).

Does crossover to simplex if asked

4.72.4.35 `int ClpSimplex::reducedGradient (int phase = 0)`

Solves non-linear using reduced gradient.

Phase = 0 get feasible, =1 use solution

4.72.4.36 `int ClpSimplex::solve (CoinStructuredModel * model)`

Solve using structure of model and maybe in parallel.

4.72.4.37 `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 keepSolution true and size is same as current then keeps current status and solution

4.72.4.38 `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

4.72.4.39 `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

4.72.4.40 `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 variables 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

4.72.4.41 `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` \geq `numberColumns` are slacks (obviously no coefficients) status array is (char) Status enum

4.72.4.42 `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

4.72.4.43 `double ClpSimplex::moveTowardsPrimalFeasible ()`

Try simple crash like techniques to get closer to primal feasibility returns final sum of infeasibilities.

4.72.4.44 `void ClpSimplex::removeSuperBasicSlacks (int threshold = 0)`

Try simple crash like techniques to remove super basic slacks but only if $>$ threshold.

4.72.4.45 `ClpSimplex* 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.

4.72.4.46 void ClpSimplex::miniPostsolve (const ClpSimplex * *presolvedModel*, void * *info*)

After mini presolve.

4.72.4.47 void ClpSimplex::scaleRealObjective (double *multiplier*)

Scale real objective.

4.72.4.48 void ClpSimplex::scaleRealRhs (double *maxValue*, double *killIfSmaller*)

Scale so no RHS (abs not infinite) $>$ value.

4.72.4.49 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

4.72.4.50 int ClpSimplex::readBasis (const char * *filename*)

Read a basis from the given filename, returns -1 on file error, 0 if no values, 1 if values.

4.72.4.51 CoinWarmStartBasis* ClpSimplex::getBasis () const

Returns a basis (to be deleted by user)

4.72.4.52 void ClpSimplex::setFactorization (ClpFactorization & *factorization*)

Passes in factorization.

4.72.4.53 ClpFactorization* ClpSimplex::swapFactorization (ClpFactorization * *factorization*)

4.72.4.54 void ClpSimplex::copyFactorization (ClpFactorization & *factorization*)

Copies in factorization to existing one.

4.72.4.55 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)

4.72.4.56 `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

4.72.4.57 `void ClpSimplex::setDualRowPivotAlgorithm (ClpDualRowPivot & choice)`

Sets row pivot choice algorithm in dual.

4.72.4.58 `void ClpSimplex::setPrimalColumnPivotAlgorithm (ClpPrimalColumnPivot & choice)`

Sets column pivot choice algorithm in primal.

4.72.4.59 `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

4.72.4.60 `int ClpSimplex::fathom (void * stuff)`

Fathom - 1 if solution.

4.72.4.61 `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)

4.72.4.62 `double ClpSimplex::doubleCheck ()`

Double checks OK.

4.72.4.63 `int ClpSimplex::startFastDual2 (ClpNodeStuff * stuff)`

Starts Fast dual2.

4.72.4.64 `int ClpSimplex::fastDual2 (ClpNodeStuff * stuff)`

Like Fast dual.

4.72.4.65 `void ClpSimplex::stopFastDual2 (ClpNodeStuff * stuff)`

Stops Fast dual2.

4.72.4.66 `ClpSimplex* 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.

4.72.4.67 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.

4.72.4.68 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

4.72.4.69 int ClpSimplex::dualPivotResultPart1 ()

Pivot out a variable and choose an incoming 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

4.72.4.70 int ClpSimplex::pivotResultPart2 (int *algorithm*, int *state*)

Do actual pivot state is 0 if need tableau column, 1 if in rowArray_[1].

4.72.4.71 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

4.72.4.72 void ClpSimplex::finish (int *startFinishOptions* = 0)**4.72.4.73 bool ClpSimplex::statusOfProblem (bool *initial* = false)**

Factorizes and returns true if optimal.

Used by user

4.72.4.74 void ClpSimplex::defaultFactorizationFrequency ()

If user left factorization frequency then compute.

4.72.4.75 void ClpSimplex::copyEnabledStuff (const ClpSimplex * *rhs*)

Copy across enabled stuff from one solver to another.

4.72.4.76 bool ClpSimplex::primalFeasible () const [inline]

If problem is primal feasible.

Definition at line 578 of file ClpSimplex.hpp.

4.72.4.77 bool ClpSimplex::dualFeasible () const [inline]

If problem is dual feasible.

Definition at line 582 of file ClpSimplex.hpp.

4.72.4.78 `ClpFactorization* ClpSimplex::factorization () const` `[inline]`

factorization

Definition at line 586 of file ClpSimplex.hpp.

4.72.4.79 `bool ClpSimplex::sparseFactorization () const`

Sparsity on or off.

4.72.4.80 `void ClpSimplex::setSparseFactorization (bool value)`

4.72.4.81 `int ClpSimplex::factorizationFrequency () const`

Factorization frequency.

4.72.4.82 `void ClpSimplex::setFactorizationFrequency (int value)`

4.72.4.83 `double ClpSimplex::dualBound () const` `[inline]`

Dual bound.

Definition at line 596 of file ClpSimplex.hpp.

4.72.4.84 `void ClpSimplex::setDualBound (double value)`

4.72.4.85 `double ClpSimplex::infeasibilityCost () const` `[inline]`

Infeasibility cost.

Definition at line 601 of file ClpSimplex.hpp.

4.72.4.86 `void ClpSimplex::setInfeasibilityCost (double value)`

4.72.4.87 `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 621 of file ClpSimplex.hpp.

4.72.4.88 `void ClpSimplex::setPerturbation (int value)`

4.72.4.89 `int ClpSimplex::algorithm () const` `[inline]`

Current (or last) algorithm.

Definition at line 626 of file ClpSimplex.hpp.

4.72.4.90 `void ClpSimplex::setAlgorithm (int value)` `[inline]`

Set algorithm.

Definition at line 630 of file ClpSimplex.hpp.

4.72.4.91 `bool ClpSimplex::isObjectiveLimitTestValid () const`

Return true if the objective limit test can be relied upon.

4.72.4.92 `double ClpSimplex::sumDualInfeasibilities () const [inline]`

Sum of dual infeasibilities.

Definition at line 636 of file ClpSimplex.hpp.

4.72.4.93 `void ClpSimplex::setSumDualInfeasibilities (double value) [inline]`

Definition at line 639 of file ClpSimplex.hpp.

4.72.4.94 `double ClpSimplex::sumOfRelaxedDualInfeasibilities () const [inline]`

Sum of relaxed dual infeasibilities.

Definition at line 643 of file ClpSimplex.hpp.

4.72.4.95 `void ClpSimplex::setSumOfRelaxedDualInfeasibilities (double value) [inline]`

Definition at line 646 of file ClpSimplex.hpp.

4.72.4.96 `int ClpSimplex::numberDualInfeasibilities () const [inline]`

Number of dual infeasibilities.

Definition at line 650 of file ClpSimplex.hpp.

4.72.4.97 `void ClpSimplex::setNumberDualInfeasibilities (int value) [inline]`

Definition at line 653 of file ClpSimplex.hpp.

4.72.4.98 `int ClpSimplex::numberDualInfeasibilitiesWithoutFree () const [inline]`

Number of dual infeasibilities (without free)

Definition at line 657 of file ClpSimplex.hpp.

4.72.4.99 `double ClpSimplex::sumPrimalInfeasibilities () const [inline]`

Sum of primal infeasibilities.

Definition at line 661 of file ClpSimplex.hpp.

4.72.4.100 `void ClpSimplex::setSumPrimalInfeasibilities (double value) [inline]`

Definition at line 664 of file ClpSimplex.hpp.

4.72.4.101 `double ClpSimplex::sumOfRelaxedPrimalInfeasibilities () const [inline]`

Sum of relaxed primal infeasibilities.

Definition at line 668 of file ClpSimplex.hpp.

4.72.4.102 `void ClpSimplex::setSumOfRelaxedPrimalInfeasibilities (double value) [inline]`

Definition at line 671 of file ClpSimplex.hpp.

4.72.4.103 `int ClpSimplex::numberPrimalInfeasibilities () const [inline]`

Number of primal infeasibilities.

Definition at line 675 of file ClpSimplex.hpp.

4.72.4.104 `void ClpSimplex::setNumberPrimalInfeasibilities (int value) [inline]`

Definition at line 678 of file ClpSimplex.hpp.

4.72.4.105 `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.

4.72.4.106 `int ClpSimplex::restoreModel (const char * fileName)`

Restore model from file, returns 0 if success, deletes current model.

4.72.4.107 `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.

4.72.4.108 `void ClpSimplex::checkSolutionInternal ()`

Just check solution (for internal use) - sets sum of infeasibilities etc.

4.72.4.109 `void ClpSimplex::checkUnscaledSolution ()`

Check unscaled primal solution but allow for rounding error.

4.72.4.110 `CoinIndexedVector* ClpSimplex::rowArray (int index) const [inline]`

Useful row length arrays (0,1,2,3,4,5)

Definition at line 706 of file ClpSimplex.hpp.

4.72.4.111 `CoinIndexedVector* ClpSimplex::columnArray (int index) const [inline]`

Useful column length arrays (0,1,2,3,4,5)

Definition at line 710 of file ClpSimplex.hpp.

4.72.4.112 `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

4.72.4.113 `int ClpSimplex::getSolution ()`

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

Uses current problem arrays for bounds. Returns feasibility states

4.72.4.114 `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

4.72.4.115 `ClpDualRowPivot* ClpSimplex::dualRowPivot () const [inline]`

dual row pivot choice

Definition at line 736 of file ClpSimplex.hpp.

4.72.4.116 `ClpPrimalColumnPivot* ClpSimplex::primalColumnPivot () const [inline]`

primal column pivot choice

Definition at line 740 of file ClpSimplex.hpp.

4.72.4.117 `bool ClpSimplex::goodAccuracy () const [inline]`

Returns true if model looks OK.

Definition at line 744 of file ClpSimplex.hpp.

4.72.4.118 `void ClpSimplex::returnModel (ClpSimplex & otherModel)`

Return model - updates any scalars.

4.72.4.119 `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.

4.72.4.120 `ClpDataSave ClpSimplex::saveData ()`

Save data.

4.72.4.121 `void ClpSimplex::restoreData (ClpDataSave saved)`

Restore data.

4.72.4.122 `void ClpSimplex::cleanStatus ()`

Clean up status.

4.72.4.123 `int ClpSimplex::factorize ()`

Factorizes using current basis. For external use.

4.72.4.124 `void ClpSimplex::computeDuals (double * givenDjs)`

Computes duals from scratch.

If *givenDjs* then allows for nonzero basic djs

4.72.4.125 void ClpSimplex::computePrimals (const double * *rowActivities*, const double * *columnActivities*)

Computes primals from scratch.

4.72.4.126 void ClpSimplex::add (double * *array*, int *column*, double *multiplier*) const

Adds multiple of a column into an array.

4.72.4.127 void ClpSimplex::unpack (CoinIndexedVector * *rowArray*) const

Unpacks one column of the matrix into indexed array Uses sequenceIn_ Also applies scaling if needed.

4.72.4.128 void ClpSimplex::unpack (CoinIndexedVector * *rowArray*, int *sequence*) const

Unpacks one column of the matrix into indexed array Slack if sequence >= numberColumns Also applies scaling if needed.

4.72.4.129 void ClpSimplex::unpackPacked (CoinIndexedVector * *rowArray*)

Unpacks one column of the matrix into indexed array as packed vector Uses sequenceIn_ Also applies scaling if needed.

4.72.4.130 void ClpSimplex::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.

4.72.4.131 int ClpSimplex::housekeeping (double *objectiveChange*) [protected]

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

Can also decide to re-factorize

4.72.4.132 void ClpSimplex::checkPrimalSolution (const double * *rowActivities* = NULL, const double * *columnActivities* = NULL) [protected]

This sets largest infeasibility and most infeasible and sum and number of infeasibilities (Primal)

4.72.4.133 void ClpSimplex::checkDualSolution () [protected]

This sets largest infeasibility and most infeasible and sum and number of infeasibilities (Dual)

4.72.4.134 void ClpSimplex::checkBothSolutions () [protected]

This sets sum and number of infeasibilities (Dual and Primal)

4.72.4.135 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

4.72.4.136 int ClpSimplex::solveDW (CoinStructuredModel * *model*) [protected]

Solve using Dantzig-Wolfe decomposition and maybe in parallel.

4.72.4.137 int ClpSimplex::solveBenders (CoinStructuredModel * *model*) [protected]

Solve using Benders decomposition and maybe in parallel.

4.72.4.138 `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 although you may end up with an all slack basis.

Defaults are 1.0,10.0

4.72.4.139 `int ClpSimplex::cleanFactorization (int ifValuesPass)`

Get a clean factorization - i.e.

throw out singularities may do more later

4.72.4.140 `double ClpSimplex::alphaAccuracy () const [inline]`

Initial value for alpha accuracy calculation (-1.0 off)

Definition at line 847 of file ClpSimplex.hpp.

4.72.4.141 `void ClpSimplex::setAlphaAccuracy (double value) [inline]`

Definition at line 850 of file ClpSimplex.hpp.

4.72.4.142 `void ClpSimplex::setDisasterHandler (ClpDisasterHandler * handler) [inline]`

Objective value.

Set disaster handler

Definition at line 859 of file ClpSimplex.hpp.

4.72.4.143 `ClpDisasterHandler* ClpSimplex::disasterHandler () const [inline]`

Get disaster handler.

Definition at line 863 of file ClpSimplex.hpp.

4.72.4.144 `double ClpSimplex::largeValue () const [inline]`

Large bound value (for complementarity etc)

Definition at line 867 of file ClpSimplex.hpp.

4.72.4.145 `void ClpSimplex::setLargeValue (double value)`

4.72.4.146 `double ClpSimplex::largestPrimalError () const [inline]`

Largest error on Ax=b.

Definition at line 872 of file ClpSimplex.hpp.

4.72.4.147 `double ClpSimplex::largestDualError () const [inline]`

Largest error on basic duals.

Definition at line 876 of file ClpSimplex.hpp.

4.72.4.148 void ClpSimplex::setLargestPrimalError (double *value*) [inline]

Largest error on Ax-b.

Definition at line 880 of file ClpSimplex.hpp.

4.72.4.149 void ClpSimplex::setLargestDualError (double *value*) [inline]

Largest error on basic duals.

Definition at line 884 of file ClpSimplex.hpp.

4.72.4.150 double ClpSimplex::zeroTolerance () const [inline]

Get zero tolerance.

Definition at line 888 of file ClpSimplex.hpp.

4.72.4.151 void ClpSimplex::setZeroTolerance (double *value*) [inline]

Set zero tolerance.

Definition at line 892 of file ClpSimplex.hpp.

4.72.4.152 int* ClpSimplex::pivotVariable () const [inline]

Basic variables pivoting on which rows.

Definition at line 896 of file ClpSimplex.hpp.

4.72.4.153 bool ClpSimplex::automaticScaling () const [inline]

If automatic scaling on.

Definition at line 900 of file ClpSimplex.hpp.

4.72.4.154 void ClpSimplex::setAutomaticScaling (bool *onOff*) [inline]

Definition at line 903 of file ClpSimplex.hpp.

4.72.4.155 double ClpSimplex::currentDualTolerance () const [inline]

Current dual tolerance.

Definition at line 907 of file ClpSimplex.hpp.

4.72.4.156 void ClpSimplex::setCurrentDualTolerance (double *value*) [inline]

Definition at line 910 of file ClpSimplex.hpp.

4.72.4.157 double ClpSimplex::currentPrimalTolerance () const [inline]

Current primal tolerance.

Definition at line 914 of file ClpSimplex.hpp.

4.72.4.158 void ClpSimplex::setCurrentPrimalTolerance (double *value*) [inline]

Definition at line 917 of file ClpSimplex.hpp.

4.72.4.159 `int ClpSimplex::numberRefinements () const [inline]`

How many iterative refinements to do.

Definition at line 921 of file ClpSimplex.hpp.

4.72.4.160 `void ClpSimplex::setNumberRefinements (int value)`

4.72.4.161 `double ClpSimplex::alpha () const [inline]`

Alpha (pivot element) for use by classes e.g. `steepestedge`.

Definition at line 926 of file ClpSimplex.hpp.

4.72.4.162 `void ClpSimplex::setAlpha (double value) [inline]`

Definition at line 929 of file ClpSimplex.hpp.

4.72.4.163 `double ClpSimplex::dualIn () const [inline]`

Reduced cost of last incoming for use by classes e.g. `steepestedge`.

Definition at line 933 of file ClpSimplex.hpp.

4.72.4.164 `void ClpSimplex::setDualIn (double value) [inline]`

Set reduced cost of last incoming to force error.

Definition at line 937 of file ClpSimplex.hpp.

4.72.4.165 `int ClpSimplex::pivotRow () const [inline]`

Pivot Row for use by classes e.g. `steepestedge`.

Definition at line 941 of file ClpSimplex.hpp.

4.72.4.166 `void ClpSimplex::setPivotRow (int value) [inline]`

Definition at line 944 of file ClpSimplex.hpp.

4.72.4.167 `double ClpSimplex::valueIncomingDual () const`

value of incoming variable (in Dual)

4.72.4.168 `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 overridden by given pi and solution

4.72.4.169 `void ClpSimplex::gutsOfDelete (int type) [protected]`

Does most of deletion (0 = all, 1 = most, 2 most + factorization)

4.72.4.170 `void ClpSimplex::gutsOfCopy (const ClpSimplex & rhs) [protected]`

Does most of copying.

4.72.4.171 `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_

4.72.4.172 `void ClpSimplex::createRim1 (bool initial)` [protected]

Does rows and columns.

4.72.4.173 `void ClpSimplex::createRim4 (bool initial)` [protected]

Does objective.

4.72.4.174 `void ClpSimplex::createRim5 (bool initial)` [protected]

Does rows and columns and objective.

4.72.4.175 `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

4.72.4.176 `bool ClpSimplex::sanityCheck ()` [protected]

Sanity check on input rim data (after scaling) - returns true if okay.

4.72.4.177 `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 997 of file ClpSimplex.hpp.

4.72.4.178 `double* ClpSimplex::djRegion (int section) const` [inline]

Definition at line 1001 of file ClpSimplex.hpp.

4.72.4.179 `double* ClpSimplex::lowerRegion (int section) const` [inline]

Definition at line 1005 of file ClpSimplex.hpp.

4.72.4.180 `double* ClpSimplex::upperRegion (int section) const` [inline]

Definition at line 1009 of file ClpSimplex.hpp.

4.72.4.181 `double* ClpSimplex::costRegion (int section) const` [inline]

Definition at line 1013 of file ClpSimplex.hpp.

4.72.4.182 `double* ClpSimplex::solutionRegion () const` [inline]

Return region as single array.

Definition at line 1018 of file ClpSimplex.hpp.

4.72.4.183 `double* ClpSimplex::djRegion () const [inline]`

Definition at line 1021 of file ClpSimplex.hpp.

4.72.4.184 `double* ClpSimplex::lowerRegion () const [inline]`

Definition at line 1024 of file ClpSimplex.hpp.

4.72.4.185 `double* ClpSimplex::upperRegion () const [inline]`

Definition at line 1027 of file ClpSimplex.hpp.

4.72.4.186 `double* ClpSimplex::costRegion () const [inline]`

Definition at line 1030 of file ClpSimplex.hpp.

4.72.4.187 `Status ClpSimplex::getStatus (int sequence) const [inline]`

Definition at line 1033 of file ClpSimplex.hpp.

4.72.4.188 `void ClpSimplex::setStatus (int sequence, Status newstatus) [inline]`

Definition at line 1036 of file ClpSimplex.hpp.

4.72.4.189 `bool ClpSimplex::startPermanentArrays ()`

Start or reset using `maximumRows_` and `Columns_` - true if change.

4.72.4.190 `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

4.72.4.191 `bool ClpSimplex::initialDenseFactorization () const`

4.72.4.192 `int ClpSimplex::sequenceIn () const [inline]`

Return sequence In or Out.

Definition at line 1050 of file ClpSimplex.hpp.

4.72.4.193 `int ClpSimplex::sequenceOut () const [inline]`

Definition at line 1053 of file ClpSimplex.hpp.

4.72.4.194 `void ClpSimplex::setSequenceIn (int sequence) [inline]`

Set sequenceIn or Out.

Definition at line 1057 of file ClpSimplex.hpp.

4.72.4.195 `void ClpSimplex::setSequenceOut (int sequence) [inline]`

Definition at line 1060 of file ClpSimplex.hpp.

4.72.4.196 `int ClpSimplex::directionIn () const [inline]`

Return direction In or Out.

Definition at line 1064 of file ClpSimplex.hpp.

4.72.4.197 `int ClpSimplex::directionOut () const [inline]`

Definition at line 1067 of file ClpSimplex.hpp.

4.72.4.198 `void ClpSimplex::setDirectionIn (int direction) [inline]`

Set directionIn or Out.

Definition at line 1071 of file ClpSimplex.hpp.

4.72.4.199 `void ClpSimplex::setDirectionOut (int direction) [inline]`

Definition at line 1074 of file ClpSimplex.hpp.

4.72.4.200 `double ClpSimplex::valueOut () const [inline]`

Value of Out variable.

Definition at line 1078 of file ClpSimplex.hpp.

4.72.4.201 `void ClpSimplex::setValueOut (double value) [inline]`

Set value of out variable.

Definition at line 1082 of file ClpSimplex.hpp.

4.72.4.202 `double ClpSimplex::dualOut () const [inline]`

Dual value of Out variable.

Definition at line 1086 of file ClpSimplex.hpp.

4.72.4.203 `void ClpSimplex::setDualOut (double value) [inline]`

Set dual value of out variable.

Definition at line 1090 of file ClpSimplex.hpp.

4.72.4.204 `void ClpSimplex::setLowerOut (double value) [inline]`

Set lower of out variable.

Definition at line 1094 of file ClpSimplex.hpp.

4.72.4.205 `void ClpSimplex::setUpperOut (double value) [inline]`

Set upper of out variable.

Definition at line 1098 of file ClpSimplex.hpp.

4.72.4.206 `void ClpSimplex::setTheta (double value) [inline]`

Set theta of out variable.

Definition at line 1102 of file ClpSimplex.hpp.

4.72.4.207 `int ClpSimplex::isColumn (int sequence) const [inline]`

Returns 1 if sequence indicates column.

Definition at line 1106 of file ClpSimplex.hpp.

4.72.4.208 `int ClpSimplex::sequenceWithin (int sequence) const [inline]`

Returns sequence number within section.

Definition at line 1110 of file ClpSimplex.hpp.

4.72.4.209 `double ClpSimplex::solution (int sequence) [inline]`

Return row or column values.

Definition at line 1114 of file ClpSimplex.hpp.

4.72.4.210 `double& ClpSimplex::solutionAddress (int sequence) [inline]`

Return address of row or column values.

Definition at line 1118 of file ClpSimplex.hpp.

4.72.4.211 `double ClpSimplex::reducedCost (int sequence) [inline]`

Definition at line 1121 of file ClpSimplex.hpp.

4.72.4.212 `double& ClpSimplex::reducedCostAddress (int sequence) [inline]`

Definition at line 1124 of file ClpSimplex.hpp.

4.72.4.213 `double ClpSimplex::lower (int sequence) [inline]`

Definition at line 1127 of file ClpSimplex.hpp.

4.72.4.214 `double& ClpSimplex::lowerAddress (int sequence) [inline]`

Return address of row or column lower bound.

Definition at line 1131 of file ClpSimplex.hpp.

4.72.4.215 `double ClpSimplex::upper (int sequence) [inline]`

Definition at line 1134 of file ClpSimplex.hpp.

4.72.4.216 `double& ClpSimplex::upperAddress (int sequence) [inline]`

Return address of row or column upper bound.

Definition at line 1138 of file ClpSimplex.hpp.

4.72.4.217 `double ClpSimplex::cost (int sequence) [inline]`

Definition at line 1141 of file ClpSimplex.hpp.

4.72.4.218 `double& ClpSimplex::costAddress (int sequence) [inline]`

Return address of row or column cost.

Definition at line 1145 of file ClpSimplex.hpp.

4.72.4.219 `double ClpSimplex::originalLower (int iSequence) const` `[inline]`

Return original lower bound.

Definition at line 1149 of file ClpSimplex.hpp.

4.72.4.220 `double ClpSimplex::originalUpper (int iSequence) const` `[inline]`

Return original lower bound.

Definition at line 1155 of file ClpSimplex.hpp.

4.72.4.221 `double ClpSimplex::theta () const` `[inline]`

Theta (pivot change)

Definition at line 1161 of file ClpSimplex.hpp.

4.72.4.222 `double ClpSimplex::bestPossibleImprovement () const` `[inline]`

Best possible improvement using djs (primal) or obj change by flipping bounds to make dual feasible (dual)

Definition at line 1166 of file ClpSimplex.hpp.

4.72.4.223 `ClpNonLinearCost* ClpSimplex::nonLinearCost () const` `[inline]`

Return pointer to details of costs.

Definition at line 1170 of file ClpSimplex.hpp.

4.72.4.224 `int ClpSimplex::moreSpecialOptions () const` `[inline]`

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 debug 32768 bit - do dual in netlibd 65536 (*3) initial stateDualColumn 262144 bit - stop when primal feasible.

Definition at line 1194 of file ClpSimplex.hpp.

4.72.4.225 `void ClpSimplex::setMoreSpecialOptions (int value)` `[inline]`

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 debug 32768 bit - do dual in netlibd 65536 (*3) initial stateDualColumn 262144 bit - stop when primal feasible.

Definition at line 1218 of file ClpSimplex.hpp.

4.72.4.226 `void ClpSimplex::setFakeBound (int sequence, FakeBound fakeBound)` `[inline]`

Definition at line 1224 of file ClpSimplex.hpp.

4.72.4.227 FakeBound ClpSimplex::getFakeBound (int *sequence*) const [inline]

Definition at line 1229 of file ClpSimplex.hpp.

4.72.4.228 void ClpSimplex::setRowStatus (int *sequence*, Status *newstatus*) [inline]

Definition at line 1232 of file ClpSimplex.hpp.

4.72.4.229 Status ClpSimplex::getRowStatus (int *sequence*) const [inline]

Definition at line 1237 of file ClpSimplex.hpp.

4.72.4.230 void ClpSimplex::setColumnStatus (int *sequence*, Status *newstatus*) [inline]

Definition at line 1240 of file ClpSimplex.hpp.

4.72.4.231 Status ClpSimplex::getColumnStatus (int *sequence*) const [inline]

Definition at line 1245 of file ClpSimplex.hpp.

4.72.4.232 void ClpSimplex::setPivoted (int *sequence*) [inline]

Definition at line 1248 of file ClpSimplex.hpp.

4.72.4.233 void ClpSimplex::clearPivoted (int *sequence*) [inline]

Definition at line 1251 of file ClpSimplex.hpp.

4.72.4.234 bool ClpSimplex::pivoted (int *sequence*) const [inline]

Definition at line 1254 of file ClpSimplex.hpp.

4.72.4.235 void ClpSimplex::setFlagged (int *sequence*)

To flag a variable (not inline to allow for column generation)

4.72.4.236 void ClpSimplex::clearFlagged (int *sequence*) [inline]

Definition at line 1259 of file ClpSimplex.hpp.

4.72.4.237 bool ClpSimplex::flagged (int *sequence*) const [inline]

Definition at line 1262 of file ClpSimplex.hpp.

4.72.4.238 void ClpSimplex::setActive (int *iRow*) [inline]

To say row active in primal pivot row choice.

Definition at line 1266 of file ClpSimplex.hpp.

4.72.4.239 void ClpSimplex::clearActive (int *iRow*) [inline]

Definition at line 1269 of file ClpSimplex.hpp.

4.72.4.240 bool ClpSimplex::active (int *iRow*) const [inline]

Definition at line 1272 of file ClpSimplex.hpp.

4.72.4.241 `void ClpSimplex::createStatus ()`

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

Also can be used to set up all slack basis

4.72.4.242 `void ClpSimplex::allSlackBasis (bool resetSolution = false)`

Sets up all slack basis and resets solution to as it was after initial load or readMps.

4.72.4.243 `int ClpSimplex::lastBadIteration () const [inline]`

So we know when to be cautious.

Definition at line 1283 of file ClpSimplex.hpp.

4.72.4.244 `void ClpSimplex::setLastBadIteration (int value) [inline]`

Set so we know when to be cautious.

Definition at line 1287 of file ClpSimplex.hpp.

4.72.4.245 `int ClpSimplex::progressFlag () const [inline]`

Progress flag - at present 0 bit says artificials out.

Definition at line 1291 of file ClpSimplex.hpp.

4.72.4.246 `ClpSimplexProgress* ClpSimplex::progress () [inline]`

For dealing with all issues of cycling etc.

Definition at line 1295 of file ClpSimplex.hpp.

4.72.4.247 `int ClpSimplex::forceFactorization () const [inline]`

Force re-factorization early value.

Definition at line 1298 of file ClpSimplex.hpp.

4.72.4.248 `void ClpSimplex::forceFactorization (int value) [inline]`

Force re-factorization early.

Definition at line 1302 of file ClpSimplex.hpp.

4.72.4.249 `double ClpSimplex::rawObjectiveValue () const [inline]`

Raw objective value (so always minimize in primal)

Definition at line 1306 of file ClpSimplex.hpp.

4.72.4.250 `void ClpSimplex::computeObjectiveValue (bool useWorkingSolution = false)`

Compute objective value from solution and put in objectiveValue_.

4.72.4.251 `double ClpSimplex::computeInternalObjectiveValue ()`

Compute minimization objective value from internal solution without perturbation.

4.72.4.252 `double* ClpSimplex::infeasibilityRay (bool fullRay = false) const`

Infeasibility/unbounded ray (NULL returned if none/wrong) Up to user to use delete [] on these arrays.

4.72.4.253 `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 1319 of file ClpSimplex.hpp.

4.72.4.254 `int ClpSimplex::maximumBasic () const` `[inline]`

Maximum number of basic variables - can be more than number of rows if GUB.

Definition at line 1324 of file ClpSimplex.hpp.

4.72.4.255 `int ClpSimplex::baseIteration () const` `[inline]`

Iteration when we entered dual or primal.

Definition at line 1328 of file ClpSimplex.hpp.

4.72.4.256 `void ClpSimplex::generateCpp (FILE * fp, bool defaultFactor = false)`

Create C++ lines to get to current state.

4.72.4.257 `ClpFactorization* ClpSimplex::getEmptyFactorization ()`

Gets clean and emptyish factorization.

4.72.4.258 `void ClpSimplex::setEmptyFactorization ()`

May delete or may make clean and emptyish factorization.

4.72.4.259 `void ClpSimplex::moveInfo (const ClpSimplex & rhs, bool justStatus = false)`

Move status and solution across.

4.72.4.260 `void ClpSimplex::getBInvARow (int row, double * z, double * slack = NULL)`

Get a row of the tableau (slack part in slack if not NULL)

4.72.4.261 `void ClpSimplex::getBInvRow (int row, double * z)`

Get a row of the basis inverse.

4.72.4.262 `void ClpSimplex::getBInvACol (int col, double * vec)`

Get a column of the tableau.

4.72.4.263 `void ClpSimplex::getBInvCol (int col, double * vec)`

Get a column of the basis inverse.

4.72.4.264 `void ClpSimplex::getBasics (int * index)`

Get basic indices (order of indices corresponds to the order of elements in a vector returned by [getBInvACol\(\)](#) and [getBInvCol\(\)](#)).

4.72.4.265 void ClpSimplex::setObjectiveCoefficient (int *elementIndex*, double *elementValue*)

Set an objective function coefficient.

4.72.4.266 void ClpSimplex::setObjCoeff (int *elementIndex*, double *elementValue*) [inline]

Set an objective function coefficient.

Definition at line 1370 of file ClpSimplex.hpp.

4.72.4.267 void ClpSimplex::setColumnLower (int *elementIndex*, double *elementValue*)

Set a single column lower bound

Use -DBL_MAX for -infinity.

4.72.4.268 void ClpSimplex::setColumnUpper (int *elementIndex*, double *elementValue*)

Set a single column upper bound

Use DBL_MAX for infinity.

4.72.4.269 void ClpSimplex::setColumnBounds (int *elementIndex*, double *lower*, double *upper*)

Set a single column lower and upper bound.

4.72.4.270 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

<i>indexFirst, index-Last</i>	pointers to the beginning and after the end of the array of the indices of the variables whose <i>either</i> bound changes
<i>boundList</i>	the new lower/upper bound pairs for the variables

4.72.4.271 void ClpSimplex::setColLower (int *elementIndex*, double *elementValue*) [inline]

Set a single column lower bound

Use -DBL_MAX for -infinity.

Definition at line 1400 of file ClpSimplex.hpp.

4.72.4.272 void ClpSimplex::setColUpper (int *elementIndex*, double *elementValue*) [inline]

Set a single column upper bound

Use DBL_MAX for infinity.

Definition at line 1405 of file ClpSimplex.hpp.

4.72.4.273 void ClpSimplex::setColBounds (int *elementIndex*, double *newlower*, double *newupper*) [inline]

Set a single column lower and upper bound.

Definition at line 1410 of file ClpSimplex.hpp.

4.72.4.274 `void ClpSimplex::setColSetBounds (const int * indexFirst, const int * indexLast, const double * boundList)`
`[inline]`

Set the bounds on a number of columns simultaneously

Parameters

<i>indexFirst, index-Last</i>	pointers to the beginning and after the end of the array of the indices of the variables whose <i>either</i> bound changes
<i>boundList</i>	the new lower/upper bound pairs for the variables

Definition at line 1421 of file ClpSimplex.hpp.

4.72.4.275 void ClpSimplex::setRowLower (int *elementIndex*, double *elementValue*)

Set a single row lower bound

Use -DBL_MAX for -infinity.

4.72.4.276 void ClpSimplex::setRowUpper (int *elementIndex*, double *elementValue*)

Set a single row upper bound

Use DBL_MAX for infinity.

4.72.4.277 void ClpSimplex::setRowBounds (int *elementIndex*, double *lower*, double *upper*)

Set a single row lower and upper bound.

4.72.4.278 void ClpSimplex::setRowSetBounds (const int * *indexFirst*, const int * *indexLast*, const double * *boundList*)

Set the bounds on a number of rows simultaneously

Parameters

<i>indexFirst, index-Last</i>	pointers to the beginning and after the end of the array of the indices of the constraints whose <i>either</i> bound changes
<i>boundList</i>	the new lower/upper bound pairs for the constraints

4.72.4.279 void ClpSimplex::resize (int *newNumberRows*, int *newNumberColumns*)

Resizes rim part of model.

4.72.5 Friends And Related Function Documentation

4.72.5.1 friend class OsiClpSolverInterface [friend]

Allow OsiClp certain perks.

Definition at line 1673 of file ClpSimplex.hpp.

4.72.5.2 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 double ClpSimplex::bestPossibleImprovement_ [protected]

Best possible improvement using djs (primal) or obj change by flipping bounds to make dual feasible (dual)

Definition at line 1464 of file ClpSimplex.hpp.

4.72.6.2 double ClpSimplex::zeroTolerance_ [protected]

Zero tolerance.

Definition at line 1466 of file ClpSimplex.hpp.

4.72.6.3 int ClpSimplex::columnPrimalSequence_ [protected]

Sequence of worst (-1 if feasible)

Definition at line 1468 of file ClpSimplex.hpp.

4.72.6.4 int ClpSimplex::rowPrimalSequence_ [protected]

Sequence of worst (-1 if feasible)

Definition at line 1470 of file ClpSimplex.hpp.

4.72.6.5 double ClpSimplex::bestObjectiveValue_ [protected]

"Best" objective value

Definition at line 1472 of file ClpSimplex.hpp.

4.72.6.6 int ClpSimplex::moreSpecialOptions_ [protected]

More special options - see set for details.

Definition at line 1474 of file ClpSimplex.hpp.

4.72.6.7 int ClpSimplex::baseIteration_ [protected]

Iteration when we entered dual or primal.

Definition at line 1476 of file ClpSimplex.hpp.

4.72.6.8 double ClpSimplex::primalToleranceToGetOptimal_ [protected]

Primal tolerance needed to make dual feasible (<largeTolerance)

Definition at line 1478 of file ClpSimplex.hpp.

4.72.6.9 double ClpSimplex::largeValue_ [protected]

Large bound value (for complementarity etc)

Definition at line 1480 of file ClpSimplex.hpp.

4.72.6.10 double ClpSimplex::largestPrimalError_ [protected]

Largest error on $Ax=b$.

Definition at line 1482 of file ClpSimplex.hpp.

4.72.6.11 double ClpSimplex::largestDualError_ [protected]

Largest error on basic duals.

Definition at line 1484 of file ClpSimplex.hpp.

4.72.6.12 double ClpSimplex::alphaAccuracy_ [protected]

For computing whether to re-factorize.

Definition at line 1486 of file ClpSimplex.hpp.

4.72.6.13 double ClpSimplex::dualBound_ [protected]

Dual bound.

Definition at line 1488 of file ClpSimplex.hpp.

4.72.6.14 double ClpSimplex::alpha_ [protected]

Alpha (pivot element)

Definition at line 1490 of file ClpSimplex.hpp.

4.72.6.15 double ClpSimplex::theta_ [protected]

Theta (pivot change)

Definition at line 1492 of file ClpSimplex.hpp.

4.72.6.16 double ClpSimplex::lowerIn_ [protected]

Lower Bound on In variable.

Definition at line 1494 of file ClpSimplex.hpp.

4.72.6.17 double ClpSimplex::valueIn_ [protected]

Value of In variable.

Definition at line 1496 of file ClpSimplex.hpp.

4.72.6.18 double ClpSimplex::upperIn_ [protected]

Upper Bound on In variable.

Definition at line 1498 of file ClpSimplex.hpp.

4.72.6.19 double ClpSimplex::dualIn_ [protected]

Reduced cost of In variable.

Definition at line 1500 of file ClpSimplex.hpp.

4.72.6.20 double ClpSimplex::lowerOut_ [protected]

Lower Bound on Out variable.

Definition at line 1502 of file ClpSimplex.hpp.

4.72.6.21 double ClpSimplex::valueOut_ [protected]

Value of Out variable.

Definition at line 1504 of file ClpSimplex.hpp.

4.72.6.22 double ClpSimplex::upperOut_ [protected]

Upper Bound on Out variable.

Definition at line 1506 of file ClpSimplex.hpp.

4.72.6.23 double ClpSimplex::dualOut_ [protected]

Infeasibility (dual) or ? (primal) of Out variable.

Definition at line 1508 of file ClpSimplex.hpp.

4.72.6.24 double ClpSimplex::dualTolerance_ [protected]

Current dual tolerance for algorithm.

Definition at line 1510 of file ClpSimplex.hpp.

4.72.6.25 double ClpSimplex::primalTolerance_ [protected]

Current primal tolerance for algorithm.

Definition at line 1512 of file ClpSimplex.hpp.

4.72.6.26 double ClpSimplex::sumDualInfeasibilities_ [protected]

Sum of dual infeasibilities.

Definition at line 1514 of file ClpSimplex.hpp.

4.72.6.27 double ClpSimplex::sumPrimalInfeasibilities_ [protected]

Sum of primal infeasibilities.

Definition at line 1516 of file ClpSimplex.hpp.

4.72.6.28 double ClpSimplex::infeasibilityCost_ [protected]

Weight assigned to being infeasible in primal.

Definition at line 1518 of file ClpSimplex.hpp.

4.72.6.29 double ClpSimplex::sumOfRelaxedDualInfeasibilities_ [protected]

Sum of Dual infeasibilities using tolerance based on error in duals.

Definition at line 1520 of file ClpSimplex.hpp.

4.72.6.30 double ClpSimplex::sumOfRelaxedPrimalInfeasibilities_ [protected]

Sum of Primal infeasibilities using tolerance based on error in primals.

Definition at line 1522 of file ClpSimplex.hpp.

4.72.6.31 `double ClpSimplex::acceptablePivot_` [protected]

Acceptable pivot value just after factorization.

Definition at line 1524 of file ClpSimplex.hpp.

4.72.6.32 `double* ClpSimplex::lower_` [protected]

Working copy of lower bounds (Owner of arrays below)

Definition at line 1526 of file ClpSimplex.hpp.

4.72.6.33 `double* ClpSimplex::rowLowerWork_` [protected]

Row lower bounds - working copy.

Definition at line 1528 of file ClpSimplex.hpp.

4.72.6.34 `double* ClpSimplex::columnLowerWork_` [protected]

Column lower bounds - working copy.

Definition at line 1530 of file ClpSimplex.hpp.

4.72.6.35 `double* ClpSimplex::upper_` [protected]

Working copy of upper bounds (Owner of arrays below)

Definition at line 1532 of file ClpSimplex.hpp.

4.72.6.36 `double* ClpSimplex::rowUpperWork_` [protected]

Row upper bounds - working copy.

Definition at line 1534 of file ClpSimplex.hpp.

4.72.6.37 `double* ClpSimplex::columnUpperWork_` [protected]

Column upper bounds - working copy.

Definition at line 1536 of file ClpSimplex.hpp.

4.72.6.38 `double* ClpSimplex::cost_` [protected]

Working copy of objective (Owner of arrays below)

Definition at line 1538 of file ClpSimplex.hpp.

4.72.6.39 `double* ClpSimplex::rowObjectiveWork_` [protected]

Row objective - working copy.

Definition at line 1540 of file ClpSimplex.hpp.

4.72.6.40 `double* ClpSimplex::objectiveWork_` [protected]

Column objective - working copy.

Definition at line 1542 of file ClpSimplex.hpp.

4.72.6.41 CoinIndexedVector* ClpSimplex::rowArray_[6] [protected]

Useful row length arrays.

Definition at line 1544 of file ClpSimplex.hpp.

4.72.6.42 CoinIndexedVector* ClpSimplex::columnArray_[6] [protected]

Useful column length arrays.

Definition at line 1546 of file ClpSimplex.hpp.

4.72.6.43 int ClpSimplex::sequenceIn_ [protected]

Sequence of In variable.

Definition at line 1548 of file ClpSimplex.hpp.

4.72.6.44 int ClpSimplex::directionIn_ [protected]

Direction of In, 1 going up, -1 going down, 0 not a clude.

Definition at line 1550 of file ClpSimplex.hpp.

4.72.6.45 int ClpSimplex::sequenceOut_ [protected]

Sequence of Out variable.

Definition at line 1552 of file ClpSimplex.hpp.

4.72.6.46 int ClpSimplex::directionOut_ [protected]

Direction of Out, 1 to upper bound, -1 to lower bound, 0 - superbasic.

Definition at line 1554 of file ClpSimplex.hpp.

4.72.6.47 int ClpSimplex::pivotRow_ [protected]

Pivot Row.

Definition at line 1556 of file ClpSimplex.hpp.

4.72.6.48 int ClpSimplex::lastGoodIteration_ [protected]

Last good iteration (immediately after a re-factorization)

Definition at line 1558 of file ClpSimplex.hpp.

4.72.6.49 double* ClpSimplex::dj_ [protected]

Working copy of reduced costs (Owner of arrays below)

Definition at line 1560 of file ClpSimplex.hpp.

4.72.6.50 double* ClpSimplex::rowReducedCost_ [protected]

Reduced costs of slacks not same as duals (or - duals)

Definition at line 1562 of file ClpSimplex.hpp.

4.72.6.51 `double* ClpSimplex::reducedCostWork_` [protected]

Possible scaled reduced costs.

Definition at line 1564 of file ClpSimplex.hpp.

4.72.6.52 `double* ClpSimplex::solution_` [protected]

Working copy of primal solution (Owner of arrays below)

Definition at line 1566 of file ClpSimplex.hpp.

4.72.6.53 `double* ClpSimplex::rowActivityWork_` [protected]

Row activities - working copy.

Definition at line 1568 of file ClpSimplex.hpp.

4.72.6.54 `double* ClpSimplex::columnActivityWork_` [protected]

Column activities - working copy.

Definition at line 1570 of file ClpSimplex.hpp.

4.72.6.55 `int ClpSimplex::numberDualInfeasibilities_` [protected]

Number of dual infeasibilities.

Definition at line 1572 of file ClpSimplex.hpp.

4.72.6.56 `int ClpSimplex::numberDualInfeasibilitiesWithoutFree_` [protected]

Number of dual infeasibilities (without free)

Definition at line 1574 of file ClpSimplex.hpp.

4.72.6.57 `int ClpSimplex::numberPrimalInfeasibilities_` [protected]

Number of primal infeasibilities.

Definition at line 1576 of file ClpSimplex.hpp.

4.72.6.58 `int ClpSimplex::numberRefinements_` [protected]

How many iterative refinements to do.

Definition at line 1578 of file ClpSimplex.hpp.

4.72.6.59 `ClpDualRowPivot* ClpSimplex::dualRowPivot_` [protected]

dual row pivot choice

Definition at line 1580 of file ClpSimplex.hpp.

4.72.6.60 `ClpPrimalColumnPivot* ClpSimplex::primalColumnPivot_` [protected]

primal column pivot choice

Definition at line 1582 of file ClpSimplex.hpp.

4.72.6.61 `int* ClpSimplex::pivotVariable_` `[protected]`

Basic variables pivoting on which rows.

Definition at line 1584 of file ClpSimplex.hpp.

4.72.6.62 `ClpFactorization* ClpSimplex::factorization_` `[protected]`

factorization

Definition at line 1586 of file ClpSimplex.hpp.

4.72.6.63 `double* ClpSimplex::savedSolution_` `[protected]`

Saved version of solution.

Definition at line 1588 of file ClpSimplex.hpp.

4.72.6.64 `int ClpSimplex::numberOfTimesOptimal_` `[protected]`

Number of times code has tentatively thought optimal.

Definition at line 1590 of file ClpSimplex.hpp.

4.72.6.65 `ClpDisasterHandler* ClpSimplex::disasterArea_` `[protected]`

Disaster handler.

Definition at line 1592 of file ClpSimplex.hpp.

4.72.6.66 `int ClpSimplex::changeMade_` `[protected]`

If change has been made (first attempt at stopping looping)

Definition at line 1594 of file ClpSimplex.hpp.

4.72.6.67 `int ClpSimplex::algorithm_` `[protected]`

Algorithm >0 == Primal, <0 == Dual.

Definition at line 1596 of file ClpSimplex.hpp.

4.72.6.68 `int ClpSimplex::forceFactorization_` `[protected]`

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

Definition at line 1599 of file ClpSimplex.hpp.

4.72.6.69 `int ClpSimplex::perturbation_` `[protected]`

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.

Definition at line 1607 of file ClpSimplex.hpp.

4.72.6.70 `unsigned char* ClpSimplex::saveStatus_` `[protected]`

Saved status regions.

Definition at line 1609 of file ClpSimplex.hpp.

4.72.6.71 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 1614 of file ClpSimplex.hpp.

4.72.6.72 int ClpSimplex::lastBadIteration_ [protected]

So we know when to be cautious.

Definition at line 1616 of file ClpSimplex.hpp.

4.72.6.73 int ClpSimplex::lastFlaggedIteration_ [protected]

So we know when to open up again.

Definition at line 1618 of file ClpSimplex.hpp.

4.72.6.74 int ClpSimplex::numberFake_ [protected]

Can be used for count of fake bounds (dual) or fake costs (primal)

Definition at line 1620 of file ClpSimplex.hpp.

4.72.6.75 int ClpSimplex::numberChanged_ [protected]

Can be used for count of changed costs (dual) or changed bounds (primal)

Definition at line 1622 of file ClpSimplex.hpp.

4.72.6.76 int ClpSimplex::progressFlag_ [protected]

Progress flag - at present 0 bit says artificials out, 1 free in.

Definition at line 1624 of file ClpSimplex.hpp.

4.72.6.77 int ClpSimplex::firstFree_ [protected]

First free/super-basic variable (-1 if none)

Definition at line 1626 of file ClpSimplex.hpp.

4.72.6.78 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 1630 of file ClpSimplex.hpp.

4.72.6.79 int ClpSimplex::maximumBasic_ [protected]

Maximum number of basic variables - can be more than number of rows if GUB.

Definition at line 1633 of file ClpSimplex.hpp.

4.72.6.80 int ClpSimplex::dontFactorizePivots_ [protected]

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

Definition at line 1635 of file ClpSimplex.hpp.

4.72.6.81 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 allowedInfeasibility >= incomingInfeasibility this is always possible although you may end up with an all slack basis.

Defaults are 1.0,10.0

Definition at line 1645 of file ClpSimplex.hpp.

4.72.6.82 double ClpSimplex::allowedInfeasibility_ [protected]

Definition at line 1646 of file ClpSimplex.hpp.

4.72.6.83 int ClpSimplex::automaticScale_ [protected]

Automatic scaling of objective and rhs and bounds.

Definition at line 1648 of file ClpSimplex.hpp.

4.72.6.84 int ClpSimplex::maximumPerturbationSize_ [protected]

Maximum perturbation array size (take out when code rewritten)

Definition at line 1650 of file ClpSimplex.hpp.

4.72.6.85 double* ClpSimplex::perturbationArray_ [protected]

Perturbation array (maximumPerturbationSize_)

Definition at line 1652 of file ClpSimplex.hpp.

4.72.6.86 ClpSimplex* ClpSimplex::baseModel_ [protected]

A copy of model with certain state - normally without cuts.

Definition at line 1654 of file ClpSimplex.hpp.

4.72.6.87 ClpSimplexProgress ClpSimplex::progress_ [protected]

For dealing with all issues of cycling etc.

Definition at line 1656 of file ClpSimplex.hpp.

4.72.6.88 int ClpSimplex::abcState_ [protected]

Definition at line 1665 of file ClpSimplex.hpp.

4.72.6.89 int ClpSimplex::spareIntArray_[4] [mutable]

Spare int array for passing information [0]!=0 switches on.

Definition at line 1668 of file ClpSimplex.hpp.

4.72.6.90 double ClpSimplex::spareDoubleArray_[4] [mutable]

Spare double array for passing information [0]!=0 switches on.

Definition at line 1670 of file ClpSimplex.hpp.

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

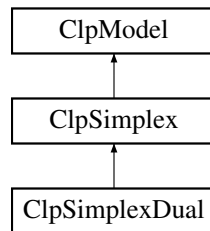
- [src/ClpSimplex.hpp](#)

4.73 ClpSimplexDual Class Reference

This solves LPs using the dual simplex method.

```
#include <ClpSimplexDual.hpp>
```

Inheritance 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 alwaysFinish=false, int startFinishOptions=0)`
For strong branching.
- `ClpFactorization * setupForStrongBranching (char *arrays, int numberOfRows, int numberOfColumns, bool solveLp=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, CoinIndexedVector *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 acceptablePivot, CoinBigIndex *dubiousWeights)`
Row array has row part of pivot row Column array has column part.
- `int dualColumn0 (const CoinIndexedVector *rowArray, const CoinIndexedVector *columnArray, CoinIndexedVector *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 ifValuesPass)
 - 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 updatedDualBound_ 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 ClpFactorization* ClpSimplexDual::setupForStrongBranching ( char * arrays, int numberOfRows, int numberColumns,
bool solveLp = false )
```

This does first part of StrongBranching.

```
4.73.2.4 void ClpSimplexDual::cleanupAfterStrongBranching ( ClpFactorization * factorization )
```

This cleans up after strong branching.

```
4.73.2.5 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

```
4.73.2.6 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)

```
4.73.2.7 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

```
4.73.2.8 void ClpSimplexDual::flipBounds ( CoinIndexedVector * rowArray, CoinIndexedVector * columnArray )
```

While updateDualsInDual sees what effect is of flip this does actual flipping.

```
4.73.2.9 double ClpSimplexDual::dualColumn ( CoinIndexedVector * rowArray, CoinIndexedVector * columnArray,
CoinIndexedVector * spareArray, CoinIndexedVector * spareArray2, double acceptablePivot, 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

4.73.2.10 `int ClpSimplexDual::dualColumn0 (const CoinIndexedVector * rowArray, const CoinIndexedVector * columnArray, CoinIndexedVector * spareArray, double acceptablePivot, double & upperReturn, double & bestReturn, double & badFree)`

Does first bit of dualColumn.

4.73.2.11 `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

4.73.2.12 `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

4.73.2.13 `void ClpSimplexDual::doEasyOnesInValuesPass (double * givenReducedCosts)`

This sees if we can move duals in dual values pass.

This is done before any pivoting

4.73.2.14 `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

4.73.2.15 `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)

4.73.2.16 `bool ClpSimplexDual::changeBound (int iSequence)`

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

Returns true if change

4.73.2.17 `void ClpSimplexDual::originalBound (int iSequence)`

Restores bound to original bound.

4.73.2.18 `int ClpSimplexDual::checkUnbounded (CoinIndexedVector * ray, CoinIndexedVector * spare, double changeCost)`

Checks if tentative optimal actually means unbounded in dual Returns -3 if not, 2 if is unbounded.

4.73.2.19 `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 took place.

type - 0 initial so set up save arrays etc

- 1 normal -if good update save

2 restoring from saved

4.73.2.20 `int ClpSimplexDual::perturb ()`

Perturbs problem (method depends on [perturbation\(\)](#)) returns nonzero if should go to dual.

4.73.2.21 `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.

4.73.2.22 `int ClpSimplexDual::numberAtFakeBound ()`

Checks number of variables at fake bounds.

This is used by fastDual so can exit gracefully before end

4.73.2.23 `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

4.73.2.24 `int ClpSimplexDual::nextSuperBasic ()`

Get next free , -1 if none.

4.73.2.25 `int ClpSimplexDual::startupSolve (int ifValuesPass, double * saveDuals, int startFinishOptions)`

Startup part of dual (may be extended to other algorithms) returns 0 if good, 1 if bad.

4.73.2.26 `void ClpSimplexDual::finishSolve (int startFinishOptions)`

4.73.2.27 `void ClpSimplexDual::gutsOfDual (int ifValuesPass, double *& saveDuals, int initialStatus, ClpDataSave & saveData)`

4.73.2.28 `void ClpSimplexDual::resetFakeBounds (int type)`

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

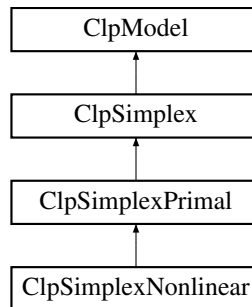
- [src/ClpSimplexDual.hpp](#)

4.74 ClpSimplexNonlinear Class Reference

This solves non-linear LPs using the primal simplex method.

```
#include <ClpSimplexNonlinear.hpp>
```

Inheritance 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)
Primal algorithm for quadratic Using a semi-trust region approach as for pooling problem This is in because I have it lying around.
- 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 numberPasses, double deltaTolerance)`

Primal algorithm for quadratic Using a semi-trust region approach as for pooling problem This is in because I have it lying around.

4.74.2.3 `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.

4.74.2.4 `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 $> 1.0e3 * \text{dualTolerance}$ If *mode* is nonzero then just largest *dj*

4.74.2.5 `int ClpSimplexNonlinear::whileIterating (int & pivotMode)`

Main part.

4.74.2.6 `int ClpSimplexNonlinear::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

4.74.2.7 `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 took place.

type - 0 initial so set up save arrays etc

- 1 normal -if good update save

2 restoring from saved

4.74.2.8 `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:

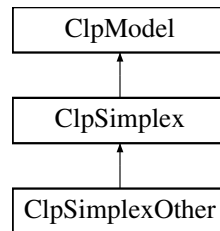
- [src/ClpSimplexNonlinear.hpp](#)

4.75 ClpSimplexOther Class Reference

This is for Simplex stuff which is neither dual nor primal.

```
#include <ClpSimplexOther.hpp>
```

Inheritance diagram for ClpSimplexOther:



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.

- `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 numberOfRows whichColumns has to be 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 variables 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 (const char * dataFile)`

Version of parametrics which reads from file See CbcClpParam.cpp for details of format Returns -2 if unable to open file.

4.75.2.5 `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)

4.75.2.6 `int ClpSimplexOther::parametricsObj (double startingTheta, double & endingTheta, const double * changeObjective)`

4.75.2.7 `double ClpSimplexOther::bestPivot (bool justColumns = false)`

Finds best possible pivot.

4.75.2.8 `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

4.75.2.9 `int ClpSimplexOther::readBasis (const char * filename)`

Read a basis from the given filename.

4.75.2.10 `ClpSimplex* ClpSimplexOther::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.

4.75.2.11 `int ClpSimplexOther::restoreFromDual (const ClpSimplex * dualProblem, bool checkAccuracy = false)`

Restores solution from dualized problem non-zero return code indicates minor problems.

4.75.2.12 `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.

4.75.2.13 `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.

4.75.2.14 `ClpSimplex* ClpSimplexOther::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 numberRows+numberColumns.

4.75.2.15 `void ClpSimplexOther::setGubBasis (ClpSimplex & original, const int * whichRows, const int * whichColumns)`

Sets basis from original.

4.75.2.16 `void ClpSimplexOther::getGubBasis (ClpSimplex & original, const int * whichRows, const int * whichColumns) const`

Restores basis to original.

4.75.2.17 `void ClpSimplexOther::cleanupAfterPostsolve ()`

Quick try at cleaning up duals if postsolve gets wrong.

4.75.2.18 `int ClpSimplexOther::tightenIntegerBounds (double * rhsSpace)`

Tightens integer bounds - returns number tightened or -1 if infeasible.

4.75.2.19 `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 entries which make up item "reConstruct" in expanded knapsack. Values in buildRow and buildElement;

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

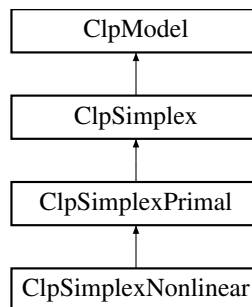
- [src/ClpSimplexOther.hpp](#)

4.76 ClpSimplexPrimal Class Reference

This solves LPs using the primal simplex method.

```
#include <ClpSimplexPrimal.hpp>
```

Inheritance 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 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, CoinIndexedVector *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 infeasibilityCost_ 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::alwaysOptimal (bool *onOff*)

Do not change infeasibility cost and always say optimal.

4.76.2.3 bool ClpSimplexPrimal::alwaysOptimal () const

4.76.2.4 void ClpSimplexPrimal::exactOutgoing (bool *onOff*)

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

This can be switched off

4.76.2.5 bool ClpSimplexPrimal::exactOutgoing () const

4.76.2.6 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*

4.76.2.7 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_*

4.76.2.8 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

4.76.2.9 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

4.76.2.10 void ClpSimplexPrimal::primalColumn (CoinIndexedVector * *updateArray*, CoinIndexedVector * *spareRow1*, CoinIndexedVector * *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.

4.76.2.11 int ClpSimplexPrimal::checkUnbounded (CoinIndexedVector * *ray*, CoinIndexedVector * *spare*, double *changeCost*)

Checks if tentative optimal actually means unbounded in primal Returns -3 if not, 2 if is unbounded.

4.76.2.12 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 took 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

4.76.2.13 void ClpSimplexPrimal::perturb (int *type*)

Perturbs problem (method depends on [perturbation\(\)](#))

4.76.2.14 bool ClpSimplexPrimal::unPerturb ()

Take off effect of perturbation and say whether to try dual.

4.76.2.15 int ClpSimplexPrimal::unflag ()

Unflag all variables and return number unflagged.

4.76.2.16 int ClpSimplexPrimal::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.

4.76.2.17 void ClpSimplexPrimal::primalRay (CoinIndexedVector * *rowArray*)

Create primal ray.

4.76.2.18 void ClpSimplexPrimal::clearAll ()

Clears all bits and clears rowArray[1] etc.

4.76.2.19 int ClpSimplexPrimal::lexSolve ()

Sort of lexicographic resolve.

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

- [src/ClpSimplexPrimal.hpp](#)

4.77 ClpSimplexProgress Class Reference

For saving extra information to see if looping.

```
#include <ClpSolve.hpp>
```

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)
- 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 `numberOfInfeasibilities_` [`CLP_PROGRESS`]
Number of infeasibilities.
- int `iterationNumber_` [`CLP_PROGRESS`]
Iteration number at which occurred.
- int `numberOfTimes_`
Number of times checked (so won't stop too early)
- int `numberOfBadTimes_`
Number of times it looked like loop.
- int `numberOfReallyBadTimes_`
Number really bad times.
- int `numberOfTimesFlagged_`
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 252 of file `ClpSolve.hpp`.

4.77.2 Constructor & Destructor Documentation

4.77.2.1 `ClpSimplexProgress::ClpSimplexProgress ()`

Default constructor.

4.77.2.2 `ClpSimplexProgress::ClpSimplexProgress (ClpSimplex * model)`

Constructor from model.

4.77.2.3 `ClpSimplexProgress::ClpSimplexProgress (const ClpSimplexProgress &)`

Copy constructor.

4.77.2.4 `ClpSimplexProgress::~~ClpSimplexProgress ()`

Destructor.

4.77.3 Member Function Documentation

4.77.3.1 `ClpSimplexProgress& ClpSimplexProgress::operator= (const ClpSimplexProgress & rhs)`

Assignment operator. This copies the data.

4.77.3.2 void ClpSimplexProgress::reset ()

Resets as much as possible.

4.77.3.3 void ClpSimplexProgress::fillFromModel (ClpSimplex * model)

Fill from model.

4.77.3.4 int ClpSimplexProgress::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.

4.77.3.5 void ClpSimplexProgress::startCheck ()

Start check at beginning of whileiterating.

4.77.3.6 int ClpSimplexProgress::cycle (int in, int out, int wayIn, int wayOut)

Returns cycle length in whileiterating.

4.77.3.7 double ClpSimplexProgress::lastObjective (int back = 1) const

Returns previous objective (if -1) - current if (0)

4.77.3.8 void ClpSimplexProgress::setInfeasibility (double value)

Set real primal infeasibility and move back.

4.77.3.9 double ClpSimplexProgress::lastInfeasibility (int back = 1) const

Returns real primal infeasibility (if -1) - current if (0)

4.77.3.10 void ClpSimplexProgress::modifyObjective (double value)

Modify objective e.g. if dual infeasible in dual.

4.77.3.11 int ClpSimplexProgress::lastIterationNumber (int back = 1) const

Returns previous iteration number (if -1) - current if (0)

4.77.3.12 void ClpSimplexProgress::clearIterationNumbers ()

clears all iteration numbers (to switch off panic)

4.77.3.13 void ClpSimplexProgress::newOddState () [inline]

Odd state.

Definition at line 303 of file ClpSolve.hpp.

4.77.3.14 void ClpSimplexProgress::endOddState () [inline]

Definition at line 306 of file ClpSolve.hpp.

4.77.3.15 void ClpSimplexProgress::clearOddState () [inline]

Definition at line 309 of file ClpSolve.hpp.

4.77.3.16 `int ClpSimplexProgress::oddState () const [inline]`

Definition at line 312 of file ClpSolve.hpp.

4.77.3.17 `int ClpSimplexProgress::badTimes () const [inline]`

number of bad times

Definition at line 316 of file ClpSolve.hpp.

4.77.3.18 `void ClpSimplexProgress::clearBadTimes () [inline]`

Definition at line 319 of file ClpSolve.hpp.

4.77.3.19 `int ClpSimplexProgress::reallyBadTimes () const [inline]`

number of really bad times

Definition at line 323 of file ClpSolve.hpp.

4.77.3.20 `void ClpSimplexProgress::incrementReallyBadTimes () [inline]`

Definition at line 326 of file ClpSolve.hpp.

4.77.3.21 `int ClpSimplexProgress::timesFlagged () const [inline]`

number of times flagged

Definition at line 330 of file ClpSolve.hpp.

4.77.3.22 `void ClpSimplexProgress::clearTimesFlagged () [inline]`

Definition at line 333 of file ClpSolve.hpp.

4.77.3.23 `void ClpSimplexProgress::incrementTimesFlagged () [inline]`

Definition at line 336 of file ClpSolve.hpp.

4.77.4 Member Data Documentation

4.77.4.1 `double ClpSimplexProgress::objective_[CLP_PROGRESS]`

Objective values.

Definition at line 346 of file ClpSolve.hpp.

4.77.4.2 `double ClpSimplexProgress::infeasibility_[CLP_PROGRESS]`

Sum of infeasibilities for algorithm.

Definition at line 348 of file ClpSolve.hpp.

4.77.4.3 `double ClpSimplexProgress::realInfeasibility_[CLP_PROGRESS]`

Sum of real primal infeasibilities for primal.

Definition at line 350 of file ClpSolve.hpp.

4.77.4.4 double ClpSimplexProgress::initialWeight_

Initial weight for weights.

Definition at line 364 of file ClpSolve.hpp.

4.77.4.5 int ClpSimplexProgress::in_[CLP_CYCLE]

For cycle checking.

Definition at line 368 of file ClpSolve.hpp.

4.77.4.6 int ClpSimplexProgress::out_[CLP_CYCLE]

Definition at line 369 of file ClpSolve.hpp.

4.77.4.7 char ClpSimplexProgress::way_[CLP_CYCLE]

Definition at line 370 of file ClpSolve.hpp.

4.77.4.8 ClpSimplex* ClpSimplexProgress::model_

Pointer back to model so we can get information.

Definition at line 372 of file ClpSolve.hpp.

4.77.4.9 int ClpSimplexProgress::numberInfeasibilities_[CLP_PROGRESS]

Number of infeasibilities.

Definition at line 374 of file ClpSolve.hpp.

4.77.4.10 int ClpSimplexProgress::iterationNumber_[CLP_PROGRESS]

Iteration number at which occurred.

Definition at line 376 of file ClpSolve.hpp.

4.77.4.11 int ClpSimplexProgress::numberTimes_

Number of times checked (so won't stop too early)

Definition at line 384 of file ClpSolve.hpp.

4.77.4.12 int ClpSimplexProgress::numberBadTimes_

Number of times it looked like loop.

Definition at line 386 of file ClpSolve.hpp.

4.77.4.13 int ClpSimplexProgress::numberReallyBadTimes_

Number really bad times.

Definition at line 388 of file ClpSolve.hpp.

4.77.4.14 int ClpSimplexProgress::numberTimesFlagged_

Number of times no iterations as flagged.

Definition at line 390 of file ClpSolve.hpp.

4.77.4.15 int ClpSimplexProgress::oddState_

If things are in an odd state.

Definition at line 392 of file ClpSolve.hpp.

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

- src/ClpSolve.hpp

4.78 ClpSolve Class Reference

This is a very simple class to guide algorithms.

```
#include <ClpSolve.hpp>
```

Public Types

- enum [SolveType](#) {
[useDual](#) = 0, [usePrimal](#), [usePrimalorSprint](#), [useBarrier](#),
[useBarrierNoCross](#), [automatic](#), [notImplemented](#) }
enums for solve function
- enum [PresolveType](#) { [presolveOn](#) = 0, [presolveOff](#), [presolveNumber](#), [presolveNumberCost](#) }

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 `doTripletion` () const
Whether we want to do tripletion part of presolve.
- void `setDoTripletion` (bool doTripletion_)
- 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)

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 Enumeration Documentation

4.78.2.1 enum ClpSolve::SolveType

enums for solve function

Enumerator

useDual
usePrimal
usePrimalorSprint
useBarrier
useBarrierNoCross
automatic
notImplemented

Definition at line 25 of file ClpSolve.hpp.

4.78.2.2 enum ClpSolve::PresolveType

Enumerator

presolveOn
presolveOff
presolveNumber
presolveNumberCost

Definition at line 34 of file ClpSolve.hpp.

4.78.3 Constructor & Destructor Documentation

4.78.3.1 ClpSolve::ClpSolve ()

Default constructor.

4.78.3.2 ClpSolve::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.

4.78.3.3 ClpSolve::ClpSolve (**const** ClpSolve &)

Copy constructor.

4.78.3.4 ClpSolve::~~ClpSolve ()

Destructor.

4.78.4 Member Function Documentation

4.78.4.1 void ClpSolve::generateCpp (**FILE** * *fp*)

Generates code for above constructor.

4.78.4.2 **ClpSolve & ClpSolve::operator= (const ClpSolve & rhs)**

Assignment operator. This copies the data.

4.78.4.3 **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 - for detailed printout (initially just presolve) 1 - presolve statistics

4.78.4.4 **int ClpSolve::getSpecialOption (int which) const**

4.78.4.5 **void ClpSolve::setSolveType (SolveType method, int extraInfo = -1)**

Solve types.

4.78.4.6 **SolveType ClpSolve::getSolveType ()**

4.78.4.7 **void ClpSolve::setPresolveType (PresolveType amount, int extraInfo = -1)**

4.78.4.8 **PresolveType ClpSolve::getPresolveType ()**

4.78.4.9 **int ClpSolve::getPresolvePasses () const**

4.78.4.10 **int ClpSolve::getExtraInfo (int which) const**

Extra info for idiot (or sprint)

4.78.4.11 **void ClpSolve::setInfeasibleReturn (bool trueFalse)**

Say to return at once if infeasible, default is to solve.

4.78.4.12 **bool ClpSolve::infeasibleReturn () const [inline]**

Definition at line 119 of file ClpSolve.hpp.

4.78.4.13 **bool ClpSolve::doDual () const [inline]**

Whether we want to do dual part of presolve.

Definition at line 123 of file ClpSolve.hpp.

4.78.4.14 **void ClpSolve::setDoDual (bool doDual_) [inline]**

Definition at line 126 of file ClpSolve.hpp.

4.78.4.15 `bool ClpSolve::doSingleton () const [inline]`

Whether we want to do singleton part of presolve.

Definition at line 131 of file ClpSolve.hpp.

4.78.4.16 `void ClpSolve::setDoSingleton (bool doSingleton_) [inline]`

Definition at line 134 of file ClpSolve.hpp.

4.78.4.17 `bool ClpSolve::doDoubleton () const [inline]`

Whether we want to do doubleton part of presolve.

Definition at line 139 of file ClpSolve.hpp.

4.78.4.18 `void ClpSolve::setDoDoubleton (bool doDoubleton_) [inline]`

Definition at line 142 of file ClpSolve.hpp.

4.78.4.19 `bool ClpSolve::doTripletion () const [inline]`

Whether we want to do tripletion part of presolve.

Definition at line 147 of file ClpSolve.hpp.

4.78.4.20 `void ClpSolve::setDoTripletion (bool doTripletion_) [inline]`

Definition at line 150 of file ClpSolve.hpp.

4.78.4.21 `bool ClpSolve::doTighten () const [inline]`

Whether we want to do tighten part of presolve.

Definition at line 155 of file ClpSolve.hpp.

4.78.4.22 `void ClpSolve::setDoTighten (bool doTighten_) [inline]`

Definition at line 158 of file ClpSolve.hpp.

4.78.4.23 `bool ClpSolve::doForcing () const [inline]`

Whether we want to do forcing part of presolve.

Definition at line 163 of file ClpSolve.hpp.

4.78.4.24 `void ClpSolve::setDoForcing (bool doForcing_) [inline]`

Definition at line 166 of file ClpSolve.hpp.

4.78.4.25 `bool ClpSolve::doImpliedFree () const [inline]`

Whether we want to do impliedfree part of presolve.

Definition at line 171 of file ClpSolve.hpp.

4.78.4.26 `void ClpSolve::setDoImpliedFree (bool doImpliedfree) [inline]`

Definition at line 174 of file ClpSolve.hpp.

4.78.4.27 `bool ClpSolve::doDupcol () const [inline]`

Whether we want to do dupcol part of presolve.

Definition at line 179 of file ClpSolve.hpp.

4.78.4.28 `void ClpSolve::setDoDupcol (bool doDupcol_) [inline]`

Definition at line 182 of file ClpSolve.hpp.

4.78.4.29 `bool ClpSolve::doDuprow () const [inline]`

Whether we want to do duprow part of presolve.

Definition at line 187 of file ClpSolve.hpp.

4.78.4.30 `void ClpSolve::setDoDuprow (bool doDuprow_) [inline]`

Definition at line 190 of file ClpSolve.hpp.

4.78.4.31 `bool ClpSolve::doSingletonColumn () const [inline]`

Whether we want to do singleton column part of presolve.

Definition at line 195 of file ClpSolve.hpp.

4.78.4.32 `void ClpSolve::setDoSingletonColumn (bool doSingleton_) [inline]`

Definition at line 198 of file ClpSolve.hpp.

4.78.4.33 `bool ClpSolve::doKillSmall () const [inline]`

Whether we want to kill small substitutions.

Definition at line 203 of file ClpSolve.hpp.

4.78.4.34 `void ClpSolve::setDoKillSmall (bool doKill) [inline]`

Definition at line 206 of file ClpSolve.hpp.

4.78.4.35 `int ClpSolve::presolveActions () const [inline]`

Set whole group.

Definition at line 211 of file ClpSolve.hpp.

4.78.4.36 `void ClpSolve::setPresolveActions (int action) [inline]`

Definition at line 214 of file ClpSolve.hpp.

4.78.4.37 `int ClpSolve::substitution () const [inline]`

Largest column for substitution (normally 3)

Definition at line 218 of file ClpSolve.hpp.

4.78.4.38 `void ClpSolve::setSubstitution (int value) [inline]`

Definition at line 221 of file ClpSolve.hpp.

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

- [src/ClpSolve.hpp](#)

4.79 ClpTrustedData Struct Reference

For a structure to be used by trusted code.

```
#include <ClpParameters.hpp>
```

Public Attributes

- int [typeStruct](#)
- int [typeCall](#)
- void * [data](#)

4.79.1 Detailed Description

For a structure to be used by trusted code.

Definition at line 119 of file ClpParameters.hpp.

4.79.2 Member Data Documentation

4.79.2.1 int ClpTrustedData::typeStruct

Definition at line 120 of file ClpParameters.hpp.

4.79.2.2 int ClpTrustedData::typeCall

Definition at line 121 of file ClpParameters.hpp.

4.79.2.3 void* ClpTrustedData::data

Definition at line 122 of file ClpParameters.hpp.

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

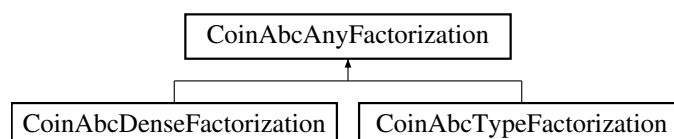
- [src/ClpParameters.hpp](#)

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 [setUsefullInformation](#) (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 ®ionSparse)=0
Updates one column (FTRAN) from unpacked regionSparse Tries to do FT update number returned is negative if no room.
- virtual int [updateColumnFTPart1](#) (CoinIndexedVector ®ionSparse)=0
- virtual void [updateColumnFTPart2](#) (CoinIndexedVector ®ionSparse)=0
- virtual void [updateColumnFT](#) (CoinIndexedVector ®ionSparseFT, CoinIndexedVector &partialUpdate, int which)=0
- virtual int [updateColumn](#) (CoinIndexedVector ®ionSparse) const =0
This version has same effect as above with FTUpdate==false so number returned is always >=0.
- virtual int [updateTwoColumnsFT](#) (CoinIndexedVector ®ionFT, CoinIndexedVector ®ionOther)=0
does FTRAN on two unpacked columns
- virtual int [updateColumnTranspose](#) (CoinIndexedVector ®ionSparse) const =0
Updates one column (BTRAN) from unpacked regionSparse.
- virtual void [updateFullColumn](#) (CoinIndexedVector ®ionSparse) const =0
This version does FTRAN on array when indices not set up.
- virtual void [updateFullColumnTranspose](#) (CoinIndexedVector ®ionSparse) const =0
Updates one column (BTRAN) from unpacked regionSparse.
- virtual void [updateWeights](#) (CoinIndexedVector ®ionSparse) const =0
Updates one column for dual steepest edge weights (FTRAN)
- virtual void [updateColumnCpu](#) (CoinIndexedVector ®ionSparse, int whichCpu) const
Updates one column (FTRAN)
- virtual void [updateColumnTransposeCpu](#) (CoinIndexedVector ®ionSparse, 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 $\max R * \max R + \max \text{Space}$ will always be long enough so can have $nR * nR$ ints in $\max \text{Space}$.*
- 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 Constructor & Destructor Documentation

4.80.2.1 CoinAbcAnyFactorization::CoinAbcAnyFactorization ()

Default constructor.

4.80.2.2 CoinAbcAnyFactorization::CoinAbcAnyFactorization (const CoinAbcAnyFactorization & other)

Copy constructor.

4.80.2.3 virtual CoinAbcAnyFactorization::~~CoinAbcAnyFactorization () [virtual]

Destructor.

4.80.3 Member Function Documentation

4.80.3.1 `CoinAbcAnyFactorization& CoinAbcAnyFactorization::operator= (const CoinAbcAnyFactorization & other)`

= copy

4.80.3.2 `virtual CoinAbcAnyFactorization* CoinAbcAnyFactorization::clone () const` [pure virtual]

Clone.

Implemented in [CoinAbcDenseFactorization](#), and [CoinAbcTypeFactorization](#).

4.80.3.3 `int CoinAbcAnyFactorization::status () const` [inline]

Returns status.

Definition at line 47 of file `CoinAbcDenseFactorization.hpp`.

4.80.3.4 `void CoinAbcAnyFactorization::setStatus (int value)` [inline]

Sets status.

Definition at line 51 of file `CoinAbcDenseFactorization.hpp`.

4.80.3.5 `int CoinAbcAnyFactorization::pivots () const` [inline]

Returns number of pivots since factorization.

Definition at line 54 of file `CoinAbcDenseFactorization.hpp`.

4.80.3.6 `void CoinAbcAnyFactorization::setPivots (int value)` [inline]

Sets number of pivots since factorization.

Definition at line 63 of file `CoinAbcDenseFactorization.hpp`.

4.80.3.7 `int CoinAbcAnyFactorization::numberSlacks () const` [inline]

Returns number of slacks.

Definition at line 66 of file `CoinAbcDenseFactorization.hpp`.

4.80.3.8 `void CoinAbcAnyFactorization::setNumberSlacks (int value)` [inline]

Sets number of slacks.

Definition at line 70 of file `CoinAbcDenseFactorization.hpp`.

4.80.3.9 `void CoinAbcAnyFactorization::setNumberRows (int value)` [inline]

Set number of Rows after factorization.

Definition at line 73 of file `CoinAbcDenseFactorization.hpp`.

4.80.3.10 `int CoinAbcAnyFactorization::numberRows () const` [inline]

Number of Rows after factorization.

Definition at line 76 of file `CoinAbcDenseFactorization.hpp`.

4.80.3.11 `CoinSimplexInt CoinAbcAnyFactorization::numberDense () const [inline]`

Number of dense rows after factorization.

Definition at line 80 of file `CoinAbcDenseFactorization.hpp`.

4.80.3.12 `int CoinAbcAnyFactorization::numberGoodColumns () const [inline]`

Number of good columns in factorization.

Definition at line 84 of file `CoinAbcDenseFactorization.hpp`.

4.80.3.13 `void CoinAbcAnyFactorization::relaxAccuracyCheck (double value) [inline]`

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

Definition at line 88 of file `CoinAbcDenseFactorization.hpp`.

4.80.3.14 `double CoinAbcAnyFactorization::getAccuracyCheck () const [inline]`

Definition at line 90 of file `CoinAbcDenseFactorization.hpp`.

4.80.3.15 `int CoinAbcAnyFactorization::maximumPivots () const [inline]`

Maximum number of pivots between factorizations.

Definition at line 93 of file `CoinAbcDenseFactorization.hpp`.

4.80.3.16 `virtual void CoinAbcAnyFactorization::maximumPivots (int value) [virtual]`

Set maximum pivots.

Reimplemented in [CoinAbcTypeFactorization](#).

4.80.3.17 `double CoinAbcAnyFactorization::pivotTolerance () const [inline]`

Pivot tolerance.

Definition at line 100 of file `CoinAbcDenseFactorization.hpp`.

4.80.3.18 `void CoinAbcAnyFactorization::pivotTolerance (double value)`

4.80.3.19 `double CoinAbcAnyFactorization::minimumPivotTolerance () const [inline]`

Minimum pivot tolerance.

Definition at line 105 of file `CoinAbcDenseFactorization.hpp`.

4.80.3.20 `void CoinAbcAnyFactorization::minimumPivotTolerance (double value)`

4.80.3.21 `virtual CoinFactorizationDouble* CoinAbcAnyFactorization::pivotRegion () const [inline],[virtual]`

Reimplemented in [CoinAbcTypeFactorization](#).

Definition at line 109 of file `CoinAbcDenseFactorization.hpp`.

4.80.3.22 `double CoinAbcAnyFactorization::areaFactor () const [inline]`

Area factor.

Definition at line 112 of file `CoinAbcDenseFactorization.hpp`.

4.80.3.23 `void CoinAbcAnyFactorization::areaFactor (CoinSimplexDouble value) [inline]`

Definition at line 115 of file CoinAbcDenseFactorization.hpp.

4.80.3.24 `double CoinAbcAnyFactorization::zeroTolerance () const [inline]`

Zero tolerance.

Definition at line 119 of file CoinAbcDenseFactorization.hpp.

4.80.3.25 `void CoinAbcAnyFactorization::zeroTolerance (double value)`

4.80.3.26 `virtual CoinFactorizationDouble* CoinAbcAnyFactorization::elements () const [virtual]`

Returns array to put basis elements in.

Reimplemented in [CoinAbcTypeFactorization](#).

4.80.3.27 `virtual int* CoinAbcAnyFactorization::pivotRow () const [virtual]`

Returns pivot row.

4.80.3.28 `virtual CoinFactorizationDouble* CoinAbcAnyFactorization::workArea () const [virtual]`

Returns work area.

4.80.3.29 `virtual int* CoinAbcAnyFactorization::intWorkArea () const [virtual]`

Returns int work area.

4.80.3.30 `virtual int* CoinAbcAnyFactorization::numberInRow () const [virtual]`

Number of entries in each row.

Reimplemented in [CoinAbcTypeFactorization](#).

4.80.3.31 `virtual int* CoinAbcAnyFactorization::numberInColumn () const [virtual]`

Number of entries in each column.

Reimplemented in [CoinAbcTypeFactorization](#).

4.80.3.32 `virtual CoinBigIndex* CoinAbcAnyFactorization::starts () const [virtual]`

Returns array to put basis starts in.

Reimplemented in [CoinAbcTypeFactorization](#).

4.80.3.33 `virtual int* CoinAbcAnyFactorization::permuteBack () const [virtual]`

Returns permute back.

4.80.3.34 `virtual void CoinAbcAnyFactorization::goSparse () [inline],[virtual]`

Sees whether to go sparse.

Reimplemented in [CoinAbcTypeFactorization](#).

Definition at line 140 of file CoinAbcDenseFactorization.hpp.

4.80.3.35 `virtual void CoinAbcAnyFactorization::checkMarkArrays () const [inline],[virtual]`

Reimplemented in [CoinAbcTypeFactorization](#).

Definition at line 142 of file `CoinAbcDenseFactorization.hpp`.

4.80.3.36 `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`.

4.80.3.37 `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`.

4.80.3.38 `virtual bool CoinAbcAnyFactorization::wantsTableauColumn () const [virtual]`

Returns true if wants tableauColumn in replaceColumn.

Reimplemented in [CoinAbcTypeFactorization](#).

4.80.3.39 `virtual void CoinAbcAnyFactorization::setUsefullInformation (const int * info, int whereFrom) [virtual]`

Useful information for factorization 0 - iteration number whereFrom is 0 for factorize and 1 for replaceColumn.

4.80.3.40 `virtual void CoinAbcAnyFactorization::clearArrays () [inline],[virtual]`

Get rid of all memory.

Reimplemented in [CoinAbcDenseFactorization](#), and [CoinAbcTypeFactorization](#).

Definition at line 164 of file `CoinAbcDenseFactorization.hpp`.

4.80.3.41 `virtual int* CoinAbcAnyFactorization::indices () const [pure virtual]`

Returns array to put basis indices in.

Implemented in [CoinAbcDenseFactorization](#), and [CoinAbcTypeFactorization](#).

4.80.3.42 `virtual int* CoinAbcAnyFactorization::permute () const [pure virtual]`

Returns permute in.

Implemented in [CoinAbcDenseFactorization](#), and [CoinAbcTypeFactorization](#).

4.80.3.43 `virtual int* CoinAbcAnyFactorization::pivotColumn () const [virtual]`

Returns pivotColumn or permute.

Reimplemented in [CoinAbcTypeFactorization](#).

4.80.3.44 `virtual int CoinAbcAnyFactorization::numberElements () const [pure virtual]`

Total number of elements in factorization.

Implemented in [CoinAbcDenseFactorization](#), and [CoinAbcTypeFactorization](#).

4.80.3.45 `virtual void CoinAbcAnyFactorization::getAreas (int numberRows, int numberColumns, CoinBigIndex maximumL, CoinBigIndex maximumU) [pure virtual]`

Gets space for a factorization.

Implemented in [CoinAbcTypeFactorization](#), and [CoinAbcDenseFactorization](#).

4.80.3.46 `virtual void CoinAbcAnyFactorization::preProcess () [pure virtual]`

PreProcesses column ordered copy of basis.

Implemented in [CoinAbcTypeFactorization](#), and [CoinAbcDenseFactorization](#).

4.80.3.47 `virtual int CoinAbcAnyFactorization::factor (AbcSimplex * model) [pure virtual]`

Does most of factorization returning status 0 - OK -99 - needs more memory -1 - singular - use numberGoodColumns and redo.

Implemented in [CoinAbcTypeFactorization](#), and [CoinAbcDenseFactorization](#).

4.80.3.48 `virtual void CoinAbcAnyFactorization::postProcess (const int * sequence, int * pivotVariable) [pure virtual]`

Does post processing on valid factorization - putting variables on correct rows.

Implemented in [CoinAbcTypeFactorization](#), and [CoinAbcDenseFactorization](#).

4.80.3.49 `virtual void CoinAbcAnyFactorization::makeNonSingular (int * sequence) [pure virtual]`

Makes a non-singular basis by replacing variables.

Implemented in [CoinAbcTypeFactorization](#), and [CoinAbcDenseFactorization](#).

4.80.3.50 `virtual double CoinAbcAnyFactorization::checkReplacePart1 (CoinIndexedVector *, int) [inline], [virtual]`

Checks if can replace one Column to basis, returns update alpha Fills in region for use later partial update already in U.

Reimplemented in [CoinAbcTypeFactorization](#).

Definition at line 245 of file [CoinAbcDenseFactorization.hpp](#).

4.80.3.51 `virtual double CoinAbcAnyFactorization::checkReplacePart1 (CoinIndexedVector *, CoinIndexedVector *, int) [inline], [virtual]`

Reimplemented in [CoinAbcTypeFactorization](#).

Definition at line 252 of file [CoinAbcDenseFactorization.hpp](#).

4.80.3.52 `virtual void CoinAbcAnyFactorization::checkReplacePart1a (CoinIndexedVector *, int) [inline], [virtual]`

Definition at line 256 of file [CoinAbcDenseFactorization.hpp](#).

4.80.3.53 `virtual double CoinAbcAnyFactorization::checkReplacePart1b (CoinIndexedVector *, int) [inline], [virtual]`

Definition at line 259 of file [CoinAbcDenseFactorization.hpp](#).

4.80.3.54 `virtual int CoinAbcAnyFactorization::checkReplacePart2 (int pivotRow, double btranAlpha, double ftranAlpha, double ftAlpha, double acceptablePivot = 1.0e-8) [pure virtual]`

Checks if can replace one Column to basis, returns 0=OK, 1=Probably OK, 2=singular, 3=no room, 5 max pivots.

Implemented in [CoinAbcDenseFactorization](#), and [CoinAbcTypeFactorization](#).

4.80.3.55 `virtual void CoinAbcAnyFactorization::replaceColumnPart3 (const AbcSimplex * model, CoinIndexedVector * regionSparse, CoinIndexedVector * tableauColumn, int pivotRow, double alpha) [pure virtual]`

Replaces one Column to basis, partial update already in U.

Implemented in [CoinAbcDenseFactorization](#), and [CoinAbcTypeFactorization](#).

4.80.3.56 `virtual void CoinAbcAnyFactorization::replaceColumnPart3 (const AbcSimplex * model, CoinIndexedVector * regionSparse, CoinIndexedVector * tableauColumn, CoinIndexedVector * partialUpdate, int pivotRow, double alpha) [pure virtual]`

Replaces one Column to basis, partial update in vector.

Implemented in [CoinAbcDenseFactorization](#), and [CoinAbcTypeFactorization](#).

4.80.3.57 `virtual int CoinAbcAnyFactorization::updateColumnFT (CoinIndexedVector & regionSparse) [pure virtual]`

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

Implemented in [CoinAbcDenseFactorization](#), and [CoinAbcTypeFactorization](#).

4.80.3.58 `virtual int CoinAbcAnyFactorization::updateColumnFTPPart1 (CoinIndexedVector & regionSparse) [pure virtual]`

Implemented in [CoinAbcDenseFactorization](#), and [CoinAbcTypeFactorization](#).

4.80.3.59 `virtual void CoinAbcAnyFactorization::updateColumnFTPPart2 (CoinIndexedVector & regionSparse) [pure virtual]`

Implemented in [CoinAbcDenseFactorization](#), and [CoinAbcTypeFactorization](#).

4.80.3.60 `virtual void CoinAbcAnyFactorization::updateColumnFT (CoinIndexedVector & regionSparseFT, CoinIndexedVector & partialUpdate, int which) [pure virtual]`

Implemented in [CoinAbcDenseFactorization](#), and [CoinAbcTypeFactorization](#).

4.80.3.61 `virtual int CoinAbcAnyFactorization::updateColumn (CoinIndexedVector & regionSparse) const [pure virtual]`

This version has same effect as above with FTUpdate==false so number returned is always >=0.

Implemented in [CoinAbcDenseFactorization](#), and [CoinAbcTypeFactorization](#).

4.80.3.62 `virtual int CoinAbcAnyFactorization::updateTwoColumnsFT (CoinIndexedVector & regionFT, CoinIndexedVector & regionOther) [pure virtual]`

does FTRAN on two unpacked columns

Implemented in [CoinAbcDenseFactorization](#), and [CoinAbcTypeFactorization](#).

4.80.3.63 `virtual int CoinAbcAnyFactorization::updateColumnTranspose (CoinIndexedVector & regionSparse) const [pure virtual]`

Updates one column (BTRAN) from unpacked regionSparse.

Implemented in [CoinAbcDenseFactorization](#), and [CoinAbcTypeFactorization](#).

4.80.3.64 `virtual void CoinAbcAnyFactorization::updateFullColumn (CoinIndexedVector & regionSparse) const` `[pure virtual]`

This version does FTRAN on array when indices not set up.

Implemented in [CoinAbcDenseFactorization](#), and [CoinAbcTypeFactorization](#).

4.80.3.65 `virtual void CoinAbcAnyFactorization::updateFullColumnTranspose (CoinIndexedVector & regionSparse) const` `[pure virtual]`

Updates one column (BTRAN) from unpacked regionSparse.

Implemented in [CoinAbcDenseFactorization](#), and [CoinAbcTypeFactorization](#).

4.80.3.66 `virtual void CoinAbcAnyFactorization::updateWeights (CoinIndexedVector & regionSparse) const` `[pure virtual]`

Updates one column for dual steepest edge weights (FTRAN)

Implemented in [CoinAbcDenseFactorization](#), and [CoinAbcTypeFactorization](#).

4.80.3.67 `virtual void CoinAbcAnyFactorization::updateColumnCpu (CoinIndexedVector & regionSparse, int whichCpu) const` `[virtual]`

Updates one column (FTRAN)

Reimplemented in [CoinAbcTypeFactorization](#).

4.80.3.68 `virtual void CoinAbcAnyFactorization::updateColumnTransposeCpu (CoinIndexedVector & regionSparse, int whichCpu) const` `[virtual]`

Updates one column (BTRAN)

Reimplemented in [CoinAbcTypeFactorization](#).

4.80.4 Member Data Documentation

4.80.4.1 `double CoinAbcAnyFactorization::pivotTolerance_` `[protected]`

Pivot tolerance.

Definition at line 336 of file [CoinAbcDenseFactorization.hpp](#).

4.80.4.2 `double CoinAbcAnyFactorization::minimumPivotTolerance_` `[protected]`

Minimum pivot tolerance.

Definition at line 338 of file [CoinAbcDenseFactorization.hpp](#).

4.80.4.3 `double CoinAbcAnyFactorization::areaFactor_` `[protected]`

Area factor.

Definition at line 340 of file [CoinAbcDenseFactorization.hpp](#).

4.80.4.4 `double CoinAbcAnyFactorization::zeroTolerance_` `[protected]`

Zero tolerance.

Definition at line 342 of file [CoinAbcDenseFactorization.hpp](#).

4.80.4.5 `double CoinAbcAnyFactorization::relaxCheck_` `[protected]`

Relax check on accuracy in replaceColumn.

Definition at line 347 of file CoinAbcDenseFactorization.hpp.

4.80.4.6 `CoinBigIndex CoinAbcAnyFactorization::factorElements_` `[protected]`

Number of elements after factorization.

Definition at line 349 of file CoinAbcDenseFactorization.hpp.

4.80.4.7 `int CoinAbcAnyFactorization::numberRows_` `[protected]`

Number of Rows in factorization.

Definition at line 351 of file CoinAbcDenseFactorization.hpp.

4.80.4.8 `int CoinAbcAnyFactorization::numberDense_` `[protected]`

Number of dense rows in factorization.

Definition at line 353 of file CoinAbcDenseFactorization.hpp.

4.80.4.9 `int CoinAbcAnyFactorization::numberGoodU_` `[protected]`

Number factorized in U (not row singletons)

Definition at line 355 of file CoinAbcDenseFactorization.hpp.

4.80.4.10 `int CoinAbcAnyFactorization::maximumPivots_` `[protected]`

Maximum number of pivots before factorization.

Definition at line 357 of file CoinAbcDenseFactorization.hpp.

4.80.4.11 `int CoinAbcAnyFactorization::numberPivots_` `[protected]`

Number pivots since last factorization.

Definition at line 359 of file CoinAbcDenseFactorization.hpp.

4.80.4.12 `int CoinAbcAnyFactorization::numberSlacks_` `[protected]`

Number slacks.

Definition at line 361 of file CoinAbcDenseFactorization.hpp.

4.80.4.13 `int CoinAbcAnyFactorization::status_` `[protected]`

Status of factorization.

Definition at line 363 of file CoinAbcDenseFactorization.hpp.

4.80.4.14 `int CoinAbcAnyFactorization::maximumRows_` `[protected]`

Maximum rows ever (i.e. use to copy arrays etc)

Definition at line 365 of file CoinAbcDenseFactorization.hpp.

4.80.4.15 `int* CoinAbcAnyFactorization::pivotRow_` [protected]

Pivot row.

Definition at line 370 of file `CoinAbcDenseFactorization.hpp`.

4.80.4.16 `CoinFactorizationDouble* CoinAbcAnyFactorization::elements_` [protected]

Elements of factorization and updates length is $\max R * \max R + \max \text{Space}$ will always be long enough so can have $nR * nR$ ints in `maxSpace`.

Definition at line 375 of file `CoinAbcDenseFactorization.hpp`.

4.80.4.17 `CoinFactorizationDouble* CoinAbcAnyFactorization::workArea_` [protected]

Work area of `numberRows_`.

Definition at line 377 of file `CoinAbcDenseFactorization.hpp`.

4.80.4.18 `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:

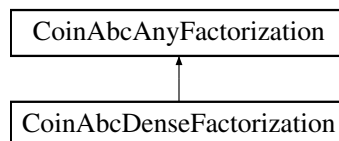
- [src/CoinAbcDenseFactorization.hpp](#)

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`:



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 ®ionSparse)

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

- virtual int [updateColumnFTPart1](#) (CoinIndexedVector ®ionSparse)
- virtual void [updateColumnFTPart2](#) (CoinIndexedVector &)
- virtual void [updateColumnFT](#) (CoinIndexedVector ®ionSparseFT, CoinIndexedVector &, int)
- virtual int [updateColumn](#) (CoinIndexedVector ®ionSparse) const
This version has same effect as above with FTUpdate==false so number returned is always ≥ 0 .
- virtual int [updateTwoColumnsFT](#) (CoinIndexedVector ®ionFT, CoinIndexedVector ®ionOther)
does FTRAN on two unpacked columns
- virtual int [updateColumnTranspose](#) (CoinIndexedVector ®ionSparse) const
Updates one column (BTRAN) from unpacked regionSparse.
- virtual void [updateFullColumn](#) (CoinIndexedVector ®ionSparse) const
This version does FTRAN on array when indices not set up.
- virtual void [updateFullColumnTranspose](#) (CoinIndexedVector ®ionSparse) const
Updates one column (BTRAN) from unpacked regionSparse.
- virtual void [updateWeights](#) (CoinIndexedVector ®ionSparse) 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.

Friends

- void [CoinAbcDenseFactorizationUnitTest](#) (const std::string &mpsDir)

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.

4.81.2 Constructor & Destructor Documentation

4.81.2.1 CoinAbcDenseFactorization::CoinAbcDenseFactorization ()

Default constructor.

4.81.2.2 CoinAbcDenseFactorization::CoinAbcDenseFactorization (const CoinAbcDenseFactorization & *other*)

Copy constructor.

4.81.2.3 virtual CoinAbcDenseFactorization::~CoinAbcDenseFactorization () [virtual]

Destructor.

4.81.3 Member Function Documentation

4.81.3.1 CoinAbcDenseFactorization& CoinAbcDenseFactorization::operator= (const CoinAbcDenseFactorization & *other*)

= copy

4.81.3.2 virtual CoinAbcAnyFactorization* CoinAbcDenseFactorization::clone () const [virtual]

Clone.

Implements [CoinAbcAnyFactorization](#).

4.81.3.3 virtual void CoinAbcDenseFactorization::getAreas (int *numberOfRows*, int *numberOfColumns*, CoinBigIndex *maximumL*, CoinBigIndex *maximumU*) [virtual]

Gets space for a factorization.

Implements [CoinAbcAnyFactorization](#).

4.81.3.4 virtual void CoinAbcDenseFactorization::preProcess () [virtual]

PreProcesses column ordered copy of basis.

Implements [CoinAbcAnyFactorization](#).

4.81.3.5 virtual int CoinAbcDenseFactorization::factor (AbcSimplex * *model*) [virtual]

Does most of factorization returning status 0 - OK -99 - needs more memory -1 - singular - use numberGoodColumns and redo.

Implements [CoinAbcAnyFactorization](#).

4.81.3.6 virtual void CoinAbcDenseFactorization::postProcess (const int * *sequence*, int * *pivotVariable*) [virtual]

Does post processing on valid factorization - putting variables on correct rows.

Implements [CoinAbcAnyFactorization](#).

4.81.3.7 virtual void CoinAbcDenseFactorization::makeNonSingular (int * *sequence*) [virtual]

Makes a non-singular basis by replacing variables.

Implements [CoinAbcAnyFactorization](#).

4.81.3.8 `virtual int CoinAbcDenseFactorization::numberElements () const [inline],[virtual]`

Total number of elements in factorization.

Implements [CoinAbcAnyFactorization](#).

Definition at line 439 of file `CoinAbcDenseFactorization.hpp`.

4.81.3.9 `double CoinAbcDenseFactorization::maximumCoefficient () const`

Returns maximum absolute value in factorization.

4.81.3.10 `virtual int CoinAbcDenseFactorization::replaceColumn (CoinIndexedVector * regionSparse, int pivotRow, double pivotCheck, bool skipBtranU = false, double acceptablePivot = 1.0e-8) [virtual]`

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.

4.81.3.11 `virtual int CoinAbcDenseFactorization::checkReplacePart2 (int pivotRow, double btranAlpha, double ftranAlpha, double ftAlpha, double acceptablePivot = 1.0e-8) [virtual]`

Checks if can replace one Column to basis, returns 0=OK, 1=Probably OK, 2=singular, 3=no room, 5 max pivots.

Implements [CoinAbcAnyFactorization](#).

4.81.3.12 `virtual void CoinAbcDenseFactorization::replaceColumnPart3 (const AbcSimplex * model, CoinIndexedVector * regionSparse, CoinIndexedVector * tableauColumn, int pivotRow, double alpha) [virtual]`

Replaces one Column to basis, partial update already in U.

Implements [CoinAbcAnyFactorization](#).

4.81.3.13 `virtual void CoinAbcDenseFactorization::replaceColumnPart3 (const AbcSimplex * model, CoinIndexedVector * regionSparse, CoinIndexedVector * tableauColumn, CoinIndexedVector *, int pivotRow, double alpha) [inline],[virtual]`

Replaces one Column to basis, partial update in vector.

Implements [CoinAbcAnyFactorization](#).

Definition at line 480 of file `CoinAbcDenseFactorization.hpp`.

4.81.3.14 `virtual int CoinAbcDenseFactorization::updateColumnFT (CoinIndexedVector & regionSparse) [inline],[virtual]`

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

Implements [CoinAbcAnyFactorization](#).

Definition at line 499 of file `CoinAbcDenseFactorization.hpp`.

4.81.3.15 `virtual int CoinAbcDenseFactorization::updateColumnFTPart1 (CoinIndexedVector & regionSparse) [inline],[virtual]`

Implements [CoinAbcAnyFactorization](#).

Definition at line 501 of file `CoinAbcDenseFactorization.hpp`.

4.81.3.16 `virtual void CoinAbcDenseFactorization::updateColumnFTPart2 (CoinIndexedVector &) [inline],[virtual]`

Implements [CoinAbcAnyFactorization](#).

Definition at line 503 of file CoinAbcDenseFactorization.hpp.

4.81.3.17 `virtual void CoinAbcDenseFactorization::updateColumnFT (CoinIndexedVector & regionSparseFT, CoinIndexedVector &, int) [inline],[virtual]`

Implements [CoinAbcAnyFactorization](#).

Definition at line 505 of file CoinAbcDenseFactorization.hpp.

4.81.3.18 `virtual int CoinAbcDenseFactorization::updateColumn (CoinIndexedVector & regionSparse) const [virtual]`

This version has same effect as above with FTUpdate==false so number returned is always >=0.

Implements [CoinAbcAnyFactorization](#).

4.81.3.19 `virtual int CoinAbcDenseFactorization::updateTwoColumnsFT (CoinIndexedVector & regionFT, CoinIndexedVector & regionOther) [virtual]`

does FTRAN on two unpacked columns

Implements [CoinAbcAnyFactorization](#).

4.81.3.20 `virtual int CoinAbcDenseFactorization::updateColumnTranspose (CoinIndexedVector & regionSparse) const [virtual]`

Updates one column (BTRAN) from unpacked regionSparse.

Implements [CoinAbcAnyFactorization](#).

4.81.3.21 `virtual void CoinAbcDenseFactorization::updateFullColumn (CoinIndexedVector & regionSparse) const [inline],[virtual]`

This version does FTRAN on array when indices not set up.

Implements [CoinAbcAnyFactorization](#).

Definition at line 517 of file CoinAbcDenseFactorization.hpp.

4.81.3.22 `virtual void CoinAbcDenseFactorization::updateFullColumnTranspose (CoinIndexedVector & regionSparse) const [inline],[virtual]`

Updates one column (BTRAN) from unpacked regionSparse.

Implements [CoinAbcAnyFactorization](#).

Definition at line 521 of file CoinAbcDenseFactorization.hpp.

4.81.3.23 `virtual void CoinAbcDenseFactorization::updateWeights (CoinIndexedVector & regionSparse) const [virtual]`

Updates one column for dual steepest edge weights (FTRAN)

Implements [CoinAbcAnyFactorization](#).

4.81.3.24 `void CoinAbcDenseFactorization::clearArrays () [inline],[virtual]`

Get rid of all memory.

Reimplemented from [CoinAbcAnyFactorization](#).

Definition at line 532 of file CoinAbcDenseFactorization.hpp.

4.81.3.25 `virtual int* CoinAbcDenseFactorization::indices () const [inline],[virtual]`

Returns array to put basis indices in.

Implements [CoinAbcAnyFactorization](#).

Definition at line 535 of file CoinAbcDenseFactorization.hpp.

4.81.3.26 `virtual int* CoinAbcDenseFactorization::permute () const [inline],[virtual]`

Returns permute in.

Implements [CoinAbcAnyFactorization](#).

Definition at line 538 of file CoinAbcDenseFactorization.hpp.

4.81.3.27 `void CoinAbcDenseFactorization::gutsOfDestructor ()`

The real work of desstructor.

4.81.3.28 `void CoinAbcDenseFactorization::gutsOfInitialize ()`

The real work of constructor.

4.81.3.29 `void CoinAbcDenseFactorization::gutsOfCopy (const CoinAbcDenseFactorization & other)`

The real work of copy.

4.81.3.30 `int CoinAbcDenseFactorization::checkPivot (double saveFromU, double oldPivot) const [protected]`

Returns accuracy status of replaceColumn returns 0=OK, 1=Probably OK, 2=singular.

4.81.4 Friends And Related Function Documentation

4.81.4.1 `void CoinAbcDenseFactorizationUnitTest (const std::string & mpsDir) [friend]`

4.81.5 Member Data Documentation

4.81.5.1 `CoinBigIndex CoinAbcDenseFactorization::maximumSpace_ [protected]`

Maximum length of iterating area.

Definition at line 557 of file CoinAbcDenseFactorization.hpp.

4.81.5.2 `CoinSimplexInt CoinAbcDenseFactorization::maximumRowsAdjusted_ [protected]`

Use for array size to get multiple of 8.

Definition at line 559 of file CoinAbcDenseFactorization.hpp.

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

- [src/CoinAbcDenseFactorization.hpp](#)

4.82 CoinAbcStack Struct Reference

```
#include <CoinAbcCommonFactorization.hpp>
```

Public Attributes

- CoinBigIndex [next](#)
- CoinBigIndex [start](#)
- [CoinSimplexUnsignedInt](#) [stack](#)

4.82.1 Detailed Description

Definition at line 71 of file `CoinAbcCommonFactorization.hpp`.

4.82.2 Member Data Documentation

4.82.2.1 CoinBigIndex CoinAbcStack::next

Definition at line 72 of file `CoinAbcCommonFactorization.hpp`.

4.82.2.2 CoinBigIndex CoinAbcStack::start

Definition at line 73 of file `CoinAbcCommonFactorization.hpp`.

4.82.2.3 CoinSimplexUnsignedInt CoinAbcStack::stack

Definition at line 74 of file `CoinAbcCommonFactorization.hpp`.

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

- `src/CoinAbcCommonFactorization.hpp`

4.83 CoinAbcStatistics Struct Reference

```
#include <CoinAbcCommonFactorization.hpp>
```

Public Attributes

- double [countInput_](#)
- double [countAfterL_](#)
- double [countAfterR_](#)
- double [countAfterU_](#)
- double [averageAfterL_](#)
- double [averageAfterR_](#)
- double [averageAfterU_](#)
- [CoinSimplexInt](#) [numberCounts_](#)

4.83.1 Detailed Description

Definition at line 32 of file `CoinAbcCommonFactorization.hpp`.

4.83.2 Member Data Documentation

4.83.2.1 `double CoinAbcStatistics::countInput_`

Definition at line 33 of file `CoinAbcCommonFactorization.hpp`.

4.83.2.2 `double CoinAbcStatistics::countAfterL_`

Definition at line 34 of file `CoinAbcCommonFactorization.hpp`.

4.83.2.3 `double CoinAbcStatistics::countAfterR_`

Definition at line 35 of file `CoinAbcCommonFactorization.hpp`.

4.83.2.4 `double CoinAbcStatistics::countAfterU_`

Definition at line 36 of file `CoinAbcCommonFactorization.hpp`.

4.83.2.5 `double CoinAbcStatistics::averageAfterL_`

Definition at line 37 of file `CoinAbcCommonFactorization.hpp`.

4.83.2.6 `double CoinAbcStatistics::averageAfterR_`

Definition at line 38 of file `CoinAbcCommonFactorization.hpp`.

4.83.2.7 `double CoinAbcStatistics::averageAfterU_`

Definition at line 39 of file `CoinAbcCommonFactorization.hpp`.

4.83.2.8 `CoinSimplexInt CoinAbcStatistics::numberCounts_`

Definition at line 44 of file `CoinAbcCommonFactorization.hpp`.

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

- [src/CoinAbcCommonFactorization.hpp](#)

4.84 `CoinAbcThreadInfo` Struct Reference

```
#include <AbcSimplex.hpp>
```

Public Attributes

- `double` [result](#)
- `int` [status](#)
- `int` [stuff](#) [4]

4.84.1 Detailed Description

Definition at line 62 of file `AbcSimplex.hpp`.

4.84.2 Member Data Documentation

4.84.2.1 double CoinAbcThreadInfo::result

Definition at line 63 of file AbcSimplex.hpp.

4.84.2.2 int CoinAbcThreadInfo::status

Definition at line 66 of file AbcSimplex.hpp.

4.84.2.3 int CoinAbcThreadInfo::stuff[4]

Definition at line 67 of file AbcSimplex.hpp.

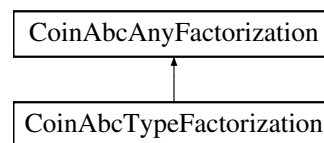
The documentation for this struct was generated from the following file:

- src/[AbcSimplex.hpp](#)

4.85 CoinAbcTypeFactorization Class Reference

```
#include <CoinAbcBaseFactorization.hpp>
```

Inheritance 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 ®ionSparse)
Later take out return codes (apart from +- 1 on FT)
- virtual int [updateColumnFTPart1](#) (CoinIndexedVector ®ionSparse)
- virtual void [updateColumnFTPart2](#) (CoinIndexedVector ®ionSparse)
- virtual void [updateColumnFT](#) (CoinIndexedVector ®ionSparseFT, CoinIndexedVector &partialUpdate, int which)
Updates one column (FTRAN) Tries to do FT update puts partial update in vector.
- virtual [CoinSimplexInt](#) [updateColumn](#) (CoinIndexedVector ®ionSparse) const
This version has same effect as above with FTUpdate==false so number returned is always >=0.
- virtual [CoinSimplexInt](#) [updateTwoColumnsFT](#) (CoinIndexedVector ®ionFT, CoinIndexedVector ®ion-Other)
Updates one column (FTRAN) from region2 Tries to do FT update number returned is negative if no room.
- virtual [CoinSimplexInt](#) [updateColumnTranspose](#) (CoinIndexedVector ®ionSparse) const
Updates one column (BTRAN) from regionSparse2 regionSparse starts as zero and is zero at end Note - if region-Sparse2 packed on input - will be packed on output.
- virtual void [updateFullColumn](#) (CoinIndexedVector ®ionSparse) const
Updates one full column (FTRAN)
- virtual void [updateFullColumnTranspose](#) (CoinIndexedVector ®ionSparse) const
Updates one full column (BTRAN)
- virtual void [updateWeights](#) (CoinIndexedVector ®ionSparse) const
Updates one column for dual steepest edge weights (FTRAN)
- virtual void [updateColumnCpu](#) (CoinIndexedVector ®ionSparse, int whichCpu) const
Updates one column (FTRAN)
- virtual void [updateColumnTransposeCpu](#) (CoinIndexedVector ®ionSparse, 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

Friends

- void [CoinAbcFactorizationUnitTest](#) (const std::string &mpsDir)

used by factorization

- virtual void [getAreas](#) ([CoinSimplexInt](#) numberOfRows, [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, [CoinSimplex-Double](#) 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 `getColumnSpacelaterateR` (`CoinSimplexInt` iColumn, `CoinFactorizationDouble` value, `CoinSimplexInt` iRow)

getColumnSpacelaterateR.
- `CoinBigIndex` `getColumnSpacelaterate` (`CoinSimplexInt` iColumn, `CoinFactorizationDouble` value, `CoinSimplexInt` iRow)

getColumnSpacelaterate.
- 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 `getRowSpacelaterate` (`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 - returns 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 [numberOfRowsExtra_](#)
Number of Rows after iterating.
- CoinSimplexInt [maximumRowsExtra_](#)
Maximum number of Rows after iterating.
- CoinSimplexInt [numberOfRowsSmall_](#)
Size of small inverse.
- CoinSimplexInt [numberOfGoodL_](#)
Number factorized in L.
- CoinSimplexInt [numberOfRowsLeft_](#)
Number Rows left (numberOfRows-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 [numberOfTrials_](#)
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 [ftranFTAveragAfterL_](#)
- double [ftranFTAveragAfterR_](#)
- double [ftranFTAveragAfterU_](#)
- [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

4.85.1 Detailed Description

Definition at line 28 of file `CoinAbcBaseFactorization.hpp`.

4.85.2 Constructor & Destructor Documentation

4.85.2.1 CoinAbcTypeFactorization::CoinAbcTypeFactorization ()

Default constructor.

4.85.2.2 CoinAbcTypeFactorization::CoinAbcTypeFactorization (const CoinAbcTypeFactorization & other)

Copy constructor.

4.85.2.3 CoinAbcTypeFactorization::CoinAbcTypeFactorization (const CoinFactorization & other)

Copy constructor.

4.85.2.4 virtual CoinAbcTypeFactorization::~~CoinAbcTypeFactorization () [virtual]

Destructor.

4.85.3 Member Function Documentation

4.85.3.1 virtual CoinAbcAnyFactorization* CoinAbcTypeFactorization::clone () const [virtual]

Clone.

Implements [CoinAbcAnyFactorization](#).

4.85.3.2 void CoinAbcTypeFactorization::almostDestructor ()

Delete all stuff (leaves as after CoinAbcFactorization())

4.85.3.3 void CoinAbcTypeFactorization::show_self () const

Debug show object (shows one representation)

4.85.3.4 void CoinAbcTypeFactorization::sort () const

Debug - sort so can compare.

4.85.3.5 CoinAbcTypeFactorization& CoinAbcTypeFactorization::operator= (const CoinAbcTypeFactorization & other)

= copy

4.85.3.6 CoinSimplexDouble CoinAbcTypeFactorization::conditionNumber () const

Condition number - product of pivots after factorization.

4.85.3.7 CoinSimplexInt* CoinAbcTypeFactorization::permute () const [inline],[virtual]

Returns address of permute region.

Implements [CoinAbcAnyFactorization](#).

Definition at line 68 of file CoinAbcBaseFactorization.hpp.

4.85.3.8 virtual CoinSimplexInt* CoinAbcTypeFactorization::indices () const [inline],[virtual]

Returns array to put basis indices in.

Implements [CoinAbcAnyFactorization](#).

Definition at line 72 of file CoinAbcBaseFactorization.hpp.

4.85.3.9 `virtual CoinSimplexInt* CoinAbcTypeFactorization::pivotColumn () const [inline], [virtual]`

Returns address of pivotColumn region (also used for permuting)

Reimplemented from [CoinAbcAnyFactorization](#).

Definition at line 75 of file CoinAbcBaseFactorization.hpp.

4.85.3.10 `virtual CoinFactorizationDouble* CoinAbcTypeFactorization::pivotRegion () const [inline], [virtual]`

Returns address of pivot region.

Reimplemented from [CoinAbcAnyFactorization](#).

Definition at line 79 of file CoinAbcBaseFactorization.hpp.

4.85.3.11 `CoinBigIndex* CoinAbcTypeFactorization::startRowL () const [inline]`

Start of each row in L.

Definition at line 84 of file CoinAbcBaseFactorization.hpp.

4.85.3.12 `CoinBigIndex* CoinAbcTypeFactorization::startColumnL () const [inline]`

Start of each column in L.

Definition at line 89 of file CoinAbcBaseFactorization.hpp.

4.85.3.13 `CoinSimplexInt* CoinAbcTypeFactorization::indexColumnL () const [inline]`

Index of column in row for L.

Definition at line 94 of file CoinAbcBaseFactorization.hpp.

4.85.3.14 `CoinSimplexInt* CoinAbcTypeFactorization::indexRowL () const [inline]`

Row indices of L.

Definition at line 99 of file CoinAbcBaseFactorization.hpp.

4.85.3.15 `CoinFactorizationDouble* CoinAbcTypeFactorization::elementByRowL () const [inline]`

Elements in L (row copy)

Definition at line 104 of file CoinAbcBaseFactorization.hpp.

4.85.3.16 `CoinSimplexInt* CoinAbcTypeFactorization::pivotLinkedBackwards () const [inline]`

Forward and backward linked lists (numberRows_+2)

Definition at line 110 of file CoinAbcBaseFactorization.hpp.

4.85.3.17 `CoinSimplexInt* CoinAbcTypeFactorization::pivotLinkedForwards () const [inline]`

Definition at line 112 of file CoinAbcBaseFactorization.hpp.

4.85.3.18 `CoinSimplexInt* CoinAbcTypeFactorization::pivotLOrder () const [inline]`

Definition at line 114 of file CoinAbcBaseFactorization.hpp.

4.85.3.19 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.

4.85.3.20 CoinSimplexInt* CoinAbcTypeFactorization::nextCount () const [inline]

Next Row/Column with count.

Definition at line 147 of file CoinAbcBaseFactorization.hpp.

4.85.3.21 CoinSimplexInt* CoinAbcTypeFactorization::lastCount () const [inline]

Previous Row/Column with count.

Definition at line 151 of file CoinAbcBaseFactorization.hpp.

4.85.3.22 CoinSimplexInt CoinAbcTypeFactorization::numberRowsExtra () const [inline]

Number of Rows after iterating.

Definition at line 155 of file CoinAbcBaseFactorization.hpp.

4.85.3.23 CoinBigIndex CoinAbcTypeFactorization::numberL () const [inline]

Number in L.

Definition at line 159 of file CoinAbcBaseFactorization.hpp.

4.85.3.24 CoinBigIndex CoinAbcTypeFactorization::baseL () const [inline]

Base of L.

Definition at line 163 of file CoinAbcBaseFactorization.hpp.

4.85.3.25 CoinSimplexInt CoinAbcTypeFactorization::maximumRowsExtra () const [inline]

Maximum of Rows after iterating.

Definition at line 166 of file CoinAbcBaseFactorization.hpp.

4.85.3.26 virtual CoinBigIndex CoinAbcTypeFactorization::numberElements () const [inline],[virtual]

Total number of elements in factorization.

Implements [CoinAbcAnyFactorization](#).

Definition at line 170 of file CoinAbcBaseFactorization.hpp.

4.85.3.27 CoinSimplexInt CoinAbcTypeFactorization::numberForrestTomlin () const [inline]

Length of FT vector.

Definition at line 174 of file CoinAbcBaseFactorization.hpp.

4.85.3.28 CoinSimplexDouble CoinAbcTypeFactorization::adjustedAreaFactor () const

Returns areaFactor but adjusted for dense.

4.85.3.29 `CoinSimplexInt CoinAbcTypeFactorization::messageLevel () const` `[inline]`

Level of detail of messages.

Definition at line 180 of file `CoinAbcBaseFactorization.hpp`.

4.85.3.30 `void CoinAbcTypeFactorization::messageLevel (CoinSimplexInt value)`

4.85.3.31 `virtual void CoinAbcTypeFactorization::maximumPivots (CoinSimplexInt value)` `[virtual]`

Set maximum pivots.

Reimplemented from [CoinAbcAnyFactorization](#).

4.85.3.32 `CoinSimplexInt CoinAbcTypeFactorization::denseThreshold () const` `[inline]`

Gets dense threshold.

Definition at line 189 of file `CoinAbcBaseFactorization.hpp`.

4.85.3.33 `void CoinAbcTypeFactorization::setDenseThreshold (CoinSimplexInt value)` `[inline]`

Sets dense threshold.

Definition at line 192 of file `CoinAbcBaseFactorization.hpp`.

4.85.3.34 `CoinSimplexDouble CoinAbcTypeFactorization::maximumCoefficient () const`

Returns maximum absolute value in factorization.

4.85.3.35 `bool CoinAbcTypeFactorization::spaceForForrestTomlin () const` `[inline]`

True if FT update and space.

Definition at line 205 of file `CoinAbcBaseFactorization.hpp`.

4.85.3.36 `CoinBigIndex CoinAbcTypeFactorization::numberElementsU () const` `[inline]`

Returns number in U area.

Definition at line 218 of file `CoinAbcBaseFactorization.hpp`.

4.85.3.37 `void CoinAbcTypeFactorization::setNumberElementsU (CoinBigIndex value)` `[inline]`

Setss number in U area.

Definition at line 222 of file `CoinAbcBaseFactorization.hpp`.

4.85.3.38 `CoinBigIndex CoinAbcTypeFactorization::lengthAreaU () const` `[inline]`

Returns length of U area.

Definition at line 225 of file `CoinAbcBaseFactorization.hpp`.

4.85.3.39 `CoinBigIndex CoinAbcTypeFactorization::numberElementsL () const` `[inline]`

Returns number in L area.

Definition at line 229 of file `CoinAbcBaseFactorization.hpp`.

4.85.3.40 `CoinBigIndex CoinAbcTypeFactorization::lengthAreaL () const [inline]`

Returns length of L area.

Definition at line 233 of file CoinAbcBaseFactorization.hpp.

4.85.3.41 `CoinBigIndex CoinAbcTypeFactorization::numberElementsR () const [inline]`

Returns number in R area.

Definition at line 237 of file CoinAbcBaseFactorization.hpp.

4.85.3.42 `CoinBigIndex CoinAbcTypeFactorization::numberCompressions () const [inline]`

Number of compressions done.

Definition at line 241 of file CoinAbcBaseFactorization.hpp.

4.85.3.43 `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.

4.85.3.44 `virtual CoinSimplexInt* CoinAbcTypeFactorization::numberInRow () const [inline],[virtual]`

Number of entries in each row.

Reimplemented from [CoinAbcAnyFactorization](#).

Definition at line 253 of file CoinAbcBaseFactorization.hpp.

4.85.3.45 `virtual CoinSimplexInt* CoinAbcTypeFactorization::numberInColumn () const [inline],[virtual]`

Number of entries in each column.

Reimplemented from [CoinAbcAnyFactorization](#).

Definition at line 256 of file CoinAbcBaseFactorization.hpp.

4.85.3.46 `virtual CoinFactorizationDouble* CoinAbcTypeFactorization::elements () const [inline],[virtual]`

Returns array to put basis elements in.

Reimplemented from [CoinAbcAnyFactorization](#).

Definition at line 259 of file CoinAbcBaseFactorization.hpp.

4.85.3.47 `CoinBigIndex* CoinAbcTypeFactorization::startColumnR () const [inline]`

Start of columns for R.

Definition at line 262 of file CoinAbcBaseFactorization.hpp.

4.85.3.48 `CoinFactorizationDouble* CoinAbcTypeFactorization::elementU () const [inline]`

Elements of U.

Definition at line 265 of file CoinAbcBaseFactorization.hpp.

4.85.3.49 `CoinSimplexInt* CoinAbcTypeFactorization::indexRowU () const` `[inline]`

Row indices of U.

Definition at line 268 of file `CoinAbcBaseFactorization.hpp`.

4.85.3.50 `CoinBigIndex* CoinAbcTypeFactorization::startColumnU () const` `[inline]`

Start of each column in U.

Definition at line 271 of file `CoinAbcBaseFactorization.hpp`.

4.85.3.51 `double* CoinAbcTypeFactorization::denseVector (CoinIndexedVector * vector) const` `[inline]`

Returns double * associated with vector.

Definition at line 294 of file `CoinAbcBaseFactorization.hpp`.

4.85.3.52 `double* CoinAbcTypeFactorization::denseVector (CoinIndexedVector & vector) const` `[inline]`

Definition at line 296 of file `CoinAbcBaseFactorization.hpp`.

4.85.3.53 `const double* CoinAbcTypeFactorization::denseVector (const CoinIndexedVector * vector) const` `[inline]`

Returns double * associated with vector.

Definition at line 299 of file `CoinAbcBaseFactorization.hpp`.

4.85.3.54 `const double* CoinAbcTypeFactorization::denseVector (const CoinIndexedVector & vector) const` `[inline]`

Definition at line 301 of file `CoinAbcBaseFactorization.hpp`.

4.85.3.55 `void CoinAbcTypeFactorization::toLongArray (CoinIndexedVector * vector, int which) const` `[inline]`

To a work array and associate vector.

Definition at line 304 of file `CoinAbcBaseFactorization.hpp`.

4.85.3.56 `void CoinAbcTypeFactorization::fromLongArray (CoinIndexedVector * vector) const` `[inline]`

From a work array and dis-associate vector.

Definition at line 306 of file `CoinAbcBaseFactorization.hpp`.

4.85.3.57 `void CoinAbcTypeFactorization::fromLongArray (int which) const` `[inline]`

From a work array and dis-associate vector.

Definition at line 308 of file `CoinAbcBaseFactorization.hpp`.

4.85.3.58 `void CoinAbcTypeFactorization::scan (CoinIndexedVector * vector) const` `[inline]`

Scans region to find nonzeros.

Definition at line 310 of file `CoinAbcBaseFactorization.hpp`.

4.85.3.59 `virtual double CoinAbcTypeFactorization::checkReplacePart1 (CoinIndexedVector * regionSparse, int pivotRow)`
`[virtual]`

Checks if can replace one Column to basis, returns update alpha Fills in region for use later partial update already in U.

Reimplemented from [CoinAbcAnyFactorization](#).

4.85.3.60 `virtual double CoinAbcTypeFactorization::checkReplacePart1 (CoinIndexedVector * regionSparse, CoinIndexedVector * partialUpdate, int pivotRow) [virtual]`

Checks if can replace one Column to basis, returns update alpha Fills in region for use later partial update in vector.
Reimplemented from [CoinAbcAnyFactorization](#).

4.85.3.61 `virtual int CoinAbcTypeFactorization::checkReplacePart2 (int pivotRow, CoinSimplexDouble btranAlpha, double ftranAlpha, double ftAlpha, double acceptablePivot = 1.0e-8) [virtual]`

Checks if can replace one Column to basis, returns 0=OK, 1=Probably OK, 2=singular, 3=no room, 5 max pivots.
Implements [CoinAbcAnyFactorization](#).

4.85.3.62 `virtual void CoinAbcTypeFactorization::replaceColumnPart3 (const AbcSimplex * model, CoinIndexedVector * regionSparse, CoinIndexedVector * tableauColumn, int pivotRow, double alpha) [virtual]`

Replaces one Column to basis, partial update already in U.
Implements [CoinAbcAnyFactorization](#).

4.85.3.63 `virtual void CoinAbcTypeFactorization::replaceColumnPart3 (const AbcSimplex * model, CoinIndexedVector * regionSparse, CoinIndexedVector * tableauColumn, CoinIndexedVector * partialUpdate, int pivotRow, double alpha) [virtual]`

Replaces one Column to basis, partial update in vector.
Implements [CoinAbcAnyFactorization](#).

4.85.3.64 `void CoinAbcTypeFactorization::updatePartialUpdate (CoinIndexedVector & partialUpdate)`

Update partial Ftran by R update.

4.85.3.65 `virtual bool CoinAbcTypeFactorization::wantsTableauColumn () const [inline],[virtual]`

Returns true if wants tableauColumn in replaceColumn.
Reimplemented from [CoinAbcAnyFactorization](#).
Definition at line 427 of file CoinAbcBaseFactorization.hpp.

4.85.3.66 `int CoinAbcTypeFactorization::replaceColumnU (CoinIndexedVector * regionSparse, CoinBigIndex * deletedPosition, CoinSimplexInt * deletedColumns, CoinSimplexInt pivotRow)`

Combines BtranU and store which elements are to be deleted returns number to be deleted.

4.85.3.67 `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 region-Sparse starts as zero and is zero at end. Note - if regionSparse2 packed on input - will be packed on output
Implements [CoinAbcAnyFactorization](#).

4.85.3.68 `virtual int CoinAbcTypeFactorization::updateColumnFTPart1 (CoinIndexedVector & regionSparse) [virtual]`

Implements [CoinAbcAnyFactorization](#).

4.85.3.69 `virtual void CoinAbcTypeFactorization::updateColumnFTPart2 (CoinIndexedVector & regionSparse) [virtual]`

Implements [CoinAbcAnyFactorization](#).

4.85.3.70 `virtual void CoinAbcTypeFactorization::updateColumnFT (CoinIndexedVector & regionSparseFT, CoinIndexedVector & partialUpdate, int which) [virtual]`

Updates one column (FTRAN) Tries to do FT update puts partial update in vector.

Implements [CoinAbcAnyFactorization](#).

4.85.3.71 `virtual CoinSimplexInt CoinAbcTypeFactorization::updateColumn (CoinIndexedVector & regionSparse) const [virtual]`

This version has same effect as above with FTUpdate==false so number returned is always ≥ 0 .

Implements [CoinAbcAnyFactorization](#).

4.85.3.72 `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](#).

4.85.3.73 `virtual CoinSimplexInt CoinAbcTypeFactorization::updateColumnTranspose (CoinIndexedVector & regionSparse) const [virtual]`

Updates one column (BTRAN) from regionSparse2 regionSparse starts as zero and is zero at end Note - if region-Sparse2 packed on input - will be packed on output.

Implements [CoinAbcAnyFactorization](#).

4.85.3.74 `virtual void CoinAbcTypeFactorization::updateFullColumn (CoinIndexedVector & regionSparse) const [virtual]`

Updates one full column (FTRAN)

Implements [CoinAbcAnyFactorization](#).

4.85.3.75 `virtual void CoinAbcTypeFactorization::updateFullColumnTranspose (CoinIndexedVector & regionSparse) const [virtual]`

Updates one full column (BTRAN)

Implements [CoinAbcAnyFactorization](#).

4.85.3.76 `virtual void CoinAbcTypeFactorization::updateWeights (CoinIndexedVector & regionSparse) const [virtual]`

Updates one column for dual steepest edge weights (FTRAN)

Implements [CoinAbcAnyFactorization](#).

4.85.3.77 `virtual void CoinAbcTypeFactorization::updateColumnCpu (CoinIndexedVector & regionSparse, int whichCpu) const [virtual]`

Updates one column (FTRAN)

Reimplemented from [CoinAbcAnyFactorization](#).

4.85.3.78 `virtual void CoinAbcTypeFactorization::updateColumnTransposeCpu (CoinIndexedVector & regionSparse, int whichCpu) const [virtual]`

Updates one column (BTRAN)

Reimplemented from [CoinAbcAnyFactorization](#).

4.85.3.79 `void CoinAbcTypeFactorization::unpack (CoinIndexedVector * regionFrom, CoinIndexedVector * regionTo) const`

4.85.3.80 `void CoinAbcTypeFactorization::pack (CoinIndexedVector * regionFrom, CoinIndexedVector * regionTo) const`

4.85.3.81 `void CoinAbcTypeFactorization::goSparse () [inline],[virtual]`

makes a row copy of L for speed and to allow very sparse problems

Reimplemented from [CoinAbcAnyFactorization](#).

Definition at line 487 of file `CoinAbcBaseFactorization.hpp`.

4.85.3.82 `void CoinAbcTypeFactorization::goSparse2 ()`

4.85.3.83 `virtual void CoinAbcTypeFactorization::checkMarkArrays () const [virtual]`

Reimplemented from [CoinAbcAnyFactorization](#).

4.85.3.84 `CoinSimplexInt CoinAbcTypeFactorization::sparseThreshold () const [inline]`

get sparse threshold

Definition at line 494 of file `CoinAbcBaseFactorization.hpp`.

4.85.3.85 `void CoinAbcTypeFactorization::sparseThreshold (CoinSimplexInt value)`

set sparse threshold

4.85.3.86 `void CoinAbcTypeFactorization::clearArrays () [inline],[virtual]`

Get rid of all memory.

Reimplemented from [CoinAbcAnyFactorization](#).

Definition at line 506 of file `CoinAbcBaseFactorization.hpp`.

4.85.3.87 `void CoinAbcTypeFactorization::checkSparse ()`

See if worth going sparse.

4.85.3.88 `void CoinAbcTypeFactorization::gutsOfDestructor (CoinSimplexInt type = 1)`

The real work of constructors etc 0 just scalars, 1 bit normal.

4.85.3.89 `void CoinAbcTypeFactorization::gutsOfInitialize (CoinSimplexInt type)`

1 bit - tolerances etc, 2 more, 4 dummy arrays

4.85.3.90 `void CoinAbcTypeFactorization::gutsOfCopy (const CoinAbcTypeFactorization & other)`

4.85.3.91 `void CoinAbcTypeFactorization::resetStatistics ()`

Reset all sparsity etc statistics.

4.85.3.92 void CoinAbcTypeFactorization::printRegion (const CoinIndexedVector & *vector*, const char * *where*) const

4.85.3.93 virtual void CoinAbcTypeFactorization::getAreas (CoinSimplexInt *numberRows*, CoinSimplexInt *numberColumns*, CoinBigIndex *maximumL*, CoinBigIndex *maximumU*) [virtual]

Gets space for a factorization, called by constructors.

Implements [CoinAbcAnyFactorization](#).

4.85.3.94 virtual void CoinAbcTypeFactorization::preProcess () [virtual]

PreProcesses column ordered copy of basis.

Implements [CoinAbcAnyFactorization](#).

4.85.3.95 void CoinAbcTypeFactorization::preProcess (CoinSimplexInt)

4.85.3.96 double CoinAbcTypeFactorization::preProcess3 ()

Return largest element.

4.85.3.97 void CoinAbcTypeFactorization::preProcess4 ()

4.85.3.98 virtual CoinSimplexInt CoinAbcTypeFactorization::factor (AbcSimplex * *model*) [virtual]

Does most of factorization.

Implements [CoinAbcAnyFactorization](#).

4.85.3.99 virtual void CoinAbcTypeFactorization::postProcess (const CoinSimplexInt * *sequence*, CoinSimplexInt * *pivotVariable*) [virtual]

Does post processing on valid factorization - putting variables on correct rows.

Implements [CoinAbcAnyFactorization](#).

4.85.3.100 virtual void CoinAbcTypeFactorization::makeNonSingular (CoinSimplexInt * *sequence*) [virtual]

Makes a non-singular basis by replacing variables.

Implements [CoinAbcAnyFactorization](#).

4.85.3.101 CoinSimplexInt CoinAbcTypeFactorization::factorSparse () [protected]

Does sparse phase of factorization return code is <0 error, 0= finished.

4.85.3.102 CoinSimplexInt CoinAbcTypeFactorization::factorDense () [protected]

Does dense phase of factorization return code is <0 error, 0= finished.

4.85.3.103 bool CoinAbcTypeFactorization::pivotOneOtherRow (CoinSimplexInt *pivotRow*, CoinSimplexInt *pivotColumn*) [protected]

Pivots when just one other row so faster?

4.85.3.104 bool CoinAbcTypeFactorization::pivotRowSingleton (CoinSimplexInt *pivotRow*, CoinSimplexInt *pivotColumn*) [protected]

Does one pivot on Row Singleton in factorization.

4.85.3.105 void CoinAbcTypeFactorization::pivotColumnSingleton (CoinSimplexInt *pivotRow*, CoinSimplexInt *pivotColumn*)
[protected]

Does one pivot on Column Singleton in factorization (can't return false)

4.85.3.106 void CoinAbcTypeFactorization::afterPivot (CoinSimplexInt *pivotRow*, CoinSimplexInt *pivotColumn*)
[protected]

After pivoting.

4.85.3.107 int CoinAbcTypeFactorization::wantToGoDense () [protected]

After pivoting - returns true if need to go dense.

4.85.3.108 bool CoinAbcTypeFactorization::getColumnSpace (CoinSimplexInt *iColumn*, CoinSimplexInt *extraNeeded*)
[protected]

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.

4.85.3.109 bool CoinAbcTypeFactorization::reorderU () [protected]

Reorders U so contiguous and in order (if there is space) Returns true if it could.

4.85.3.110 bool CoinAbcTypeFactorization::getColumnSpacelaterR (CoinSimplexInt *iColumn*, CoinFactorizationDouble *value*,
CoinSimplexInt *iRow*) [protected]

getColumnSpacelaterR.

Gets space for one extra R element in Column may have to do compression (returns true) also moves existing vector

4.85.3.111 CoinBigIndex CoinAbcTypeFactorization::getColumnSpacelater (CoinSimplexInt *iColumn*, CoinFactorizationDouble
value, CoinSimplexInt *iRow*) [protected]

getColumnSpacelater.

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)

4.85.3.112 bool CoinAbcTypeFactorization::getRowSpace (CoinSimplexInt *iRow*, CoinSimplexInt *extraNeeded*)
[protected]

Gets space for one Row with given length,

may have to do compression (returns True if successful), also moves existing vector

4.85.3.113 bool CoinAbcTypeFactorization::getRowSpacelater (CoinSimplexInt *iRow*, CoinSimplexInt *extraNeeded*)
[protected]

Gets space for one Row with given length while iterating,

may have to do compression (returns True if successful), also moves existing vector

4.85.3.114 void CoinAbcTypeFactorization::checkConsistency () [protected]

Checks that row and column copies look OK.

4.85.3.115 `void CoinAbcTypeFactorization::addLink (CoinSimplexInt index, CoinSimplexInt count) [inline], [protected]`

Adds a link in chain of equal counts.

Definition at line 611 of file CoinAbcBaseFactorization.hpp.

4.85.3.116 `void CoinAbcTypeFactorization::deleteLink (CoinSimplexInt index) [inline], [protected]`

Deletes a link in chain of equal counts.

Definition at line 623 of file CoinAbcBaseFactorization.hpp.

4.85.3.117 `void CoinAbcTypeFactorization::modifyLink (CoinSimplexInt index, CoinSimplexInt count) [inline], [protected]`

Modifies links in chain of equal counts.

Definition at line 641 of file CoinAbcBaseFactorization.hpp.

4.85.3.118 `void CoinAbcTypeFactorization::separateLinks () [protected]`

Separate out links with same row/column count.

4.85.3.119 `void CoinAbcTypeFactorization::separateLinks (CoinSimplexInt , CoinSimplexInt) [protected]`

4.85.3.120 `void CoinAbcTypeFactorization::cleanup () [protected]`

Cleans up at end of factorization.

4.85.3.121 `void CoinAbcTypeFactorization::doAddresses () [protected]`

Set up addresses from arrays.

4.85.3.122 `void CoinAbcTypeFactorization::updateColumnL (CoinIndexedVector * region, CoinAbcStatistics & statistics) const [protected]`

Updates part of column (FTRANL)

4.85.3.123 `void CoinAbcTypeFactorization::updateColumnLDensish (CoinIndexedVector * region) const [protected]`

Updates part of column (FTRANL) when densish.

4.85.3.124 `void CoinAbcTypeFactorization::updateColumnLDense (CoinIndexedVector * region) const [protected]`

Updates part of column (FTRANL) when dense (i.e. do as inner products)

4.85.3.125 `void CoinAbcTypeFactorization::updateColumnLSparse (CoinIndexedVector * region) const [protected]`

Updates part of column (FTRANL) when sparse.

4.85.3.126 `void CoinAbcTypeFactorization::updateColumnR (CoinIndexedVector * region, CoinAbcStatistics & statistics) const [protected]`

Updates part of column (FTRANR) without FT update.

4.85.3.127 `bool CoinAbcTypeFactorization::storeFT (const CoinIndexedVector * regionFT) [protected]`

Store update after doing L and R - returns false if no room.

4.85.3.128 void CoinAbcTypeFactorization::updateColumnU (CoinIndexedVector * *region*, CoinAbcStatistics & *statistics*) const [protected]

Updates part of column (FTRANU)

4.85.3.129 void CoinAbcTypeFactorization::updateColumnUSparse (CoinIndexedVector * *regionSparse*) const [protected]

Updates part of column (FTRANU) when sparse.

4.85.3.130 void CoinAbcTypeFactorization::updateColumnUDensish (CoinIndexedVector * *regionSparse*) const [protected]

Updates part of column (FTRANU)

4.85.3.131 void CoinAbcTypeFactorization::updateColumnUDense (CoinIndexedVector * *regionSparse*) const [protected]

Updates part of column (FTRANU) when dense (i.e. do as inner products)

4.85.3.132 void CoinAbcTypeFactorization::updateTwoColumnsUDensish (CoinSimplexInt & *numberNonZero1*, CoinFactorizationDouble *COIN_RESTRICT *region1*, CoinSimplexInt *COIN_RESTRICT *index1*, CoinSimplexInt & *numberNonZero2*, CoinFactorizationDouble *COIN_RESTRICT *region2*, CoinSimplexInt *COIN_RESTRICT *index2*) const [protected]

Updates part of 2 columns (FTRANU) real work.

4.85.3.133 void CoinAbcTypeFactorization::updateColumnPFI (CoinIndexedVector * *regionSparse*) const [protected]

Updates part of column PFI (FTRAN) (after rest)

4.85.3.134 void CoinAbcTypeFactorization::updateColumnTransposePFI (CoinIndexedVector * *region*) const [protected]

Updates part of column transpose PFI (BTRAN) (before rest)

4.85.3.135 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

4.85.3.136 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

4.85.3.137 void CoinAbcTypeFactorization::updateColumnTransposeUSparse (CoinIndexedVector * *region*) const [protected]

Updates part of column transpose (BTRANU) when sparse, assumes index is sorted i.e. region is correct

4.85.3.138 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

4.85.3.139 void CoinAbcTypeFactorization::updateColumnTransposeR (CoinIndexedVector * *region*, CoinAbcStatistics & *statistics*) const [protected]

Updates part of column transpose (BTRANR)

4.85.3.140 void CoinAbcTypeFactorization::updateColumnTransposeRDensish (CoinIndexedVector * *region*) const [protected]

Updates part of column transpose (BTRANR) when dense.

4.85.3.141 void CoinAbcTypeFactorization::updateColumnTransposeRSparse (CoinIndexedVector * *region*) const [protected]

Updates part of column transpose (BTRANR) when sparse.

4.85.3.142 void CoinAbcTypeFactorization::updateColumnTransposeL (CoinIndexedVector * *region*, CoinAbcStatistics & *statistics*) const [protected]

Updates part of column transpose (BTRANL)

4.85.3.143 void CoinAbcTypeFactorization::updateColumnTransposeLDensish (CoinIndexedVector * *region*) const [protected]

Updates part of column transpose (BTRANL) when densish by column.

4.85.3.144 void CoinAbcTypeFactorization::updateColumnTransposeLByRow (CoinIndexedVector * *region*) const [protected]

Updates part of column transpose (BTRANL) when densish by row.

4.85.3.145 void CoinAbcTypeFactorization::updateColumnTransposeLSparse (CoinIndexedVector * *region*) const [protected]

Updates part of column transpose (BTRANL) when sparse (by Row)

4.85.3.146 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)

4.85.3.147 CoinSimplexInt CoinAbcTypeFactorization::checkPivot (CoinSimplexDouble *saveFromU*, CoinSimplexDouble *oldPivot*) const [protected]

Returns accuracy status of replaceColumn returns 0=OK, 1=Probably OK, 2=singular.

4.85.3.148 `int CoinAbcTypeFactorization::pivot (CoinSimplexInt pivotRow, CoinSimplexInt pivotColumn, CoinBigIndex pivotRowPosition, CoinBigIndex pivotColumnPosition, CoinFactorizationDouble *COIN_RESTRICT work, CoinSimplexUnsignedInt *COIN_RESTRICT workArea2, CoinSimplexInt increment2, int *COIN_RESTRICT markRow)` [protected]

0 fine, -99 singular, 2 dense

4.85.3.149 `int CoinAbcTypeFactorization::pivot (CoinSimplexInt & pivotRow, CoinSimplexInt & pivotColumn, CoinBigIndex pivotRowPosition, CoinBigIndex pivotColumnPosition, int *COIN_RESTRICT markRow)` [protected]

4.85.3.150 `bool CoinAbcTypeFactorization::gotLCopy () const` [inline]

Definition at line 1142 of file CoinAbcBaseFactorization.hpp.

4.85.3.151 `void CoinAbcTypeFactorization::setNoGotLCopy ()` [inline]

Definition at line 1143 of file CoinAbcBaseFactorization.hpp.

4.85.3.152 `void CoinAbcTypeFactorization::setYesGotLCopy ()` [inline]

Definition at line 1144 of file CoinAbcBaseFactorization.hpp.

4.85.3.153 `bool CoinAbcTypeFactorization::gotRCopy () const` [inline]

Definition at line 1145 of file CoinAbcBaseFactorization.hpp.

4.85.3.154 `void CoinAbcTypeFactorization::setNoGotRCopy ()` [inline]

Definition at line 1146 of file CoinAbcBaseFactorization.hpp.

4.85.3.155 `void CoinAbcTypeFactorization::setYesGotRCopy ()` [inline]

Definition at line 1147 of file CoinAbcBaseFactorization.hpp.

4.85.3.156 `bool CoinAbcTypeFactorization::gotUCopy () const` [inline]

Definition at line 1148 of file CoinAbcBaseFactorization.hpp.

4.85.3.157 `void CoinAbcTypeFactorization::setNoGotUCopy ()` [inline]

Definition at line 1149 of file CoinAbcBaseFactorization.hpp.

4.85.3.158 `void CoinAbcTypeFactorization::setYesGotUCopy ()` [inline]

Definition at line 1150 of file CoinAbcBaseFactorization.hpp.

4.85.3.159 `bool CoinAbcTypeFactorization::gotSparse () const` [inline]

Definition at line 1151 of file CoinAbcBaseFactorization.hpp.

4.85.3.160 `void CoinAbcTypeFactorization::setNoGotSparse ()` [inline]

Definition at line 1152 of file CoinAbcBaseFactorization.hpp.

4.85.3.161 `void CoinAbcTypeFactorization::setYesGotSparse ()` [inline]

Definition at line 1153 of file CoinAbcBaseFactorization.hpp.

4.85.4 Friends And Related Function Documentation

4.85.4.1 `void CoinAbcFactorizationUnitTest (const std::string & mpsDir)` [friend]

4.85.5 Member Data Documentation

4.85.5.1 `CoinSimplexInt* CoinAbcTypeFactorization::pivotColumnAddress_` [protected]

Definition at line 837 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.2 `CoinSimplexInt* CoinAbcTypeFactorization::permuteAddress_` [protected]

Definition at line 838 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.3 `CoinFactorizationDouble* CoinAbcTypeFactorization::pivotRegionAddress_` [protected]

Definition at line 839 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.4 `CoinFactorizationDouble* CoinAbcTypeFactorization::elementUAddress_` [protected]

Definition at line 840 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.5 `CoinSimplexInt* CoinAbcTypeFactorization::indexRowUAddress_` [protected]

Definition at line 841 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.6 `CoinSimplexInt* CoinAbcTypeFactorization::numberInColumnAddress_` [protected]

Definition at line 842 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.7 `CoinSimplexInt* CoinAbcTypeFactorization::numberInColumnPlusAddress_` [protected]

Definition at line 843 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.8 `CoinBigIndex* CoinAbcTypeFactorization::startColumnUAddress_` [protected]

Definition at line 849 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.9 `CoinBigIndex* CoinAbcTypeFactorization::convertRowToColumnUAddress_` [protected]

Definition at line 851 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.10 `CoinBigIndex* CoinAbcTypeFactorization::convertColumnToRowUAddress_` [protected]

Definition at line 853 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.11 `CoinFactorizationDouble* CoinAbcTypeFactorization::elementRowUAddress_` [protected]

Definition at line 857 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.12 `CoinBigIndex* CoinAbcTypeFactorization::startRowUAddress_` [protected]

Definition at line 859 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.13 `CoinSimplexInt* CoinAbcTypeFactorization::numberInRowAddress_` [protected]

Definition at line 860 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.14 CoinSimplexInt* CoinAbcTypeFactorization::indexColumnUAddress_ [protected]

Definition at line 861 of file CoinAbcBaseFactorization.hpp.

4.85.5.15 CoinSimplexInt* CoinAbcTypeFactorization::firstCountAddress_ [protected]

Definition at line 862 of file CoinAbcBaseFactorization.hpp.

4.85.5.16 CoinSimplexInt* CoinAbcTypeFactorization::nextCountAddress_ [protected]

Next Row/Column with count.

Definition at line 864 of file CoinAbcBaseFactorization.hpp.

4.85.5.17 CoinSimplexInt* CoinAbcTypeFactorization::lastCountAddress_ [protected]

Previous Row/Column with count.

Definition at line 866 of file CoinAbcBaseFactorization.hpp.

4.85.5.18 CoinSimplexInt* CoinAbcTypeFactorization::nextColumnAddress_ [protected]

Definition at line 867 of file CoinAbcBaseFactorization.hpp.

4.85.5.19 CoinSimplexInt* CoinAbcTypeFactorization::lastColumnAddress_ [protected]

Definition at line 868 of file CoinAbcBaseFactorization.hpp.

4.85.5.20 CoinSimplexInt* CoinAbcTypeFactorization::nextRowAddress_ [protected]

Definition at line 869 of file CoinAbcBaseFactorization.hpp.

4.85.5.21 CoinSimplexInt* CoinAbcTypeFactorization::lastRowAddress_ [protected]

Definition at line 870 of file CoinAbcBaseFactorization.hpp.

4.85.5.22 CoinSimplexInt* CoinAbcTypeFactorization::saveColumnAddress_ [protected]

Definition at line 871 of file CoinAbcBaseFactorization.hpp.

4.85.5.23 CoinCheckZero* CoinAbcTypeFactorization::markRowAddress_ [protected]

Definition at line 873 of file CoinAbcBaseFactorization.hpp.

4.85.5.24 CoinSimplexInt* CoinAbcTypeFactorization::listAddress_ [protected]

Definition at line 874 of file CoinAbcBaseFactorization.hpp.

4.85.5.25 CoinFactorizationDouble* CoinAbcTypeFactorization::elementLAddress_ [protected]

Definition at line 875 of file CoinAbcBaseFactorization.hpp.

4.85.5.26 CoinSimplexInt* CoinAbcTypeFactorization::indexRowLAddress_ [protected]

Definition at line 876 of file CoinAbcBaseFactorization.hpp.

4.85.5.27 CoinBigIndex* CoinAbcTypeFactorization::startColumnLAddress_ [protected]

Definition at line 877 of file CoinAbcBaseFactorization.hpp.

4.85.5.28 `CoinBigIndex* CoinAbcTypeFactorization::startRowLAddress_` [protected]

Definition at line 879 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.29 `CoinSimplexInt* CoinAbcTypeFactorization::pivotLinkedBackwardsAddress_` [protected]

Definition at line 881 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.30 `CoinSimplexInt* CoinAbcTypeFactorization::pivotLinkedForwardsAddress_` [protected]

Definition at line 882 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.31 `CoinSimplexInt* CoinAbcTypeFactorization::pivotLOrderAddress_` [protected]

Definition at line 883 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.32 `CoinBigIndex* CoinAbcTypeFactorization::startColumnRAddress_` [protected]

Definition at line 884 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.33 `CoinFactorizationDouble* CoinAbcTypeFactorization::elementRAddress_` [protected]

Elements of R .

Definition at line 886 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.34 `CoinSimplexInt* CoinAbcTypeFactorization::indexRowRAddress_` [protected]

Row indices for R .

Definition at line 888 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.35 `CoinSimplexInt* CoinAbcTypeFactorization::indexColumnLAddress_` [protected]

Definition at line 889 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.36 `CoinFactorizationDouble* CoinAbcTypeFactorization::elementByRowLAddress_` [protected]

Definition at line 890 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.37 `CoinFactorizationDouble* CoinAbcTypeFactorization::denseAreaAddress_` [protected]

Definition at line 892 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.38 `CoinFactorizationDouble* CoinAbcTypeFactorization::workAreaAddress_` [protected]

Definition at line 894 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.39 `CoinSimplexUnsignedInt* CoinAbcTypeFactorization::workArea2Address_` [protected]

Definition at line 895 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.40 `CoinSimplexInt* CoinAbcTypeFactorization::sparseAddress_` [mutable], [protected]

Definition at line 896 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.41 `CoinSimplexInt CoinAbcTypeFactorization::numberOfRowsExtra_` [protected]

Number of Rows after iterating.

Definition at line 902 of file CoinAbcBaseFactorization.hpp.

4.85.5.42 CoinSimplexInt CoinAbcTypeFactorization::maximumRowsExtra_ [protected]

Maximum number of Rows after iterating.

Definition at line 904 of file CoinAbcBaseFactorization.hpp.

4.85.5.43 CoinSimplexInt CoinAbcTypeFactorization::numberRowsSmall_ [protected]

Size of small inverse.

Definition at line 906 of file CoinAbcBaseFactorization.hpp.

4.85.5.44 CoinSimplexInt CoinAbcTypeFactorization::numberGoodL_ [protected]

Number factorized in L.

Definition at line 908 of file CoinAbcBaseFactorization.hpp.

4.85.5.45 CoinSimplexInt CoinAbcTypeFactorization::numberRowsLeft_ [protected]

Number Rows left (numberRows-numberGood)

Definition at line 910 of file CoinAbcBaseFactorization.hpp.

4.85.5.46 CoinBigIndex CoinAbcTypeFactorization::totalElements_ [protected]

Number of elements in U (to go) or while iterating total overall.

Definition at line 913 of file CoinAbcBaseFactorization.hpp.

4.85.5.47 CoinBigIndex CoinAbcTypeFactorization::firstZeroed_ [protected]

First place in funny copy zeroed out.

Definition at line 915 of file CoinAbcBaseFactorization.hpp.

4.85.5.48 CoinSimplexInt CoinAbcTypeFactorization::sparseThreshold_ [protected]

Below this use sparse technology - if 0 then no L row copy.

Definition at line 918 of file CoinAbcBaseFactorization.hpp.

4.85.5.49 CoinSimplexInt CoinAbcTypeFactorization::numberR_ [protected]

Number in R.

Definition at line 921 of file CoinAbcBaseFactorization.hpp.

4.85.5.50 CoinBigIndex CoinAbcTypeFactorization::lengthR_ [protected]

Length of R stuff.

Definition at line 923 of file CoinAbcBaseFactorization.hpp.

4.85.5.51 CoinBigIndex CoinAbcTypeFactorization::lengthAreaR_ [protected]

length of area reserved for R

Definition at line 925 of file CoinAbcBaseFactorization.hpp.

4.85.5.52 CoinBigIndex CoinAbcTypeFactorization::numberL_ [protected]

Number in L.

Definition at line 927 of file CoinAbcBaseFactorization.hpp.

4.85.5.53 CoinBigIndex CoinAbcTypeFactorization::baseL_ [protected]

Base of L.

Definition at line 929 of file CoinAbcBaseFactorization.hpp.

4.85.5.54 CoinBigIndex CoinAbcTypeFactorization::lengthL_ [protected]

Length of L.

Definition at line 931 of file CoinAbcBaseFactorization.hpp.

4.85.5.55 CoinBigIndex CoinAbcTypeFactorization::lengthAreaL_ [protected]

Length of area reserved for L.

Definition at line 933 of file CoinAbcBaseFactorization.hpp.

4.85.5.56 CoinSimplexInt CoinAbcTypeFactorization::numberU_ [protected]

Number in U.

Definition at line 935 of file CoinAbcBaseFactorization.hpp.

4.85.5.57 CoinBigIndex CoinAbcTypeFactorization::maximumU_ [protected]

Maximum space used in U.

Definition at line 937 of file CoinAbcBaseFactorization.hpp.

4.85.5.58 CoinBigIndex CoinAbcTypeFactorization::lengthU_ [protected]

Length of U.

Definition at line 939 of file CoinAbcBaseFactorization.hpp.

4.85.5.59 CoinBigIndex CoinAbcTypeFactorization::lengthAreaU_ [protected]

Length of area reserved for U.

Definition at line 941 of file CoinAbcBaseFactorization.hpp.

4.85.5.60 CoinBigIndex CoinAbcTypeFactorization::lastEntryByColumnU_ [protected]

Last entry by column for U.

Definition at line 943 of file CoinAbcBaseFactorization.hpp.

4.85.5.61 CoinBigIndex CoinAbcTypeFactorization::lastEntryByRowU_ [protected]

Last entry by row for U.

Definition at line 951 of file CoinAbcBaseFactorization.hpp.

4.85.5.62 CoinSimplexInt CoinAbcTypeFactorization::numberTrials_ [protected]

Number of trials before rejection.

Definition at line 953 of file CoinAbcBaseFactorization.hpp.

4.85.5.63 CoinSimplexInt CoinAbcTypeFactorization::leadingDimension_ [protected]

Leading dimension for dense.

Definition at line 956 of file CoinAbcBaseFactorization.hpp.

4.85.5.64 CoinIntArrayWithLength CoinAbcTypeFactorization::pivotColumn_ [protected]

Pivot order for each Column.

Definition at line 965 of file CoinAbcBaseFactorization.hpp.

4.85.5.65 CoinIntArrayWithLength CoinAbcTypeFactorization::permute_ [protected]

Permutation vector for pivot row order.

Definition at line 967 of file CoinAbcBaseFactorization.hpp.

4.85.5.66 CoinBigIndexArrayWithLength CoinAbcTypeFactorization::startRowU_ [protected]

Start of each Row as pointer.

Definition at line 969 of file CoinAbcBaseFactorization.hpp.

4.85.5.67 CoinIntArrayWithLength CoinAbcTypeFactorization::numberInRow_ [protected]

Number in each Row.

Definition at line 971 of file CoinAbcBaseFactorization.hpp.

4.85.5.68 CoinIntArrayWithLength CoinAbcTypeFactorization::numberInColumn_ [protected]

Number in each Column.

Definition at line 973 of file CoinAbcBaseFactorization.hpp.

4.85.5.69 CoinIntArrayWithLength CoinAbcTypeFactorization::numberInColumnPlus_ [protected]

Number in each Column including pivoted.

Definition at line 975 of file CoinAbcBaseFactorization.hpp.

4.85.5.70 CoinIntArrayWithLength CoinAbcTypeFactorization::firstCount_ [protected]

First Row/Column with count of k, can tell which by offset - Rows then Columns.

Definition at line 978 of file CoinAbcBaseFactorization.hpp.

4.85.5.71 CoinIntArrayWithLength CoinAbcTypeFactorization::nextColumn_ [protected]

Next Column in memory order.

Definition at line 980 of file CoinAbcBaseFactorization.hpp.

4.85.5.72 CoinIntArrayWithLength CoinAbcTypeFactorization::lastColumn_ [protected]

Previous Column in memory order.

Definition at line 982 of file CoinAbcBaseFactorization.hpp.

4.85.5.73 CoinIntArrayWithLength CoinAbcTypeFactorization::nextRow_ [protected]

Next Row in memory order.

Definition at line 984 of file CoinAbcBaseFactorization.hpp.

4.85.5.74 CoinIntArrayWithLength CoinAbcTypeFactorization::lastRow_ [protected]

Previous Row in memory order.

Definition at line 986 of file CoinAbcBaseFactorization.hpp.

4.85.5.75 CoinIntArrayWithLength CoinAbcTypeFactorization::saveColumn_ [protected]

Columns left to do in a single pivot.

Definition at line 988 of file CoinAbcBaseFactorization.hpp.

4.85.5.76 CoinIntArrayWithLength CoinAbcTypeFactorization::markRow_ [protected]

Marks rows to be updated.

Definition at line 990 of file CoinAbcBaseFactorization.hpp.

4.85.5.77 CoinIntArrayWithLength CoinAbcTypeFactorization::indexColumnU_ [protected]

Base address for U (may change)

Definition at line 992 of file CoinAbcBaseFactorization.hpp.

4.85.5.78 CoinFactorizationDoubleArrayWithLength CoinAbcTypeFactorization::pivotRegion_ [protected]

Inverses of pivot values.

Definition at line 994 of file CoinAbcBaseFactorization.hpp.

4.85.5.79 CoinFactorizationDoubleArrayWithLength CoinAbcTypeFactorization::elementU_ [protected]

Elements of U.

Definition at line 996 of file CoinAbcBaseFactorization.hpp.

4.85.5.80 CoinIntArrayWithLength CoinAbcTypeFactorization::indexRowU_ [protected]

Row indices of U.

Definition at line 998 of file CoinAbcBaseFactorization.hpp.

4.85.5.81 CoinBigIndexArrayWithLength CoinAbcTypeFactorization::startColumnU_ [protected]

Start of each column in U.

Definition at line 1000 of file CoinAbcBaseFactorization.hpp.

4.85.5.82 CoinBigIndexArrayWithLength CoinAbcTypeFactorization::convertRowToColumnU_ [protected]

Converts rows to columns in U.

Definition at line 1007 of file CoinAbcBaseFactorization.hpp.

4.85.5.83 CoinBigIndexArrayWithLength CoinAbcTypeFactorization::convertColumnToRowU_ [protected]

Converts columns to rows in U.

Definition at line 1010 of file CoinAbcBaseFactorization.hpp.

4.85.5.84 CoinFactorizationDoubleArrayWithLength CoinAbcTypeFactorization::elementRowU_ [protected]

Elements of U by row.

Definition at line 1015 of file CoinAbcBaseFactorization.hpp.

4.85.5.85 CoinFactorizationDoubleArrayWithLength CoinAbcTypeFactorization::elementL_ [protected]

Elements of L.

Definition at line 1018 of file CoinAbcBaseFactorization.hpp.

4.85.5.86 CoinIntArrayWithLength CoinAbcTypeFactorization::indexRowL_ [protected]

Row indices of L.

Definition at line 1020 of file CoinAbcBaseFactorization.hpp.

4.85.5.87 CoinBigIndexArrayWithLength CoinAbcTypeFactorization::startColumnL_ [protected]

Start of each column in L.

Definition at line 1022 of file CoinAbcBaseFactorization.hpp.

4.85.5.88 CoinFactorizationDoubleArrayWithLength CoinAbcTypeFactorization::denseArea_ [protected]

Dense area.

Definition at line 1025 of file CoinAbcBaseFactorization.hpp.

4.85.5.89 CoinFactorizationDoubleArrayWithLength CoinAbcTypeFactorization::workArea_ [protected]

First work area.

Definition at line 1028 of file CoinAbcBaseFactorization.hpp.

4.85.5.90 CoinUnsignedIntArrayWithLength CoinAbcTypeFactorization::workArea2_ [protected]

Second work area.

Definition at line 1030 of file CoinAbcBaseFactorization.hpp.

4.85.5.91 CoinBigIndexArrayWithLength CoinAbcTypeFactorization::startRowL_ [protected]

Start of each row in L.

Definition at line 1033 of file CoinAbcBaseFactorization.hpp.

4.85.5.92 `CoinIntArrayWithLength CoinAbcTypeFactorization::indexColumnL_` [protected]

Index of column in row for L.

Definition at line 1035 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.93 `CoinFactorizationDoubleArrayWithLength CoinAbcTypeFactorization::elementByRowL_` [protected]

Elements in L (row copy)

Definition at line 1037 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.94 `CoinIntArrayWithLength CoinAbcTypeFactorization::sparse_` [mutable], [protected]

Sparse regions.

Definition at line 1039 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.95 `CoinSimplexInt CoinAbcTypeFactorization::messageLevel_` [protected]

Detail in messages.

Definition at line 1042 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.96 `CoinBigIndex CoinAbcTypeFactorization::numberCompressions_` [protected]

Number of compressions done.

Definition at line 1044 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.97 `CoinSimplexInt CoinAbcTypeFactorization::lastSlack_` [protected]

Definition at line 1046 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.98 `double CoinAbcTypeFactorization::ftranCountInput_` [mutable], [protected]

To decide how to solve.

Definition at line 1049 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.99 `double CoinAbcTypeFactorization::ftranCountAfterL_` [mutable], [protected]

Definition at line 1050 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.100 `double CoinAbcTypeFactorization::ftranCountAfterR_` [mutable], [protected]

Definition at line 1051 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.101 `double CoinAbcTypeFactorization::ftranCountAfterU_` [mutable], [protected]

Definition at line 1052 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.102 `double CoinAbcTypeFactorization::ftranAverageAfterL_` [protected]

Definition at line 1053 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.103 `double CoinAbcTypeFactorization::ftranAverageAfterR_` [protected]

Definition at line 1054 of file `CoinAbcBaseFactorization.hpp`.

4.85.5.104 `double CoinAbcTypeFactorization::ftranAverageAfterU_` [protected]

Definition at line 1055 of file CoinAbcBaseFactorization.hpp.

4.85.5.105 `CoinSimplexInt CoinAbcTypeFactorization::numberFtranCounts_` [mutable], [protected]

Definition at line 1060 of file CoinAbcBaseFactorization.hpp.

4.85.5.106 `CoinSimplexInt CoinAbcTypeFactorization::maximumRows_` [protected]

Maximum rows (ever) (here to use double alignment)

Definition at line 1063 of file CoinAbcBaseFactorization.hpp.

4.85.5.107 `double CoinAbcTypeFactorization::ftranFTCountInput_` [mutable], [protected]

Definition at line 1065 of file CoinAbcBaseFactorization.hpp.

4.85.5.108 `double CoinAbcTypeFactorization::ftranFTCountAfterL_` [mutable], [protected]

Definition at line 1066 of file CoinAbcBaseFactorization.hpp.

4.85.5.109 `double CoinAbcTypeFactorization::ftranFTCountAfterR_` [mutable], [protected]

Definition at line 1067 of file CoinAbcBaseFactorization.hpp.

4.85.5.110 `double CoinAbcTypeFactorization::ftranFTCountAfterU_` [mutable], [protected]

Definition at line 1068 of file CoinAbcBaseFactorization.hpp.

4.85.5.111 `double CoinAbcTypeFactorization::ftranFTAverageAfterL_` [protected]

Definition at line 1069 of file CoinAbcBaseFactorization.hpp.

4.85.5.112 `double CoinAbcTypeFactorization::ftranFTAverageAfterR_` [protected]

Definition at line 1070 of file CoinAbcBaseFactorization.hpp.

4.85.5.113 `double CoinAbcTypeFactorization::ftranFTAverageAfterU_` [protected]

Definition at line 1071 of file CoinAbcBaseFactorization.hpp.

4.85.5.114 `CoinSimplexInt CoinAbcTypeFactorization::numberFtranFTCounts_` [mutable], [protected]

Definition at line 1076 of file CoinAbcBaseFactorization.hpp.

4.85.5.115 `CoinSimplexInt CoinAbcTypeFactorization::denseThreshold_` [protected]

Dense threshold (here to use double alignment)

Definition at line 1080 of file CoinAbcBaseFactorization.hpp.

4.85.5.116 `double CoinAbcTypeFactorization::btranCountInput_` [mutable], [protected]

Definition at line 1083 of file CoinAbcBaseFactorization.hpp.

4.85.5.117 `double CoinAbcTypeFactorization::btranCountAfterU_` [mutable], [protected]

Definition at line 1084 of file CoinAbcBaseFactorization.hpp.

4.85.5.118 **double** **CoinAbcTypeFactorization::btranCountAfterR_** [mutable], [protected]

Definition at line 1085 of file CoinAbcBaseFactorization.hpp.

4.85.5.119 **double** **CoinAbcTypeFactorization::btranCountAfterL_** [mutable], [protected]

Definition at line 1086 of file CoinAbcBaseFactorization.hpp.

4.85.5.120 **double** **CoinAbcTypeFactorization::btranAverageAfterU_** [protected]

Definition at line 1087 of file CoinAbcBaseFactorization.hpp.

4.85.5.121 **double** **CoinAbcTypeFactorization::btranAverageAfterR_** [protected]

Definition at line 1088 of file CoinAbcBaseFactorization.hpp.

4.85.5.122 **double** **CoinAbcTypeFactorization::btranAverageAfterL_** [protected]

Definition at line 1089 of file CoinAbcBaseFactorization.hpp.

4.85.5.123 **CoinSimplexInt** **CoinAbcTypeFactorization::numberBtranCounts_** [mutable], [protected]

Definition at line 1094 of file CoinAbcBaseFactorization.hpp.

4.85.5.124 **CoinSimplexInt** **CoinAbcTypeFactorization::maximumMaximumPivots_** [protected]

Maximum maximum pivots.

Definition at line 1097 of file CoinAbcBaseFactorization.hpp.

4.85.5.125 **double** **CoinAbcTypeFactorization::ftranFullCountInput_** [mutable], [protected]

To decide how to solve.

Definition at line 1100 of file CoinAbcBaseFactorization.hpp.

4.85.5.126 **double** **CoinAbcTypeFactorization::ftranFullCountAfterL_** [mutable], [protected]

Definition at line 1101 of file CoinAbcBaseFactorization.hpp.

4.85.5.127 **double** **CoinAbcTypeFactorization::ftranFullCountAfterR_** [mutable], [protected]

Definition at line 1102 of file CoinAbcBaseFactorization.hpp.

4.85.5.128 **double** **CoinAbcTypeFactorization::ftranFullCountAfterU_** [mutable], [protected]

Definition at line 1103 of file CoinAbcBaseFactorization.hpp.

4.85.5.129 **double** **CoinAbcTypeFactorization::ftranFullAverageAfterL_** [protected]

Definition at line 1104 of file CoinAbcBaseFactorization.hpp.

4.85.5.130 **double** **CoinAbcTypeFactorization::ftranFullAverageAfterR_** [protected]

Definition at line 1105 of file CoinAbcBaseFactorization.hpp.

4.85.5.131 **double** **CoinAbcTypeFactorization::ftranFullAverageAfterU_** [protected]

Definition at line 1106 of file CoinAbcBaseFactorization.hpp.

4.85.5.132 `CoinSimplexInt CoinAbcTypeFactorization::numberFtranFullCounts_` [mutable], [protected]

Definition at line 1111 of file CoinAbcBaseFactorization.hpp.

4.85.5.133 `CoinSimplexInt CoinAbcTypeFactorization::initialNumberRows_` [protected]

Rows first time nonzero.

Definition at line 1114 of file CoinAbcBaseFactorization.hpp.

4.85.5.134 `double CoinAbcTypeFactorization::btranFullCountInput_` [mutable], [protected]

To decide how to solve.

Definition at line 1117 of file CoinAbcBaseFactorization.hpp.

4.85.5.135 `double CoinAbcTypeFactorization::btranFullCountAfterL_` [mutable], [protected]

Definition at line 1118 of file CoinAbcBaseFactorization.hpp.

4.85.5.136 `double CoinAbcTypeFactorization::btranFullCountAfterR_` [mutable], [protected]

Definition at line 1119 of file CoinAbcBaseFactorization.hpp.

4.85.5.137 `double CoinAbcTypeFactorization::btranFullCountAfterU_` [mutable], [protected]

Definition at line 1120 of file CoinAbcBaseFactorization.hpp.

4.85.5.138 `double CoinAbcTypeFactorization::btranFullAverageAfterL_` [protected]

Definition at line 1121 of file CoinAbcBaseFactorization.hpp.

4.85.5.139 `double CoinAbcTypeFactorization::btranFullAverageAfterR_` [protected]

Definition at line 1122 of file CoinAbcBaseFactorization.hpp.

4.85.5.140 `double CoinAbcTypeFactorization::btranFullAverageAfterU_` [protected]

Definition at line 1123 of file CoinAbcBaseFactorization.hpp.

4.85.5.141 `CoinSimplexInt CoinAbcTypeFactorization::numberBtranFullCounts_` [mutable], [protected]

Definition at line 1128 of file CoinAbcBaseFactorization.hpp.

4.85.5.142 `CoinSimplexInt CoinAbcTypeFactorization::state_` [protected]

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...

Definition at line 1136 of file CoinAbcBaseFactorization.hpp.

4.85.5.143 `CoinBigIndex CoinAbcTypeFactorization::sizeSparseArray_` [protected]

Size in bytes of a sparseArray.

Definition at line 1138 of file CoinAbcBaseFactorization.hpp.

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

- [src/CoinAbcBaseFactorization.hpp](#)

4.86 ClpHashValue::CoinHashLink Struct Reference

Data.

```
#include <ClpNode.hpp>
```

Public Attributes

- double [value](#)
- int [index](#)
- int [next](#)

4.86.1 Detailed Description

Data.

Definition at line 335 of file ClpNode.hpp.

4.86.2 Member Data Documentation

4.86.2.1 double ClpHashValue::CoinHashLink::value

Definition at line 336 of file ClpNode.hpp.

4.86.2.2 int ClpHashValue::CoinHashLink::index

Definition at line 337 of file ClpNode.hpp.

4.86.2.3 int ClpHashValue::CoinHashLink::next

Definition at line 337 of file ClpNode.hpp.

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

- [src/ClpNode.hpp](#)

4.87 dualColumnResult Struct Reference

```
#include <AbcSimplexDual.hpp>
```

Public Attributes

- double [theta](#)
- double [totalThru](#)
- double [useThru](#)
- double [bestEverPivot](#)
- double [increaseInObjective](#)
- double [tentativeTheta](#)
- double [lastPivotValue](#)
- double [thisPivotValue](#)
- double [thruThis](#)
- double [increaseInThis](#)

- int [lastSequence](#)
- int [sequence](#)
- int [block](#)
- int [numberSwapped](#)
- int [numberRemaining](#)
- int [numberLastSwapped](#)
- bool [modifyCosts](#)

4.87.1 Detailed Description

Definition at line 23 of file `AbcSimplexDual.hpp`.

4.87.2 Member Data Documentation

4.87.2.1 double dualColumnResult::theta

Definition at line 24 of file `AbcSimplexDual.hpp`.

4.87.2.2 double dualColumnResult::totalThru

Definition at line 25 of file `AbcSimplexDual.hpp`.

4.87.2.3 double dualColumnResult::useThru

Definition at line 26 of file `AbcSimplexDual.hpp`.

4.87.2.4 double dualColumnResult::bestEverPivot

Definition at line 27 of file `AbcSimplexDual.hpp`.

4.87.2.5 double dualColumnResult::increaseInObjective

Definition at line 28 of file `AbcSimplexDual.hpp`.

4.87.2.6 double dualColumnResult::tentativeTheta

Definition at line 29 of file `AbcSimplexDual.hpp`.

4.87.2.7 double dualColumnResult::lastPivotValue

Definition at line 30 of file `AbcSimplexDual.hpp`.

4.87.2.8 double dualColumnResult::thisPivotValue

Definition at line 31 of file `AbcSimplexDual.hpp`.

4.87.2.9 double dualColumnResult::thruThis

Definition at line 32 of file `AbcSimplexDual.hpp`.

4.87.2.10 double dualColumnResult::increaseInThis

Definition at line 33 of file `AbcSimplexDual.hpp`.

4.87.2.11 int dualColumnResult::lastSequence

Definition at line 34 of file AbcSimplexDual.hpp.

4.87.2.12 int dualColumnResult::sequence

Definition at line 35 of file AbcSimplexDual.hpp.

4.87.2.13 int dualColumnResult::block

Definition at line 36 of file AbcSimplexDual.hpp.

4.87.2.14 int dualColumnResult::numberSwapped

Definition at line 37 of file AbcSimplexDual.hpp.

4.87.2.15 int dualColumnResult::numberRemaining

Definition at line 38 of file AbcSimplexDual.hpp.

4.87.2.16 int dualColumnResult::numberLastSwapped

Definition at line 39 of file AbcSimplexDual.hpp.

4.87.2.17 bool dualColumnResult::modifyCosts

Definition at line 40 of file AbcSimplexDual.hpp.

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

- [src/AbcSimplexDual.hpp](#)

4.88 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 doCrossover=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)

4.88.1 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`.

4.88.2 Constructor & Destructor Documentation

4.88.2.1 `Idiot::Idiot ()`

Default constructor.

4.88.2.2 `Idiot::Idiot (OsiSolverInterface & model)`

Constructor with model.

4.88.2.3 `Idiot::Idiot (const Idiot &)`

Copy constructor.

4.88.2.4 `Idiot::~Idiot ()`

Destructor.

4.88.3 Member Function Documentation

4.88.3.1 `Idiot& Idiot::operator= (const Idiot & rhs)`

Assignment operator. This copies the data.

4.88.3.2 `void Idiot::solve ()`

Get an approximate solution with the idiot code.

4.88.3.3 `void Idiot::crash (int numberPass, CoinMessageHandler * handler, const CoinMessages * messages, bool doCrossover = true)`

Lightweight "crash".

4.88.3.4 `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

4.88.3.5 `double Idiot::getStartingWeight () const [inline]`

Starting weight - small emphasizes feasibility, default 1.0e-4.

Definition at line 96 of file Idiot.hpp.

4.88.3.6 `void Idiot::setStartingWeight (double value) [inline]`

Definition at line 99 of file Idiot.hpp.

4.88.3.7 `double Idiot::getWeightFactor () const [inline]`

Weight factor - weight multiplied by this when changes, default 0.333.

Definition at line 104 of file Idiot.hpp.

4.88.3.8 `void Idiot::setWeightFactor (double value) [inline]`

Definition at line 107 of file Idiot.hpp.

4.88.3.9 `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.

4.88.3.10 `void Idiot::setFeasibilityTolerance (double value) [inline]`

Definition at line 116 of file Idiot.hpp.

4.88.3.11 `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.

4.88.3.12 `void Idiot::setReasonablyFeasible (double value) [inline]`

Definition at line 125 of file Idiot.hpp.

4.88.3.13 `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.

4.88.3.14 `void Idiot::setExitInfeasibility (double value) [inline]`

Definition at line 133 of file Idiot.hpp.

4.88.3.15 `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.

4.88.3.16 `void Idiot::setMajorIterations (int value) [inline]`

Definition at line 141 of file Idiot.hpp.

4.88.3.17 `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.

4.88.3.18 `void Idiot::setMinorIterations (int value) [inline]`

Definition at line 153 of file Idiot.hpp.

4.88.3.19 `int Idiot::getMinorIterations0 () const [inline]`

Definition at line 157 of file Idiot.hpp.

4.88.3.20 `void Idiot::setMinorIterations0 (int value) [inline]`

Definition at line 160 of file Idiot.hpp.

4.88.3.21 `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.

4.88.3.22 `void Idiot::setReduceIterations (int value) [inline]`

Definition at line 169 of file Idiot.hpp.

4.88.3.23 `int Idiot::getLogLevel () const [inline]`

Amount of information - default of 1 should be okay.

Definition at line 173 of file Idiot.hpp.

4.88.3.24 `void Idiot::setLogLevel (int value) [inline]`

Definition at line 176 of file Idiot.hpp.

4.88.3.25 `int Idiot::getLightweight () const [inline]`

How lightweight - 0 not, 1 yes, 2 very lightweight.

Definition at line 180 of file Idiot.hpp.

4.88.3.26 `void Idiot::setLightweight (int value) [inline]`

Definition at line 183 of file Idiot.hpp.

4.88.3.27 `int Idiot::getStrategy () const` `[inline]`

strategy

Definition at line 187 of file Idiot.hpp.

4.88.3.28 `void Idiot::setStrategy (int value)` `[inline]`

Definition at line 190 of file Idiot.hpp.

4.88.3.29 `double Idiot::getDropEnoughFeasibility () const` `[inline]`

Fine tuning - okay if feasibility drop this factor.

Definition at line 194 of file Idiot.hpp.

4.88.3.30 `void Idiot::setDropEnoughFeasibility (double value)` `[inline]`

Definition at line 197 of file Idiot.hpp.

4.88.3.31 `double Idiot::getDropEnoughWeighted () const` `[inline]`

Fine tuning - okay if weighted obj drop this factor.

Definition at line 201 of file Idiot.hpp.

4.88.3.32 `void Idiot::setDropEnoughWeighted (double value)` `[inline]`

Definition at line 204 of file Idiot.hpp.

4.88.3.33 `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:

- [src/Idiot.hpp](#)

4.89 IdiotResult Struct Reference

for use internally

```
#include <Idiot.hpp>
```

Public Attributes

- double [infeas](#)
- double [objval](#)
- double [dropThis](#)
- double [weighted](#)
- double [sumSquared](#)
- double [djAtBeginning](#)
- double [djAtEnd](#)
- int [iteration](#)

4.89.1 Detailed Description

for use internally

Definition at line 22 of file `Idiot.hpp`.

4.89.2 Member Data Documentation

4.89.2.1 `double IdiotResult::infeas`

Definition at line 23 of file `Idiot.hpp`.

4.89.2.2 `double IdiotResult::objval`

Definition at line 24 of file `Idiot.hpp`.

4.89.2.3 `double IdiotResult::dropThis`

Definition at line 25 of file `Idiot.hpp`.

4.89.2.4 `double IdiotResult::weighted`

Definition at line 26 of file `Idiot.hpp`.

4.89.2.5 `double IdiotResult::sumSquared`

Definition at line 27 of file `Idiot.hpp`.

4.89.2.6 `double IdiotResult::djAtBeginning`

Definition at line 28 of file `Idiot.hpp`.

4.89.2.7 `double IdiotResult::djAtEnd`

Definition at line 29 of file `Idiot.hpp`.

4.89.2.8 `int IdiotResult::iteration`

Definition at line 30 of file `Idiot.hpp`.

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

- [src/Idiot.hpp](#)

4.90 Info Struct Reference

***** DATA to be moved into protected section of [ClpInterior](#)

```
#include <ClpInterior.hpp>
```

Public Attributes

- `double` [atolmin](#)
- `double` [r3norm](#)
- `double` [LSdamp](#)

- double * [deltay](#)

4.90.1 Detailed Description

***** DATA to be moved into protected section of [ClpInterior](#)

Definition at line 27 of file ClpInterior.hpp.

4.90.2 Member Data Documentation

4.90.2.1 double Info::atolmin

Definition at line 28 of file ClpInterior.hpp.

4.90.2.2 double Info::r3norm

Definition at line 29 of file ClpInterior.hpp.

4.90.2.3 double Info::LSdamp

Definition at line 30 of file ClpInterior.hpp.

4.90.2.4 double* Info::deltay

Definition at line 31 of file ClpInterior.hpp.

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

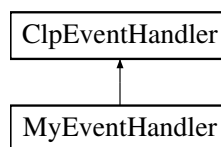
- [src/ClpInterior.hpp](#)

4.91 MyEventHandler Class Reference

This is so user can trap events and do useful stuff.

```
#include <MyEventHandler.hpp>
```

Inheritance 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

4.91.1 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.

4.91.2 Constructor & Destructor Documentation

4.91.2.1 `MyEventHandler::MyEventHandler ()`

Default constructor.

4.91.2.2 `MyEventHandler::MyEventHandler (ClpSimplex * model)`

Constructor with pointer to model (redundant as setEventHandler does)

4.91.2.3 `virtual MyEventHandler::~~MyEventHandler () [virtual]`

Destructor.

4.91.2.4 `MyEventHandler::MyEventHandler (const MyEventHandler & rhs)`

The copy constructor.

4.91.3 Member Function Documentation

4.91.3.1 `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`.

4.91.3.2 `MyEventHandler& MyEventHandler::operator= (const MyEventHandler & rhs)`

Assignment.

4.91.3.3 `virtual ClpEventHandler* MyEventHandler::clone () const` [virtual]

Clone.

Reimplemented from [ClpEventHandler](#).

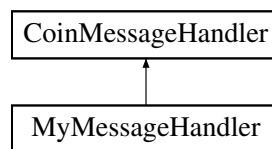
The documentation for this class was generated from the following file:

- [src/MyEventHandler.hpp](#)

4.92 MyMessageHandler Class Reference

```
#include <MyMessageHandler.hpp>
```

Inheritance 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.

4.92.1 Detailed Description

Definition at line 28 of file `MyMessageHandler.hpp`.

4.92.2 Constructor & Destructor Documentation

4.92.2.1 `MyMessageHandler::MyMessageHandler ()`

Default constructor.

4.92.2.2 `MyMessageHandler::MyMessageHandler (ClpSimplex * model, FILE * userPointer = NULL)`

Constructor with pointer to model.

4.92.2.3 `virtual MyMessageHandler::~MyMessageHandler () [virtual]`

Destructor.

4.92.2.4 `MyMessageHandler::MyMessageHandler (const MyMessageHandler &)`

The copy constructor.

4.92.2.5 `MyMessageHandler::MyMessageHandler (const CoinMessageHandler &)`

The copy constructor from an `CoinSimplexMessageHandler`.

4.92.3 Member Function Documentation

4.92.3.1 `virtual int MyMessageHandler::print () [virtual]`

4.92.3.2 `const ClpSimplex* MyMessageHandler::model () const`

Model.

4.92.3.3 `void MyMessageHandler::setModel (ClpSimplex * model)`

4.92.3.4 `const std::deque<StdVectorDouble>& MyMessageHandler::getFeasibleExtremePoints () const`

Get queue of feasible extreme points.

4.92.3.5 `void MyMessageHandler::clearFeasibleExtremePoints ()`

Empty queue of feasible extreme points.

4.92.3.6 **MyMessageHandler& MyMessageHandler::operator= (const MyMessageHandler &)**

4.92.3.7 **virtual CoinMessageHandler* MyMessageHandler::clone () const** [virtual]

Clone.

4.92.4 Member Data Documentation

4.92.4.1 **ClpSimplex* MyMessageHandler::model_** [protected]

Pointer back to model.

Definition at line 75 of file MyMessageHandler.hpp.

4.92.4.2 **std::deque<StdVectorDouble> MyMessageHandler::feasibleExtremePoints_** [protected]

Saved extreme points.

Definition at line 77 of file MyMessageHandler.hpp.

4.92.4.3 **int MyMessageHandler::iterationNumber_** [protected]

Iteration number so won't do same one twice.

Definition at line 79 of file MyMessageHandler.hpp.

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

- [src/MyMessageHandler.hpp](#)

4.93 Options Struct Reference

***** DATA to be moved into protected section of [ClpInterior](#)

```
#include <ClpInterior.hpp>
```

Public Attributes

- double [gamma](#)
- double [delta](#)
- int [MaxIter](#)
- double [FeaTol](#)
- double [OptTol](#)
- double [StepTol](#)
- double [x0min](#)
- double [z0min](#)
- double [mu0](#)
- int [LSmethod](#)
- int [LSproblem](#)
- int [LSQRMaxIter](#)
- double [LSQRatol1](#)
- double [LSQRatol2](#)
- double [LSQRconlim](#)
- int [wait](#)

4.93.1 Detailed Description

***** DATA to be moved into protected section of [ClpInterior](#)

Definition at line 44 of file ClpInterior.hpp.

4.93.2 Member Data Documentation

4.93.2.1 double Options::gamma

Definition at line 45 of file ClpInterior.hpp.

4.93.2.2 double Options::delta

Definition at line 46 of file ClpInterior.hpp.

4.93.2.3 int Options::MaxIter

Definition at line 47 of file ClpInterior.hpp.

4.93.2.4 double Options::FeaTol

Definition at line 48 of file ClpInterior.hpp.

4.93.2.5 double Options::OptTol

Definition at line 49 of file ClpInterior.hpp.

4.93.2.6 double Options::StepTol

Definition at line 50 of file ClpInterior.hpp.

4.93.2.7 double Options::x0min

Definition at line 51 of file ClpInterior.hpp.

4.93.2.8 double Options::z0min

Definition at line 52 of file ClpInterior.hpp.

4.93.2.9 double Options::mu0

Definition at line 53 of file ClpInterior.hpp.

4.93.2.10 int Options::LSmethod

Definition at line 54 of file ClpInterior.hpp.

4.93.2.11 int Options::LSproblem

Definition at line 55 of file ClpInterior.hpp.

4.93.2.12 int Options::LSQRMaxIter

Definition at line 56 of file ClpInterior.hpp.

4.93.2.13 double Options::LSQRato1

Definition at line 57 of file ClpInterior.hpp.

4.93.2.14 double Options::LSQRato2

Definition at line 58 of file ClpInterior.hpp.

4.93.2.15 double Options::LSQRconlim

Definition at line 59 of file ClpInterior.hpp.

4.93.2.16 int Options::wait

Definition at line 60 of file ClpInterior.hpp.

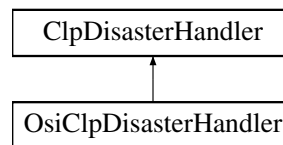
The documentation for this struct was generated from the following file:

- [src/ClpInterior.hpp](#)

4.94 OsiClpDisasterHandler Class Reference

```
#include <OsiClpSolverInterface.hpp>
```

Inheritance 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.

4.94.1 Detailed Description

Definition at line 1408 of file `OsiClpSolverInterface.hpp`.

4.94.2 Constructor & Destructor Documentation

4.94.2.1 `OsiClpDisasterHandler::OsiClpDisasterHandler (OsiClpSolverInterface * model = NULL)`

Default constructor.

4.94.2.2 `virtual OsiClpDisasterHandler::~OsiClpDisasterHandler ()` `[virtual]`

Destructor.

4.94.2.3 `OsiClpDisasterHandler::OsiClpDisasterHandler (const OsiClpDisasterHandler &)`

4.94.3 Member Function Documentation

4.94.3.1 `virtual void OsiClpDisasterHandler::intoSimplex () [virtual]`

Into simplex.

Implements [ClpDisasterHandler](#).

4.94.3.2 `virtual bool OsiClpDisasterHandler::check () const [virtual]`

Checks if disaster.

Implements [ClpDisasterHandler](#).

4.94.3.3 `virtual void OsiClpDisasterHandler::saveInfo () [virtual]`

saves information for next attempt

Implements [ClpDisasterHandler](#).

4.94.3.4 `virtual int OsiClpDisasterHandler::typeOfDisaster () [virtual]`

Type of disaster 0 can fix, 1 abort.

Reimplemented from [ClpDisasterHandler](#).

4.94.3.5 `OsiClpDisasterHandler& OsiClpDisasterHandler::operator= (const OsiClpDisasterHandler &)`

4.94.3.6 `virtual ClpDisasterHandler* OsiClpDisasterHandler::clone () const [virtual]`

Clone.

Implements [ClpDisasterHandler](#).

4.94.3.7 `void OsiClpDisasterHandler::setOsiModel (OsiClpSolverInterface * model)`

set model.

4.94.3.8 `OsiClpSolverInterface* OsiClpDisasterHandler::osiModel () const [inline]`

Get model.

Definition at line 1446 of file [OsiClpSolverInterface.hpp](#).

4.94.3.9 `void OsiClpDisasterHandler::setWhereFrom (int value) [inline]`

Set where from.

Definition at line 1449 of file [OsiClpSolverInterface.hpp](#).

4.94.3.10 `int OsiClpDisasterHandler::whereFrom () const [inline]`

Get where from.

Definition at line 1452 of file [OsiClpSolverInterface.hpp](#).

4.94.3.11 `void OsiClpDisasterHandler::setPhase (int value) [inline]`

Set phase.

Definition at line 1455 of file OsiClpSolverInterface.hpp.

4.94.3.12 `int OsiClpDisasterHandler::phase () const [inline]`

Get phase.

Definition at line 1458 of file OsiClpSolverInterface.hpp.

4.94.3.13 `bool OsiClpDisasterHandler::inTrouble () const [inline]`

are we in trouble

Definition at line 1461 of file OsiClpSolverInterface.hpp.

4.94.4 Member Data Documentation

4.94.4.1 `OsiClpSolverInterface* OsiClpDisasterHandler::osiModel_ [protected]`

Pointer to model.

Definition at line 1472 of file OsiClpSolverInterface.hpp.

4.94.4.2 `int OsiClpDisasterHandler::whereFrom_ [protected]`

Where from 0 dual (resolve) 1 crunch 2 primal (resolve) 4 dual (initialSolve) 6 primal (initialSolve)

Definition at line 1480 of file OsiClpSolverInterface.hpp.

4.94.4.3 `int OsiClpDisasterHandler::phase_ [protected]`

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

Definition at line 1488 of file OsiClpSolverInterface.hpp.

4.94.4.4 `bool OsiClpDisasterHandler::inTrouble_ [protected]`

Are we in trouble.

Definition at line 1490 of file OsiClpSolverInterface.hpp.

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

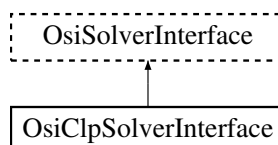
- [src/OsiClp/OsiClpSolverInterface.hpp](#)

4.95 OsiClpSolverInterface Class Reference

Clp Solver Interface.

```
#include <OsiClpSolverInterface.hpp>
```

Inheritance 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 [getBlvARow](#) (int row, double *z, double *slack=NULL) const
Get a row of the tableau (slack part in slack if not NULL)
- virtual void [getBlvARow](#) (int row, CoinIndexedVector *z, CoinIndexedVector *slack=NULL, bool keepScaled=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 [getBlvRow](#) (int row, double *z) const
Get a row of the basis inverse.
- virtual void [getBlvACol](#) (int col, double *vec) const
Get a column of the tableau.
- virtual void [getBlvACol](#) (int col, CoinIndexedVector *vec) const

- *Get a column of the tableau.*
virtual void [getBlvACol](#) (CoinIndexedVector *vec) const
Update (i.e.
- virtual void [getBlvCol](#) (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 returned by [getBlvACol\(\)](#) and [getBlvCol\(\)](#)).
- 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 colln, int colOut, int outStatus)
Perform a pivot by substituting a colln for colOut in the basis.
- virtual int [primalPivotResult](#) (int colln, 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 &colln, 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 status – 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 [setDbIParam](#) (OsiDbIParam key, double value)
- bool [setStrParam](#) (OsiStrParam key, const std::string &value)
- bool [getIntParam](#) (OsiIntParam key, int &value) const
- bool [getDbIParam](#) (OsiDbIParam 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 infeasibility proven?
- virtual bool [isProvenDualInfeasible](#) () const
Is dual infeasibility 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.

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 numberOfRows)
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.*
- 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=N-ULL)
Pass in sos stuff from AMPLI.
- 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 numberOfRows, const CoinBigIndex *starts, const int *indices, const double *elements)
- *Extend scale factors.*

4.95.1 Detailed Description

Clp Solver Interface.

Instantiation of [OsiClpSolverInterface](#) for the Model Algorithm.

Definition at line 38 of file OsiClpSolverInterface.hpp.

4.95.2 Constructor & Destructor Documentation

4.95.2.1 OsiClpSolverInterface::OsiClpSolverInterface ()

Default Constructor.

4.95.2.2 OsiClpSolverInterface::OsiClpSolverInterface (const OsiClpSolverInterface &)

Copy constructor.

4.95.2.3 OsiClpSolverInterface::OsiClpSolverInterface (ClpSimplex * rhs, bool reallyOwn = false)

Borrow constructor - only delete one copy.

4.95.2.4 `virtual OsiClpSolverInterface::~~OsiClpSolverInterface () [virtual]`

Destructor.

4.95.3 Member Function Documentation

4.95.3.1 `virtual void OsiClpSolverInterface::initialSolve () [virtual]`

Solve initial LP relaxation.

4.95.3.2 `virtual void OsiClpSolverInterface::resolve () [virtual]`

Resolve an LP relaxation after problem modification.

4.95.3.3 `virtual void OsiClpSolverInterface::resolveGub (int needed) [virtual]`

Resolve an LP relaxation after problem modification (try GUB)

4.95.3.4 `virtual void OsiClpSolverInterface::branchAndBound () [virtual]`

Invoke solver's built-in enumeration algorithm.

4.95.3.5 `void OsiClpSolverInterface::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.

4.95.3.6 `virtual int OsiClpSolverInterface::canDoSimplexInterface () const [virtual]`

Simplex API capability.

Returns

- 0 if no simplex API
- 1 if can just do getBlncv etc
- 2 if has all OsiSimplex methods

4.95.3.7 `virtual void OsiClpSolverInterface::enableFactorization () const [virtual]`

Enables simplex mode 1 (tableau access)

Tells solver that calls to getBlncv 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.

4.95.3.8 `virtual void OsiClpSolverInterface::disableFactorization () const [virtual]`

Undo any setting changes made by [enableFactorization](#).

4.95.3.9 `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 BlncvARow type operations are possible and meaningful.

4.95.3.10 `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 \leq 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

4.95.3.11 `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 \leq 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 ...

4.95.3.12 `virtual void OsiClpSolverInterface::getReducedGradient (double * columnReducedCosts, double * duals, const double * c) const` [virtual]

Get the reduced gradient for the cost vector c.

4.95.3.13 `virtual void OsiClpSolverInterface::getBlvARow (int row, double * z, double * slack = NULL) const` [virtual]

Get a row of the tableau (slack part in slack if not NULL)

4.95.3.14 `virtual void OsiClpSolverInterface::getBlvARow (int row, CoinIndexedVector * z, CoinIndexedVector * slack = NULL, bool keepScaled = false) const` [virtual]

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.

4.95.3.15 `virtual void OsiClpSolverInterface::getBlvRow (int row, double * z) const` [virtual]

Get a row of the basis inverse.

4.95.3.16 `virtual void OsiClpSolverInterface::getBlvACol (int col, double * vec) const` [virtual]

Get a column of the tableau.

4.95.3.17 `virtual void OsiClpSolverInterface::getBlvACol (int col, CoinIndexedVector * vec) const` [virtual]

Get a column of the tableau.

4.95.3.18 `virtual void OsiClpSolverInterface::getBlvACol (CoinIndexedVector * vec) const` [virtual]

Update (i.e.

frtran) the vector passed in. Unscaling is applied after - can't be applied before

4.95.3.19 `virtual void OsiClpSolverInterface::getBlvCol (int col, double * vec) const` [virtual]

Get a column of the basis inverse.

4.95.3.20 `virtual void OsiClpSolverInterface::getBasics (int * index) const` [virtual]

Get basic indices (order of indices corresponds to the order of elements in a vector returned by `getBlvACol()` and `getBlvCol()`).

4.95.3.21 `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.

4.95.3.22 `void OsiClpSolverInterface::copyEnabledStuff (OsiClpSolverInterface & rhs)`

Copy across enabled stuff from one solver to another.

4.95.3.23 `virtual void OsiClpSolverInterface::disableSimplexInterface ()` [virtual]

Undo setting changes made by `enableSimplexInterface`.

4.95.3.24 `void OsiClpSolverInterface::copyEnabledStuff (ClpSimplex & rhs)`

Copy across enabled stuff from one solver to another.

4.95.3.25 `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

4.95.3.26 `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*!=NULL, **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

4.95.3.27 `virtual int OsiClpSolverInterface::dualPivotResult (int & colIn, int & sign, int colOut, int outStatus, double & t, CoinPackedVector * dx)` [virtual]

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 status – the inputs If *dx*!=NULL, then **dx* contains dual ray Return code: same.

4.95.3.28 `bool OsiClpSolverInterface::setIntParam (OsiIntParam key, int value)`

4.95.3.29 `bool OsiClpSolverInterface::setDbParam (OsiDbParam key, double value)`

4.95.3.30 `bool OsiClpSolverInterface::setStrParam (OsiStrParam key, const std::string & value)`

4.95.3.31 `bool OsiClpSolverInterface::getIntParam (OsiIntParam key, int & value) const`

4.95.3.32 `bool OsiClpSolverInterface::getDbParam (OsiDbParam key, double & value) const`

4.95.3.33 `bool OsiClpSolverInterface::getStrParam (OsiStrParam key, std::string & value) const`

4.95.3.34 `virtual bool OsiClpSolverInterface::setHintParam (OsiHintParam key, bool yesNo = true, OsiHintStrength strength = OsiHintTry, void * otherInformation = NULL) [virtual]`

4.95.3.35 `virtual bool OsiClpSolverInterface::isAbandoned () const [virtual]`

Are there a numerical difficulties?

4.95.3.36 `virtual bool OsiClpSolverInterface::isProvenOptimal () const [virtual]`

Is optimality proven?

4.95.3.37 `virtual bool OsiClpSolverInterface::isProvenPrimalInfeasible () const [virtual]`

Is primal infeasibility proven?

4.95.3.38 `virtual bool OsiClpSolverInterface::isProvenDualInfeasible () const [virtual]`

Is dual infeasibility proven?

4.95.3.39 `virtual bool OsiClpSolverInterface::isPrimalObjectiveLimitReached () const [virtual]`

Is the given primal objective limit reached?

4.95.3.40 `virtual bool OsiClpSolverInterface::isDualObjectiveLimitReached () const [virtual]`

Is the given dual objective limit reached?

4.95.3.41 `virtual bool OsiClpSolverInterface::isIterationLimitReached () const [virtual]`

Iteration limit reached?

4.95.3.42 `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 be resized and modified as desired.

4.95.3.43 `virtual CoinWarmStart* OsiClpSolverInterface::getWarmStart () const [virtual]`

Get warmstarting information.

4.95.3.44 `CoinWarmStartBasis* OsiClpSolverInterface::getPointerToWarmStart () [inline]`

Get warmstarting information.

Definition at line 292 of file `OsiClpSolverInterface.hpp`.

4.95.3.45 `const CoinWarmStartBasis* OsiClpSolverInterface::getConstPointerToWarmStart () const [inline]`

Get warmstarting information.

Definition at line 295 of file `OsiClpSolverInterface.hpp`.

4.95.3.46 `virtual bool OsiClpSolverInterface::setWarmStart (const CoinWarmStart * warmstart) [virtual]`

Set warmstarting information.

Return true/false depending on whether the warmstart information was accepted or not.

4.95.3.47 `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 will 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.

4.95.3.48 `void OsiClpSolverInterface::setColumnStatus (int iColumn, ClpSimplex::Status status)`

Set column status in [ClpSimplex](#) and warmStart.

4.95.3.49 `virtual void OsiClpSolverInterface::markHotStart () [virtual]`

Create a hotstart point of the optimization process.

4.95.3.50 `virtual void OsiClpSolverInterface::solveFromHotStart () [virtual]`

Optimize starting from the hotstart.

4.95.3.51 `virtual void OsiClpSolverInterface::unmarkHotStart () [virtual]`

Delete the snapshot.

4.95.3.52 `int OsiClpSolverInterface::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.

4.95.3.53 `void OsiClpSolverInterface::stopFastDual ()`

Stop fast dual.

4.95.3.54 `void OsiClpSolverInterface::setStuff (double tolerance, double increment)`

Sets integer tolerance and increment.

4.95.3.55 `virtual int OsiClpSolverInterface::getNumCols () const [inline],[virtual]`

Get number of columns.

Definition at line 360 of file [OsiClpSolverInterface.hpp](#).

4.95.3.56 `virtual int OsiClpSolverInterface::getNumRows () const [inline],[virtual]`

Get number of rows.

Definition at line 364 of file [OsiClpSolverInterface.hpp](#).

4.95.3.57 `virtual int OsiClpSolverInterface::getNumElements () const [inline],[virtual]`

Get number of nonzero elements.

Definition at line 368 of file [OsiClpSolverInterface.hpp](#).

4.95.3.58 `virtual std::string OsiClpSolverInterface::getRowName (int rowIndex, unsigned maxLen = static_cast< unsigned >(std::string::npos)) const [virtual]`

Return name of row if one exists or Rnnnnnnn maxLen is currently ignored and only there to match the signature from the base class!

4.95.3.59 `virtual std::string OsiClpSolverInterface::getColName (int colIndex, unsigned maxLen = static_cast< unsigned >(std::string::npos)) const [virtual]`

Return name of column if one exists or Cnnnnnnn maxLen is currently ignored and only there to match the signature from the base class!

4.95.3.60 `virtual const double* OsiClpSolverInterface::getColLower () const [inline],[virtual]`

Get pointer to array[getNumCols()] of column lower bounds.

Definition at line 386 of file OsiClpSolverInterface.hpp.

4.95.3.61 `virtual const double* OsiClpSolverInterface::getColUpper () const [inline],[virtual]`

Get pointer to array[getNumCols()] of column upper bounds.

Definition at line 389 of file OsiClpSolverInterface.hpp.

4.95.3.62 `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

4.95.3.63 `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

4.95.3.64 `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

4.95.3.65 `virtual const double* OsiClpSolverInterface::getRowLower () const [inline],[virtual]`

Get pointer to array[getNumRows()] of row lower bounds.

Definition at line 423 of file OsiClpSolverInterface.hpp.

4.95.3.66 `virtual const double* OsiClpSolverInterface::getRowUpper () const [inline],[virtual]`

Get pointer to array[getNumRows()] of row upper bounds.

Definition at line 426 of file OsiClpSolverInterface.hpp.

4.95.3.67 `virtual const double* OsiClpSolverInterface::getObjCoefficients () const [inline],[virtual]`

Get pointer to array[getNumCols()] of objective function coefficients.

Definition at line 429 of file OsiClpSolverInterface.hpp.

4.95.3.68 `virtual double OsiClpSolverInterface::getObjSense () const [inline],[virtual]`

Get objective function sense (1 for min (default), -1 for max)

Definition at line 436 of file OsiClpSolverInterface.hpp.

4.95.3.69 `virtual bool OsiClpSolverInterface::isContinuous (int colNumber) const [virtual]`

Return true if column is continuous.

4.95.3.70 `virtual bool OsiClpSolverInterface::isBinary (int colIndex) const [virtual]`

Return true if variable is binary.

4.95.3.71 `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.

4.95.3.72 `virtual bool OsiClpSolverInterface::isIntegerNonBinary (int colIndex) const [virtual]`

Return true if variable is general integer.

4.95.3.73 `virtual bool OsiClpSolverInterface::isFreeBinary (int colIndex) const [virtual]`

Return true if variable is binary and not fixed at either bound.

4.95.3.74 `virtual const char* OsiClpSolverInterface::getColType (bool refresh = false) const [virtual]`

Return array of column length 0 - continuous 1 - binary (may get fixed later) 2 - general integer (may get fixed later)

4.95.3.75 `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.

4.95.3.76 `void OsiClpSolverInterface::setOptionalInteger (int index)`

Set the index-th variable to be an optional integer variable.

4.95.3.77 `virtual const CoinPackedMatrix* OsiClpSolverInterface::getMatrixByRow () const [virtual]`

Get pointer to row-wise copy of matrix.

4.95.3.78 `virtual const CoinPackedMatrix* OsiClpSolverInterface::getMatrixByCol () const [virtual]`

Get pointer to column-wise copy of matrix.

4.95.3.79 `virtual CoinPackedMatrix* OsiClpSolverInterface::getMutableMatrixByCol () const [virtual]`

Get pointer to mutable column-wise copy of matrix.

4.95.3.80 `virtual double OsiClpSolverInterface::getInfinity () const [inline],[virtual]`

Get solver's value for infinity.

Definition at line 482 of file `OsiClpSolverInterface.hpp`.

4.95.3.81 `virtual const double* OsiClpSolverInterface::getColSolution () const [virtual]`

Get pointer to array[`getNumCols()`] of primal solution vector.

4.95.3.82 `virtual const double* OsiClpSolverInterface::getRowPrice () const [virtual]`

Get pointer to array[`getNumRows()`] of dual prices.

4.95.3.83 `virtual const double* OsiClpSolverInterface::getReducedCost () const [virtual]`

Get a pointer to array[`getNumCols()`] of reduced costs.

4.95.3.84 `virtual const double* OsiClpSolverInterface::getRowActivity () const [virtual]`

Get pointer to array[`getNumRows()`] of row activity levels (constraint matrix times the solution vector).

4.95.3.85 `virtual double OsiClpSolverInterface::getObjValue () const [virtual]`

Get objective function value.

4.95.3.86 `virtual int OsiClpSolverInterface::getIterationCount () const [inline],[virtual]`

Get how many iterations it took to solve the problem (whatever "iteration" mean to the solver).

Definition at line 505 of file `OsiClpSolverInterface.hpp`.

4.95.3.87 `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[].

4.95.3.88 `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[].

4.95.3.89 `virtual void OsiClpSolverInterface::setObjCoeff (int elementIndex, double elementValue) [virtual]`

Set an objective function coefficient.

4.95.3.90 `virtual void OsiClpSolverInterface::setColLower (int elementIndex, double elementValue) [virtual]`

Set a single column lower bound

Use -DBL_MAX for -infinity.

4.95.3.91 `virtual void OsiClpSolverInterface::setColUpper (int elementIndex, double elementValue) [virtual]`

Set a single column upper bound

Use DBL_MAX for infinity.

4.95.3.92 `virtual void OsiClpSolverInterface::setColBounds (int elementIndex, double lower, double upper) [virtual]`

Set a single column lower and upper bound.

4.95.3.93 `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

<i>indexFirst, index-Last</i>	pointers to the beginning and after the end of the array of the indices of the variables whose <i>either</i> bound changes
<i>boundList</i>	the new lower/upper bound pairs for the variables

4.95.3.94 `virtual void OsiClpSolverInterface::setRowLower (int elementIndex, double elementValue) [virtual]`

Set a single row lower bound

Use -DBL_MAX for -infinity.

4.95.3.95 `virtual void OsiClpSolverInterface::setRowUpper (int elementIndex, double elementValue) [virtual]`

Set a single row upper bound

Use DBL_MAX for infinity.

4.95.3.96 `virtual void OsiClpSolverInterface::setRowBounds (int elementIndex, double lower, double upper)` [virtual]

Set a single row lower and upper bound.

4.95.3.97 `virtual void OsiClpSolverInterface::setRowType (int index, char sense, double rightHandSide, double range)` [virtual]

Set the type of a single row

4.95.3.98 `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

<i>indexFirst, index-Last</i>	pointers to the beginning and after the end of the array of the indices of the constraints whose <i>either</i> bound changes
<i>boundList</i>	the new lower/upper bound pairs for the constraints

4.95.3.99 `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

<i>indexFirst, index-Last</i>	pointers to the beginning and after the end of the array of the indices of the constraints whose <i>any</i> characteristics changes
<i>senseList</i>	the new senses
<i>rhsList</i>	the new right hand sides
<i>rangeList</i>	the new ranges

4.95.3.100 `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.

4.95.3.101 `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.

4.95.3.102 `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.

4.95.3.103 `virtual void OsiClpSolverInterface::setRowName (int rowIndex, std::string name)` [virtual]

Set name of row.

4.95.3.104 `virtual void OsiClpSolverInterface::setColName (int colIndex, std::string name) [virtual]`

Set name of column.

4.95.3.105 `virtual void OsiClpSolverInterface::setContinuous (int index) [virtual]`

Set the index-th variable to be a continuous variable.

4.95.3.106 `virtual void OsiClpSolverInterface::setInteger (int index) [virtual]`

Set the index-th variable to be an integer variable.

4.95.3.107 `virtual void OsiClpSolverInterface::setContinuous (const int * indices, int len) [virtual]`

Set the variables listed in indices (which is of length len) to be continuous variables.

4.95.3.108 `virtual void OsiClpSolverInterface::setInteger (const int * indices, int len) [virtual]`

Set the variables listed in indices (which is of length len) to be integer variables.

4.95.3.109 `int OsiClpSolverInterface::numberSOS () const [inline]`

Number of SOS sets.

Definition at line 664 of file OsiClpSolverInterface.hpp.

4.95.3.110 `const CoinSet* OsiClpSolverInterface::setInfo () const [inline]`

SOS set info.

Definition at line 667 of file OsiClpSolverInterface.hpp.

4.95.3.111 `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

4.95.3.112 `virtual void OsiClpSolverInterface::setObjSense (double s) [inline],[virtual]`

Set objective function sense (1 for min (default), -1 for max,)

Definition at line 683 of file OsiClpSolverInterface.hpp.

4.95.3.113 `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.

4.95.3.114 `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.

4.95.3.115 `virtual void OsiClpSolverInterface::addCol (const CoinPackedVectorBase & vec, const double collb, const double colub, const double obj) [virtual]`

4.95.3.116 `virtual void OsiClpSolverInterface::addCol (const CoinPackedVectorBase & vec, const double collb, const double colub, const double obj, std::string name) [virtual]`

Add a named column (primal variable) to the problem.

4.95.3.117 `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.

4.95.3.118 `virtual void OsiClpSolverInterface::addCol (int numberElements, const int * rows, const double * elements, const double collb, const double colub, const double obj, std::string name) [virtual]`

Add a named column (primal variable) to the problem.

4.95.3.119 `virtual void OsiClpSolverInterface::addCols (const int numcols, const CoinPackedVectorBase *const * cols, const double * collb, const double * colub, const double * obj) [virtual]`

4.95.3.120 `virtual void OsiClpSolverInterface::addCols (const int numcols, const int * columnStarts, const int * rows, const double * elements, const double * collb, const double * colub, const double * obj) [virtual]`

4.95.3.121 `virtual void OsiClpSolverInterface::deleteCols (const int num, const int * colIndices) [virtual]`

4.95.3.122 `virtual void OsiClpSolverInterface::addRow (const CoinPackedVectorBase & vec, const double rowlb, const double rowub) [virtual]`

4.95.3.123 `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.

4.95.3.124 `virtual void OsiClpSolverInterface::addRow (const CoinPackedVectorBase & vec, const char rowsen, const double rowrhs, const double rowrng) [virtual]`

4.95.3.125 `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.

4.95.3.126 `virtual void OsiClpSolverInterface::addRow (const CoinPackedVectorBase & vec, const char rowsen, const double rowrhs, const double rowrng, std::string name) [virtual]`

Add a named row (constraint) to the problem.

4.95.3.127 `virtual void OsiClpSolverInterface::addRows (const int numRows, const CoinPackedVectorBase *const * rows, const double * rowlb, const double * rowub) [virtual]`

4.95.3.128 `virtual void OsiClpSolverInterface::addRows (const int numrows, const CoinPackedVectorBase *const * rows, const char * rowSEN, const double * rowRHS, const double * rowrng)` [virtual]

4.95.3.129 `virtual void OsiClpSolverInterface::addRows (const int numrows, const int * rowStarts, const int * columns, const double * element, const double * rowlb, const double * rowub)` [virtual]

4.95.3.130 `void OsiClpSolverInterface::modifyCoefficient (int row, int column, double newElement, bool keepZero = false)` [inline]

Definition at line 787 of file OsiClpSolverInterface.hpp.

4.95.3.131 `virtual void OsiClpSolverInterface::deleteRows (const int num, const int * rowIndices)` [virtual]

4.95.3.132 `virtual void OsiClpSolverInterface::saveBaseModel ()` [virtual]

If solver wants it can save a copy of "base" (continuous) model here.

4.95.3.133 `virtual void OsiClpSolverInterface::restoreBaseModel (int numberOfRows)` [virtual]

Strip off rows to get to this number of rows.

If solver wants it can restore a copy of "base" (continuous) model here

4.95.3.134 `virtual void OsiClpSolverInterface::applyRowCuts (int numberOfCuts, 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.

4.95.3.135 `virtual void OsiClpSolverInterface::applyRowCuts (int numberOfCuts, 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

4.95.3.136 `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() + getNuminfeasible() + getNumApplied()`

4.95.3.137 `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

4.95.3.138 `virtual void OsiClpSolverInterface::assignProblem (CoinPackedMatrix * & matrix, double * & collb, double * & colub, double * & obj, double * & rowlb, double * & rowub) [virtual]`

Load in a 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.

4.95.3.139 `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 a 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 \geq
- `rowrhs`: all right hand sides are 0
- `rowrng`: 0 for the ranged rows

4.95.3.140 `virtual void OsiClpSolverInterface::assignProblem (CoinPackedMatrix * & matrix, double * & collb, double * & colub, double * & obj, char * & rowsen, double * & rowrhs, double * & rowrng) [virtual]`

Load in a 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.

4.95.3.141 `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](#).

4.95.3.142 `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).

4.95.3.143 `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).

4.95.3.144 `virtual int OsiClpSolverInterface::loadFromCoinModel (CoinModel & modelObject, bool keepSolution = false) [virtual]`

This loads a model from a coinModel object - returns number of errors.

4.95.3.145 `virtual int OsiClpSolverInterface::readMps (const char * filename, const char * extension = "mps") [virtual]`

Read an mps file from the given filename (defaults to Osi reader) - returns number of errors (see OsiMpsReader class)

4.95.3.146 `int OsiClpSolverInterface::readMps (const char * filename, bool keepNames, bool allowErrors)`

Read an mps file from the given filename returns number of errors (see OsiMpsReader class)

4.95.3.147 `virtual int OsiClpSolverInterface::readMps (const char * filename, const char * extension, int & numberSets, CoinSet **& sets) [virtual]`

Read an mps file.

4.95.3.148 `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

4.95.3.149 `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

4.95.3.150 `virtual int OsiClpSolverInterface::readLp (const char * filename, const double epsilon = 1e-5) [virtual]`

Read file in LP format (with names)

4.95.3.151 `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

4.95.3.152 `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.

4.95.3.153 `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

4.95.3.154 `virtual void OsiClpSolverInterface::replaceMatrix (const CoinPackedMatrix & matrix) [virtual]`

And if it does matter (not used at present)

4.95.3.155 `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.

4.95.3.156 `void OsiClpSolverInterface::newLanguage (CoinMessages::Language language)`

Set language.

4.95.3.157 `void OsiClpSolverInterface::setLanguage (CoinMessages::Language language) [inline]`

Definition at line 1008 of file `OsiClpSolverInterface.hpp`.

4.95.3.158 `void OsiClpSolverInterface::setLogLevel (int value)`

Set log level (will also set underlying solver's log level)

4.95.3.159 `void OsiClpSolverInterface::generateCpp (FILE * fp)`

Create C++ lines to get to current state.

4.95.3.160 `ClpSimplex* OsiClpSolverInterface::getModelPtr () const`

Get pointer to Clp model.

4.95.3.161 `ClpSimplex* OsiClpSolverInterface::swapModelPtr (ClpSimplex * newModel) [inline]`

Set pointer to Clp model and return old.

Definition at line 1022 of file `OsiClpSolverInterface.hpp`.

4.95.3.162 `unsigned int OsiClpSolverInterface::specialOptions () const [inline]`

Get special options.

Definition at line 1025 of file `OsiClpSolverInterface.hpp`.

4.95.3.163 `void OsiClpSolverInterface::setSpecialOptions (unsigned int value)`

4.95.3.164 `int OsiClpSolverInterface::lastAlgorithm () const [inline]`

Last algorithm used , 1 = primal, 2 = dual other unknown.

Definition at line 1029 of file `OsiClpSolverInterface.hpp`.

4.95.3.165 `void OsiClpSolverInterface::setLastAlgorithm (int value) [inline]`

Set last algorithm used , 1 = primal, 2 = dual other unknown.

Definition at line 1032 of file OsiClpSolverInterface.hpp.

4.95.3.166 `int OsiClpSolverInterface::cleanupScaling () const [inline]`

Get scaling action option.

Definition at line 1035 of file OsiClpSolverInterface.hpp.

4.95.3.167 `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 1049 of file OsiClpSolverInterface.hpp.

4.95.3.168 `double OsiClpSolverInterface::smallestElementInCut () const [inline]`

Get smallest allowed element in cut.

If smaller than this then ignored

Definition at line 1053 of file OsiClpSolverInterface.hpp.

4.95.3.169 `void OsiClpSolverInterface::setSmallestElementInCut (double value) [inline]`

Set smallest allowed element in cut.

If smaller than this then ignored

Definition at line 1057 of file OsiClpSolverInterface.hpp.

4.95.3.170 `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 1065 of file OsiClpSolverInterface.hpp.

4.95.3.171 `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 1073 of file OsiClpSolverInterface.hpp.

4.95.3.172 `void OsiClpSolverInterface::setSolveOptions (const ClpSolve & options) [inline]`

Pass in initial solve options.

Definition at line 1076 of file OsiClpSolverInterface.hpp.

4.95.3.173 `virtual int OsiClpSolverInterface::tightenBounds (int lightweight = 0) [virtual]`

Tighten bounds - lightweight or very lightweight 0 - normal, 1 lightweight but just integers, 2 lightweight and all.

4.95.3.174 `virtual CoinBigIndex OsiClpSolverInterface::getSizeL () const [virtual]`

Return number of entries in L part of current factorization.

4.95.3.175 `virtual CoinBigIndex OsiClpSolverInterface::getSizeU () const [virtual]`

Return number of entries in U part of current factorization.

4.95.3.176 `const OsiClpDisasterHandler* OsiClpSolverInterface::disasterHandler () const [inline]`

Get disaster handler.

Definition at line 1087 of file OsiClpSolverInterface.hpp.

4.95.3.177 `void OsiClpSolverInterface::passInDisasterHandler (OsiClpDisasterHandler * handler)`

Pass in disaster handler.

4.95.3.178 `ClpLinearObjective* OsiClpSolverInterface::fakeObjective () const [inline]`

Get fake objective.

Definition at line 1092 of file OsiClpSolverInterface.hpp.

4.95.3.179 `void OsiClpSolverInterface::setFakeObjective (ClpLinearObjective * fakeObjective)`

Set fake objective (and take ownership)

4.95.3.180 `void OsiClpSolverInterface::setFakeObjective (double * fakeObjective)`

Set fake objective.

4.95.3.181 `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

4.95.3.182 `virtual void OsiClpSolverInterface::synchronizeModel () [virtual]`

Synchronize model (really if no cuts in tree)

4.95.3.183 `void OsiClpSolverInterface::setSpecialOptionsMutable (unsigned int value) const`

Set special options in underlying clp solver.

Safe as const because `modelPtr_` is mutable.

4.95.3.184 `virtual OsiSolverInterface* OsiClpSolverInterface::clone (bool copyData =true) const [virtual]`

Clone.

4.95.3.185 `void OsiClpSolverInterface::releaseClp ()`

Releases so won't error.

4.95.3.186 `OsiClpSolverInterface& OsiClpSolverInterface::operator= (const OsiClpSolverInterface & rhs)`

Assignment operator.

4.95.3.187 `virtual void OsiClpSolverInterface::reset () [virtual]`

Resets as if default constructor.

4.95.3.188 `virtual void OsiClpSolverInterface::applyRowCut (const OsiRowCut & rc) [protected], [virtual]`

Apply a row cut (append to constraint matrix).

4.95.3.189 `virtual void OsiClpSolverInterface::applyColCut (const OsiColCut & cc) [protected], [virtual]`

Apply a column cut (adjust one or more bounds).

4.95.3.190 `void OsiClpSolverInterface::gutsOfDestructor () [protected]`

The real work of a copy constructor (used by copy and assignment)

4.95.3.191 `void OsiClpSolverInterface::freeCachedResults () const [protected]`

Deletes all mutable stuff.

4.95.3.192 `void OsiClpSolverInterface::freeCachedResults0 () const [protected]`

Deletes all mutable stuff for row ranges etc.

4.95.3.193 `void OsiClpSolverInterface::freeCachedResults1 () const [protected]`

Deletes all mutable stuff for matrix etc.

4.95.3.194 `void OsiClpSolverInterface::extractSenseRhsRange () const [protected]`

A method that fills up the `rowsense_`, `rhs_` and `rowrange_` arrays.

4.95.3.195 `void OsiClpSolverInterface::fillParamMaps () [protected]`

4.95.3.196 `CoinWarmStartBasis OsiClpSolverInterface::getBasis (ClpSimplex * model) const` [protected]

Warm start.

NOTE artificials are treated as +1 elements so for \leq 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

4.95.3.197 `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 \leq 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

4.95.3.198 `void OsiClpSolverInterface::crunch ()` [protected]

Crunch down problem a bit.

4.95.3.199 `void OsiClpSolverInterface::redoScaleFactors (int numberOfRows, const CoinBigIndex * starts, const int * indices, const double * elements)` [protected]

Extend scale factors.

4.95.3.200 `void OsiClpSolverInterface::setBasis (const CoinWarmStartBasis & basis)`

Sets up working basis as a copy of input and puts in as basis.

4.95.3.201 `void OsiClpSolverInterface::setBasis ()` [inline]

Just puts current basis_ into [ClpSimplex](#) model.

Definition at line 1219 of file `OsiClpSolverInterface.hpp`.

4.95.3.202 `CoinWarmStartDiff* OsiClpSolverInterface::getBasisDiff (const unsigned char * statusArray) const`

Warm start difference from basis_ to statusArray.

4.95.3.203 `CoinWarmStartBasis* OsiClpSolverInterface::getBasis (const unsigned char * statusArray) const`

Warm start from statusArray.

4.95.3.204 `void OsiClpSolverInterface::deleteScaleFactors ()`

Delete all scale factor stuff and reset option.

4.95.3.205 `const double* OsiClpSolverInterface::upRange () const` [inline]

If doing fast hot start then ranges are computed.

Definition at line 1228 of file `OsiClpSolverInterface.hpp`.

4.95.3.206 `const double* OsiClpSolverInterface::downRange () const` [inline]

Definition at line 1230 of file `OsiClpSolverInterface.hpp`.

4.95.3.207 `void OsiClpSolverInterface::passInRanges (int * array)` [inline]

Pass in range array.

Definition at line 1233 of file OsiClpSolverInterface.hpp.

4.95.3.208 void OsiClpSolverInterface::setSOSData (int *numberSOS*, const char * *type*, const int * *start*, const int * *indices*, const double * *weights* = NULL)

Pass in sos stuff from AMPL.

4.95.3.209 void OsiClpSolverInterface::computeLargestAway ()

Compute largest amount any at continuous away from bound.

4.95.3.210 double OsiClpSolverInterface::largestAway () const [inline]

Get largest amount continuous away from bound.

Definition at line 1241 of file OsiClpSolverInterface.hpp.

4.95.3.211 void OsiClpSolverInterface::setLargestAway (double *value*) [inline]

Set largest amount continuous away from bound.

Definition at line 1244 of file OsiClpSolverInterface.hpp.

4.95.3.212 void OsiClpSolverInterface::lexSolve ()

Sort of lexicographic resolve.

4.95.4 Friends And Related Function Documentation

4.95.4.1 void OsiClpSolverInterfaceUnitTest (const std::string & *mpsDir*, const std::string & *netlibDir*) [friend]

A function that tests the methods in the [OsiClpSolverInterface](#) class.

4.95.5 Member Data Documentation

4.95.5.1 ClpSimplex* OsiClpSolverInterface::modelPtr_ [mutable], [protected]

Clp model represented by this class instance.

Definition at line 1254 of file OsiClpSolverInterface.hpp.

4.95.5.2 char* OsiClpSolverInterface::rowsense_ [mutable], [protected]

Pointer to dense vector of row sense indicators.

Definition at line 1259 of file OsiClpSolverInterface.hpp.

4.95.5.3 double* OsiClpSolverInterface::rhs_ [mutable], [protected]

Pointer to dense vector of row right-hand side values.

Definition at line 1262 of file OsiClpSolverInterface.hpp.

4.95.5.4 double* OsiClpSolverInterface::rowrange_ [mutable], [protected]

Pointer to dense vector of slack upper bounds for range constraints (undefined for non-range rows)

Definition at line 1267 of file OsiClpSolverInterface.hpp.

4.95.5.5 **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 1271 of file OsiClpSolverInterface.hpp.

4.95.5.6 **double*** **OsiClpSolverInterface::rowActivity_** [mutable],[protected]

also save row and column information for hot starts only used in hotstarts so can be casual

Definition at line 1274 of file OsiClpSolverInterface.hpp.

4.95.5.7 **double*** **OsiClpSolverInterface::columnActivity_** [mutable],[protected]

Definition at line 1275 of file OsiClpSolverInterface.hpp.

4.95.5.8 **ClpNodeStuff** **OsiClpSolverInterface::stuff_** [protected]

Stuff for fast dual.

Definition at line 1277 of file OsiClpSolverInterface.hpp.

4.95.5.9 **int** **OsiClpSolverInterface::numberSOS_** [protected]

Number of SOS sets.

Definition at line 1279 of file OsiClpSolverInterface.hpp.

4.95.5.10 **CoinSet*** **OsiClpSolverInterface::setInfo_** [protected]

SOS set info.

Definition at line 1281 of file OsiClpSolverInterface.hpp.

4.95.5.11 **ClpSimplex*** **OsiClpSolverInterface::smallModel_** [protected]

Alternate model (hot starts) - but also could be permanent and used for crunch.

Definition at line 1283 of file OsiClpSolverInterface.hpp.

4.95.5.12 **ClpFactorization*** **OsiClpSolverInterface::factorization_** [protected]

factorization for hot starts

Definition at line 1285 of file OsiClpSolverInterface.hpp.

4.95.5.13 **double** **OsiClpSolverInterface::smallestElementInCut_** [protected]

Smallest allowed element in cut.

If smaller than this then ignored

Definition at line 1288 of file OsiClpSolverInterface.hpp.

4.95.5.14 **double** **OsiClpSolverInterface::smallestChangeInCut_** [protected]

Smallest change in cut.

If (upper-lower)*element < this then element is taken out and cut relaxed.

Definition at line 1292 of file OsiClpSolverInterface.hpp.

4.95.5.15 `double OsiClpSolverInterface::largestAway_` `[protected]`

Largest amount continuous away from bound.

Definition at line 1294 of file OsiClpSolverInterface.hpp.

4.95.5.16 `char* OsiClpSolverInterface::spareArrays_` `[protected]`

Arrays for hot starts.

Definition at line 1296 of file OsiClpSolverInterface.hpp.

4.95.5.17 `CoinWarmStartBasis OsiClpSolverInterface::basis_` `[protected]`

Warmstart information to be used in resolves.

Definition at line 1298 of file OsiClpSolverInterface.hpp.

4.95.5.18 `int OsiClpSolverInterface::itlimOrig_` `[protected]`

The original iteration limit before hotstarts started.

Definition at line 1300 of file OsiClpSolverInterface.hpp.

4.95.5.19 `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 1312 of file OsiClpSolverInterface.hpp.

4.95.5.20 `bool OsiClpSolverInterface::notOwned_` `[protected]`

To say if destructor should delete underlying model.

Definition at line 1315 of file OsiClpSolverInterface.hpp.

4.95.5.21 `CoinPackedMatrix* OsiClpSolverInterface::matrixByRow_` `[mutable], [protected]`

Pointer to row-wise copy of problem matrix coefficients.

Definition at line 1318 of file OsiClpSolverInterface.hpp.

4.95.5.22 `CoinPackedMatrix* OsiClpSolverInterface::matrixByRowAtContinuous_` `[protected]`

Pointer to row-wise copy of continuous problem matrix coefficients.

Definition at line 1321 of file OsiClpSolverInterface.hpp.

4.95.5.23 `char* OsiClpSolverInterface::integerInformation_` `[protected]`

Pointer to integer information.

Definition at line 1324 of file OsiClpSolverInterface.hpp.

4.95.5.24 `int* OsiClpSolverInterface::whichRange_` `[protected]`

Pointer to variables for which we want range information The number is in [0] memory is not owned by OsiClp.

Definition at line 1330 of file OsiClpSolverInterface.hpp.

4.95.5.25 `bool OsiClpSolverInterface::fakeMinInSimplex_` `[mutable], [protected]`

Faking min to get proper dual solution signs in simplex API.

Definition at line 1337 of file OsiClpSolverInterface.hpp.

4.95.5.26 `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 1344 of file OsiClpSolverInterface.hpp.

4.95.5.27 `ClpDataSave OsiClpSolverInterface::saveData_` `[mutable], [protected]`

To save data in OsiSimplex stuff.

Definition at line 1347 of file OsiClpSolverInterface.hpp.

4.95.5.28 `ClpSolve OsiClpSolverInterface::solveOptions_` `[protected]`

[Options](#) for initialSolve.

Definition at line 1349 of file OsiClpSolverInterface.hpp.

4.95.5.29 `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 1362 of file OsiClpSolverInterface.hpp.

4.95.5.30 `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 1390 of file OsiClpSolverInterface.hpp.

4.95.5.31 ClpSimplex* OsiClpSolverInterface::baseModel_ [protected]

Copy of model when option 131072 set.

Definition at line 1392 of file OsiClpSolverInterface.hpp.

4.95.5.32 int OsiClpSolverInterface::lastNumberRows_ [protected]

Number of rows when last "scaled".

Definition at line 1394 of file OsiClpSolverInterface.hpp.

4.95.5.33 ClpSimplex* OsiClpSolverInterface::continuousModel_ [protected]

Continuous model.

Definition at line 1396 of file OsiClpSolverInterface.hpp.

4.95.5.34 OsiClpDisasterHandler* OsiClpSolverInterface::disasterHandler_ [protected]

Possible disaster handler.

Definition at line 1398 of file OsiClpSolverInterface.hpp.

4.95.5.35 ClpLinearObjective* OsiClpSolverInterface::fakeObjective_ [protected]

Fake objective.

Definition at line 1400 of file OsiClpSolverInterface.hpp.

4.95.5.36 CoinDoubleArrayWithLength OsiClpSolverInterface::rowScale_ [protected]

Row scale factors (has inverse at end)

Definition at line 1402 of file OsiClpSolverInterface.hpp.

4.95.5.37 CoinDoubleArrayWithLength OsiClpSolverInterface::columnScale_ [protected]

Column scale factors (has inverse at end)

Definition at line 1404 of file OsiClpSolverInterface.hpp.

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

- [src/OsiClp/OsiClpSolverInterface.hpp](#)

4.96 Outfo Struct Reference

***** DATA to be moved into protected section of [ClpInterior](#)

```
#include <ClpInterior.hpp>
```

Public Attributes

- double [atolold](#)
- double [atolnew](#)
- double [r3ratio](#)
- int [istop](#)
- int [itncg](#)

4.96.1 Detailed Description

***** DATA to be moved into protected section of [ClpInterior](#)

Definition at line 35 of file ClpInterior.hpp.

4.96.2 Member Data Documentation

4.96.2.1 double Outfo::atolold

Definition at line 36 of file ClpInterior.hpp.

4.96.2.2 double Outfo::atolnew

Definition at line 37 of file ClpInterior.hpp.

4.96.2.3 double Outfo::r3ratio

Definition at line 38 of file ClpInterior.hpp.

4.96.2.4 int Outfo::istop

Definition at line 39 of file ClpInterior.hpp.

4.96.2.5 int Outfo::itncg

Definition at line 40 of file ClpInterior.hpp.

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

- [src/ClpInterior.hpp](#)

4.97 ClpSimplexOther::parametricsData Struct Reference

```
#include <ClpSimplexOther.hpp>
```

Public Attributes

- double [startingTheta](#)
- double [endingTheta](#)
- double [maxTheta](#)
- double [acceptableMaxTheta](#)
- double * [lowerChange](#)
- int * [lowerList](#)
- double * [upperChange](#)
- int * [upperList](#)
- char * [markDone](#)
- int * [backwardBasic](#)
- int * [lowerActive](#)
- double * [lowerGap](#)
- double * [lowerCoefficient](#)
- int * [upperActive](#)
- double * [upperGap](#)

- double * [upperCoefficient](#)
- int [unscaledChangesOffset](#)
- bool [firstIteration](#)

4.97.1 Detailed Description

Definition at line 107 of file ClpSimplexOther.hpp.

4.97.2 Member Data Documentation

4.97.2.1 double ClpSimplexOther::parametricsData::startingTheta

Definition at line 108 of file ClpSimplexOther.hpp.

4.97.2.2 double ClpSimplexOther::parametricsData::endingTheta

Definition at line 109 of file ClpSimplexOther.hpp.

4.97.2.3 double ClpSimplexOther::parametricsData::maxTheta

Definition at line 110 of file ClpSimplexOther.hpp.

4.97.2.4 double ClpSimplexOther::parametricsData::acceptableMaxTheta

Definition at line 111 of file ClpSimplexOther.hpp.

4.97.2.5 double* ClpSimplexOther::parametricsData::lowerChange

Definition at line 112 of file ClpSimplexOther.hpp.

4.97.2.6 int* ClpSimplexOther::parametricsData::lowerList

Definition at line 113 of file ClpSimplexOther.hpp.

4.97.2.7 double* ClpSimplexOther::parametricsData::upperChange

Definition at line 114 of file ClpSimplexOther.hpp.

4.97.2.8 int* ClpSimplexOther::parametricsData::upperList

Definition at line 115 of file ClpSimplexOther.hpp.

4.97.2.9 char* ClpSimplexOther::parametricsData::markDone

Definition at line 116 of file ClpSimplexOther.hpp.

4.97.2.10 int* ClpSimplexOther::parametricsData::backwardBasic

Definition at line 117 of file ClpSimplexOther.hpp.

4.97.2.11 int* ClpSimplexOther::parametricsData::lowerActive

Definition at line 118 of file ClpSimplexOther.hpp.

4.97.2.12 `double* ClpSimplexOther::parametricsData::lowerGap`

Definition at line 119 of file `ClpSimplexOther.hpp`.

4.97.2.13 `double* ClpSimplexOther::parametricsData::lowerCoefficient`

Definition at line 120 of file `ClpSimplexOther.hpp`.

4.97.2.14 `int* ClpSimplexOther::parametricsData::upperActive`

Definition at line 121 of file `ClpSimplexOther.hpp`.

4.97.2.15 `double* ClpSimplexOther::parametricsData::upperGap`

Definition at line 122 of file `ClpSimplexOther.hpp`.

4.97.2.16 `double* ClpSimplexOther::parametricsData::upperCoefficient`

Definition at line 123 of file `ClpSimplexOther.hpp`.

4.97.2.17 `int ClpSimplexOther::parametricsData::unscaledChangesOffset`

Definition at line 124 of file `ClpSimplexOther.hpp`.

4.97.2.18 `bool ClpSimplexOther::parametricsData::firstIteration`

Definition at line 125 of file `ClpSimplexOther.hpp`.

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

- [src/ClpSimplexOther.hpp](#)

4.98 `AbcSimplexPrimal::pivotStruct` Struct Reference

```
#include <AbcSimplexPrimal.hpp>
```

Public Attributes

- `double theta_`
- `double alpha_`
- `double saveDualIn_`
- `double dualIn_`
- `double lowerIn_`
- `double upperIn_`
- `double valueIn_`
- `int sequenceIn_`
- `int directionIn_`
- `double dualOut_`
- `double lowerOut_`
- `double upperOut_`
- `double valueOut_`
- `int sequenceOut_`
- `int directionOut_`
- `int pivotRow_`
- `int valuesPass_`

4.98.1 Detailed Description

Definition at line 210 of file AbcSimplexPrimal.hpp.

4.98.2 Member Data Documentation

4.98.2.1 `double AbcSimplexPrimal::pivotStruct::theta_`

Definition at line 211 of file AbcSimplexPrimal.hpp.

4.98.2.2 `double AbcSimplexPrimal::pivotStruct::alpha_`

Definition at line 212 of file AbcSimplexPrimal.hpp.

4.98.2.3 `double AbcSimplexPrimal::pivotStruct::saveDualIn_`

Definition at line 213 of file AbcSimplexPrimal.hpp.

4.98.2.4 `double AbcSimplexPrimal::pivotStruct::dualIn_`

Definition at line 214 of file AbcSimplexPrimal.hpp.

4.98.2.5 `double AbcSimplexPrimal::pivotStruct::lowerIn_`

Definition at line 215 of file AbcSimplexPrimal.hpp.

4.98.2.6 `double AbcSimplexPrimal::pivotStruct::upperIn_`

Definition at line 216 of file AbcSimplexPrimal.hpp.

4.98.2.7 `double AbcSimplexPrimal::pivotStruct::valueIn_`

Definition at line 217 of file AbcSimplexPrimal.hpp.

4.98.2.8 `int AbcSimplexPrimal::pivotStruct::sequenceIn_`

Definition at line 218 of file AbcSimplexPrimal.hpp.

4.98.2.9 `int AbcSimplexPrimal::pivotStruct::directionIn_`

Definition at line 219 of file AbcSimplexPrimal.hpp.

4.98.2.10 `double AbcSimplexPrimal::pivotStruct::dualOut_`

Definition at line 220 of file AbcSimplexPrimal.hpp.

4.98.2.11 `double AbcSimplexPrimal::pivotStruct::lowerOut_`

Definition at line 221 of file AbcSimplexPrimal.hpp.

4.98.2.12 `double AbcSimplexPrimal::pivotStruct::upperOut_`

Definition at line 222 of file AbcSimplexPrimal.hpp.

4.98.2.13 `double AbcSimplexPrimal::pivotStruct::valueOut_`

Definition at line 223 of file AbcSimplexPrimal.hpp.

4.98.2.14 int `AbcSimplexPrimal::pivotStruct::sequenceOut_`

Definition at line 224 of file `AbcSimplexPrimal.hpp`.

4.98.2.15 int `AbcSimplexPrimal::pivotStruct::directionOut_`

Definition at line 225 of file `AbcSimplexPrimal.hpp`.

4.98.2.16 int `AbcSimplexPrimal::pivotStruct::pivotRow_`

Definition at line 226 of file `AbcSimplexPrimal.hpp`.

4.98.2.17 int `AbcSimplexPrimal::pivotStruct::valuesPass_`

Definition at line 227 of file `AbcSimplexPrimal.hpp`.

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

- [src/AbcSimplexPrimal.hpp](#)

4.99 scatterStruct Struct Reference

```
#include <CoinAbcHelperFunctions.hpp>
```

Public Attributes

- [scatterUpdate](#) [functionPointer](#)
- `CoinBigIndex` [offset](#)
- int [number](#)

4.99.1 Detailed Description

Definition at line 534 of file `CoinAbcHelperFunctions.hpp`.

4.99.2 Member Data Documentation

4.99.2.1 `scatterUpdate` `scatterStruct::functionPointer`

Definition at line 535 of file `CoinAbcHelperFunctions.hpp`.

4.99.2.2 `CoinBigIndex` `scatterStruct::offset`

Definition at line 536 of file `CoinAbcHelperFunctions.hpp`.

4.99.2.3 int `scatterStruct::number`

Definition at line 537 of file `CoinAbcHelperFunctions.hpp`.

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

- [src/CoinAbcHelperFunctions.hpp](#)

5 File Documentation

5.1 src/AbcCommon.hpp File Reference

```
#include "ClpConfig.h"
```

5.2 src/AbcDualRowDantzig.hpp File Reference

```
#include "AbcDualRowPivot.hpp"
```

Classes

- class [AbcDualRowDantzig](#)
Dual Row Pivot Dantzig Algorithm Class.

5.3 src/AbcDualRowPivot.hpp File Reference

```
#include "AbcCommon.hpp"
```

Classes

- class [AbcDualRowPivot](#)
Dual Row Pivot Abstract Base Class.

5.4 src/AbcDualRowSteepest.hpp File Reference

```
#include "AbcDualRowPivot.hpp"
```

Classes

- class [AbcDualRowSteepest](#)
Dual Row Pivot Steepest Edge Algorithm Class.

Macros

- #define [DEVEX_TRY_NORM](#) 1.0e-8
- #define [DEVEX_ADD_ONE](#) 1.0

5.4.1 Macro Definition Documentation

5.4.1.1 #define DEVEX_TRY_NORM 1.0e-8

Definition at line 155 of file AbcDualRowSteepest.hpp.

5.4.1.2 #define DEVEX_ADD_ONE 1.0

Definition at line 156 of file AbcDualRowSteepest.hpp.

5.5 src/AbcMatrix.hpp File Reference

```
#include "CoinPragma.hpp"
#include "ClpMatrixBase.hpp"
#include "AbcSimplex.hpp"
#include "CoinAbcHelperFunctions.hpp"
```

Classes

- class [AbcMatrix](#)
- class [AbcMatrix2](#)
- struct [blockStruct3](#)
- class [AbcMatrix3](#)

Macros

Data members

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

- #define [NUMBER_ROW_BLOCKS](#) 1
- #define [NUMBER_COLUMN_BLOCKS](#) 1

5.5.1 Macro Definition Documentation

5.5.1.1 #define NUMBER_ROW_BLOCKS 1

Definition at line 407 of file AbcMatrix.hpp.

5.5.1.2 #define NUMBER_COLUMN_BLOCKS 1

Definition at line 408 of file AbcMatrix.hpp.

5.6 src/AbcNonLinearCost.hpp File Reference

```
#include "CoinPragma.hpp"
#include "AbcCommon.hpp"
```

Classes

- class [AbcNonLinearCost](#)

Macros

- `#define CLP_BELOW_LOWER 0`
Trivial class to deal with non linear costs.
- `#define CLP_FEASIBLE 1`
- `#define CLP_ABOVE_UPPER 2`
- `#define CLP_SAME 4`

Functions

- `int originalStatus` (unsigned char status)
- `int currentStatus` (unsigned char status)
- `void setOriginalStatus` (unsigned char &status, int value)
- `void setCurrentStatus` (unsigned char &status, int value)
- `void setInitialStatus` (unsigned char &status)
- `void setSameStatus` (unsigned char &status)

5.6.1 Macro Definition Documentation

5.6.1.1 `#define CLP_BELOW_LOWER 0`

Trivial class to deal with non linear costs.

I don't make any explicit assumptions about convexity but I am sure I do make implicit ones.

One interesting idea for normal LP's will be to allow non-basic variables to come into basis as infeasible i.e. if variable at lower bound has very large positive reduced cost (when problem is infeasible) could it reduce overall problem infeasibility more by bringing it into basis below its lower bound.

Another feature would be to automatically discover when problems are convex piecewise linear and re-formulate to use non-linear. I did some work on this many years ago on "grade" problems, but while it improved primal interior point algorithms were much better for that particular problem.

Definition at line 39 of file AbcNonLinearCost.hpp.

5.6.1.2 `#define CLP_FEASIBLE 1`

Definition at line 40 of file AbcNonLinearCost.hpp.

5.6.1.3 `#define CLP_ABOVE_UPPER 2`

Definition at line 41 of file AbcNonLinearCost.hpp.

5.6.1.4 `#define CLP_SAME 4`

Definition at line 42 of file AbcNonLinearCost.hpp.

5.6.2 Function Documentation

5.6.2.1 `int originalStatus (unsigned char status) [inline]`

Definition at line 43 of file AbcNonLinearCost.hpp.

5.6.2.2 `int currentStatus (unsigned char status) [inline]`

Definition at line 47 of file `AbcNonLinearCost.hpp`.

5.6.2.3 `void setOriginalStatus (unsigned char & status, int value) [inline]`

Definition at line 51 of file `AbcNonLinearCost.hpp`.

5.6.2.4 `void setCurrentStatus (unsigned char & status, int value) [inline]`

Definition at line 56 of file `AbcNonLinearCost.hpp`.

5.6.2.5 `void setInitialStatus (unsigned char & status) [inline]`

Definition at line 61 of file `AbcNonLinearCost.hpp`.

5.6.2.6 `void setSameStatus (unsigned char & status) [inline]`

Definition at line 65 of file `AbcNonLinearCost.hpp`.

5.7 `src/AbcPrimalColumnDantzig.hpp` File Reference

```
#include "AbcPrimalColumnPivot.hpp"
```

Classes

- class [AbcPrimalColumnDantzig](#)
Primal Column Pivot Dantzig Algorithm Class.

5.8 `src/AbcPrimalColumnPivot.hpp` File Reference

```
#include "AbcCommon.hpp"
```

Classes

- class [AbcPrimalColumnPivot](#)
Primal Column Pivot Abstract Base Class.

Macros

- `#define` [CLP_PRIMAL_SLACK_MULTIPLIER](#) 1.01

5.8.1 Macro Definition Documentation

5.8.1.1 `#define` [CLP_PRIMAL_SLACK_MULTIPLIER](#) 1.01

Definition at line 152 of file `AbcPrimalColumnPivot.hpp`.

5.9 src/AbcPrimalColumnSteepest.hpp File Reference

```
#include "AbcPrimalColumnPivot.hpp"
#include <bitset>
```

Classes

- class [AbcPrimalColumnSteepest](#)
Primal Column Pivot Steepest Edge Algorithm Class.

5.10 src/AbcSimplex.hpp File Reference

```
#include <iostream>
#include <cfloat>
#include "ClpModel.hpp"
#include "ClpMatrixBase.hpp"
#include "CoinIndexedVector.hpp"
#include "AbcCommon.hpp"
#include "ClpSolve.hpp"
#include "CoinAbcCommon.hpp"
#include "ClpSimplex.hpp"
```

Classes

- struct [CoinAbcThreadInfo](#)
- class [AbcSimplex](#)

Macros

- #define [PAN](#)
This solves LPs using the simplex method.
- #define [TRY_ABC_GUS](#)
- #define [HEAVY_PERTURBATION](#) 57

Functions less likely to be useful to casual user

- #define [rowUseScale_](#) scaleFromExternal_
- #define [inverseRowUseScale_](#) scaleToExternal_

status methods

- #define [NUMBER_THREADS](#) 3

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

- #define [startAtLowerNoOther_](#) maximumAbcNumberRows_

Start of variables at lower bound with no upper.

- `#define ALL_STATUS_OK 2048`

State of problem State of external arrays 2048 - status OK 4096 - row primal solution OK 8192 - row dual solution OK 16384 - column primal solution OK 32768 - column dual solution OK 65536 - Everything not going smoothly (when smooth we forget about tiny bad djs) 131072 - when increasing rows add a bit 262144 - scale matrix and create new one 524288 - do basis and order 1048576 - just status (and check if order needed) 2097152 - just solution 4194304 - just redo bounds (and offset) Bottom bits say if usefulArray in use.

- `#define ROW_PRIMAL_OK 4096`
- `#define ROW_DUAL_OK 8192`
- `#define COLUMN_PRIMAL_OK 16384`
- `#define COLUMN_DUAL_OK 32768`
- `#define PESSIMISTIC 65536`
- `#define ADD_A_BIT 131072`
- `#define DO_SCALE_AND_MATRIX 262144`
- `#define DO_BASIS_AND_ORDER 524288`
- `#define DO_STATUS 1048576`
- `#define DO_SOLUTION 2097152`
- `#define DO_JUST_BOUNDS 0x400000`
- `#define NEED_BASIS_SORT 0x800000`
- `#define FAKE_SUPERBASIC 0x1000000`
- `#define VALUES_PASS 0x2000000`
- `#define VALUES_PASS2 0x4000000`
- `#define ABC_NUMBER_USEFUL 8`

Useful arrays (all of row+column+2 length)

Functions

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

A function that tests the methods in the [AbcSimplex](#) class.

5.10.1 Macro Definition Documentation

5.10.1.1 `#define PAN`

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 [AbcSimplexDual.hpp](#) and at top of `AbcSimplexDual.cpp` for primal see [AbcSimplexPrimal.hpp](#) and at top of `AbcSimplexPrimal.cpp`

There is an algorithm data member. + for primal variations and - for dual variations

Definition at line 52 of file `AbcSimplex.hpp`.

5.10.1.2 `#define TRY_ABC_GUS`

Definition at line 56 of file `AbcSimplex.hpp`.

5.10.1.3 `#define HEAVY_PERTURBATION 57`

Definition at line 57 of file `AbcSimplex.hpp`.

5.10.1.4 #define rowUseScale_ scaleFromExternal_

Definition at line 383 of file AbcSimplex.hpp.

5.10.1.5 #define inverseRowUseScale_ scaleToExternal_

Definition at line 384 of file AbcSimplex.hpp.

5.10.1.6 #define NUMBER_THREADS 3

Definition at line 864 of file AbcSimplex.hpp.

5.10.1.7 #define startAtLowerNoOther_ maximumAbcNumberRows_

Start of variables at lower bound with no upper.

Definition at line 1059 of file AbcSimplex.hpp.

5.10.1.8 #define ALL_STATUS_OK 2048

State of problem State of external arrays 2048 - status OK 4096 - row primal solution OK 8192 - row dual solution OK 16384 - column primal solution OK 32768 - column dual solution OK 65536 - Everything not going smoothly (when smooth we forget about tiny bad djs) 131072 - when increasing rows add a bit 262144 - scale matrix and create new one 524288 - do basis and order 1048576 - just status (and check if order needed) 2097152 - just solution 4194304 - just redo bounds (and offset) Bottom bits say if usefulArray in use.

Definition at line 1091 of file AbcSimplex.hpp.

5.10.1.9 #define ROW_PRIMAL_OK 4096

Definition at line 1092 of file AbcSimplex.hpp.

5.10.1.10 #define ROW_DUAL_OK 8192

Definition at line 1093 of file AbcSimplex.hpp.

5.10.1.11 #define COLUMN_PRIMAL_OK 16384

Definition at line 1094 of file AbcSimplex.hpp.

5.10.1.12 #define COLUMN_DUAL_OK 32768

Definition at line 1095 of file AbcSimplex.hpp.

5.10.1.13 #define PESSIMISTIC 65536

Definition at line 1096 of file AbcSimplex.hpp.

5.10.1.14 #define ADD_A_BIT 131072

Definition at line 1097 of file AbcSimplex.hpp.

5.10.1.15 #define DO_SCALE_AND_MATRIX 262144

Definition at line 1098 of file AbcSimplex.hpp.

5.10.1.16 `#define DO_BASIS_AND_ORDER 524288`

Definition at line 1099 of file `AbcSimplex.hpp`.

5.10.1.17 `#define DO_STATUS 1048576`

Definition at line 1100 of file `AbcSimplex.hpp`.

5.10.1.18 `#define DO_SOLUTION 2097152`

Definition at line 1101 of file `AbcSimplex.hpp`.

5.10.1.19 `#define DO_JUST_BOUNDS 0x400000`

Definition at line 1102 of file `AbcSimplex.hpp`.

5.10.1.20 `#define NEED_BASIS_SORT 0x800000`

Definition at line 1103 of file `AbcSimplex.hpp`.

5.10.1.21 `#define FAKE_SUPERBASIC 0x1000000`

Definition at line 1104 of file `AbcSimplex.hpp`.

5.10.1.22 `#define VALUES_PASS 0x2000000`

Definition at line 1105 of file `AbcSimplex.hpp`.

5.10.1.23 `#define VALUES_PASS2 0x4000000`

Definition at line 1106 of file `AbcSimplex.hpp`.

5.10.1.24 `#define ABC_NUMBER_USEFUL 8`

Useful arrays (all of row+column+2 length)

Definition at line 1239 of file `AbcSimplex.hpp`.

5.10.2 Function Documentation

5.10.2.1 `void AbcSimplexUnitTest (const std::string & mpsDir)`

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

5.11 `src/AbcSimplexDual.hpp` File Reference

```
#include "AbcSimplex.hpp"
```

Classes

- struct [dualColumnResult](#)
- class [AbcSimplexDual](#)

This solves LPs using the dual simplex method.

5.12 src/AbcSimplexFactorization.hpp File Reference

```
#include "CoinPragma.hpp"
#include "CoinAbcCommon.hpp"
#include "CoinAbcFactorization.hpp"
#include "AbcMatrix.hpp"
#include "AbcSimplex.hpp"
```

Classes

- class [AbcSimplexFactorization](#)

This just implements AbcFactorization when an [AbcMatrix](#) object is passed.

5.13 src/AbcSimplexPrimal.hpp File Reference

```
#include "AbcSimplex.hpp"
```

Classes

- class [AbcSimplexPrimal](#)

This solves LPs using the primal simplex method.

- struct [AbcSimplexPrimal::pivotStruct](#)

5.14 src/AbcWarmStart.hpp File Reference

```
#include "AbcCommon.hpp"
#include "CoinWarmStartBasis.hpp"
#include "ClpSimplex.hpp"
```

Classes

- class [AbcWarmStartOrganizer](#)
- class [AbcWarmStart](#)

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


```

AM_DBL_DEXTRA5,
CLP_PARAM_INT_SOLVERLOGLEVEL = 101, CLP_PARAM_INT_LOGLEVEL = 101, CLP_PARAM_INT_MA-
XFACTOR, CLP_PARAM_INT_PERTVALUE,
CLP_PARAM_INT_MAXITERATION, CLP_PARAM_INT_PRESOLVEPASS, CLP_PARAM_INT_IDIOT, CLP_-
PARAM_INT_SPRINT,
CLP_PARAM_INT_OUTPUTFORMAT, CLP_PARAM_INT_SLPVALUE, CLP_PARAM_INT_PRESOLVEOPTI-
ONS, CLP_PARAM_INT_PRINTOPTIONS,
CLP_PARAM_INT_SPECIALOPTIONS, CLP_PARAM_INT_SUBSTITUTION, CLP_PARAM_INT_DUALIZE, C-
LP_PARAM_INT_VERBOSE,
CLP_PARAM_INT_CPP, CLP_PARAM_INT_PROCESSTUNE, CLP_PARAM_INT_USESOLUTION, CLP_PAR-
AM_INT_RANDOMSEED,
CLP_PARAM_INT_MORESPECIALOPTIONS, CLP_PARAM_INT_DECOMPOSE_BLOCKS, CBC_PARAM_IN-
T_STRONGBRANCHING = 151, CBC_PARAM_INT_CUTDEPTH,
CBC_PARAM_INT_MAXNODES, CBC_PARAM_INT_NUMBERBEFORE, CBC_PARAM_INT_NUMBERANAL-
YZE, CBC_PARAM_INT_MIPOPTIONS,
CBC_PARAM_INT_MOREMIPOPTIONS, CBC_PARAM_INT_MAXHOTITS, CBC_PARAM_INT_FPUMPITS,
CBC_PARAM_INT_MAXSOLS,
CBC_PARAM_INT_FPUMPTUNE, CBC_PARAM_INT_TESTOSI, CBC_PARAM_INT_EXTRA1, CBC_PARAM-
_INT_EXTRA2,
CBC_PARAM_INT_EXTRA3, CBC_PARAM_INT_EXTRA4, CBC_PARAM_INT_DEPTHMINIBAB, CBC PARA-
M_INT_CUTPASSINTREE,
CBC_PARAM_INT_THREADS, CBC_PARAM_INT_CUTPASS, CBC_PARAM_INT_VUBTRY, CBC_PARAM_I-
NT_DENSE,
CBC_PARAM_INT_EXPERIMENT, CBC_PARAM_INT_DIVEOPT, CBC_PARAM_INT_STRATEGY, CBC PA-
RAM_INT_SMALLFACT,
CBC_PARAM_INT_HOPTIONS, CBC_PARAM_INT_CUTLENGTH, CBC_PARAM_INT_FPUMPTUNE2, CBC_-
PARAM_INT_MAXSAVEDSOLS,
CBC_PARAM_INT_RANDOMSEED, CBC_PARAM_INT_MULTIPLEROOT, CBC_PARAM_INT_STRONG_S-
TRATEGY, CBC_PARAM_INT_EXTRA_VARIABLES,
CBC_PARAM_INT_MAX_SLOW_CUTS, CBC_PARAM_INT_MOREMOREMIPOPTIONS, CLP_PARAM_STR-
_DIRECTION = 201, CLP_PARAM_STR_DUALPIVOT,
CLP_PARAM_STR_SCALING, CLP_PARAM_STR_ERRORSALLOWED, CLP_PARAM_STR_KEEPNAMES,
CLP_PARAM_STR_SPARSEFACTOR,
CLP_PARAM_STR_PRIMALPIVOT, CLP_PARAM_STR_PRESOLVE, CLP_PARAM_STR_CRASH, CLP PAR-
AM_STR_BIASLU,
CLP_PARAM_STR_PERTURBATION, CLP_PARAM_STR_MESSAGES, CLP_PARAM_STR_AUTOSCALE,
CLP_PARAM_STR_CHOLESKY,
CLP_PARAM_STR_KKT, CLP_PARAM_STR_BARRIERSCALE, CLP_PARAM_STR_GAMMA, CLP_PARAM_-
STR_CROSSOVER,
CLP_PARAM_STR_PFI, CLP_PARAM_STR_INTPRINT, CLP_PARAM_STR_VECTOR, CLP_PARAM_STR_F-
ACTORIZATION,
CLP_PARAM_STR_ALLCOMMANDS, CLP_PARAM_STR_TIME_MODE, CLP_PARAM_STR_ABCWANTED,
CBC_PARAM_STR_NODESTRATEGY = 251,
CBC_PARAM_STR_BRANCHSTRATEGY, CBC_PARAM_STR_CUTSSTRATEGY, CBC_PARAM_STR_HEU-
RISTICSTRATEGY, CBC_PARAM_STR_GOMORYCUTS,
CBC_PARAM_STR_PROBINGCUTS, CBC_PARAM_STR_KNAPSACKCUTS, CBC_PARAM_STR_REDSPLI-
TCUTS, CBC_PARAM_STR_ROUNDING,
CBC_PARAM_STR_SOLVER, CBC_PARAM_STR_CLIQUCUTS, CBC_PARAM_STR_COSTSTRATEGY, C-
BC_PARAM_STR_FLOWCUTS,
CBC_PARAM_STR_MIXEDCUTS, CBC_PARAM_STR_TWOMIRCUTS, CBC_PARAM_STR_PREPROCESS,
CBC_PARAM_STR_FPUMP,
CBC_PARAM_STR_GREEDY, CBC_PARAM_STR_COMBINE, CBC_PARAM_STR_PROXIMITY, CBC PAR-
AM_STR_LOCALTREE,
CBC_PARAM_STR_SOS, CBC_PARAM_STR_LANDPCUTS, CBC_PARAM_STR_RINS, CBC_PARAM_STR-

```

```

_RESIDCUTS,
CBC_PARAM_STR_RENS, CBC_PARAM_STR_DIVINGS, CBC_PARAM_STR_DIVINGC, CBC_PARAM_STR_DIVINGF,
CBC_PARAM_STR_DIVINGG, CBC_PARAM_STR_DIVINGL, CBC_PARAM_STR_DIVINGP, CBC_PARAM_STR_DIVINGV,
CBC_PARAM_STR_DINS, CBC_PARAM_STR_PIVOTANDFIX, CBC_PARAM_STR_RANDROUND, CBC_PARAM_STR_NAIVE,
CBC_PARAM_STR_ZEROHALFCUTS, CBC_PARAM_STR_CPX, CBC_PARAM_STR_CROSSOVER2, CBC_PARAM_STR_PIVOTANDCOMPLEMENT,
CBC_PARAM_STR_VND, CBC_PARAM_STR_LAGOMORYCUTS, CBC_PARAM_STR_LATWOMIRCUTS, CBC_PARAM_STR_REDSPLIT2CUTS,
CBC_PARAM_STR_GMICUTS, CBC_PARAM_STR_CUTOFF_CONSTRAINT, CBC_PARAM_STR_DW, CLP_PARAM_ACTION_DIRECTORY = 301,
CLP_PARAM_ACTION_DIRSAMPLE, CLP_PARAM_ACTION_DIRNETLIB, CBC_PARAM_ACTION_DIRMIPLIB, CLP_PARAM_ACTION_IMPORT,
CLP_PARAM_ACTION_EXPORT, CLP_PARAM_ACTION_RESTORE, CLP_PARAM_ACTION_SAVE, CLP_PARAM_ACTION_DUALSIMPLEX,
CLP_PARAM_ACTION_PRIMALSIMPLEX, CLP_PARAM_ACTION_EITHERSIMPLEX, CLP_PARAM_ACTION_MAXIMIZE, CLP_PARAM_ACTION_MINIMIZE,
CLP_PARAM_ACTION_EXIT, CLP_PARAM_ACTION_STDIN, CLP_PARAM_ACTION_UNITTEST, CLP_PARAM_ACTION_NETLIB_EITHER,
CLP_PARAM_ACTION_NETLIB_DUAL, CLP_PARAM_ACTION_NETLIB_PRIMAL, CLP_PARAM_ACTION_SOLUTION, CLP_PARAM_ACTION_SAVESOL,
CLP_PARAM_ACTION_TIGHTEN, CLP_PARAM_ACTION_FAKEBOUND, CLP_PARAM_ACTION_HELP, CLP_PARAM_ACTION_PLUSMINUS,
CLP_PARAM_ACTION_NETWORK, CLP_PARAM_ACTION_ALLSLACK, CLP_PARAM_ACTION_REVERSE, CLP_PARAM_ACTION_BARRIER,
CLP_PARAM_ACTION_NETLIB_BARRIER, CLP_PARAM_ACTION_NETLIB_TUNE, CLP_PARAM_ACTION_REALLY_SCALE, CLP_PARAM_ACTION_BASISIN,
CLP_PARAM_ACTION_BASISOUT, CLP_PARAM_ACTION_SOLVECONTINUOUS, CLP_PARAM_ACTION_CLEARCUTS, CLP_PARAM_ACTION_VERSION,
CLP_PARAM_ACTION_STATISTICS, CLP_PARAM_ACTION_DEBUG, CLP_PARAM_ACTION_DUMMY, CLP_PARAM_ACTION_PRINTMASK,
CLP_PARAM_ACTION_OUTDUPROWS, CLP_PARAM_ACTION_USERCLP, CLP_PARAM_ACTION_MODELIN, CLP_PARAM_ACTION_CSVSTATISTICS,
CLP_PARAM_ACTION_STOREDFILE, CLP_PARAM_ACTION_ENVIRONMENT, CLP_PARAM_ACTION_PARAMETERS, CLP_PARAM_ACTION_GMPL_SOLUTION,
CBC_PARAM_ACTION_BAB = 351, CBC_PARAM_ACTION_MIPLIB, CBC_PARAM_ACTION_STRENGTHEN, CBC_PARAM_ACTION_PRIORITYIN,
CBC_PARAM_ACTION_MIPSTART, CBC_PARAM_ACTION_USERCBC, CBC_PARAM_ACTION_DOHEURISTIC, CLP_PARAM_ACTION_NEXTBESTSOLUTION,
CBC_PARAM_NOTUSED_OSLSTUFF = 401, CBC_PARAM_NOTUSED_CBCSTUFF, CBC_PARAM_NOTUSED_INVALID = 1000 }

```

Parameter codes.

Functions

- std::string [CoinReadNextField](#) ()
Simple read stuff.
- std::string [CoinReadGetCommand](#) (int argc, const char *argv[])
- std::string [CoinReadGetString](#) (int argc, const char *argv[])
- int [CoinReadGetIntField](#) (int argc, const char *argv[], int *valid)
- double [CoinReadGetDoubleField](#) (int argc, const char *argv[], int *valid)

- void [CoinReadPrintit](#) (const char *input)
- void [setCbcOrClpPrinting](#) (bool yesNo)
- void [establishParams](#) (int &numberParameters, [CbcOrClpParam](#) *const parameters)
- int [whichParam](#) ([CbcOrClpParameterType](#) name, int numberParameters, [CbcOrClpParam](#) *const parameters)
- void [saveSolution](#) (const [ClpSimplex](#) *lpSolver, std::string fileName)

5.15.1 Macro Definition Documentation

5.15.1.1 #define CBCMAXPARAMETERS 250

Definition at line 510 of file CbcOrClpParam.hpp.

5.15.2 Enumeration Type Documentation

5.15.2.1 enum CbcOrClpParameterType

Parameter codes.

Parameter type ranges are allocated as follows

- 1 – 100 double parameters
- 101 – 200 integer parameters
- 201 – 250 string parameters
- 251 – 300 cuts etc(string but broken out for clarity)
- 301 – 400 'actions'

'Actions' do not necessarily invoke an immediate action; it's just that they don't fit neatly into the parameters array.

This coding scheme is in flux.

Enumerator

```
CBC_PARAM_GENERALQUERY
CBC_PARAM_FULLGENERALQUERY
CLP_PARAM_DBL_PRIMALTOLERANCE
CLP_PARAM_DBL_DUALTOLERANCE
CLP_PARAM_DBL_TIMELIMIT
CLP_PARAM_DBL_DUALBOUND
CLP_PARAM_DBL_PRIMALWEIGHT
CLP_PARAM_DBL_OBJSCALE
CLP_PARAM_DBL_RHSSCALE
CLP_PARAM_DBL_ZEROTOLERANCE
CBC_PARAM_DBL_INFEASIBILITYWEIGHT
CBC_PARAM_DBL_CUTOFF
CBC_PARAM_DBL_INTEGERTOLERANCE
CBC_PARAM_DBL_INCREMENT
CBC_PARAM_DBL_ALLOWABLEGAP
```

CBC_PARAM_DBL_TIMELIMIT_BAB
CBC_PARAM_DBL_GAPRATIO
CBC_PARAM_DBL_DJFIX
CBC_PARAM_DBL_TIGHTENFACTOR
CLP_PARAM_DBL_PRESVOLVETOLERANCE
CLP_PARAM_DBL_OBJSCALE2
CBC_PARAM_DBL_FAKEINCREMENT
CBC_PARAM_DBL_FAKECUTOFF
CBC_PARAM_DBL_ARTIFICIALCOST
CBC_PARAM_DBL_DEXTRA3
CBC_PARAM_DBL_SMALLBAB
CBC_PARAM_DBL_DEXTRA4
CBC_PARAM_DBL_DEXTRA5
CLP_PARAM_INT_SOLVERLOGLEVEL
CLP_PARAM_INT_LOGLEVEL
CLP_PARAM_INT_MAXFACTOR
CLP_PARAM_INT_PERTVALUE
CLP_PARAM_INT_MAXITERATION
CLP_PARAM_INT_PREOLVEPASS
CLP_PARAM_INT_IDIOT
CLP_PARAM_INT_SPRINT
CLP_PARAM_INT_OUTPUTFORMAT
CLP_PARAM_INT_SLPVALUE
CLP_PARAM_INT_PREOLVEOPTIONS
CLP_PARAM_INT_PRINTOPTIONS
CLP_PARAM_INT_SPECIALOPTIONS
CLP_PARAM_INT_SUBSTITUTION
CLP_PARAM_INT_DUALIZE
CLP_PARAM_INT_VERBOSE
CLP_PARAM_INT_CPP
CLP_PARAM_INT_PROCESSTUNE
CLP_PARAM_INT_USESOLUTION
CLP_PARAM_INT_RANDOMSEED
CLP_PARAM_INT_MORESPECIALOPTIONS
CLP_PARAM_INT_DECOMPOSE_BLOCKS
CBC_PARAM_INT_STRONGBRANCHING
CBC_PARAM_INT_CUTDEPTH
CBC_PARAM_INT_MAXNODES
CBC_PARAM_INT_NUMBERBEFORE
CBC_PARAM_INT_NUMBERANALYZE
CBC_PARAM_INT_MIPOPTIONS
CBC_PARAM_INT_MOREMIPOPTIONS

CBC_PARAM_INT_MAXHOTITS
CBC_PARAM_INT_FPUMPITS
CBC_PARAM_INT_MAXSOLS
CBC_PARAM_INT_FPUMPTUNE
CBC_PARAM_INT_TESTOSI
CBC_PARAM_INT_EXTRA1
CBC_PARAM_INT_EXTRA2
CBC_PARAM_INT_EXTRA3
CBC_PARAM_INT_EXTRA4
CBC_PARAM_INT_DEPTHMINIBAB
CBC_PARAM_INT_CUTPASSINTREE
CBC_PARAM_INT_THREADS
CBC_PARAM_INT_CUTPASS
CBC_PARAM_INT_VUBTRY
CBC_PARAM_INT_DENSE
CBC_PARAM_INT_EXPERIMENT
CBC_PARAM_INT_DIVEOPT
CBC_PARAM_INT_STRATEGY
CBC_PARAM_INT_SMALLFACT
CBC_PARAM_INT_HOPTIONS
CBC_PARAM_INT_CUTLENGTH
CBC_PARAM_INT_FPUMPTUNE2
CBC_PARAM_INT_MAXSAVEDSOLS
CBC_PARAM_INT_RANDOMSEED
CBC_PARAM_INT_MULTIPLEROOTS
CBC_PARAM_INT_STRONG_STRATEGY
CBC_PARAM_INT_EXTRA_VARIABLES
CBC_PARAM_INT_MAX_SLOW_CUTS
CBC_PARAM_INT_MOREMOREMIPOPTIONS
CLP_PARAM_STR_DIRECTION
CLP_PARAM_STR_DUALPIVOT
CLP_PARAM_STR_SCALING
CLP_PARAM_STR_ERRORSALLOWED
CLP_PARAM_STR_KEEPNAMES
CLP_PARAM_STR_SPARSEFACTOR
CLP_PARAM_STR_PRIMALPIVOT
CLP_PARAM_STR_PRESOLVE
CLP_PARAM_STR_CRASH
CLP_PARAM_STR_BIASLU
CLP_PARAM_STR_PERTURBATION
CLP_PARAM_STR_MESSAGES
CLP_PARAM_STR_AUTOSCALE

CLP_PARAM_STR_CHOLESKY
CLP_PARAM_STR_KKT
CLP_PARAM_STR_BARRIERSCALE
CLP_PARAM_STR_GAMMA
CLP_PARAM_STR_CROSSOVER
CLP_PARAM_STR_PFI
CLP_PARAM_STR_INTPRINT
CLP_PARAM_STR_VECTOR
CLP_PARAM_STR_FACTORIZATION
CLP_PARAM_STR_ALLCOMMANDS
CLP_PARAM_STR_TIME_MODE
CLP_PARAM_STR_ABCWANTED
CBC_PARAM_STR_NODESTRATEGY
CBC_PARAM_STR_BRANCHSTRATEGY
CBC_PARAM_STR_CUTSSTRATEGY
CBC_PARAM_STR_HEURISTICSTRATEGY
CBC_PARAM_STR_GOMORYCUTS
CBC_PARAM_STR_PROBINGCUTS
CBC_PARAM_STR_KNAPSACKCUTS
CBC_PARAM_STR_REDSPLITCUTS
CBC_PARAM_STR_ROUNDING
CBC_PARAM_STR_SOLVER
CBC_PARAM_STR_CLIQUECUTS
CBC_PARAM_STR_COSTSTRATEGY
CBC_PARAM_STR_FLOWCUTS
CBC_PARAM_STR_MIXEDCUTS
CBC_PARAM_STR_TWOMIRCUTS
CBC_PARAM_STR_PREPROCESS
CBC_PARAM_STR_FPUMP
CBC_PARAM_STR_GREEDY
CBC_PARAM_STR_COMBINE
CBC_PARAM_STR_PROXIMITY
CBC_PARAM_STR_LOCALTREE
CBC_PARAM_STR_SOS
CBC_PARAM_STR_LANDPCUTS
CBC_PARAM_STR_RINS
CBC_PARAM_STR_RESIDCUTS
CBC_PARAM_STR_RENS
CBC_PARAM_STR_DIVINGS
CBC_PARAM_STR_DIVINGC
CBC_PARAM_STR_DIVINGF
CBC_PARAM_STR_DIVINGG

CBC_PARAM_STR_DIVINGL
CBC_PARAM_STR_DIVINGP
CBC_PARAM_STR_DIVINGV
CBC_PARAM_STR_DINS
CBC_PARAM_STR_PIVOTANDFIX
CBC_PARAM_STR_RANDROUND
CBC_PARAM_STR_NAIVE
CBC_PARAM_STR_ZEROHALFCUTS
CBC_PARAM_STR_CPX
CBC_PARAM_STR_CROSSOVER2
CBC_PARAM_STR_PIVOTANDCOMPLEMENT
CBC_PARAM_STR_VND
CBC_PARAM_STR_LAGOMORYCUTS
CBC_PARAM_STR_LATWOMIRCUTS
CBC_PARAM_STR_REDSPLIT2CUTS
CBC_PARAM_STR_GMICUTS
CBC_PARAM_STR_CUTOFF_CONSTRAINT
CBC_PARAM_STR_DW
CLP_PARAM_ACTION_DIRECTORY
CLP_PARAM_ACTION_DIRSAMPLE
CLP_PARAM_ACTION_DIRNETLIB
CBC_PARAM_ACTION_DIRMIPLIB
CLP_PARAM_ACTION_IMPORT
CLP_PARAM_ACTION_EXPORT
CLP_PARAM_ACTION_RESTORE
CLP_PARAM_ACTION_SAVE
CLP_PARAM_ACTION_DUALSIMPLEX
CLP_PARAM_ACTION_PRIMALSIMPLEX
CLP_PARAM_ACTION_EITHERSIMPLEX
CLP_PARAM_ACTION_MAXIMIZE
CLP_PARAM_ACTION_MINIMIZE
CLP_PARAM_ACTION_EXIT
CLP_PARAM_ACTION_STDIN
CLP_PARAM_ACTION_UNITTEST
CLP_PARAM_ACTION_NETLIB_EITHER
CLP_PARAM_ACTION_NETLIB_DUAL
CLP_PARAM_ACTION_NETLIB_PRIMAL
CLP_PARAM_ACTION_SOLUTION
CLP_PARAM_ACTION_SAVESOL
CLP_PARAM_ACTION_TIGHTEN
CLP_PARAM_ACTION_FAKEBOUND
CLP_PARAM_ACTION_HELP

CLP_PARAM_ACTION_PLUSMINUS
CLP_PARAM_ACTION_NETWORK
CLP_PARAM_ACTION_ALLSLACK
CLP_PARAM_ACTION_REVERSE
CLP_PARAM_ACTION_BARRIER
CLP_PARAM_ACTION_NETLIB_BARRIER
CLP_PARAM_ACTION_NETLIB_TUNE
CLP_PARAM_ACTION_REALLY_SCALE
CLP_PARAM_ACTION_BASISIN
CLP_PARAM_ACTION_BASISOUT
CLP_PARAM_ACTION_SOLVECONTINUOUS
CLP_PARAM_ACTION_CLEARCUTS
CLP_PARAM_ACTION_VERSION
CLP_PARAM_ACTION_STATISTICS
CLP_PARAM_ACTION_DEBUG
CLP_PARAM_ACTION_DUMMY
CLP_PARAM_ACTION_PRINTMASK
CLP_PARAM_ACTION_OUTDUPROWS
CLP_PARAM_ACTION_USERCLP
CLP_PARAM_ACTION_MODELIN
CLP_PARAM_ACTION_CSVSTATISTICS
CLP_PARAM_ACTION_STOREDFILE
CLP_PARAM_ACTION_ENVIRONMENT
CLP_PARAM_ACTION_PARAMETRICS
CLP_PARAM_ACTION_GMPL_SOLUTION
CBC_PARAM_ACTION_BAB
CBC_PARAM_ACTION_MIPLIB
CBC_PARAM_ACTION_STRENGTHEN
CBC_PARAM_ACTION_PRIORITYIN
CBC_PARAM_ACTION_MIPSTART
CBC_PARAM_ACTION_USERCBC
CBC_PARAM_ACTION_DOHEURISTIC
CLP_PARAM_ACTION_NEXTBESTSOLUTION
CBC_PARAM_NOTUSED_OSLSTUFF
CBC_PARAM_NOTUSED_CBCSTUFF
CBC_PARAM_NOTUSED_INVALID

Definition at line 47 of file CbcOrClpParam.hpp.

5.15.3 Function Documentation

5.15.3.1 `std::string CoinReadNextField ()`

Simple read stuff.

- 5.15.3.2 `std::string CoinReadGetCommand (int argc, const char * argv[])`
- 5.15.3.3 `std::string CoinReadGetString (int argc, const char * argv[])`
- 5.15.3.4 `int CoinReadGetIntField (int argc, const char * argv[], int * valid)`
- 5.15.3.5 `double CoinReadGetDoubleField (int argc, const char * argv[], int * valid)`
- 5.15.3.6 `void CoinReadPrintit (const char * input)`
- 5.15.3.7 `void setCbcOrClpPrinting (bool yesNo)`
- 5.15.3.8 `void establishParams (int & numberParameters, CbcOrClpParam *const parameters)`
- 5.15.3.9 `int whichParam (CbcOrClpParameterType name, int numberParameters, CbcOrClpParam *const parameters)`
- 5.15.3.10 `void saveSolution (const ClpSimplex * lpSolver, std::string fileName)`

5.16 src/Clp_ampl.h File Reference

Classes

- struct [ampl_info](#)

Functions

- int [readAmpl](#) ([ampl_info](#) *info, int argc, char **argv, void **coinModel)
- void [freeArrays1](#) ([ampl_info](#) *info)
- void [freeArrays2](#) ([ampl_info](#) *info)
- void [freeArgs](#) ([ampl_info](#) *info)
- void [writeAmpl](#) ([ampl_info](#) *info)
- int [ampl_obj_prec](#) ()

5.16.1 Function Documentation

- 5.16.1.1 `int readAmpl (ampl_info * info, int argc, char ** argv, void ** coinModel)`
- 5.16.1.2 `void freeArrays1 (ampl_info * info)`
- 5.16.1.3 `void freeArrays2 (ampl_info * info)`
- 5.16.1.4 `void freeArgs (ampl_info * info)`
- 5.16.1.5 `void writeAmpl (ampl_info * info)`
- 5.16.1.6 `int ampl_obj_prec ()`

5.17 src/Clp_C_Interface.h File Reference

```
#include "Coin_C_defines.h"
```

Typedefs

- typedef void [Clp_Solve](#)

Functions

Constructors and destructor

This is a first "C" interface to Clp.

It has similarities to the OSL V3 interface and only has most common functions. These do not have an exact analogue in C++. The user does not need to know structure of Clp_Simplex or Clp_Solve.

For (almost) all Clp_ functions outside this group there is an exact C++ analogue created by taking the first parameter out, removing the Clp_ from name and applying the method to an object of type [ClpSimplex](#).*

Similarly, for all ClpSolve_ functions there is an exact C++ analogue created by taking the first parameter out, removing the ClpSolve_ from name and applying the method to an object of type [ClpSolve](#).*

- COINLIBAPI Clp_Simplex *COINLINKAGE [Clp_newModel](#) (void)
Default constructor.
- COINLIBAPI void COINLINKAGE [Clp_deleteModel](#) (Clp_Simplex *model)
Destructor.
- COINLIBAPI [Clp_Solve](#) *COINLINKAGE [ClpSolve_new](#) ()
Default constructor.
- COINLIBAPI void COINLINKAGE [ClpSolve_delete](#) (Clp_Solve *solve)
Destructor.

Load model - loads some stuff and initializes others

- COINLIBAPI void COINLINKAGE [Clp_loadProblem](#) (Clp_Simplex *model, 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)
Loads a problem (the constraints on the rows are given by lower and upper bounds).
- COINLIBAPI void COINLINKAGE [Clp_loadQuadraticObjective](#) (Clp_Simplex *model, const int numberColumns, const CoinBigIndex *start, const int *column, const double *element)
- COINLIBAPI int COINLINKAGE [Clp_readMps](#) (Clp_Simplex *model, const char *filename, int keepNames, int ignoreErrors)
Read an mps file from the given filename.
- COINLIBAPI void COINLINKAGE [Clp_copyInIntegerInformation](#) (Clp_Simplex *model, const char *information)
Copy in integer informations.
- COINLIBAPI void COINLINKAGE [Clp_deleteIntegerInformation](#) (Clp_Simplex *model)
Drop integer informations.
- COINLIBAPI void COINLINKAGE [Clp_resize](#) (Clp_Simplex *model, int newNumberRows, int newNumberColumns)
Resizes rim part of model.
- COINLIBAPI void COINLINKAGE [Clp_deleteRows](#) (Clp_Simplex *model, int number, const int *which)
Deletes rows.
- COINLIBAPI void COINLINKAGE [Clp_addRows](#) (Clp_Simplex *model, int number, const double *rowLower, const double *rowUpper, const int *rowStarts, const int *columns, const double *elements)
Add rows.
- COINLIBAPI void COINLINKAGE [Clp_deleteColumns](#) (Clp_Simplex *model, int number, const int *which)
Deletes columns.
- COINLIBAPI void COINLINKAGE [Clp_addColumns](#) (Clp_Simplex *model, int number, const double *columnLower, const double *columnUpper, const double *objective, const int *columnStarts, const int *rows, const double *elements)

- Add columns.*
- COINLIBAPI void COINLINKAGE [Clp_chgRowLower](#) (Clp_Simplex *model, const double *rowLower)
Change row lower bounds.
- COINLIBAPI void COINLINKAGE [Clp_chgRowUpper](#) (Clp_Simplex *model, const double *rowUpper)
Change row upper bounds.
- COINLIBAPI void COINLINKAGE [Clp_chgColumnLower](#) (Clp_Simplex *model, const double *columnLower)
Change column lower bounds.
- COINLIBAPI void COINLINKAGE [Clp_chgColumnUpper](#) (Clp_Simplex *model, const double *columnUpper)
Change column upper bounds.
- COINLIBAPI void COINLINKAGE [Clp_chgObjCoefficients](#) (Clp_Simplex *model, const double *objIn)
Change objective coefficients.
- COINLIBAPI void COINLINKAGE [Clp_dropNames](#) (Clp_Simplex *model)
Drops names - makes lengthnames 0 and names empty.
- COINLIBAPI void COINLINKAGE [Clp_copyNames](#) (Clp_Simplex *model, const char *const *rowNames, const char *const *columnNames)
Copies in names.

gets and sets - you will find some synonyms at the end of this file

- COINLIBAPI int COINLINKAGE [Clp_numberRows](#) (Clp_Simplex *model)
Number of rows.
- COINLIBAPI int COINLINKAGE [Clp_numberColumns](#) (Clp_Simplex *model)
Number of columns.
- COINLIBAPI double COINLINKAGE [Clp_primalTolerance](#) (Clp_Simplex *model)
Primal tolerance to use.
- COINLIBAPI void COINLINKAGE [Clp_setPrimalTolerance](#) (Clp_Simplex *model, double value)
- COINLIBAPI double COINLINKAGE [Clp_dualTolerance](#) (Clp_Simplex *model)
Dual tolerance to use.
- COINLIBAPI void COINLINKAGE [Clp_setDualTolerance](#) (Clp_Simplex *model, double value)
- COINLIBAPI double COINLINKAGE [Clp_dualObjectiveLimit](#) (Clp_Simplex *model)
Dual objective limit.
- COINLIBAPI void COINLINKAGE [Clp_setDualObjectiveLimit](#) (Clp_Simplex *model, double value)
- COINLIBAPI double COINLINKAGE [Clp_objectiveOffset](#) (Clp_Simplex *model)
Objective offset.
- COINLIBAPI void COINLINKAGE [Clp_setObjectiveOffset](#) (Clp_Simplex *model, double value)
- COINLIBAPI void COINLINKAGE [Clp_problemName](#) (Clp_Simplex *model, int maxNumberCharacters, char *array)
Fills in array with problem name.
- COINLIBAPI int COINLINKAGE [Clp_setProblemName](#) (Clp_Simplex *model, int maxNumberCharacters, char *array)
- COINLIBAPI int COINLINKAGE [Clp_numberIterations](#) (Clp_Simplex *model)
Number of iterations.
- COINLIBAPI void COINLINKAGE [Clp_setNumberIterations](#) (Clp_Simplex *model, int numberIterations)
- COINLIBAPI int [maximumIterations](#) (Clp_Simplex *model)
Maximum number of iterations.
- COINLIBAPI void COINLINKAGE [Clp_setMaximumIterations](#) (Clp_Simplex *model, int value)
- COINLIBAPI double COINLINKAGE [Clp_maximumSeconds](#) (Clp_Simplex *model)
Maximum time in seconds (from when set called)
- COINLIBAPI void COINLINKAGE [Clp_setMaximumSeconds](#) (Clp_Simplex *model, double value)
- COINLIBAPI int COINLINKAGE [Clp_hitMaximumIterations](#) (Clp_Simplex *model)
Returns true if hit maximum iterations (or time)
- COINLIBAPI int COINLINKAGE [Clp_status](#) (Clp_Simplex *model)
Status of problem: 0 - optimal 1 - primal infeasible 2 - dual infeasible 3 - stopped on iterations etc 4 - stopped due to errors.
- COINLIBAPI void COINLINKAGE [Clp_setProblemStatus](#) (Clp_Simplex *model, int problemStatus)

- Set problem status.*
- COINLIBAPI int COINLINKAGE [Clp_secondaryStatus](#) (Clp_Simplex *model)
 - Secondary status of problem - may get extended 0 - none 1 - primal infeasible because dual limit reached 2 - scaled problem optimal - unscaled has primal infeasibilities 3 - scaled problem optimal - unscaled has dual infeasibilities 4 - scaled problem optimal - unscaled has both dual and primal infeasibilities.*
- COINLIBAPI void COINLINKAGE [Clp_setSecondaryStatus](#) (Clp_Simplex *model, int status)
- COINLIBAPI double COINLINKAGE [Clp_optimizationDirection](#) (Clp_Simplex *model)
 - Direction of optimization (1 - minimize, -1 - maximize, 0 - ignore.*
- COINLIBAPI void COINLINKAGE [Clp_setOptimizationDirection](#) (Clp_Simplex *model, double value)
- COINLIBAPI double *COINLINKAGE [Clp_primalRowSolution](#) (Clp_Simplex *model)
 - Primal row solution.*
- COINLIBAPI double *COINLINKAGE [Clp_primalColumnSolution](#) (Clp_Simplex *model)
 - Primal column solution.*
- COINLIBAPI double *COINLINKAGE [Clp_dualRowSolution](#) (Clp_Simplex *model)
 - Dual row solution.*
- COINLIBAPI double *COINLINKAGE [Clp_dualColumnSolution](#) (Clp_Simplex *model)
 - Reduced costs.*
- COINLIBAPI double *COINLINKAGE [Clp_rowLower](#) (Clp_Simplex *model)
 - Row lower.*
- COINLIBAPI double *COINLINKAGE [Clp_rowUpper](#) (Clp_Simplex *model)
 - Row upper.*
- COINLIBAPI double *COINLINKAGE [Clp_objective](#) (Clp_Simplex *model)
 - Objective.*
- COINLIBAPI double *COINLINKAGE [Clp_columnLower](#) (Clp_Simplex *model)
 - Column Lower.*
- COINLIBAPI double *COINLINKAGE [Clp_columnUpper](#) (Clp_Simplex *model)
 - Column Upper.*
- COINLIBAPI int COINLINKAGE [Clp_getNumElements](#) (Clp_Simplex *model)
 - Number of elements in matrix.*
- COINLIBAPI const CoinBigIndex
 - *COINLINKAGE [Clp_getVectorStarts](#) (Clp_Simplex *model)
- COINLIBAPI const int *COINLINKAGE [Clp_getIndices](#) (Clp_Simplex *model)
- COINLIBAPI const int *COINLINKAGE [Clp_getVectorLengths](#) (Clp_Simplex *model)
- COINLIBAPI const double
 - *COINLINKAGE [Clp_getElements](#) (Clp_Simplex *model)
- COINLIBAPI double COINLINKAGE [Clp_objectiveValue](#) (Clp_Simplex *model)
 - Objective value.*
- COINLIBAPI char *COINLINKAGE [Clp_integerInformation](#) (Clp_Simplex *model)
 - Integer information.*
- COINLIBAPI double *COINLINKAGE [Clp_infeasibilityRay](#) (Clp_Simplex *model)
 - Infeasibility/unbounded ray (NULL returned if none/wrong) Up to user to use free() on these arrays.*
- COINLIBAPI double *COINLINKAGE [Clp_unboundedRay](#) (Clp_Simplex *model)
- COINLIBAPI int COINLINKAGE [Clp_statusExists](#) (Clp_Simplex *model)
 - See if status array exists (partly for OsiClp)*
- COINLIBAPI unsigned char
 - *COINLINKAGE [Clp_statusArray](#) (Clp_Simplex *model)
 - Return address of status array (char[numberRows+numberColumns])*
- COINLIBAPI void COINLINKAGE [Clp_copyinStatus](#) (Clp_Simplex *model, const unsigned char *statusArray)
 - Copy in status vector.*
- COINLIBAPI int COINLINKAGE [Clp_getColumnStatus](#) (Clp_Simplex *model, int sequence)
- COINLIBAPI int COINLINKAGE [Clp_getRowStatus](#) (Clp_Simplex *model, int sequence)
- COINLIBAPI void COINLINKAGE [Clp_setColumnStatus](#) (Clp_Simplex *model, int sequence, int value)
- COINLIBAPI void COINLINKAGE [Clp_setRowStatus](#) (Clp_Simplex *model, int sequence, int value)
- COINLIBAPI void COINLINKAGE [Clp_setUserPointer](#) (Clp_Simplex *model, void *pointer)
 - User pointer for whatever reason.*
- COINLIBAPI void *COINLINKAGE [Clp_getUserPointer](#) (Clp_Simplex *model)

Message handling. Call backs are handled by ONE function

- COINLIBAPI void COINLINKAGE [Clp_registerCallBack](#) (Clp_Simplex *model, clp_callback userCallBack)
Pass in Callback function.
- COINLIBAPI void COINLINKAGE [Clp_clearCallBack](#) (Clp_Simplex *model)
Unset Callback function.
- COINLIBAPI void COINLINKAGE [Clp_setLogLevel](#) (Clp_Simplex *model, 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.
- COINLIBAPI int COINLINKAGE [Clp_logLevel](#) (Clp_Simplex *model)
- COINLIBAPI int COINLINKAGE [Clp_lengthNames](#) (Clp_Simplex *model)
length of names (0 means no names)
- COINLIBAPI void COINLINKAGE [Clp_rowName](#) (Clp_Simplex *model, int iRow, char *name)
Fill in array (at least lengthNames+1 long) with a row name.
- COINLIBAPI void COINLINKAGE [Clp_columnName](#) (Clp_Simplex *model, int iColumn, char *name)
Fill in array (at least lengthNames+1 long) with a column name.

Functions most useful to user

- COINLIBAPI int COINLINKAGE [Clp_initialSolve](#) (Clp_Simplex *model)
General solve algorithm which can do presolve.
- COINLIBAPI int COINLINKAGE [Clp_initialSolveWithOptions](#) (Clp_Simplex *model, [Clp_Solve](#) *)
Pass solve options.
- COINLIBAPI int COINLINKAGE [Clp_initialDualSolve](#) (Clp_Simplex *model)
Dual initial solve.
- COINLIBAPI int COINLINKAGE [Clp_initialPrimalSolve](#) (Clp_Simplex *model)
Primal initial solve.
- COINLIBAPI int COINLINKAGE [Clp_initialBarrierSolve](#) (Clp_Simplex *model)
Barrier initial solve.
- COINLIBAPI int COINLINKAGE [Clp_initialBarrierNoCrossSolve](#) (Clp_Simplex *model)
Barrier initial solve, no crossover.
- COINLIBAPI int COINLINKAGE [Clp_dual](#) (Clp_Simplex *model, int ifValuesPass)
Dual algorithm - see [ClpSimplexDual.hpp](#) for method.
- COINLIBAPI int COINLINKAGE [Clp_primal](#) (Clp_Simplex *model, int ifValuesPass)
Primal algorithm - see [ClpSimplexPrimal.hpp](#) for method.
- COINLIBAPI void COINLINKAGE [Clp_idiot](#) (Clp_Simplex *model, int tryhard)
Solve the problem with the idiot code.
- COINLIBAPI void COINLINKAGE [Clp_scaling](#) (Clp_Simplex *model, int mode)
Sets or unsets scaling, 0 -off, 1 equilibrium, 2 geometric, 3, auto, 4 dynamic(later)
- COINLIBAPI int COINLINKAGE [Clp_scalingFlag](#) (Clp_Simplex *model)
Gets scalingFlag.
- COINLIBAPI int COINLINKAGE [Clp_crash](#) (Clp_Simplex *model, 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.

most useful gets and sets

- COINLIBAPI int COINLINKAGE [Clp_primalFeasible](#) (Clp_Simplex *model)
If problem is primal feasible.
- COINLIBAPI int COINLINKAGE [Clp_dualFeasible](#) (Clp_Simplex *model)
If problem is dual feasible.
- COINLIBAPI double COINLINKAGE [Clp_dualBound](#) (Clp_Simplex *model)
Dual bound.
- COINLIBAPI void COINLINKAGE [Clp_setDualBound](#) (Clp_Simplex *model, double value)

- COINLIBAPI double COINLINKAGE [Clp_infeasibilityCost](#) (Clp_Simplex *model)
Infeasibility cost.
- COINLIBAPI void COINLINKAGE [Clp_setInfeasibilityCost](#) (Clp_Simplex *model, double value)
- COINLIBAPI int COINLINKAGE [Clp_perturbation](#) (Clp_Simplex *model)
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.
- COINLIBAPI void COINLINKAGE [Clp_setPerturbation](#) (Clp_Simplex *model, int value)
- COINLIBAPI int COINLINKAGE [Clp_algorithm](#) (Clp_Simplex *model)
Current (or last) algorithm.
- COINLIBAPI void COINLINKAGE [Clp_setAlgorithm](#) (Clp_Simplex *model, int value)
Set algorithm.
- COINLIBAPI double COINLINKAGE [Clp_sumDualInfeasibilities](#) (Clp_Simplex *model)
Sum of dual infeasibilities.
- COINLIBAPI int COINLINKAGE [Clp_numberDualInfeasibilities](#) (Clp_Simplex *model)
Number of dual infeasibilities.
- COINLIBAPI double COINLINKAGE [Clp_sumPrimalInfeasibilities](#) (Clp_Simplex *model)
Sum of primal infeasibilities.
- COINLIBAPI int COINLINKAGE [Clp_numberPrimalInfeasibilities](#) (Clp_Simplex *model)
Number of primal infeasibilities.
- COINLIBAPI int COINLINKAGE [Clp_saveModel](#) (Clp_Simplex *model, const char *fileName)
Save model to file, returns 0 if success.
- COINLIBAPI int COINLINKAGE [Clp_restoreModel](#) (Clp_Simplex *model, const char *fileName)
Restore model from file, returns 0 if success, deletes current model.
- COINLIBAPI void COINLINKAGE [Clp_checkSolution](#) (Clp_Simplex *model)
Just check solution (for external use) - sets sum of infeasibilities etc.

gets and sets - some synonyms

- COINLIBAPI int COINLINKAGE [Clp_getNumRows](#) (Clp_Simplex *model)
Number of rows.
- COINLIBAPI int COINLINKAGE [Clp_getNumCols](#) (Clp_Simplex *model)
Number of columns.
- COINLIBAPI int COINLINKAGE [Clp_getIterationCount](#) (Clp_Simplex *model)
Number of iterations.
- COINLIBAPI int COINLINKAGE [Clp_isAbandoned](#) (Clp_Simplex *model)
Are there a numerical difficulties?
- COINLIBAPI int COINLINKAGE [Clp_isProvenOptimal](#) (Clp_Simplex *model)
Is optimality proven?
- COINLIBAPI int COINLINKAGE [Clp_isProvenPrimalInfeasible](#) (Clp_Simplex *model)
Is primal infeasibility proven?
- COINLIBAPI int COINLINKAGE [Clp_isProvenDualInfeasible](#) (Clp_Simplex *model)
Is dual infeasibility proven?
- COINLIBAPI int COINLINKAGE [Clp_isPrimalObjectiveLimitReached](#) (Clp_Simplex *model)
Is the given primal objective limit reached?
- COINLIBAPI int COINLINKAGE [Clp_isDualObjectiveLimitReached](#) (Clp_Simplex *model)
Is the given dual objective limit reached?
- COINLIBAPI int COINLINKAGE [Clp_isIterationLimitReached](#) (Clp_Simplex *model)
Iteration limit reached?
- COINLIBAPI double COINLINKAGE [Clp_getObjSense](#) (Clp_Simplex *model)
Direction of optimization (1 - minimize, -1 - maximize, 0 - ignore.
- COINLIBAPI void COINLINKAGE [Clp_setObjSense](#) (Clp_Simplex *model, double objsen)
Direction of optimization (1 - minimize, -1 - maximize, 0 - ignore.
- COINLIBAPI const double
*COINLINKAGE [Clp_getRowActivity](#) (Clp_Simplex *model)
Primal row solution.

- COINLIBAPI const double
*COINLINKAGE [Clp_getColSolution](#) (Clp_Simplex *model)
Primal column solution.
- COINLIBAPI void COINLINKAGE [Clp_setColSolution](#) (Clp_Simplex *model, const double *input)
- COINLIBAPI const double
*COINLINKAGE [Clp_getRowPrice](#) (Clp_Simplex *model)
Dual row solution.
- COINLIBAPI const double
*COINLINKAGE [Clp_getReducedCost](#) (Clp_Simplex *model)
Reduced costs.
- COINLIBAPI const double
*COINLINKAGE [Clp_getRowLower](#) (Clp_Simplex *model)
Row lower.
- COINLIBAPI const double
*COINLINKAGE [Clp_getRowUpper](#) (Clp_Simplex *model)
Row upper.
- COINLIBAPI const double
*COINLINKAGE [Clp_getObjCoefficients](#) (Clp_Simplex *model)
Objective.
- COINLIBAPI const double
*COINLINKAGE [Clp_getColLower](#) (Clp_Simplex *model)
Column Lower.
- COINLIBAPI const double
*COINLINKAGE [Clp_getColUpper](#) (Clp_Simplex *model)
Column Upper.
- COINLIBAPI double COINLINKAGE [Clp_getObjValue](#) (Clp_Simplex *model)
Objective value.
- COINLIBAPI void COINLINKAGE [Clp_printModel](#) (Clp_Simplex *model, const char *prefix)
Print model for debugging purposes.
- COINLIBAPI double COINLINKAGE [Clp_getSmallElementValue](#) (Clp_Simplex *model)
- COINLIBAPI void COINLINKAGE [Clp_setSmallElementValue](#) (Clp_Simplex *model, double value)

Get and set ClpSolve options

- COINLIBAPI void COINLINKAGE [ClpSolve_setSpecialOption](#) (Clp_Solve *, int which, int value, int extraInfo)
- COINLIBAPI int COINLINKAGE [ClpSolve_getSpecialOption](#) (Clp_Solve *, int which)
- COINLIBAPI void COINLINKAGE [ClpSolve_setSolveType](#) (Clp_Solve *, int method, int extraInfo)
method: (see [ClpSolve::SolveType](#)) 0 - dual simplex 1 - primal simplex 2 - primal or sprint 3 - barrier 4 - barrier no crossover 5 - automatic 6 - not implemented – pass extraInfo == -1 for default behavior
- COINLIBAPI int COINLINKAGE [ClpSolve_getSolveType](#) (Clp_Solve *)
- COINLIBAPI void COINLINKAGE [ClpSolve_setPresolveType](#) (Clp_Solve *, int amount, int extraInfo)
amount: (see [ClpSolve::PresolveType](#)) 0 - presolve on 1 - presolve off 2 - presolve number 3 - presolve number cost – pass extraInfo == -1 for default behavior
- COINLIBAPI int COINLINKAGE [ClpSolve_getPresolveType](#) (Clp_Solve *)
- COINLIBAPI int COINLINKAGE [ClpSolve_getPresolvePasses](#) (Clp_Solve *)
- COINLIBAPI int COINLINKAGE [ClpSolve_getExtraInfo](#) (Clp_Solve *, int which)
- COINLIBAPI void COINLINKAGE [ClpSolve_setInfeasibleReturn](#) (Clp_Solve *, int trueFalse)
- COINLIBAPI int COINLINKAGE [ClpSolve_infeasibleReturn](#) (Clp_Solve *)
- COINLIBAPI int COINLINKAGE [ClpSolve_doDual](#) (Clp_Solve *)
- COINLIBAPI void COINLINKAGE [ClpSolve_setDoDual](#) (Clp_Solve *, int doDual)
- COINLIBAPI int COINLINKAGE [ClpSolve_doSingleton](#) (Clp_Solve *)
- COINLIBAPI void COINLINKAGE [ClpSolve_setDoSingleton](#) (Clp_Solve *, int doSingleton)
- COINLIBAPI int COINLINKAGE [ClpSolve_doDoubleton](#) (Clp_Solve *)
- COINLIBAPI void COINLINKAGE [ClpSolve_setDoDoubleton](#) (Clp_Solve *, int doDoubleton)
- COINLIBAPI int COINLINKAGE [ClpSolve_doTripleton](#) (Clp_Solve *)
- COINLIBAPI void COINLINKAGE [ClpSolve_setDoTripleton](#) (Clp_Solve *, int doTripleton)

- COINLIBAPI int COINLINKAGE [ClpSolve_doTighten](#) ([Clp_Solve](#) *)
- COINLIBAPI void COINLINKAGE [ClpSolve_setDoTighten](#) ([Clp_Solve](#) *, int doTighten)
- COINLIBAPI int COINLINKAGE [ClpSolve_doForcing](#) ([Clp_Solve](#) *)
- COINLIBAPI void COINLINKAGE [ClpSolve_setDoForcing](#) ([Clp_Solve](#) *, int doForcing)
- COINLIBAPI int COINLINKAGE [ClpSolve_doImpliedFree](#) ([Clp_Solve](#) *)
- COINLIBAPI void COINLINKAGE [ClpSolve_setDoImpliedFree](#) ([Clp_Solve](#) *, int doImpliedFree)
- COINLIBAPI int COINLINKAGE [ClpSolve_doDupcol](#) ([Clp_Solve](#) *)
- COINLIBAPI void COINLINKAGE [ClpSolve_setDoDupcol](#) ([Clp_Solve](#) *, int doDupcol)
- COINLIBAPI int COINLINKAGE [ClpSolve_doDuprow](#) ([Clp_Solve](#) *)
- COINLIBAPI void COINLINKAGE [ClpSolve_setDoDuprow](#) ([Clp_Solve](#) *, int doDuprow)
- COINLIBAPI int COINLINKAGE [ClpSolve_doSingletonColumn](#) ([Clp_Solve](#) *)
- COINLIBAPI void COINLINKAGE [ClpSolve_setDoSingletonColumn](#) ([Clp_Solve](#) *, int doSingleton)
- COINLIBAPI int COINLINKAGE [ClpSolve_presolveActions](#) ([Clp_Solve](#) *)
- COINLIBAPI void COINLINKAGE [ClpSolve_setPresolveActions](#) ([Clp_Solve](#) *, int action)
- COINLIBAPI int COINLINKAGE [ClpSolve_substitution](#) ([Clp_Solve](#) *)
- COINLIBAPI void COINLINKAGE [ClpSolve_setSubstitution](#) ([Clp_Solve](#) *, int value)

5.17.1 Typedef Documentation

5.17.1.1 typedef void [Clp_Solve](#)

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

5.17.2 Function Documentation

5.17.2.1 COINLIBAPI [Clp_Simplex*](#) COINLINKAGE [Clp_newModel](#) (void)

Default constructor.

5.17.2.2 COINLIBAPI void COINLINKAGE [Clp_deleteModel](#) ([Clp_Simplex](#) * *model*)

Destructor.

5.17.2.3 COINLIBAPI [Clp_Solve*](#) COINLINKAGE [ClpSolve_new](#) ()

Default constructor.

5.17.2.4 COINLIBAPI void COINLINKAGE [ClpSolve_delete](#) ([Clp_Solve](#) * *solve*)

Destructor.

5.17.2.5 COINLIBAPI void COINLINKAGE [Clp_loadProblem](#) ([Clp_Simplex](#) * *model*, 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*)

Loads a problem (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

Just like the other `loadProblem()` method except that the matrix is given in a standard column major ordered format (without gaps).

5.17.2.6 COINLIBAPI void COINLINKAGE `Clp_loadQuadraticObjective` (`Clp_Simplex * model`, const int *numberColumns*, const `CoinBigIndex * start`, const int * *column*, const double * *element*)

5.17.2.7 COINLIBAPI int COINLINKAGE `Clp_readMps` (`Clp_Simplex * model`, const char * *filename*, int *keepNames*, int *ignoreErrors*)

Read an mps file from the given filename.

5.17.2.8 COINLIBAPI void COINLINKAGE `Clp_copyInIntegerInformation` (`Clp_Simplex * model`, const char * *information*)

Copy in integer informations.

5.17.2.9 COINLIBAPI void COINLINKAGE `Clp_deleteIntegerInformation` (`Clp_Simplex * model`)

Drop integer informations.

5.17.2.10 COINLIBAPI void COINLINKAGE `Clp_resize` (`Clp_Simplex * model`, int *newNumberRows*, int *newNumberColumns*)

Resizes rim part of model.

5.17.2.11 COINLIBAPI void COINLINKAGE `Clp_deleteRows` (`Clp_Simplex * model`, int *number*, const int * *which*)

Deletes rows.

5.17.2.12 COINLIBAPI void COINLINKAGE `Clp_addRows` (`Clp_Simplex * model`, int *number*, const double * *rowLower*, const double * *rowUpper*, const int * *rowStarts*, const int * *columns*, const double * *elements*)

Add rows.

5.17.2.13 COINLIBAPI void COINLINKAGE `Clp_deleteColumns` (`Clp_Simplex * model`, int *number*, const int * *which*)

Deletes columns.

5.17.2.14 COINLIBAPI void COINLINKAGE `Clp_addColumns` (`Clp_Simplex * model`, int *number*, const double * *columnLower*, const double * *columnUpper*, const double * *objective*, const int * *columnStarts*, const int * *rows*, const double * *elements*)

Add columns.

5.17.2.15 COINLIBAPI void COINLINKAGE `Clp_chgRowLower` (`Clp_Simplex * model`, const double * *rowLower*)

Change row lower bounds.

5.17.2.16 COINLIBAPI void COINLINKAGE `Clp_chgRowUpper` (`Clp_Simplex * model`, const double * *rowUpper*)

Change row upper bounds.

5.17.2.17 COINLIBAPI void COINLINKAGE `Clp_chgColumnLower` (`Clp_Simplex * model`, const double * *columnLower*)

Change column lower bounds.

5.17.2.18 COINLIBAPI void COINLINKAGE Clp_chgColumnUpper (Clp_Simplex * *model*, const double * *columnUpper*)

Change column upper bounds.

5.17.2.19 COINLIBAPI void COINLINKAGE Clp_chgObjCoefficients (Clp_Simplex * *model*, const double * *objIn*)

Change objective coefficients.

5.17.2.20 COINLIBAPI void COINLINKAGE Clp_dropNames (Clp_Simplex * *model*)

Drops names - makes lengthnames 0 and names empty.

5.17.2.21 COINLIBAPI void COINLINKAGE Clp_copyNames (Clp_Simplex * *model*, const char *const * *rowNames*, const char *const * *columnNames*)

Copies in names.

5.17.2.22 COINLIBAPI int COINLINKAGE Clp_numberRows (Clp_Simplex * *model*)

Number of rows.

5.17.2.23 COINLIBAPI int COINLINKAGE Clp_numberColumns (Clp_Simplex * *model*)

Number of columns.

5.17.2.24 COINLIBAPI double COINLINKAGE Clp_primalTolerance (Clp_Simplex * *model*)

Primal tolerance to use.

5.17.2.25 COINLIBAPI void COINLINKAGE Clp_setPrimalTolerance (Clp_Simplex * *model*, double *value*)

5.17.2.26 COINLIBAPI double COINLINKAGE Clp_dualTolerance (Clp_Simplex * *model*)

Dual tolerance to use.

5.17.2.27 COINLIBAPI void COINLINKAGE Clp_setDualTolerance (Clp_Simplex * *model*, double *value*)

5.17.2.28 COINLIBAPI double COINLINKAGE Clp_dualObjectiveLimit (Clp_Simplex * *model*)

Dual objective limit.

5.17.2.29 COINLIBAPI void COINLINKAGE Clp_setDualObjectiveLimit (Clp_Simplex * *model*, double *value*)

5.17.2.30 COINLIBAPI double COINLINKAGE Clp_objectiveOffset (Clp_Simplex * *model*)

Objective offset.

5.17.2.31 COINLIBAPI void COINLINKAGE Clp_setObjectiveOffset (Clp_Simplex * *model*, double *value*)

5.17.2.32 COINLIBAPI void COINLINKAGE Clp_problemName (Clp_Simplex * *model*, int *maxNumberCharacters*, char * *array*)

Fills in array with problem name.

5.17.2.33 COINLIBAPI int COINLINKAGE Clp_setProblemName (Clp_Simplex * *model*, int *maxNumberCharacters*, char * *array*)

5.17.2.34 COINLIBAPI int COINLINKAGE Clp_numberIterations (Clp_Simplex * *model*)

Number of iterations.

5.17.2.35 COINLIBAPI void COINLINKAGE Clp_setNumberIterations (Clp_Simplex * *model*, int *numberIterations*)

5.17.2.36 COINLIBAPI int maximumIterations (Clp_Simplex * *model*)

Maximum number of iterations.

5.17.2.37 COINLIBAPI void COINLINKAGE Clp_setMaximumIterations (Clp_Simplex * *model*, int *value*)

5.17.2.38 COINLIBAPI double COINLINKAGE Clp_maximumSeconds (Clp_Simplex * *model*)

Maximum time in seconds (from when set called)

5.17.2.39 COINLIBAPI void COINLINKAGE Clp_setMaximumSeconds (Clp_Simplex * *model*, double *value*)

5.17.2.40 COINLIBAPI int COINLINKAGE Clp_hitMaximumIterations (Clp_Simplex * *model*)

Returns true if hit maximum iterations (or time)

5.17.2.41 COINLIBAPI int COINLINKAGE Clp_status (Clp_Simplex * *model*)

Status of problem: 0 - optimal 1 - primal infeasible 2 - dual infeasible 3 - stopped on iterations etc 4 - stopped due to errors.

5.17.2.42 COINLIBAPI void COINLINKAGE Clp_setProblemStatus (Clp_Simplex * *model*, int *problemStatus*)

Set problem status.

5.17.2.43 COINLIBAPI int COINLINKAGE Clp_secondaryStatus (Clp_Simplex * *model*)

Secondary status of problem - may get extended 0 - none 1 - primal infeasible because dual limit reached 2 - scaled problem optimal - unscaled has primal infeasibilities 3 - scaled problem optimal - unscaled has dual infeasibilities 4 - scaled problem optimal - unscaled has both dual and primal infeasibilities.

5.17.2.44 COINLIBAPI void COINLINKAGE Clp_setSecondaryStatus (Clp_Simplex * *model*, int *status*)

5.17.2.45 COINLIBAPI double COINLINKAGE Clp_optimizationDirection (Clp_Simplex * *model*)

Direction of optimization (1 - minimize, -1 - maximize, 0 - ignore.

5.17.2.46 COINLIBAPI void COINLINKAGE Clp_setOptimizationDirection (Clp_Simplex * *model*, double *value*)

5.17.2.47 COINLIBAPI double* COINLINKAGE Clp_primalRowSolution (Clp_Simplex * *model*)

Primal row solution.

5.17.2.48 COINLIBAPI double* COINLINKAGE Clp_primalColumnSolution (Clp_Simplex * *model*)

Primal column solution.

5.17.2.49 COINLIBAPI double* COINLINKAGE Clp_dualRowSolution (Clp_Simplex * *model*)

Dual row solution.

5.17.2.50 COINLIBAPI double* COINLINKAGE Clp_dualColumnSolution (Clp_Simplex * *model*)

Reduced costs.

5.17.2.51 COINLIBAPI double* COINLINKAGE Clp_rowLower (Clp_Simplex * *model*)

Row lower.

5.17.2.52 COINLIBAPI double* COINLINKAGE Clp_rowUpper (Clp_Simplex * *model*)

Row upper.

5.17.2.53 COINLIBAPI double* COINLINKAGE Clp_objective (Clp_Simplex * *model*)

Objective.

5.17.2.54 COINLIBAPI double* COINLINKAGE Clp_columnLower (Clp_Simplex * *model*)

Column Lower.

5.17.2.55 COINLIBAPI double* COINLINKAGE Clp_columnUpper (Clp_Simplex * *model*)

Column Upper.

5.17.2.56 COINLIBAPI int COINLINKAGE Clp_getNumElements (Clp_Simplex * *model*)

Number of elements in matrix.

5.17.2.57 COINLIBAPI const CoinBigIndex* COINLINKAGE Clp_getVectorStarts (Clp_Simplex * *model*)

5.17.2.58 COINLIBAPI const int* COINLINKAGE Clp_getIndices (Clp_Simplex * *model*)

5.17.2.59 COINLIBAPI const int* COINLINKAGE Clp_getVectorLengths (Clp_Simplex * *model*)

5.17.2.60 COINLIBAPI const double* COINLINKAGE Clp_getElements (Clp_Simplex * *model*)

5.17.2.61 COINLIBAPI double COINLINKAGE Clp_objectiveValue (Clp_Simplex * *model*)

Objective value.

5.17.2.62 COINLIBAPI char* COINLINKAGE Clp_integerInformation (Clp_Simplex * *model*)

Integer information.

5.17.2.63 COINLIBAPI double* COINLINKAGE Clp_infeasibilityRay (Clp_Simplex * *model*)

Infeasibility/unbounded ray (NULL returned if none/wrong) Up to user to use free() on these arrays.

5.17.2.64 COINLIBAPI double* COINLINKAGE Clp_unboundedRay (Clp_Simplex * *model*)

5.17.2.65 COINLIBAPI int COINLINKAGE Clp_statusExists (Clp_Simplex * *model*)

See if status array exists (partly for OsiClp)

5.17.2.66 COINLIBAPI unsigned char* COINLINKAGE Clp_statusArray (Clp_Simplex * *model*)

Return address of status array (char[numberRows+numberColumns])

5.17.2.67 COINLIBAPI void COINLINKAGE Clp_copyinStatus (Clp_Simplex * *model*, const unsigned char * *statusArray*)

Copy in status vector.

5.17.2.68 COINLIBAPI int COINLINKAGE Clp_getColumnStatus (Clp_Simplex * *model*, int *sequence*)

5.17.2.69 COINLIBAPI int COINLINKAGE Clp_getRowStatus (Clp_Simplex * *model*, int *sequence*)

5.17.2.70 COINLIBAPI void COINLINKAGE Clp_setColumnStatus (Clp_Simplex * *model*, int *sequence*, int *value*)

5.17.2.71 COINLIBAPI void COINLINKAGE Clp_setRowStatus (Clp_Simplex * *model*, int *sequence*, int *value*)

5.17.2.72 COINLIBAPI void COINLINKAGE Clp_setUserPointer (Clp_Simplex * *model*, void * *pointer*)

User pointer for whatever reason.

5.17.2.73 COINLIBAPI void* COINLINKAGE Clp_getUserPointer (Clp_Simplex * *model*)

5.17.2.74 COINLIBAPI void COINLINKAGE Clp_registerCallBack (Clp_Simplex * *model*, clp_callback *userCallBack*)

Pass in Callback function.

Message numbers up to 1000000 are Clp, Coin ones have 1000000 added

5.17.2.75 COINLIBAPI void COINLINKAGE Clp_clearCallBack (Clp_Simplex * *model*)

Unset Callback function.

5.17.2.76 COINLIBAPI void COINLINKAGE Clp_setLogLevel (Clp_Simplex * *model*, 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.

5.17.2.77 COINLIBAPI int COINLINKAGE Clp_logLevel (Clp_Simplex * *model*)

5.17.2.78 COINLIBAPI int COINLINKAGE Clp_lengthNames (Clp_Simplex * *model*)

length of names (0 means no names)

5.17.2.79 COINLIBAPI void COINLINKAGE Clp_rowName (Clp_Simplex * *model*, int *iRow*, char * *name*)

Fill in array (at least lengthNames+1 long) with a row name.

5.17.2.80 COINLIBAPI void COINLINKAGE Clp_columnName (Clp_Simplex * *model*, int *iColumn*, char * *name*)

Fill in array (at least lengthNames+1 long) with a column name.

5.17.2.81 COINLIBAPI int COINLINKAGE Clp_initialSolve (Clp_Simplex * *model*)

General solve algorithm which can do presolve.

See [ClpSolve.hpp](#) for options

5.17.2.82 COINLIBAPI int COINLINKAGE Clp_initialSolveWithOptions (Clp_Simplex * *model*, Clp_Solve *)

Pass solve options.

(Exception to direct analogue rule)

5.17.2.83 COINLIBAPI int COINLINKAGE Clp_initialDualSolve (Clp_Simplex * *model*)

Dual initial solve.

5.17.2.84 COINLIBAPI int COINLINKAGE Clp_initialPrimalSolve (Clp_Simplex * *model*)

Primal initial solve.

5.17.2.85 COINLIBAPI int COINLINKAGE Clp_initialBarrierSolve (Clp_Simplex * *model*)

Barrier initial solve.

5.17.2.86 COINLIBAPI int COINLINKAGE Clp_initialBarrierNoCrossSolve (Clp_Simplex * *model*)

Barrier initial solve, no crossover.

5.17.2.87 COINLIBAPI int COINLINKAGE Clp_dual (Clp_Simplex * *model*, int *ifValuesPass*)

Dual algorithm - see [ClpSimplexDual.hpp](#) for method.

5.17.2.88 COINLIBAPI int COINLINKAGE Clp_primal (Clp_Simplex * *model*, int *ifValuesPass*)

Primal algorithm - see [ClpSimplexPrimal.hpp](#) for method.

5.17.2.89 COINLIBAPI void COINLINKAGE Clp_idiot (Clp_Simplex * *model*, int *tryhard*)

Solve the problem with the idiot code.

5.17.2.90 COINLIBAPI void COINLINKAGE Clp_scaling (Clp_Simplex * *model*, int *mode*)

Sets or unsets scaling, 0 -off, 1 equilibrium, 2 geometric, 3, auto, 4 dynamic(later)

5.17.2.91 COINLIBAPI int COINLINKAGE Clp_scalingFlag (Clp_Simplex * *model*)

Gets scalingFlag.

5.17.2.92 COINLIBAPI int COINLINKAGE Clp_crash (Clp_Simplex * *model*, 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" is 0 No pivoting (so will just be choice of algorithm) 1 Simple pivoting e.g. gub 2 Mini iterations

5.17.2.93 COINLIBAPI int COINLINKAGE Clp_primalFeasible (Clp_Simplex * *model*)

If problem is primal feasible.

5.17.2.94 COINLIBAPI int COINLINKAGE Clp_dualFeasible (Clp_Simplex * *model*)

If problem is dual feasible.

5.17.2.95 COINLIBAPI double COINLINKAGE Clp_dualBound (Clp_Simplex * *model*)

Dual bound.

5.17.2.96 COINLIBAPI void COINLINKAGE Clp_setDualBound (Clp_Simplex * *model*, double *value*)

5.17.2.97 COINLIBAPI double COINLINKAGE Clp_infeasibilityCost (Clp_Simplex * *model*)

Infeasibility cost.

5.17.2.98 COINLIBAPI void COINLINKAGE Clp_setInfeasibilityCost (Clp_Simplex * *model*, double *value*)

5.17.2.99 COINLIBAPI int COINLINKAGE Clp_perturbation (Clp_Simplex * *model*)

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.

5.17.2.100 COINLIBAPI void COINLINKAGE Clp_setPerturbation (Clp_Simplex * *model*, int *value*)

5.17.2.101 COINLIBAPI int COINLINKAGE Clp_algorithm (Clp_Simplex * *model*)

Current (or last) algorithm.

5.17.2.102 COINLIBAPI void COINLINKAGE Clp_setAlgorithm (Clp_Simplex * *model*, int *value*)

Set algorithm.

5.17.2.103 COINLIBAPI double COINLINKAGE Clp_sumDualInfeasibilities (Clp_Simplex * *model*)

Sum of dual infeasibilities.

5.17.2.104 COINLIBAPI int COINLINKAGE Clp_numberDualInfeasibilities (Clp_Simplex * *model*)

Number of dual infeasibilities.

5.17.2.105 COINLIBAPI double COINLINKAGE Clp_sumPrimalInfeasibilities (Clp_Simplex * *model*)

Sum of primal infeasibilities.

5.17.2.106 COINLIBAPI int COINLINKAGE Clp_numberPrimalInfeasibilities (Clp_Simplex * *model*)

Number of primal infeasibilities.

5.17.2.107 COINLIBAPI int COINLINKAGE Clp_saveModel (Clp_Simplex * *model*, 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.

5.17.2.108 COINLIBAPI int COINLINKAGE Clp_restoreModel (Clp_Simplex * *model*, const char * *fileName*)

Restore model from file, returns 0 if success, deletes current model.

5.17.2.109 COINLIBAPI void COINLINKAGE Clp_checkSolution (Clp_Simplex * *model*)

Just check solution (for external use) - sets sum of infeasibilities etc.

5.17.2.110 COINLIBAPI int COINLINKAGE Clp_getNumRows (Clp_Simplex * *model*)

Number of rows.

5.17.2.111 COINLIBAPI int COINLINKAGE Clp_getNumCols (Clp_Simplex * *model*)

Number of columns.

5.17.2.112 COINLIBAPI int COINLINKAGE Clp_getIterationCount (Clp_Simplex * *model*)

Number of iterations.

5.17.2.113 COINLIBAPI int COINLINKAGE Clp_isAbandoned (Clp_Simplex * *model*)

Are there a numerical difficulties?

5.17.2.114 COINLIBAPI int COINLINKAGE Clp_isProvenOptimal (Clp_Simplex * *model*)

Is optimality proven?

5.17.2.115 COINLIBAPI int COINLINKAGE Clp_isProvenPrimalInfeasible (Clp_Simplex * *model*)

Is primal infeasibility proven?

5.17.2.116 COINLIBAPI int COINLINKAGE Clp_isProvenDualInfeasible (Clp_Simplex * *model*)

Is dual infeasibility proven?

5.17.2.117 COINLIBAPI int COINLINKAGE Clp_isPrimalObjectiveLimitReached (Clp_Simplex * *model*)

Is the given primal objective limit reached?

5.17.2.118 COINLIBAPI int COINLINKAGE Clp_isDualObjectiveLimitReached (Clp_Simplex * *model*)

Is the given dual objective limit reached?

5.17.2.119 COINLIBAPI int COINLINKAGE Clp_isIterationLimitReached (Clp_Simplex * *model*)

Iteration limit reached?

5.17.2.120 COINLIBAPI double COINLINKAGE Clp_getObjSense (Clp_Simplex * *model*)

Direction of optimization (1 - minimize, -1 - maximize, 0 - ignore).

5.17.2.121 COINLIBAPI void COINLINKAGE Clp_setObjSense (Clp_Simplex * *model*, double *objsen*)

Direction of optimization (1 - minimize, -1 - maximize, 0 - ignore).

5.17.2.122 COINLIBAPI const double* COINLINKAGE Clp_getRowActivity (Clp_Simplex * *model*)

Primal row solution.

5.17.2.123 COINLIBAPI const double* COINLINKAGE Clp_getColSolution (Clp_Simplex * *model*)

Primal column solution.

5.17.2.124 COINLIBAPI void COINLINKAGE Clp_setColSolution (Clp_Simplex * *model*, const double * *input*)

5.17.2.125 COINLIBAPI const double* COINLINKAGE Clp_getRowPrice (Clp_Simplex * *model*)

Dual row solution.

5.17.2.126 COINLIBAPI const double* COINLINKAGE Clp_getReducedCost (Clp_Simplex * *model*)

Reduced costs.

5.17.2.127 COINLIBAPI const double* COINLINKAGE Clp_getRowLower (Clp_Simplex * *model*)

Row lower.

5.17.2.128 COINLIBAPI const double* COINLINKAGE Clp_getRowUpper (Clp_Simplex * *model*)

Row upper.

5.17.2.129 COINLIBAPI const double* COINLINKAGE Clp_getObjCoefficients (Clp_Simplex * *model*)

Objective.

5.17.2.130 COINLIBAPI const double* COINLINKAGE Clp_getColLower (Clp_Simplex * *model*)

Column Lower.

5.17.2.131 COINLIBAPI const double* COINLINKAGE Clp_getColUpper (Clp_Simplex * *model*)

Column Upper.

5.17.2.132 COINLIBAPI double COINLINKAGE Clp_getObjValue (Clp_Simplex * *model*)

Objective value.

5.17.2.133 COINLIBAPI void COINLINKAGE Clp_printModel (Clp_Simplex * *model*, const char * *prefix*)

Print model for debugging purposes.

5.17.2.134 COINLIBAPI double COINLINKAGE Clp_getSmallElementValue (Clp_Simplex * *model*)

5.17.2.135 COINLIBAPI void COINLINKAGE Clp_setSmallElementValue (Clp_Simplex * *model*, double *value*)

5.17.2.136 COINLIBAPI void COINLINKAGE ClpSolve_setSpecialOption (Clp_Solve * , int *which*, int *value*, int *extraInfo*)

5.17.2.137 COINLIBAPI int COINLINKAGE ClpSolve_getSpecialOption (Clp_Solve * , int *which*)

5.17.2.138 COINLIBAPI void COINLINKAGE ClpSolve_setSolveType (Clp_Solve * , int *method*, int *extraInfo*)

method: (see [ClpSolve::SolveType](#)) 0 - dual simplex 1 - primal simplex 2 - primal or sprint 3 - barrier 4 - barrier no crossover 5 - automatic 6 - not implemented – pass *extraInfo* == -1 for default behavior

5.17.2.139 COINLIBAPI int COINLINKAGE ClpSolve_getSolveType (Clp_Solve *)

5.17.2.140 COINLIBAPI void COINLINKAGE ClpSolve_setPresolveType (Clp_Solve * , int *amount*, int *extraInfo*)

amount: (see [ClpSolve::PresolveType](#)) 0 - presolve on 1 - presolve off 2 - presolve number 3 - presolve number cost – pass *extraInfo* == -1 for default behavior

5.17.2.141 COINLIBAPI int COINLINKAGE ClpSolve_getPresolveType (Clp_Solve *)

5.17.2.142 COINLIBAPI int COINLINKAGE ClpSolve_getPresolvePasses (Clp_Solve *)

5.17.2.143 COINLIBAPI int COINLINKAGE ClpSolve_getExtraInfo (Clp_Solve * , int *which*)

5.17.2.144 COINLIBAPI void COINLINKAGE ClpSolve_setInfeasibleReturn (Clp_Solve * , int *trueFalse*)

5.17.2.145 COINLIBAPI int COINLINKAGE ClpSolve_infeasibleReturn (Clp_Solve *)

- 5.17.2.146 COINLIBAPI int COINLINKAGE ClpSolve_doDual (Clp_Solve *)
- 5.17.2.147 COINLIBAPI void COINLINKAGE ClpSolve_setDoDual (Clp_Solve * , int *doDual*)
- 5.17.2.148 COINLIBAPI int COINLINKAGE ClpSolve_doSingleton (Clp_Solve *)
- 5.17.2.149 COINLIBAPI void COINLINKAGE ClpSolve_setDoSingleton (Clp_Solve * , int *doSingleton*)
- 5.17.2.150 COINLIBAPI int COINLINKAGE ClpSolve_doDoubleton (Clp_Solve *)
- 5.17.2.151 COINLIBAPI void COINLINKAGE ClpSolve_setDoDoubleton (Clp_Solve * , int *doDoubleton*)
- 5.17.2.152 COINLIBAPI int COINLINKAGE ClpSolve_doTripleton (Clp_Solve *)
- 5.17.2.153 COINLIBAPI void COINLINKAGE ClpSolve_setDoTripleton (Clp_Solve * , int *doTripleton*)
- 5.17.2.154 COINLIBAPI int COINLINKAGE ClpSolve_doTighten (Clp_Solve *)
- 5.17.2.155 COINLIBAPI void COINLINKAGE ClpSolve_setDoTighten (Clp_Solve * , int *doTighten*)
- 5.17.2.156 COINLIBAPI int COINLINKAGE ClpSolve_doForcing (Clp_Solve *)
- 5.17.2.157 COINLIBAPI void COINLINKAGE ClpSolve_setDoForcing (Clp_Solve * , int *doForcing*)
- 5.17.2.158 COINLIBAPI int COINLINKAGE ClpSolve_dolpliedFree (Clp_Solve *)
- 5.17.2.159 COINLIBAPI void COINLINKAGE ClpSolve_setDolpliedFree (Clp_Solve * , int *dolpliedFree*)
- 5.17.2.160 COINLIBAPI int COINLINKAGE ClpSolve_doDupcol (Clp_Solve *)
- 5.17.2.161 COINLIBAPI void COINLINKAGE ClpSolve_setDoDupcol (Clp_Solve * , int *doDupcol*)
- 5.17.2.162 COINLIBAPI int COINLINKAGE ClpSolve_doDuprow (Clp_Solve *)
- 5.17.2.163 COINLIBAPI void COINLINKAGE ClpSolve_setDoDuprow (Clp_Solve * , int *doDuprow*)
- 5.17.2.164 COINLIBAPI int COINLINKAGE ClpSolve_doSingletonColumn (Clp_Solve *)
- 5.17.2.165 COINLIBAPI void COINLINKAGE ClpSolve_setDoSingletonColumn (Clp_Solve * , int *doSingleton*)
- 5.17.2.166 COINLIBAPI int COINLINKAGE ClpSolve_presolveActions (Clp_Solve *)
- 5.17.2.167 COINLIBAPI void COINLINKAGE ClpSolve_setPresolveActions (Clp_Solve * , int *action*)
- 5.17.2.168 COINLIBAPI int COINLINKAGE ClpSolve_substitution (Clp_Solve *)
- 5.17.2.169 COINLIBAPI void COINLINKAGE ClpSolve_setSubstitution (Clp_Solve * , int *value*)

5.18 src/ClpCholeskyBase.hpp File Reference

```
#include "CoinPragma.hpp"
#include "CoinTypes.hpp"
```

Classes

- class [ClpCholeskyBase](#)

Base class for Clp Cholesky factorization Will do better factorization.

Macros

- `#define CLP_LONG_CHOLESKY 0`
- `#define CHOL_SMALL_VALUE 1.0e-11`

Typedefs

- `typedef double longDouble`

5.18.1 Macro Definition Documentation

5.18.1.1 `#define CLP_LONG_CHOLESKY 0`

Definition at line 13 of file ClpCholeskyBase.hpp.

5.18.1.2 `#define CHOL_SMALL_VALUE 1.0e-11`

Definition at line 41 of file ClpCholeskyBase.hpp.

5.18.2 Typedef Documentation

5.18.2.1 `typedef double longDouble`

Definition at line 40 of file ClpCholeskyBase.hpp.

5.19 src/ClpCholeskyDense.hpp File Reference

```
#include "ClpCholeskyBase.hpp"
```

Classes

- class [ClpCholeskyDense](#)
- struct [ClpCholeskyDenseC](#)

Functions

- void [ClpCholeskySpawn](#) (void *)
- void [ClpCholeskyCfactor](#) ([ClpCholeskyDenseC](#) *thisStruct, [longDouble](#) *a, int n, int numberBlocks, [longDouble](#) *diagonal, [longDouble](#) *work, int *rowsDropped)
Non leaf recursive factor.
- void [ClpCholeskyCtriRec](#) ([ClpCholeskyDenseC](#) *thisStruct, [longDouble](#) *aTri, int nThis, [longDouble](#) *aUnder, [longDouble](#) *diagonal, [longDouble](#) *work, int nLeft, int iBlock, int jBlock, int numberBlocks)
Non leaf recursive triangle rectangle update.
- void [ClpCholeskyCrecTri](#) ([ClpCholeskyDenseC](#) *thisStruct, [longDouble](#) *aUnder, int nTri, int nDo, int iBlock, int jBlock, [longDouble](#) *aTri, [longDouble](#) *diagonal, [longDouble](#) *work, int numberBlocks)
Non leaf recursive rectangle triangle update.

- void `ClpCholeskyCrecRec` (`ClpCholeskyDenseC` *thisStruct, `longDouble` *above, int nUnder, int nUnderK, int nDo, `longDouble` *aUnder, `longDouble` *aOther, `longDouble` *work, int iBlock, int jBlock, int numberBlocks)

Non leaf recursive rectangle rectangle update, nUnder is number of rows in iBlock, nUnderK is number of rows in kBlock.

- void `ClpCholeskyCfactorLeaf` (`ClpCholeskyDenseC` *thisStruct, `longDouble` *a, int n, `longDouble` *diagonal, `longDouble` *work, int *rowsDropped)

Leaf recursive factor.

- void `ClpCholeskyCtrirecLeaf` (`longDouble` *aTri, `longDouble` *aUnder, `longDouble` *diagonal, `longDouble` *work, int nUnder)

Leaf recursive triangle rectangle update.

- void `ClpCholeskyCrecTriLeaf` (`longDouble` *aUnder, `longDouble` *aTri, `longDouble` *work, int nUnder)

Leaf recursive rectangle triangle update.

- void `ClpCholeskyCrecRecLeaf` (const `longDouble` *COIN_RESTRICT above, const `longDouble` *COIN_RESTRICT aUnder, `longDouble` *COIN_RESTRICT aOther, const `longDouble` *COIN_RESTRICT work, int nUnder)

Leaf recursive rectangle rectangle update, nUnder is number of rows in iBlock, nUnderK is number of rows in kBlock.

5.19.1 Function Documentation

5.19.1.1 void `ClpCholeskySpawn` (void *)

5.19.1.2 void `ClpCholeskyCfactor` (`ClpCholeskyDenseC` * thisStruct, `longDouble` * a, int n, int numberBlocks, `longDouble` * diagonal, `longDouble` * work, int * rowsDropped)

Non leaf recursive factor.

5.19.1.3 void `ClpCholeskyCtrirec` (`ClpCholeskyDenseC` * thisStruct, `longDouble` * aTri, int nThis, `longDouble` * aUnder, `longDouble` * diagonal, `longDouble` * work, int nLeft, int iBlock, int jBlock, int numberBlocks)

Non leaf recursive triangle rectangle update.

5.19.1.4 void `ClpCholeskyCrecTri` (`ClpCholeskyDenseC` * thisStruct, `longDouble` * aUnder, int nTri, int nDo, int iBlock, int jBlock, `longDouble` * aTri, `longDouble` * diagonal, `longDouble` * work, int numberBlocks)

Non leaf recursive rectangle triangle update.

5.19.1.5 void `ClpCholeskyCrecRec` (`ClpCholeskyDenseC` * thisStruct, `longDouble` * above, int nUnder, int nUnderK, int nDo, `longDouble` * aUnder, `longDouble` * aOther, `longDouble` * work, int iBlock, int jBlock, int numberBlocks)

Non leaf recursive rectangle rectangle update, nUnder is number of rows in iBlock, nUnderK is number of rows in kBlock.

5.19.1.6 void `ClpCholeskyCfactorLeaf` (`ClpCholeskyDenseC` * thisStruct, `longDouble` * a, int n, `longDouble` * diagonal, `longDouble` * work, int * rowsDropped)

Leaf recursive factor.

5.19.1.7 void `ClpCholeskyCtrirecLeaf` (`longDouble` * aTri, `longDouble` * aUnder, `longDouble` * diagonal, `longDouble` * work, int nUnder)

Leaf recursive triangle rectangle update.

5.19.1.8 void `ClpCholeskyCrecTriLeaf` (`longDouble` * aUnder, `longDouble` * aTri, `longDouble` * work, int nUnder)

Leaf recursive rectangle triangle update.

5.19.1.9 void ClpCholeskyCrecRecLeaf (const longDouble *COIN_RESTRICT *above*, const longDouble *COIN_RESTRICT *aUnder*, longDouble *COIN_RESTRICT *aOther*, const longDouble *COIN_RESTRICT *work*, int *nUnder*)

Leaf recursive rectangle rectangle update, *nUnder* is number of rows in *iBlock*, *nUnderK* is number of rows in *kBlock*.

5.20 src/ClpCholeskyMumps.hpp File Reference

```
#include "ClpCholeskyBase.hpp"
```

Classes

- class [ClpCholeskyMumps](#)
Mumps class for Clp Cholesky factorization.

Typedefs

- typedef void [DMUMPS_STRUC_C](#)

5.20.1 Typedef Documentation

5.20.1.1 typedef void DMUMPS_STRUC_C

Definition at line 10 of file ClpCholeskyMumps.hpp.

5.21 src/ClpCholeskyTaucs.hpp File Reference

```
#include "taucs.h"  
#include "ClpCholeskyBase.hpp"
```

Classes

- class [ClpCholeskyTaucs](#)
Taucs class for Clp Cholesky factorization.

5.22 src/ClpCholeskyUfl.hpp File Reference

```
#include "ClpCholeskyBase.hpp"
```

Classes

- class [ClpCholeskyUfl](#)
Ufl class for Clp Cholesky factorization.

Typedefs

- typedef struct
cholmod_factor_struct [cholmod_factor](#)
- typedef struct
cholmod_common_struct [cholmod_common](#)

5.22.1 Typedef Documentation

5.22.1.1 typedef struct cholmod_factor_struct [cholmod_factor](#)

Definition at line 14 of file ClpCholeskyUfl.hpp.

5.22.1.2 typedef struct cholmod_common_struct [cholmod_common](#)

Definition at line 15 of file ClpCholeskyUfl.hpp.

5.23 src/ClpCholeskyWssmp.hpp File Reference

```
#include "ClpCholeskyBase.hpp"
```

Classes

- class [ClpCholeskyWssmp](#)
Wssmp class for Clp Cholesky factorization.

5.24 src/ClpCholeskyWssmpKKT.hpp File Reference

```
#include "ClpCholeskyBase.hpp"
```

Classes

- class [ClpCholeskyWssmpKKT](#)
WssmpKKT class for Clp Cholesky factorization.

5.25 src/ClpConfig.h File Reference

```
#include "config_clp_default.h"
```

5.26 src/ClpConstraint.hpp File Reference

Classes

- class [ClpConstraint](#)
Constraint Abstract Base Class.

5.27 src/ClpConstraintLinear.hpp File Reference

```
#include "ClpConstraint.hpp"
```

Classes

- class [ClpConstraintLinear](#)
Linear Constraint Class.

5.28 src/ClpConstraintQuadratic.hpp File Reference

```
#include "ClpConstraint.hpp"
```

Classes

- class [ClpConstraintQuadratic](#)
Quadratic Constraint Class.

5.29 src/ClpDualRowDantzig.hpp File Reference

```
#include "ClpDualRowPivot.hpp"
```

Classes

- class [ClpDualRowDantzig](#)
Dual Row Pivot Dantzig Algorithm Class.

5.30 src/ClpDualRowPivot.hpp File Reference

Classes

- class [ClpDualRowPivot](#)
Dual Row Pivot Abstract Base Class.

5.31 src/ClpDualRowSteepest.hpp File Reference

```
#include "ClpDualRowPivot.hpp"
```

Classes

- class [ClpDualRowSteepest](#)
Dual Row Pivot Steepest Edge Algorithm Class.

5.32 src/ClpDummyMatrix.hpp File Reference

```
#include "CoinPragma.hpp"
#include "ClpMatrixBase.hpp"
```

Classes

- class [ClpDummyMatrix](#)

This implements a dummy matrix as derived from [ClpMatrixBase](#).

5.33 src/ClpDynamicExampleMatrix.hpp File Reference

```
#include "CoinPragma.hpp"
#include "ClpDynamicMatrix.hpp"
```

Classes

- class [ClpDynamicExampleMatrix](#)

This implements a dynamic matrix when we have a limit on the number of "interesting rows".

5.34 src/ClpDynamicMatrix.hpp File Reference

```
#include "CoinPragma.hpp"
#include "ClpPackedMatrix.hpp"
```

Classes

- class [ClpDynamicMatrix](#)

This implements a dynamic matrix when we have a limit on the number of "interesting rows".

5.35 src/ClpEventHandler.hpp File Reference

```
#include "ClpSimplex.hpp"
```

Classes

- class [ClpEventHandler](#)

Base class for Clp event handling.

- class [ClpDisasterHandler](#)

Base class for Clp disaster handling.

5.36 src/ClpFactorization.hpp File Reference

```
#include "CoinPragma.hpp"
#include "CoinFactorization.hpp"
#include "CoinDenseFactorization.hpp"
#include "ClpSimplex.hpp"
```

Classes

- class [ClpFactorization](#)
This just implements CoinFactorization when an [ClpMatrixBase](#) object is passed.

Macros

- #define [CLP_MULTIPLE_FACTORIZATIONS](#) 4
- #define [COIN_FAST_CODE](#)

5.36.1 Macro Definition Documentation

5.36.1.1 #define CLP_MULTIPLE_FACTORIZATIONS 4

Definition at line 18 of file ClpFactorization.hpp.

5.36.1.2 #define COIN_FAST_CODE

Definition at line 25 of file ClpFactorization.hpp.

5.37 src/ClpGubDynamicMatrix.hpp File Reference

```
#include "CoinPragma.hpp"
#include "ClpGubMatrix.hpp"
```

Classes

- class [ClpGubDynamicMatrix](#)
This implements Gub rows plus a [ClpPackedMatrix](#).

5.38 src/ClpGubMatrix.hpp File Reference

```
#include "CoinPragma.hpp"
#include "ClpPackedMatrix.hpp"
```

Classes

- class [ClpGubMatrix](#)
This implements Gub rows plus a [ClpPackedMatrix](#).

5.39 src/ClpHelperFunctions.hpp File Reference

```
#include "ClpConfig.h"
```

Macros

- #define [ClpTraceDebug](#)(expression)

Functions

- double [maximumAbsElement](#) (const double *region, int size)
Note (JJF) I have added some operations on arrays even though they may duplicate CoinDenseVector.
- void [setElements](#) (double *region, int size, double value)
- void [multiplyAdd](#) (const double *region1, int size, double multiplier1, double *region2, double multiplier2)
- double [innerProduct](#) (const double *region1, int size, const double *region2)
- void [getNorms](#) (const double *region, int size, double &norm1, double &norm2)
- double [CoinSqrt](#) (double x)
- void [ClpTracePrint](#) (std::string fileName, std::string message, int line)
Trace.

5.39.1 Macro Definition Documentation

5.39.1.1 #define ClpTraceDebug(expression)

Value:

```
{ \
    if (!(expression)) { ClpTracePrint(__FILE__, __STRING(expression), __LINE__); } \
}
```

Definition at line 89 of file ClpHelperFunctions.hpp.

5.39.2 Function Documentation

5.39.2.1 double maximumAbsElement (const double * region, int size)

Note (JJF) I have added some operations on arrays even though they may duplicate CoinDenseVector.

I think the use of templates was a mistake as I don't think inline generic code can take as much advantage of parallelism or machine architectures or memory hierarchies.

5.39.2.2 void setElements (double * region, int size, double value)

5.39.2.3 void multiplyAdd (const double * region1, int size, double multiplier1, double * region2, double multiplier2)

5.39.2.4 double innerProduct (const double * region1, int size, const double * region2)

5.39.2.5 void getNorms (const double * region, int size, double & norm1, double & norm2)

5.39.2.6 double CoinSqrt (double x) [inline]

Definition at line 79 of file ClpHelperFunctions.hpp.

5.39.2.7 void ClpTracePrint (std::string fileName, std::string message, int line)

Trace.

5.40 src/ClpInterior.hpp File Reference

```
#include <iostream>
#include <cfloat>
#include "ClpModel.hpp"
#include "ClpMatrixBase.hpp"
#include "ClpSolve.hpp"
#include "CoinDenseVector.hpp"
```

Classes

- struct [Info](#)
 - ***** DATA to be moved into protected section of [ClpInterior](#)
- struct [Outfo](#)
 - ***** DATA to be moved into protected section of [ClpInterior](#)
- struct [Options](#)
 - ***** DATA to be moved into protected section of [ClpInterior](#)
- class [ClpInterior](#)
 - This solves LPs using interior point methods.*

Macros

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

- #define [LENGTH_HISTORY](#) 5
 - historyInfeasibility.*

Functions

- void [ClpInteriorUnitTest](#) (const std::string &mpsDir, const std::string &netlibDir)
 - A function that tests the methods in the [ClpInterior](#) class.*

5.40.1 Macro Definition Documentation

5.40.1.1 #define LENGTH_HISTORY 5

historyInfeasibility.

Definition at line 485 of file ClpInterior.hpp.

5.40.2 Function Documentation

5.40.2.1 void ClpInteriorUnitTest (const std::string & mpsDir, const std::string & netlibDir)

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

5.41 src/ClpLinearObjective.hpp File Reference

```
#include "ClpObjective.hpp"
```

Classes

- class [ClpLinearObjective](#)
Linear Objective Class.

5.42 src/ClpLsqr.hpp File Reference

```
#include "CoinDenseVector.hpp"  
#include "ClpInterior.hpp"
```

Classes

- class [ClpLsqr](#)
This class implements LSQR.

5.43 src/ClpMatrixBase.hpp File Reference

```
#include "CoinPragma.hpp"  
#include "CoinTypes.hpp"  
#include "CoinPackedMatrix.hpp"
```

Classes

- class [ClpMatrixBase](#)
Abstract base class for Clp Matrices.

Macros

- #define [FREE_BIAS](#) 1.0e1
- #define [FREE_ACCEPT](#) 1.0e2

5.43.1 Macro Definition Documentation

5.43.1.1 #define FREE_BIAS 1.0e1

Definition at line 512 of file ClpMatrixBase.hpp.

5.43.1.2 #define FREE_ACCEPT 1.0e2

Definition at line 514 of file ClpMatrixBase.hpp.

5.44 src/ClpMessage.hpp File Reference

```
#include "CoinPragma.hpp"
#include <cstring>
#include "CoinMessageHandler.hpp"
```

Classes

- class [ClpMessage](#)
This deals with Clp messages (as against Osi messages etc)

Enumerations

- enum [CLP_Message](#) {
[CLP_SIMPLEX_FINISHED](#), [CLP_SIMPLEX_INFEASIBLE](#), [CLP_SIMPLEX_UNBOUNDED](#), [CLP_SIMPLEX_STOPPED](#),
[CLP_SIMPLEX_ERROR](#), [CLP_SIMPLEX_INTERRUPT](#), [CLP_SIMPLEX_STATUS](#), [CLP_DUAL_BOUNDS](#),
[CLP_SIMPLEX_ACCURACY](#), [CLP_SIMPLEX_BADFACTOR](#), [CLP_SIMPLEX_BOUNDTIGHTEN](#), [CLP_SIMPLEX_INFEASIBILITIES](#),
[CLP_SIMPLEX_FLAG](#), [CLP_SIMPLEX_GIVINGUP](#), [CLP_DUAL_CHECKB](#), [CLP_DUAL_ORIGINAL](#),
[CLP_SIMPLEX_PERTURB](#), [CLP_PRIMAL_ORIGINAL](#), [CLP_PRIMAL_WEIGHT](#), [CLP_PRIMAL_OPTIMAL](#),
[CLP_SINGULARITIES](#), [CLP_MODIFIEDBOUNDS](#), [CLP_RIMSTATISTICS1](#), [CLP_RIMSTATISTICS2](#),
[CLP_RIMSTATISTICS3](#), [CLP_POSSIBLELOOP](#), [CLP_SMALLELEMENTS](#), [CLP_DUPLICATEELEMENTS](#),
[CLP_SIMPLEX_HOUSE1](#), [CLP_SIMPLEX_HOUSE2](#), [CLP_SIMPLEX_NONLINEAR](#), [CLP_SIMPLEX_FREEIN](#),
[CLP_SIMPLEX_PIVOTROW](#), [CLP_DUAL_CHECK](#), [CLP_PRIMAL_DJ](#), [CLP_PACKEDSCALE_INITIAL](#),
[CLP_PACKEDSCALE_WHILE](#), [CLP_PACKEDSCALE_FINAL](#), [CLP_PACKEDSCALE_FORGET](#), [CLP_INITIALIZE_STEEP](#),
[CLP_UNABLE_OPEN](#), [CLP_BAD_BOUNDS](#), [CLP_BAD_MATRIX](#), [CLP_LOOP](#),
[CLP_IMPORT_RESULT](#), [CLP_IMPORT_ERRORS](#), [CLP_EMPTY_PROBLEM](#), [CLP_CRASH](#),
[CLP_END_VALUES_PASS](#), [CLP_QUADRATIC_BOTH](#), [CLP_QUADRATIC_PRIMAL_DETAILS](#), [CLP_IDIOT_ITERATION](#),
[CLP_INFEASIBLE](#), [CLP_MATRIX_CHANGE](#), [CLP_TIMING](#), [CLP_INTERVAL_TIMING](#),
[CLP_SPRINT](#), [CLP_BARRIER_ITERATION](#), [CLP_BARRIER_OBJECTIVE_GAP](#), [CLP_BARRIER_GONE_INFEASIBLE](#),
[CLP_BARRIER_CLOSE_TO_OPTIMAL](#), [CLP_BARRIER_COMPLEMENTARITY](#), [CLP_BARRIER_EXIT2](#), [CLP_BARRIER_STOPPING](#),
[CLP_BARRIER_EXIT](#), [CLP_BARRIER_SCALING](#), [CLP_BARRIER_MU](#), [CLP_BARRIER_INFO](#),
[CLP_BARRIER_END](#), [CLP_BARRIER_ACCURACY](#), [CLP_BARRIER_SAFE](#), [CLP_BARRIER_NEGATIVE_GAPS](#),
[CLP_BARRIER_REDUCING](#), [CLP_BARRIER_DIAGONAL](#), [CLP_BARRIER_SLACKS](#), [CLP_BARRIER_DUALI-](#)

```

NF,
CLP_BARRIER_KILLED, CLP_BARRIER_ABS_DROPPED, CLP_BARRIER_ABS_ERROR, CLP_BARRIER_F-
EASIBLE,
CLP_BARRIER_STEP, CLP_BARRIER_KKT, CLP_RIM_SCALE, CLP_SLP_ITER,
CLP_COMPLICATED_MODEL, CLP_BAD_STRING_VALUES, CLP_CRUNCH_STATS, CLP_PARAMETRICS-
_STATS,
CLP_PARAMETRICS_STATS2, CLP_FATHOM_STATUS, CLP_FATHOM_SOLUTION, CLP_FATHOM_FINIS-
H,
CLP_GENERAL, CLP_GENERAL2, CLP_GENERAL_WARNING, CLP_DUMMY_END }

```

5.44.1 Enumeration Type Documentation

5.44.1.1 enum CLP_Message

Enumerator

```

CLP_SIMPLEX_FINISHED
CLP_SIMPLEX_INFEASIBLE
CLP_SIMPLEX_UNBOUNDED
CLP_SIMPLEX_STOPPED
CLP_SIMPLEX_ERROR
CLP_SIMPLEX_INTERRUPT
CLP_SIMPLEX_STATUS
CLP_DUAL_BOUNDS
CLP_SIMPLEX_ACCURACY
CLP_SIMPLEX_BADFACTOR
CLP_SIMPLEX_BOUNDTIGHTEN
CLP_SIMPLEX_INFEASIBILITIES
CLP_SIMPLEX_FLAG
CLP_SIMPLEX_GIVINGUP
CLP_DUAL_CHECKB
CLP_DUAL_ORIGINAL
CLP_SIMPLEX_PERTURB
CLP_PRIMAL_ORIGINAL
CLP_PRIMAL_WEIGHT
CLP_PRIMAL_OPTIMAL
CLP_SINGULARITIES
CLP_MODIFIEDBOUNDS
CLP_RIMSTATISTICS1
CLP_RIMSTATISTICS2
CLP_RIMSTATISTICS3
CLP_POSSIBLELOOP
CLP_SMALLELEMENTS
CLP_DUPLICATEELEMENTS
CLP_SIMPLEX_HOUSE1

```

CLP_SIMPLEX_HOUSE2
CLP_SIMPLEX_NONLINEAR
CLP_SIMPLEX_FREEIN
CLP_SIMPLEX_PIVOTROW
CLP_DUAL_CHECK
CLP_PRIMAL_DJ
CLP_PACKEDSCALE_INITIAL
CLP_PACKEDSCALE_WHILE
CLP_PACKEDSCALE_FINAL
CLP_PACKEDSCALE_FORGET
CLP_INITIALIZE_STEEP
CLP_UNABLE_OPEN
CLP_BAD_BOUNDS
CLP_BAD_MATRIX
CLP_LOOP
CLP_IMPORT_RESULT
CLP_IMPORT_ERRORS
CLP_EMPTY_PROBLEM
CLP_CRASH
CLP_END_VALUES_PASS
CLP_QUADRATIC_BOTH
CLP_QUADRATIC_PRIMAL_DETAILS
CLP_IDIOT_ITERATION
CLP_INFEASIBLE
CLP_MATRIX_CHANGE
CLP_TIMING
CLP_INTERVAL_TIMING
CLP_SPRINT
CLP_BARRIER_ITERATION
CLP_BARRIER_OBJECTIVE_GAP
CLP_BARRIER_GONE_INFEASIBLE
CLP_BARRIER_CLOSE_TO_OPTIMAL
CLP_BARRIER_COMPLEMENTARITY
CLP_BARRIER_EXIT2
CLP_BARRIER_STOPPING
CLP_BARRIER_EXIT
CLP_BARRIER_SCALING
CLP_BARRIER_MU
CLP_BARRIER_INFO
CLP_BARRIER_END
CLP_BARRIER_ACCURACY
CLP_BARRIER_SAFE

CLP_BARRIER_NEGATIVE_GAPS
CLP_BARRIER_REDUCING
CLP_BARRIER_DIAGONAL
CLP_BARRIER_SLACKS
CLP_BARRIER_DUALINF
CLP_BARRIER_KILLED
CLP_BARRIER_ABS_DROPPED
CLP_BARRIER_ABS_ERROR
CLP_BARRIER_FEASIBLE
CLP_BARRIER_STEP
CLP_BARRIER_KKT
CLP_RIM_SCALE
CLP_SLP_ITER
CLP_COMPLICATED_MODEL
CLP_BAD_STRING_VALUES
CLP_CRUNCH_STATS
CLP_PARAMETRICS_STATS
CLP_PARAMETRICS_STATS2
CLP_FATHOM_STATUS
CLP_FATHOM_SOLUTION
CLP_FATHOM_FINISH
CLP_GENERAL
CLP_GENERAL2
CLP_GENERAL_WARNING
CLP_DUMMY_END

Definition at line 16 of file ClpMessage.hpp.

5.45 src/ClpModel.hpp File Reference

```
#include "ClpConfig.h"  
#include <iostream>  
#include <cassert>  
#include <cmath>  
#include <vector>  
#include <string>  
#include "ClpPackedMatrix.hpp"  
#include "CoinMessageHandler.hpp"  
#include "CoinHelperFunctions.hpp"  
#include "CoinTypes.hpp"  
#include "CoinFinite.hpp"  
#include "ClpParameters.hpp"  
#include "ClpObjective.hpp"
```


Classes

- class [ClpModel](#)
- class [ClpDataSave](#)

This is a tiny class where data can be saved round calls.

Macros

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

- `#define COIN_CBC_USING_CLP 0x01000000`

data

- `#define ROW_COLUMN_COUNTS_SAME 1`
Whats changed since last solve.
- `#define MATRIX_SAME 2`
- `#define MATRIX_JUST_ROWS_ADDED 4`
- `#define MATRIX_JUST_COLUMNS_ADDED 8`
- `#define ROW_LOWER_SAME 16`
- `#define ROW_UPPER_SAME 32`
- `#define OBJECTIVE_SAME 64`
- `#define COLUMN_LOWER_SAME 128`
- `#define COLUMN_UPPER_SAME 256`
- `#define BASIS_SAME 512`
- `#define ALL_SAME 65339`
- `#define ALL_SAME_EXCEPT_COLUMN_BOUNDS 65337`

5.45.1 Macro Definition Documentation

5.45.1.1 `#define COIN_CBC_USING_CLP 0x01000000`

Definition at line 1056 of file ClpModel.hpp.

5.45.1.2 `#define ROW_COLUMN_COUNTS_SAME 1`

Whats changed since last solve.

This is a work in progress It is designed so careful people can make go faster. It is only used when startFinishOptions used in dual or primal. Bit 1 - number of rows/columns has not changed (so work arrays valid) 2 - matrix has not changed 4 - if matrix has changed only by adding rows 8 - if matrix has changed only by adding columns 16 - row lbs not changed 32 - row ubs not changed 64 - column objective not changed 128 - column lbs not changed 256 - column ubs not changed 512 - basis not changed (up to user to set this to 0) top bits may be used internally shift by 65336 is 3 all same, 1 all except col bounds

Definition at line 1200 of file ClpModel.hpp.

5.45.1.3 `#define MATRIX_SAME 2`

Definition at line 1201 of file ClpModel.hpp.

5.45.1.4 `#define MATRIX_JUST_ROWS_ADDED 4`

Definition at line 1202 of file ClpModel.hpp.

5.45.1.5 `#define MATRIX_JUST_COLUMNS_ADDED 8`

Definition at line 1203 of file ClpModel.hpp.

5.45.1.6 `#define ROW_LOWER_SAME 16`

Definition at line 1204 of file ClpModel.hpp.

5.45.1.7 `#define ROW_UPPER_SAME 32`

Definition at line 1205 of file ClpModel.hpp.

5.45.1.8 `#define OBJECTIVE_SAME 64`

Definition at line 1206 of file ClpModel.hpp.

5.45.1.9 `#define COLUMN_LOWER_SAME 128`

Definition at line 1207 of file ClpModel.hpp.

5.45.1.10 `#define COLUMN_UPPER_SAME 256`

Definition at line 1208 of file ClpModel.hpp.

5.45.1.11 `#define BASIS_SAME 512`

Definition at line 1209 of file ClpModel.hpp.

5.45.1.12 `#define ALL_SAME 65339`

Definition at line 1210 of file ClpModel.hpp.

5.45.1.13 `#define ALL_SAME_EXCEPT_COLUMN_BOUNDS 65337`

Definition at line 1211 of file ClpModel.hpp.

5.46 `src/ClpNetworkBasis.hpp` File Reference

```
#include "CoinTypes.hpp"
```

Classes

- class [ClpNetworkBasis](#)

This deals with Factorization and Updates for network structures.

Macros

- `#define` [COIN_FAST_CODE](#)

5.46.1 Macro Definition Documentation

5.46.1.1 `#define` COIN_FAST_CODE

Definition at line 19 of file ClpNetworkBasis.hpp.

5.47 src/ClpNetworkMatrix.hpp File Reference

```
#include "CoinPragma.hpp"
#include "ClpMatrixBase.hpp"
```

Classes

- class [ClpNetworkMatrix](#)

This implements a simple network matrix as derived from [ClpMatrixBase](#).

5.48 src/ClpNode.hpp File Reference

```
#include "CoinPragma.hpp"
```

Classes

- class [ClpNode](#)
- struct [ClpNode::branchState](#)
- class [ClpNodeStuff](#)
- class [ClpHashValue](#)
- struct [ClpHashValue::CoinHashLink](#)

Data.

5.49 src/ClpNonLinearCost.hpp File Reference

```
#include "CoinPragma.hpp"
```

Classes

- class [ClpNonLinearCost](#)

Macros

- `#define CLP_BELOW_LOWER 0`
Trivial class to deal with non linear costs.
- `#define CLP_FEASIBLE 1`
- `#define CLP_ABOVE_UPPER 2`
- `#define CLP_SAME 4`
- `#define CLP_METHOD1 ((method_&1)!=0)`
- `#define CLP_METHOD2 ((method_&2)!=0)`

Functions

- `int originalStatus` (unsigned char status)
- `int currentStatus` (unsigned char status)
- `void setOriginalStatus` (unsigned char &status, int value)
- `void setCurrentStatus` (unsigned char &status, int value)
- `void setInitialStatus` (unsigned char &status)
- `void setSameStatus` (unsigned char &status)

5.49.1 Macro Definition Documentation

5.49.1.1 `#define CLP_BELOW_LOWER 0`

Trivial class to deal with non linear costs.

I don't make any explicit assumptions about convexity but I am sure I do make implicit ones.

One interesting idea for normal LP's will be to allow non-basic variables to come into basis as infeasible i.e. if variable at lower bound has very large positive reduced cost (when problem is infeasible) could it reduce overall problem infeasibility more by bringing it into basis below its lower bound.

Another feature would be to automatically discover when problems are convex piecewise linear and re-formulate to use non-linear. I did some work on this many years ago on "grade" problems, but while it improved primal interior point algorithms were much better for that particular problem.

Definition at line 38 of file ClpNonLinearCost.hpp.

5.49.1.2 `#define CLP_FEASIBLE 1`

Definition at line 39 of file ClpNonLinearCost.hpp.

5.49.1.3 `#define CLP_ABOVE_UPPER 2`

Definition at line 40 of file ClpNonLinearCost.hpp.

5.49.1.4 `#define CLP_SAME 4`

Definition at line 41 of file ClpNonLinearCost.hpp.

5.49.1.5 `#define CLP_METHOD1 ((method_&1)!=0)`

Definition at line 72 of file ClpNonLinearCost.hpp.

5.49.1.6 `#define CLP_METHOD2 ((method_&2)!=0)`

Definition at line 73 of file ClpNonLinearCost.hpp.

5.49.2 Function Documentation

5.49.2.1 `int originalStatus (unsigned char status) [inline]`

Definition at line 42 of file ClpNonLinearCost.hpp.

5.49.2.2 `int currentStatus (unsigned char status) [inline]`

Definition at line 46 of file ClpNonLinearCost.hpp.

5.49.2.3 `void setOriginalStatus (unsigned char & status, int value) [inline]`

Definition at line 50 of file ClpNonLinearCost.hpp.

5.49.2.4 `void setCurrentStatus (unsigned char & status, int value) [inline]`

Definition at line 55 of file ClpNonLinearCost.hpp.

5.49.2.5 `void setInitialStatus (unsigned char & status) [inline]`

Definition at line 60 of file ClpNonLinearCost.hpp.

5.49.2.6 `void setSameStatus (unsigned char & status) [inline]`

Definition at line 64 of file ClpNonLinearCost.hpp.

5.50 src/ClpObjective.hpp File Reference

Classes

- class [ClpObjective](#)
Objective Abstract Base Class.

5.51 src/ClpPackedMatrix.hpp File Reference

```
#include "CoinPragma.hpp"
#include "ClpMatrixBase.hpp"
```

Classes

- class [ClpPackedMatrix](#)
- class [ClpPackedMatrix2](#)
- struct [blockStruct](#)
- class [ClpPackedMatrix3](#)

Macros

- `#define` [COIN_RESTRICT](#)

5.51.1 Macro Definition Documentation

5.51.1.1 #define COIN_RESTRICT

Definition at line 18 of file ClpPackedMatrix.hpp.

5.52 src/ClpParameters.hpp File Reference

Classes

- struct [ClpTrustedData](#)
For a structure to be used by trusted code.

Enumerations

- enum [ClpIntParam](#) { [ClpMaxNumIteration](#) = 0, [ClpMaxNumIterationHotStart](#), [ClpNameDiscipline](#), [ClpLastIntParam](#) }
This is where to put any useful stuff.
- enum [ClpDbiParam](#) { [ClpDualObjectiveLimit](#), [ClpPrimalObjectiveLimit](#), [ClpDualTolerance](#), [ClpPrimalTolerance](#), [ClpObjOffset](#), [ClpMaxSeconds](#), [ClpPresolveTolerance](#), [ClpLastDbiParam](#) }
- enum [ClpStrParam](#) { [ClpProbName](#) = 0, [ClpLastStrParam](#) }

Functions

- template<class T >
void [ClpDisjointCopyN](#) (const T *array, const int size, T *newArray)
Copy (I don't like complexity of Coin version)
- template<class T >
void [ClpFillN](#) (T *array, const int size, T value)
And set.
- template<class T >
T * [ClpCopyOfArray](#) (const T *array, const int size, T value)
This returns a non const array filled with input from scalar or actual array.
- template<class T >
T * [ClpCopyOfArray](#) (const T *array, const int size)
This returns a non const array filled with actual array (or NULL)

5.52.1 Enumeration Type Documentation

5.52.1.1 enum ClpIntParam

This is where to put any useful stuff.

Enumerator

- [ClpMaxNumIteration](#)** The maximum number of iterations Clp can execute in the simplex methods.
- [ClpMaxNumIterationHotStart](#)** The maximum number of iterations Clp can execute in hotstart before terminating.
- [ClpNameDiscipline](#)** The name discipline; specifies how the solver will handle row and column names.

- 0: Auto names: Names cannot be set by the client. Names of the form Rnnnnnnnn or Cnnnnnnnn are generated on demand when a name for a specific row or column is requested; nnnnnnnn is derived from the row or column index. Requests for a vector of names return a vector with zero entries.
- 1: Lazy names: Names supplied by the client are retained. Names of the form Rnnnnnnnn or Cnnnnnnnn are generated on demand if no name has been supplied by the client. Requests for a vector of names return a vector sized to the largest index of a name supplied by the client; some entries in the vector may be null strings.
- 2: Full names: Names supplied by the client are retained. Names of the form Rnnnnnnnn or Cnnnnnnnn are generated on demand if no name has been supplied by the client. Requests for a vector of names return a vector sized to match the constraint system, and all entries will contain either the name specified by the client or a generated name.

ClpLastIntParam Just a marker, so that we can allocate a static sized array to store parameters.

Definition at line 12 of file ClpParameters.hpp.

5.52.1.2 enum ClpDbiParam

Enumerator

ClpDualObjectiveLimit Set Dual objective limit. This is to be used as a termination criteria in methods where the dual objective monotonically changes (dual simplex).

ClpPrimalObjectiveLimit Primal objective limit. This is to be used as a termination criteria in methods where the primal objective monotonically changes (e.g., primal simplex)

ClpDualTolerance The maximum amount the dual constraints can be violated and still be considered feasible.

ClpPrimalTolerance The maximum amount the primal constraints can be violated and still be considered feasible.

ClpObjOffset Objective function constant. This the value of the constant term in the objective function.

ClpMaxSeconds Maximum time in seconds - after this action is as max iterations.

ClpPresolveTolerance Tolerance to use in presolve.

ClpLastDbiParam Just a marker, so that we can allocate a static sized array to store parameters.

Definition at line 43 of file ClpParameters.hpp.

5.52.1.3 enum ClpStrParam

Enumerator

ClpProbName Name of the problem. This is the found on the Name card of an mps file.

ClpLastStrParam Just a marker, so that we can allocate a static sized array to store parameters.

Definition at line 71 of file ClpParameters.hpp.

5.52.2 Function Documentation

5.52.2.1 template<class T> void ClpDisjointCopyN (const T * array, const int size, T * newArray) [inline]

Copy (I don't like complexity of Coin version)

Definition at line 82 of file ClpParameters.hpp.

5.52.2.2 template<class T> void ClpFillN (T * array, const int size, T value) [inline]

And set.

Definition at line 88 of file ClpParameters.hpp.

5.52.2.3 `template<class T> T* ClpCopyOfArray (const T * array, const int size, T value) [inline]`

This returns a non const array filled with input from scalar or actual array.

Definition at line 96 of file ClpParameters.hpp.

5.52.2.4 `template<class T> T* ClpCopyOfArray (const T * array, const int size) [inline]`

This returns a non const array filled with actual array (or NULL)

Definition at line 108 of file ClpParameters.hpp.

5.53 src/ClpPdco.hpp File Reference

```
#include "ClpInterior.hpp"
```

Classes

- class [ClpPdco](#)
This solves problems in Primal Dual Convex Optimization.

5.54 src/ClpPdcoBase.hpp File Reference

```
#include "CoinPragma.hpp"
#include "CoinPackedMatrix.hpp"
#include "CoinDenseVector.hpp"
```

Classes

- class [ClpPdcoBase](#)
Abstract base class for tailoring everything for Pcdco.

5.55 src/ClpPlusMinusOneMatrix.hpp File Reference

```
#include "CoinPragma.hpp"
#include "ClpMatrixBase.hpp"
```

Classes

- class [ClpPlusMinusOneMatrix](#)
This implements a simple +- one matrix as derived from [ClpMatrixBase](#).

5.56 src/ClpPredictorCorrector.hpp File Reference

```
#include "ClpInterior.hpp"
```


Classes

- class [ClpPredictorCorrector](#)

This solves LPs using the predictor-corrector method due to Mehrotra.

5.57 src/ClpPresolve.hpp File Reference

```
#include "ClpSimplex.hpp"
#include "CoinPresolveMatrix.hpp"
```

Classes

- class [ClpPresolve](#)

This is the Clp interface to CoinPresolve.

5.58 src/ClpPrimalColumnDantzig.hpp File Reference

```
#include "ClpPrimalColumnPivot.hpp"
```

Classes

- class [ClpPrimalColumnDantzig](#)

Primal Column Pivot Dantzig Algorithm Class.

5.59 src/ClpPrimalColumnPivot.hpp File Reference

Classes

- class [ClpPrimalColumnPivot](#)

Primal Column Pivot Abstract Base Class.

Macros

- `#define` [CLP_PRIMAL_SLACK_MULTIPLIER](#) 1.01

5.59.1 Macro Definition Documentation

5.59.1.1 `#define` [CLP_PRIMAL_SLACK_MULTIPLIER](#) 1.01

Definition at line 153 of file ClpPrimalColumnPivot.hpp.

5.60 src/ClpPrimalColumnSteepest.hpp File Reference

```
#include "ClpPrimalColumnPivot.hpp"
#include <bitset>
```

Classes

- class [ClpPrimalColumnSteepest](#)
Primal Column Pivot Steepest Edge Algorithm Class.

5.61 src/ClpPrimalQuadraticDantzig.hpp File Reference

```
#include "ClpPrimalColumnPivot.hpp"
```

Classes

- class [ClpPrimalQuadraticDantzig](#)
Primal Column Pivot Dantzig Algorithm Class.

5.62 src/ClpQuadraticObjective.hpp File Reference

```
#include "ClpObjective.hpp"  
#include "CoinPackedMatrix.hpp"
```

Classes

- class [ClpQuadraticObjective](#)
Quadratic Objective Class.

5.63 src/ClpSimplex.hpp File Reference

```
#include <iostream>  
#include <cfloat>  
#include "ClpModel.hpp"  
#include "ClpMatrixBase.hpp"  
#include "ClpSolve.hpp"  
#include "ClpConfig.h"
```

Classes

- class [ClpSimplex](#)
This solves LPs using the simplex method.

Macros

- #define [DEVEX_TRY_NORM](#) 1.0e-4
- #define [DEVEX_ADD_ONE](#) 1.0

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

- `#define CLP_ABC_BEEN_FEASIBLE 65536`

Functions

- `void ClpSimplexUnitTest (const std::string &mpsDir)`
A function that tests the methods in the [ClpSimplex](#) class.

5.63.1 Macro Definition Documentation

5.63.1.1 `#define CLP_ABC_BEEN_FEASIBLE 65536`

Definition at line 1664 of file `ClpSimplex.hpp`.

5.63.1.2 `#define DEVEX_TRY_NORM 1.0e-4`

Definition at line 1689 of file `ClpSimplex.hpp`.

5.63.1.3 `#define DEVEX_ADD_ONE 1.0`

Definition at line 1690 of file `ClpSimplex.hpp`.

5.63.2 Function Documentation

5.63.2.1 `void ClpSimplexUnitTest (const std::string & mpsDir)`

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

5.64 src/ClpSimplexDual.hpp File Reference

```
#include "ClpSimplex.hpp"
```

Classes

- `class ClpSimplexDual`
This solves LPs using the dual simplex method.

5.65 src/ClpSimplexNonlinear.hpp File Reference

```
#include "ClpSimplexPrimal.hpp"
```

Classes

- class [ClpSimplexNonlinear](#)

This solves non-linear LPs using the primal simplex method.

5.66 src/ClpSimplexOther.hpp File Reference

```
#include "ClpSimplex.hpp"
```

Classes

- class [ClpSimplexOther](#)

This is for Simplex stuff which is neither dual nor primal.

- struct [ClpSimplexOther::parametricsData](#)

5.67 src/ClpSimplexPrimal.hpp File Reference

```
#include "ClpSimplex.hpp"
```

Classes

- class [ClpSimplexPrimal](#)

This solves LPs using the primal simplex method.

5.68 src/ClpSolve.hpp File Reference

```
#include "ClpConfig.h"
```

Classes

- class [ClpSolve](#)

This is a very simple class to guide algorithms.

- class [ClpSimplexProgress](#)

For saving extra information to see if looping.

Macros

Data

- #define [CLP_PROGRESS](#) 5
- #define [CLP_CYCLE](#) 12

5.68.1 Macro Definition Documentation

5.68.1.1 #define CLP_PROGRESS 5

Definition at line 342 of file ClpSolve.hpp.

5.68.1.2 #define CLP_CYCLE 12

Definition at line 365 of file ClpSolve.hpp.

5.69 src/CoinAbcBaseFactorization.hpp File Reference

```
#include "AbcCommon.hpp"  
#include "CoinAbcHelperFunctions.hpp"
```

Classes

- class [CoinAbcTypeFactorization](#)

Macros

- #define [FACTOR_CPU](#) 1
This deals with Factorization and Updates.
- #define [LARGE_SET](#) COIN_INT_MAX-10
- #define [LARGE_UNSET](#) ([LARGE_SET](#)+1)

used by factorization

- #define [checkLinks](#)(x)
- #define [CONVERTROW](#) 2

5.69.1 Macro Definition Documentation

5.69.1.1 #define FACTOR_CPU 1

This deals with Factorization and Updates.

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 23 of file CoinAbcBaseFactorization.hpp.

5.69.1.2 #define LARGE_SET COIN_INT_MAX-10

Definition at line 25 of file CoinAbcBaseFactorization.hpp.

5.69.1.3 #define LARGE_UNSET (LARGE_SET+1)

Definition at line 26 of file CoinAbcBaseFactorization.hpp.

5.69.1.4 #define checkLinks(x)

Definition at line 608 of file CoinAbcBaseFactorization.hpp.

5.69.1.5 #define CONVERTROW 2

Definition at line 824 of file CoinAbcBaseFactorization.hpp.

5.70 src/CoinAbcCommon.hpp File Reference

```
#include "CoinPragma.hpp"
#include "CoinUtilsConfig.h"
#include <iostream>
#include <string>
#include <cassert>
#include <cstdio>
#include <cmath>
#include "AbcCommon.hpp"
#include "CoinHelperFunctions.hpp"
#include <endian.h>
```

Classes

- class [AbcTolerancesEtc](#)

Macros

- #define [COIN_FAC_NEW](#)
- #define [ABC_INLINE](#)
- #define [ABC_PARALLEL](#) 0
- #define [cilk_for](#) for
- #define [cilk_spawn](#)
- #define [cilk_sync](#)
- #define [SLACK_VALUE](#) 1
- #define [ABC_INSTRUMENT](#) 1
- #define [instrument_start](#)(name, x)
- #define [instrument_add](#)(x)
- #define [instrument_end](#)()
- #define [instrument_do](#)(name, x)
- #define [instrument_end_and_adjust](#)(x)
- #define [ABC_INTEL](#)
- #define [CoinFabs](#)(x) fabs(x)
- #define [TEST_DOUBLE_NONZERO](#)(x) (true)
- #define [USE_TEST_INT_ZERO](#)
- #define [TEST_INT_NONZERO](#)(x) (x)
- #define [TEST_DOUBLE REALLY_NONZERO](#)(x) (x)
- #define [TEST_DOUBLE_NONZERO_REGISTER](#)(x) (true)
- #define [USE_FIXED_ZERO_TOLERANCE](#)
- #define [TEST_LESS_THAN_TOLERANCE](#)(x) (fabs(x)<pow(0.5,43))
- #define [TEST_LESS_THAN_UPDATE_TOLERANCE](#)(x) (fabs(x)<pow(0.5,43))
- #define [TEST_LESS_THAN_TOLERANCE_REGISTER](#)(x) (fabs(x)<pow(0.5,43))
- #define [ABC_EXPONENT](#)(x) ((reinterpret_cast<int *>(&x))[1]&0x7ff00000)
- #define [TEST_EXPONENT_LESS_THAN_TOLERANCE](#)(x) (x<0x3d400000)

- `#define TEST_EXPONENT_LESS_THAN_UPDATE_TOLERANCE(x) (x<0x3d400000)`
- `#define TEST_EXPONENT_NON_ZERO(x) (x)`
- `#define COINFACTORIZATION_BITS_PER_INT 32`
- `#define COINFACTORIZATION_SHIFT_PER_INT 5`
- `#define COINFACTORIZATION_MASK_PER_INT 0x1f`
- `#define ABC_DENSE_CODE 2`

Typedefs

- `typedef double CoinSimplexDouble`
- `typedef int CoinSimplexInt`
- `typedef unsigned int CoinSimplexUnsignedInt`
- `typedef unsigned int CoinExponent`
- `typedef unsigned char CoinCheckZero`

Functions

- `template<class T >`
`void CoinAbcMemset0 (register T *to, const int size)`
- `template<class T >`
`void CoinAbcMemcpy (register T *to, register const T *from, const int size)`

5.70.1 Macro Definition Documentation

5.70.1.1 `#define COIN_FAC_NEW`

Definition at line 8 of file CoinAbcCommon.hpp.

5.70.1.2 `#define ABC_INLINE`

Definition at line 30 of file CoinAbcCommon.hpp.

5.70.1.3 `#define ABC_PARALLEL 0`

Definition at line 36 of file CoinAbcCommon.hpp.

5.70.1.4 `#define cilk_for for`

Definition at line 49 of file CoinAbcCommon.hpp.

5.70.1.5 `#define cilk_spawn`

Definition at line 50 of file CoinAbcCommon.hpp.

5.70.1.6 `#define cilk_sync`

Definition at line 51 of file CoinAbcCommon.hpp.

5.70.1.7 `#define SLACK_VALUE 1`

Definition at line 54 of file CoinAbcCommon.hpp.

5.70.1.8 #define ABC_INSTRUMENT 1

Definition at line 55 of file CoinAbcCommon.hpp.

5.70.1.9 #define instrument_start(*name*, *x*)

Definition at line 58 of file CoinAbcCommon.hpp.

5.70.1.10 #define instrument_add(*x*)

Definition at line 59 of file CoinAbcCommon.hpp.

5.70.1.11 #define instrument_end()

Definition at line 60 of file CoinAbcCommon.hpp.

5.70.1.12 #define instrument_do(*name*, *x*)

Definition at line 62 of file CoinAbcCommon.hpp.

5.70.1.13 #define instrument_end_and_adjust(*x*)

Definition at line 64 of file CoinAbcCommon.hpp.

5.70.1.14 #define ABC_INTEL

Definition at line 76 of file CoinAbcCommon.hpp.

5.70.1.15 #define CoinFabs(*x*) fabs(x)

Definition at line 86 of file CoinAbcCommon.hpp.

5.70.1.16 #define TEST_DOUBLE_NONZERO(*x*)(true)

Definition at line 96 of file CoinAbcCommon.hpp.

5.70.1.17 #define USE_TEST_INT_ZERO

Definition at line 98 of file CoinAbcCommon.hpp.

5.70.1.18 #define TEST_INT_NONZERO(*x*)(x)

Definition at line 100 of file CoinAbcCommon.hpp.

5.70.1.19 #define TEST_DOUBLE_REALLY_NONZERO(*x*)(x)

Definition at line 112 of file CoinAbcCommon.hpp.

5.70.1.20 #define TEST_DOUBLE_NONZERO_REGISTER(*x*)(true)

Definition at line 122 of file CoinAbcCommon.hpp.

5.70.1.21 #define USE_FIXED_ZERO_TOLERANCE

Definition at line 124 of file CoinAbcCommon.hpp.

5.70.1.22 `#define TEST_LESS_THAN_TOLERANCE(x) (fabs(x)<pow(0.5,43))`

Definition at line 136 of file CoinAbcCommon.hpp.

5.70.1.23 `#define TEST_LESS_THAN_UPDATE_TOLERANCE(x) (fabs(x)<pow(0.5,43))`

Definition at line 137 of file CoinAbcCommon.hpp.

5.70.1.24 `#define TEST_LESS_THAN_TOLERANCE_REGISTER(x) (fabs(x)<pow(0.5,43))`

Definition at line 146 of file CoinAbcCommon.hpp.

5.70.1.25 `#define ABC_EXPONENT(x) ((reinterpret_cast<int *>(&x))[1]&0x7ff00000)`

Definition at line 155 of file CoinAbcCommon.hpp.

5.70.1.26 `#define TEST_EXPONENT_LESS_THAN_TOLERANCE(x) (x<0x3d400000)`

Definition at line 159 of file CoinAbcCommon.hpp.

5.70.1.27 `#define TEST_EXPONENT_LESS_THAN_UPDATE_TOLERANCE(x) (x<0x3d400000)`

Definition at line 160 of file CoinAbcCommon.hpp.

5.70.1.28 `#define TEST_EXPONENT_NON_ZERO(x) (x)`

Definition at line 161 of file CoinAbcCommon.hpp.

5.70.1.29 `#define COINFACTORIZATION_BITS_PER_INT 32`

Definition at line 174 of file CoinAbcCommon.hpp.

5.70.1.30 `#define COINFACTORIZATION_SHIFT_PER_INT 5`

Definition at line 175 of file CoinAbcCommon.hpp.

5.70.1.31 `#define COINFACTORIZATION_MASK_PER_INT 0x1f`

Definition at line 176 of file CoinAbcCommon.hpp.

5.70.1.32 `#define ABC_DENSE_CODE 2`

Definition at line 220 of file CoinAbcCommon.hpp.

5.70.2 Typedef Documentation

5.70.2.1 `typedef double CoinSimplexDouble`

Definition at line 21 of file CoinAbcCommon.hpp.

5.70.2.2 `typedef int CoinSimplexInt`

Definition at line 22 of file CoinAbcCommon.hpp.

5.70.2.3 `typedef unsigned int CoinSimplexUnsignedInt`

Definition at line 23 of file CoinAbcCommon.hpp.

5.70.2.4 typedef unsigned int CoinExponent

Definition at line 153 of file CoinAbcCommon.hpp.

5.70.2.5 typedef unsigned char CoinCheckZero

Definition at line 225 of file CoinAbcCommon.hpp.

5.70.3 Function Documentation

5.70.3.1 template<class T> void CoinAbcMemset0 (register T * to, const int size) [inline]

Definition at line 227 of file CoinAbcCommon.hpp.

5.70.3.2 template<class T> void CoinAbcMemcpy (register T * to, register const T * from, const int size) [inline]

Definition at line 238 of file CoinAbcCommon.hpp.

5.71 src/CoinAbcCommonFactorization.hpp File Reference

```
#include "CoinAbcCommon.hpp"
#include "CoinAbcDenseFactorization.hpp"
```

Classes

- struct [CoinAbcStatistics](#)
- struct [CoinAbcStack](#)

Macros

- #define [INITIAL_AVERAGE](#) 1.0
- #define [INITIAL_AVERAGE2](#) 1.0
- #define [AVERAGE_SCALE_BACK](#) 0.8
- #define [setStatistics\(x\)](#)
- #define [factorizationStatistics\(\)](#) (true)
- #define [FACTORIZATION_STATISTICS](#) 0
- #define [twiddleFactor1S\(\)](#) (1.0)
- #define [twiddleFactor2S\(\)](#) (1.0)
- #define [twiddleFtranFactor1\(\)](#) (1.0)
- #define [twiddleFtranFTFactor1\(\)](#) (1.0)
- #define [twiddleBtranFactor1\(\)](#) (1.0)
- #define [twiddleFtranFactor2\(\)](#) (1.0)
- #define [twiddleFtranFTFactor2\(\)](#) (1.0)
- #define [twiddleBtranFactor2\(\)](#) (1.0)
- #define [twiddleBtranFullFactor1\(\)](#) (1.0)
- #define [ABC_FAC_GOT_LCOPY](#) 4
- #define [ABC_FAC_GOT_RCOPY](#) 8
- #define [ABC_FAC_GOT_UCOPY](#) 16
- #define [ABC_FAC_GOT_SPARSE](#) 32

- `#define SWAP_FACTOR 2`
- `#define BLOCKING8 8`
- `#define BLOCKING8X8 BLOCKING8*BLOCKING8`

Functions

- void `CoinAbcDgetrs` (char trans, int m, double *a, double *work)
- int `CoinAbcDgetrf` (int m, int n, double *a, int lda, int *ipiv)
- void `CoinAbcDgetrs` (char trans, int m, long double *a, long double *work)
- int `CoinAbcDgetrf` (int m, int n, long double *a, int lda, int *ipiv)

5.71.1 Macro Definition Documentation

5.71.1.1 `#define INITIAL_AVERAGE 1.0`

Definition at line 17 of file `CoinAbcCommonFactorization.hpp`.

5.71.1.2 `#define INITIAL_AVERAGE2 1.0`

Definition at line 18 of file `CoinAbcCommonFactorization.hpp`.

5.71.1.3 `#define AVERAGE_SCALE_BACK 0.8`

Definition at line 19 of file `CoinAbcCommonFactorization.hpp`.

5.71.1.4 `#define setStatistics(x)`

Definition at line 22 of file `CoinAbcCommonFactorization.hpp`.

5.71.1.5 `#define factorizationStatistics() (true)`

Definition at line 23 of file `CoinAbcCommonFactorization.hpp`.

5.71.1.6 `#define FACTORIZATION_STATISTICS 0`

Definition at line 31 of file `CoinAbcCommonFactorization.hpp`.

5.71.1.7 `#define twiddleFactor1S() (1.0)`

Definition at line 57 of file `CoinAbcCommonFactorization.hpp`.

5.71.1.8 `#define twiddleFactor2S() (1.0)`

Definition at line 58 of file `CoinAbcCommonFactorization.hpp`.

5.71.1.9 `#define twiddleFtranFactor1() (1.0)`

Definition at line 59 of file `CoinAbcCommonFactorization.hpp`.

5.71.1.10 `#define twiddleFtranFTFactor1() (1.0)`

Definition at line 60 of file `CoinAbcCommonFactorization.hpp`.

5.71.1.11 `#define twiddleBtranFactor1() (1.0)`

Definition at line 61 of file `CoinAbcCommonFactorization.hpp`.

5.71.1.12 `#define twiddleFtranFactor2() (1.0)`

Definition at line 62 of file `CoinAbcCommonFactorization.hpp`.

5.71.1.13 `#define twiddleFtranFTFactor2() (1.0)`

Definition at line 63 of file `CoinAbcCommonFactorization.hpp`.

5.71.1.14 `#define twiddleBtranFactor2() (1.0)`

Definition at line 64 of file `CoinAbcCommonFactorization.hpp`.

5.71.1.15 `#define twiddleBtranFullFactor1() (1.0)`

Definition at line 65 of file `CoinAbcCommonFactorization.hpp`.

5.71.1.16 `#define ABC_FAC_GOT_LCOPY 4`

Definition at line 67 of file `CoinAbcCommonFactorization.hpp`.

5.71.1.17 `#define ABC_FAC_GOT_RCOPY 8`

Definition at line 68 of file `CoinAbcCommonFactorization.hpp`.

5.71.1.18 `#define ABC_FAC_GOT_UCOPY 16`

Definition at line 69 of file `CoinAbcCommonFactorization.hpp`.

5.71.1.19 `#define ABC_FAC_GOT_SPARSE 32`

Definition at line 70 of file `CoinAbcCommonFactorization.hpp`.

5.71.1.20 `#define SWAP_FACTOR 2`

Definition at line 88 of file `CoinAbcCommonFactorization.hpp`.

5.71.1.21 `#define BLOCKING8 8`

Definition at line 89 of file `CoinAbcCommonFactorization.hpp`.

5.71.1.22 `#define BLOCKING8X8 BLOCKING8*BLOCKING8`

Definition at line 90 of file `CoinAbcCommonFactorization.hpp`.

5.71.2 Function Documentation

5.71.2.1 `void CoinAbcDgetrs (char trans, int m, double * a, double * work)`

5.71.2.2 `int CoinAbcDgetrf (int m, int n, double * a, int lda, int * ipiv)`

5.71.2.3 `void CoinAbcDgetrs (char trans, int m, long double * a, long double * work)`

5.71.2.4 `int CoinAbcDgetrf (int m, int n, long double * a, int lda, int * ipiv)`

5.72 src/CoinAbcDenseFactorization.hpp File Reference

```
#include <iostream>
#include <string>
#include <cassert>
#include "CoinTypes.hpp"
#include "CoinAbcCommon.hpp"
#include "CoinIndexedVector.hpp"
```

Classes

- class [CoinAbcAnyFactorization](#)
Abstract base class which also has some scalars so can be used from Dense or Simp.
- class [CoinAbcDenseFactorization](#)
This deals with Factorization and Updates This is a simple dense version so other people can write a better one.

Macros

data

- #define [slackValue2_](#) 1.0

5.72.1 Macro Definition Documentation

5.72.1.1 #define slackValue2_ 1.0

Definition at line 344 of file CoinAbcDenseFactorization.hpp.

5.73 src/CoinAbcFactorization.hpp File Reference

```
#include "CoinAbcCommonFactorization.hpp"
#include "CoinAbcBaseFactorization.hpp"
```

Macros

- #define [CoinAbcTypeFactorization](#) CoinAbcFactorization
- #define [ABC_SMALL](#) -1
- #define [COIN_BIG_DOUBLE](#) 1
- #define [CoinAbcTypeFactorization](#) CoinAbcLongFactorization
- #define [ABC_SMALL](#) -1
- #define [CoinAbcTypeFactorization](#) CoinAbcSmallFactorization
- #define [ABC_SMALL](#) 4
- #define [CoinAbcTypeFactorization](#) CoinAbcOrderedFactorization
- #define [ABC_SMALL](#) -1

5.73.1 Macro Definition Documentation

5.73.1.1 `#define CoinAbcTypeFactorization CoinAbcFactorization`

Definition at line 34 of file `CoinAbcFactorization.hpp`.

5.73.1.2 `#define ABC_SMALL -1`

Definition at line 35 of file `CoinAbcFactorization.hpp`.

5.73.1.3 `#define COIN_BIG_DOUBLE 1`

Definition at line 22 of file `CoinAbcFactorization.hpp`.

5.73.1.4 `#define CoinAbcTypeFactorization CoinAbcLongFactorization`

Definition at line 34 of file `CoinAbcFactorization.hpp`.

5.73.1.5 `#define ABC_SMALL -1`

Definition at line 35 of file `CoinAbcFactorization.hpp`.

5.73.1.6 `#define CoinAbcTypeFactorization CoinAbcSmallFactorization`

Definition at line 34 of file `CoinAbcFactorization.hpp`.

5.73.1.7 `#define ABC_SMALL 4`

Definition at line 35 of file `CoinAbcFactorization.hpp`.

5.73.1.8 `#define CoinAbcTypeFactorization CoinAbcOrderedFactorization`

Definition at line 34 of file `CoinAbcFactorization.hpp`.

5.73.1.9 `#define ABC_SMALL -1`

Definition at line 35 of file `CoinAbcFactorization.hpp`.

5.74 `src/CoinAbcHelperFunctions.hpp` File Reference

```
#include "ClpConfig.h"
#include <cmath>
#include "CoinAbcCommon.hpp"
```

Classes

- struct [scatterStruct](#)

Macros

- `#define` [abc_assert](#)(condition)
- `#define` [CILK_FOR_GRAINSIZE](#) 128
- `#define` [UNROLL_SCATTER](#) 2

Note (JJF) I have added some operations on arrays even though they may duplicate CoinDenseVector.

- `#define INLINE_SCATTER 1`
- `#define coin_prefetch(mem)`
- `#define coin_prefetch_const(mem)`
- `#define NEW_CHUNK_SIZE 4`
- `#define NEW_CHUNK_SIZE_INCREMENT (NEW_CHUNK_SIZE+NEW_CHUNK_SIZE/2);`
- `#define NEW_CHUNK_SIZE_OFFSET (NEW_CHUNK_SIZE/2)`
- `#define SCATTER_ATTRIBUTE`
- `#define UNROLL_GATHER 0`
- `#define INLINE_GATHER 1`
- `#define UNROLL_MULTIPLY_INDEXED 0`
- `#define INLINE_MULTIPLY_INDEXED 0`

Typedefs

- `typedef void(* scatterUpdate)(int, CoinFactorizationDouble, const CoinFactorizationDouble *, double *) SCATTER_ATTRIBUTE`

Functions

- `void ABC_INLINE CoinAbcScatterUpdate (int number, CoinFactorizationDouble pivotValue, const CoinFactorizationDouble *COIN_RESTRICT thisElement, const int *COIN_RESTRICT thisIndex, CoinFactorizationDouble *COIN_RESTRICT region)`
- `void ABC_INLINE CoinAbcScatterUpdate (int number, CoinFactorizationDouble pivotValue, const CoinFactorizationDouble *COIN_RESTRICT thisElement, CoinFactorizationDouble *COIN_RESTRICT region)`
- `void CoinAbcScatterUpdate0 (int numberIn, CoinFactorizationDouble multiplier, const CoinFactorizationDouble *COIN_RESTRICT thisElement, CoinFactorizationDouble *COIN_RESTRICT region) SCATTER_ATTRIBUTE`
- `void CoinAbcScatterUpdate1 (int numberIn, CoinFactorizationDouble multiplier, const CoinFactorizationDouble *COIN_RESTRICT thisElement, CoinFactorizationDouble *COIN_RESTRICT region) SCATTER_ATTRIBUTE`
- `void CoinAbcScatterUpdate2 (int numberIn, CoinFactorizationDouble multiplier, const CoinFactorizationDouble *COIN_RESTRICT thisElement, CoinFactorizationDouble *COIN_RESTRICT region) SCATTER_ATTRIBUTE`
- `void CoinAbcScatterUpdate3 (int numberIn, CoinFactorizationDouble multiplier, const CoinFactorizationDouble *COIN_RESTRICT thisElement, CoinFactorizationDouble *COIN_RESTRICT region) SCATTER_ATTRIBUTE`
- `void CoinAbcScatterUpdate4 (int numberIn, CoinFactorizationDouble multiplier, const CoinFactorizationDouble *COIN_RESTRICT thisElement, CoinFactorizationDouble *COIN_RESTRICT region) SCATTER_ATTRIBUTE`
- `void CoinAbcScatterUpdate5 (int numberIn, CoinFactorizationDouble multiplier, const CoinFactorizationDouble *COIN_RESTRICT thisElement, CoinFactorizationDouble *COIN_RESTRICT region) SCATTER_ATTRIBUTE`
- `void CoinAbcScatterUpdate6 (int numberIn, CoinFactorizationDouble multiplier, const CoinFactorizationDouble *COIN_RESTRICT thisElement, CoinFactorizationDouble *COIN_RESTRICT region) SCATTER_ATTRIBUTE`
- `void CoinAbcScatterUpdate7 (int numberIn, CoinFactorizationDouble multiplier, const CoinFactorizationDouble *COIN_RESTRICT thisElement, CoinFactorizationDouble *COIN_RESTRICT region) SCATTER_ATTRIBUTE`
- `void CoinAbcScatterUpdate8 (int numberIn, CoinFactorizationDouble multiplier, const CoinFactorizationDouble *COIN_RESTRICT thisElement, CoinFactorizationDouble *COIN_RESTRICT region) SCATTER_ATTRIBUTE`
- `void CoinAbcScatterUpdate4N (int numberIn, CoinFactorizationDouble multiplier, const CoinFactorizationDouble *COIN_RESTRICT thisElement, CoinFactorizationDouble *COIN_RESTRICT region) SCATTER_ATTRIBUTE`
- `void CoinAbcScatterUpdate4NPlus1 (int numberIn, CoinFactorizationDouble multiplier, const CoinFactorizationDouble *COIN_RESTRICT thisElement, CoinFactorizationDouble *COIN_RESTRICT region) SCATTER_ATTRIBUTE`
- `void CoinAbcScatterUpdate4NPlus2 (int numberIn, CoinFactorizationDouble multiplier, const CoinFactorizationDouble *COIN_RESTRICT thisElement, CoinFactorizationDouble *COIN_RESTRICT region) SCATTER_ATTRIBUTE`

- void [CoinAbcScatterUpdate6Add](#) (int numberIn, CoinFactorizationDouble multiplier, const CoinFactorizationDouble *[COIN_RESTRICT](#) thisElement, CoinFactorizationDouble *[COIN_RESTRICT](#) region) [SCATTER_ATTRIBUTE](#)
- void [CoinAbcScatterUpdate7Add](#) (int numberIn, CoinFactorizationDouble multiplier, const CoinFactorizationDouble *[COIN_RESTRICT](#) thisElement, CoinFactorizationDouble *[COIN_RESTRICT](#) region) [SCATTER_ATTRIBUTE](#)
- void [CoinAbcScatterUpdate8Add](#) (int numberIn, CoinFactorizationDouble multiplier, const CoinFactorizationDouble *[COIN_RESTRICT](#) thisElement, CoinFactorizationDouble *[COIN_RESTRICT](#) region) [SCATTER_ATTRIBUTE](#)
- void [CoinAbcScatterUpdate4NAdd](#) (int numberIn, CoinFactorizationDouble multiplier, const CoinFactorizationDouble *[COIN_RESTRICT](#) thisElement, CoinFactorizationDouble *[COIN_RESTRICT](#) region) [SCATTER_ATTRIBUTE](#)
- void [CoinAbcScatterUpdate4NPlus1Add](#) (int numberIn, CoinFactorizationDouble multiplier, const CoinFactorizationDouble *[COIN_RESTRICT](#) thisElement, CoinFactorizationDouble *[COIN_RESTRICT](#) region) [SCATTER_ATTRIBUTE](#)
- void [CoinAbcScatterUpdate4NPlus2Add](#) (int numberIn, CoinFactorizationDouble multiplier, const CoinFactorizationDouble *[COIN_RESTRICT](#) thisElement, CoinFactorizationDouble *[COIN_RESTRICT](#) region) [SCATTER_ATTRIBUTE](#)
- void [CoinAbcScatterUpdate4NPlus3Add](#) (int numberIn, CoinFactorizationDouble multiplier, const CoinFactorizationDouble *[COIN_RESTRICT](#) thisElement, CoinFactorizationDouble *[COIN_RESTRICT](#) region) [SCATTER_ATTRIBUTE](#)
- CoinFactorizationDouble [ABC_INLINE CoinAbcGatherUpdate](#) (CoinSimplexInt number, const CoinFactorizationDouble *[COIN_RESTRICT](#) thisElement, const int *[COIN_RESTRICT](#) thisIndex, CoinFactorizationDouble *[COIN_RESTRICT](#) region)
- void [CoinAbcMultiplyIndexed](#) (int number, const double *[COIN_RESTRICT](#) multiplier, const int *[COIN_RESTRICT](#) thisIndex, CoinFactorizationDouble *[COIN_RESTRICT](#) region)
- void [CoinAbcMultiplyIndexed](#) (int number, const long double *[COIN_RESTRICT](#) multiplier, const int *[COIN_RESTRICT](#) thisIndex, long double *[COIN_RESTRICT](#) region)
- double [CoinAbcMaximumAbsElement](#) (const double *region, int size)
- void [CoinAbcMinMaxAbsElement](#) (const double *region, int size, double &minimum, double &maximum)
- void [CoinAbcMinMaxAbsNormalValues](#) (const double *region, int size, double &minimum, double &maximum)
- void [CoinAbcScale](#) (double *region, double multiplier, int size)
- void [CoinAbcScaleNormalValues](#) (double *region, double multiplier, double killIfLessThanThis, int size)
- double [CoinAbcMaximumAbsElementAndScale](#) (double *region, double multiplier, int size)
maximum fabs(region[i]) and then region[i]=multiplier*
- void [CoinAbcSetElements](#) (double *region, int size, double value)
- void [CoinAbcMultiplyAdd](#) (const double *region1, int size, double multiplier1, double *regionChanged, double multiplier2)
- double [CoinAbcInnerProduct](#) (const double *region1, int size, const double *region2)
- void [CoinAbcGetNorms](#) (const double *region, int size, double &norm1, double &norm2)
- void [CoinAbcScatterTo](#) (const double *regionFrom, double *regionTo, const int *index, int number)
regionTo[index[i]]=regionFrom[i]
- void [CoinAbcGatherFrom](#) (const double *regionFrom, double *regionTo, const int *index, int number)
regionTo[i]=regionFrom[index[i]]
- void [CoinAbcScatterZeroTo](#) (double *regionTo, const int *index, int number)
regionTo[index[i]]=0.0
- void [CoinAbcScatterToList](#) (const double *regionFrom, double *regionTo, const int *indexList, const int *indexScatter, int number)
regionTo[indexScatter[indexList[i]]]=regionFrom[indexList[i]]
- void [CoinAbcInverseSqrts](#) (double *array, int n)
array[i]=1.0/sqrt(array[i])

- void [CoinAbcReciprocal](#) (double *array, int n, const double *input)
 - void [CoinAbcMemcpyLong](#) (double *array, const double *arrayFrom, int size)
 - void [CoinAbcMemcpyLong](#) (int *array, const int *arrayFrom, int size)
 - void [CoinAbcMemcpyLong](#) (unsigned char *array, const unsigned char *arrayFrom, int size)
 - void [CoinAbcMemset0Long](#) (double *array, int size)
 - void [CoinAbcMemset0Long](#) (int *array, int size)
 - void [CoinAbcMemset0Long](#) (unsigned char *array, int size)
 - void [CoinAbcMemmove](#) (double *array, const double *arrayFrom, int size)
 - void [CoinAbcMemmove](#) (int *array, const int *arrayFrom, int size)
 - void [CoinAbcMemmove](#) (unsigned char *array, const unsigned char *arrayFrom, int size)
 - void [CoinAbcMemmoveAndZero](#) (double *array, double *arrayFrom, int size)
- This moves down and zeroes out end.*
- int [CoinAbcCompact](#) (int numberSections, int alreadyDone, double *array, const int *starts, const int *lengths)
- This compacts several sections and zeroes out end (returns number)*
- int [CoinAbcCompact](#) (int numberSections, int alreadyDone, int *array, const int *starts, const int *lengths)
- This compacts several sections (returns number)*

5.74.1 Macro Definition Documentation

5.74.1.1 `#define abc_assert(condition)`

Value:

```
{ if (!condition) {printf("abc_assert in %s at line %d - %s is false\n", \
    __FILE__, __LINE__, __STRING(condition)); abort();} }
```

Definition at line 21 of file CoinAbcHelperFunctions.hpp.

5.74.1.2 `#define CILK_FOR_GRAINSIZE 128`

Definition at line 26 of file CoinAbcHelperFunctions.hpp.

5.74.1.3 `#define UNROLL_SCATTER 2`

Note (JJF) I have added some operations on arrays even though they may duplicate CoinDenseVector.

Definition at line 43 of file CoinAbcHelperFunctions.hpp.

5.74.1.4 `#define INLINE_SCATTER 1`

Definition at line 44 of file CoinAbcHelperFunctions.hpp.

5.74.1.5 `#define coin_prefetch(mem)`

Definition at line 523 of file CoinAbcHelperFunctions.hpp.

5.74.1.6 `#define coin_prefetch_const(mem)`

Definition at line 524 of file CoinAbcHelperFunctions.hpp.

5.74.1.7 `#define NEW_CHUNK_SIZE 4`

Definition at line 526 of file CoinAbcHelperFunctions.hpp.

5.74.1.8 `#define NEW_CHUNK_SIZE_INCREMENT (NEW_CHUNK_SIZE+NEW_CHUNK_SIZE/2);`

Definition at line 527 of file CoinAbcHelperFunctions.hpp.

5.74.1.9 `#define NEW_CHUNK_SIZE_OFFSET (NEW_CHUNK_SIZE/2)`

Definition at line 528 of file CoinAbcHelperFunctions.hpp.

5.74.1.10 `#define SCATTER_ATTRIBUTE`

Definition at line 532 of file CoinAbcHelperFunctions.hpp.

5.74.1.11 `#define UNROLL_GATHER 0`

Definition at line 706 of file CoinAbcHelperFunctions.hpp.

5.74.1.12 `#define INLINE_GATHER 1`

Definition at line 707 of file CoinAbcHelperFunctions.hpp.

5.74.1.13 `#define UNROLL_MULTIPLY_INDEXED 0`

Definition at line 733 of file CoinAbcHelperFunctions.hpp.

5.74.1.14 `#define INLINE_MULTIPLY_INDEXED 0`

Definition at line 734 of file CoinAbcHelperFunctions.hpp.

5.74.2 Typedef Documentation

5.74.2.1 `typedef void(* scatterUpdate)(int, CoinFactorizationDouble, const CoinFactorizationDouble *, double *)`
SCATTER_ATTRIBUTE

Definition at line 533 of file CoinAbcHelperFunctions.hpp.

5.74.3 Function Documentation

5.74.3.1 `void ABC_INLINE CoinAbcScatterUpdate (int number, CoinFactorizationDouble pivotValue, const CoinFactorizationDouble *COIN_RESTRICT thisElement, const int *COIN_RESTRICT thisIndex, CoinFactorizationDouble *COIN_RESTRICT region) [inline]`

Definition at line 51 of file CoinAbcHelperFunctions.hpp.

5.74.3.2 `void ABC_INLINE CoinAbcScatterUpdate (int number, CoinFactorizationDouble pivotValue, const CoinFactorizationDouble *COIN_RESTRICT thisElement, CoinFactorizationDouble *COIN_RESTRICT region) [inline]`

Definition at line 266 of file CoinAbcHelperFunctions.hpp.

5.74.3.3 `void CoinAbcScatterUpdate0 (int numberIn, CoinFactorizationDouble multiplier, const CoinFactorizationDouble *COIN_RESTRICT thisElement, CoinFactorizationDouble *COIN_RESTRICT region)`

5.74.3.4 `void CoinAbcScatterUpdate1 (int numberIn, CoinFactorizationDouble multiplier, const CoinFactorizationDouble *COIN_RESTRICT thisElement, CoinFactorizationDouble *COIN_RESTRICT region)`

- 5.74.3.24 void CoinAbcScatterUpdate4NSubtract (int *numberIn*, CoinFactorizationDouble *multiplier*, const CoinFactorizationDouble *COIN_RESTRICT *thisElement*, CoinFactorizationDouble *COIN_RESTRICT *region*)
- 5.74.3.25 void CoinAbcScatterUpdate4NPlus1Subtract (int *numberIn*, CoinFactorizationDouble *multiplier*, const CoinFactorizationDouble *COIN_RESTRICT *thisElement*, CoinFactorizationDouble *COIN_RESTRICT *region*)
- 5.74.3.26 void CoinAbcScatterUpdate4NPlus2Subtract (int *numberIn*, CoinFactorizationDouble *multiplier*, const CoinFactorizationDouble *COIN_RESTRICT *thisElement*, CoinFactorizationDouble *COIN_RESTRICT *region*)
- 5.74.3.27 void CoinAbcScatterUpdate4NPlus3Subtract (int *numberIn*, CoinFactorizationDouble *multiplier*, const CoinFactorizationDouble *COIN_RESTRICT *thisElement*, CoinFactorizationDouble *COIN_RESTRICT *region*)
- 5.74.3.28 void CoinAbcScatterUpdate1Add (int *numberIn*, CoinFactorizationDouble *multiplier*, const CoinFactorizationDouble *COIN_RESTRICT *thisElement*, CoinFactorizationDouble *COIN_RESTRICT *region*)
- 5.74.3.29 void CoinAbcScatterUpdate2Add (int *numberIn*, CoinFactorizationDouble *multiplier*, const CoinFactorizationDouble *COIN_RESTRICT *thisElement*, CoinFactorizationDouble *COIN_RESTRICT *region*)
- 5.74.3.30 void CoinAbcScatterUpdate3Add (int *numberIn*, CoinFactorizationDouble *multiplier*, const CoinFactorizationDouble *COIN_RESTRICT *thisElement*, CoinFactorizationDouble *COIN_RESTRICT *region*)
- 5.74.3.31 void CoinAbcScatterUpdate4Add (int *numberIn*, CoinFactorizationDouble *multiplier*, const CoinFactorizationDouble *COIN_RESTRICT *thisElement*, CoinFactorizationDouble *COIN_RESTRICT *region*)
- 5.74.3.32 void CoinAbcScatterUpdate5Add (int *numberIn*, CoinFactorizationDouble *multiplier*, const CoinFactorizationDouble *COIN_RESTRICT *thisElement*, CoinFactorizationDouble *COIN_RESTRICT *region*)
- 5.74.3.33 void CoinAbcScatterUpdate6Add (int *numberIn*, CoinFactorizationDouble *multiplier*, const CoinFactorizationDouble *COIN_RESTRICT *thisElement*, CoinFactorizationDouble *COIN_RESTRICT *region*)
- 5.74.3.34 void CoinAbcScatterUpdate7Add (int *numberIn*, CoinFactorizationDouble *multiplier*, const CoinFactorizationDouble *COIN_RESTRICT *thisElement*, CoinFactorizationDouble *COIN_RESTRICT *region*)
- 5.74.3.35 void CoinAbcScatterUpdate8Add (int *numberIn*, CoinFactorizationDouble *multiplier*, const CoinFactorizationDouble *COIN_RESTRICT *thisElement*, CoinFactorizationDouble *COIN_RESTRICT *region*)
- 5.74.3.36 void CoinAbcScatterUpdate4NAdd (int *numberIn*, CoinFactorizationDouble *multiplier*, const CoinFactorizationDouble *COIN_RESTRICT *thisElement*, CoinFactorizationDouble *COIN_RESTRICT *region*)
- 5.74.3.37 void CoinAbcScatterUpdate4NPlus1Add (int *numberIn*, CoinFactorizationDouble *multiplier*, const CoinFactorizationDouble *COIN_RESTRICT *thisElement*, CoinFactorizationDouble *COIN_RESTRICT *region*)
- 5.74.3.38 void CoinAbcScatterUpdate4NPlus2Add (int *numberIn*, CoinFactorizationDouble *multiplier*, const CoinFactorizationDouble *COIN_RESTRICT *thisElement*, CoinFactorizationDouble *COIN_RESTRICT *region*)
- 5.74.3.39 void CoinAbcScatterUpdate4NPlus3Add (int *numberIn*, CoinFactorizationDouble *multiplier*, const CoinFactorizationDouble *COIN_RESTRICT *thisElement*, CoinFactorizationDouble *COIN_RESTRICT *region*)
- 5.74.3.40 CoinFactorizationDouble ABC_INLINE CoinAbcGatherUpdate (CoinSimplexInt *number*, const CoinFactorizationDouble *COIN_RESTRICT *thisElement*, const int *COIN_RESTRICT *thisIndex*, CoinFactorizationDouble *COIN_RESTRICT *region*) [inline]

Definition at line 714 of file CoinAbcHelperFunctions.hpp.

- 5.74.3.41 void CoinAbcMultiplyIndexed (int *number*, const double *COIN_RESTRICT *multiplier*, const int *COIN_RESTRICT *thisIndex*, CoinFactorizationDouble *COIN_RESTRICT *region*)

- 5.74.3.42 void CoinAbcMultiplyIndexed (int *number*, const long double **COIN_RESTRICT multiplier*, const int **COIN_RESTRICT thisIndex*, long double **COIN_RESTRICT region*)
- 5.74.3.43 double CoinAbcMaximumAbsElement (const double * *region*, int *size*)
- 5.74.3.44 void CoinAbcMinMaxAbsElement (const double * *region*, int *size*, double & *minimum*, double & *maximum*)
- 5.74.3.45 void CoinAbcMinMaxAbsNormalValues (const double * *region*, int *size*, double & *minimum*, double & *maximum*)
- 5.74.3.46 void CoinAbcScale (double * *region*, double *multiplier*, int *size*)
- 5.74.3.47 void CoinAbcScaleNormalValues (double * *region*, double *multiplier*, double *killIfLessThanThis*, int *size*)
- 5.74.3.48 double CoinAbcMaximumAbsElementAndScale (double * *region*, double *multiplier*, int *size*)
- maximum fabs(region[i]) and then region[i]*=multiplier
- 5.74.3.49 void CoinAbcSetElements (double * *region*, int *size*, double *value*)
- 5.74.3.50 void CoinAbcMultiplyAdd (const double * *region1*, int *size*, double *multiplier1*, double * *regionChanged*, double *multiplier2*)
- 5.74.3.51 double CoinAbcInnerProduct (const double * *region1*, int *size*, const double * *region2*)
- 5.74.3.52 void CoinAbcGetNorms (const double * *region*, int *size*, double & *norm1*, double & *norm2*)
- 5.74.3.53 void CoinAbcScatterTo (const double * *regionFrom*, double * *regionTo*, const int * *index*, int *number*)
- regionTo[index[i]]=regionFrom[i]
- 5.74.3.54 void CoinAbcGatherFrom (const double * *regionFrom*, double * *regionTo*, const int * *index*, int *number*)
- regionTo[i]=regionFrom[index[i]]
- 5.74.3.55 void CoinAbcScatterZeroTo (double * *regionTo*, const int * *index*, int *number*)
- regionTo[index[i]]=0.0
- 5.74.3.56 void CoinAbcScatterToList (const double * *regionFrom*, double * *regionTo*, const int * *indexList*, const int * *indexScatter*, int *number*)
- regionTo[indexScatter[indexList[i]]]=regionFrom[indexList[i]]
- 5.74.3.57 void CoinAbcInverseSqrts (double * *array*, int *n*)
- array[i]=1.0/sqrt(array[i])
- 5.74.3.58 void CoinAbcReciprocal (double * *array*, int *n*, const double * *input*)
- 5.74.3.59 void CoinAbcMemcpyLong (double * *array*, const double * *arrayFrom*, int *size*)
- 5.74.3.60 void CoinAbcMemcpyLong (int * *array*, const int * *arrayFrom*, int *size*)
- 5.74.3.61 void CoinAbcMemcpyLong (unsigned char * *array*, const unsigned char * *arrayFrom*, int *size*)
- 5.74.3.62 void CoinAbcMemset0Long (double * *array*, int *size*)

5.74.3.63 void CoinAbcMemset0Long (int * *array*, int *size*)

5.74.3.64 void CoinAbcMemset0Long (unsigned char * *array*, int *size*)

5.74.3.65 void CoinAbcMemmove (double * *array*, const double * *arrayFrom*, int *size*)

5.74.3.66 void CoinAbcMemmove (int * *array*, const int * *arrayFrom*, int *size*)

5.74.3.67 void CoinAbcMemmove (unsigned char * *array*, const unsigned char * *arrayFrom*, int *size*)

5.74.3.68 void CoinAbcMemmoveAndZero (double * *array*, double * *arrayFrom*, int *size*)

This moves down and zeroes out end.

5.74.3.69 int CoinAbcCompact (int *numberSections*, int *alreadyDone*, double * *array*, const int * *starts*, const int * *lengths*)

This compacts several sections and zeroes out end (returns number)

5.74.3.70 int CoinAbcCompact (int *numberSections*, int *alreadyDone*, int * *array*, const int * *starts*, const int * *lengths*)

This compacts several sections (returns number)

5.75 src/config_clp_default.h File Reference

Macros

- #define CLP_VERSION "trunk"
- #define CLP_VERSION_MAJOR 9999
- #define CLP_VERSION_MINOR 9999
- #define CLP_VERSION_RELEASE 9999

5.75.1 Macro Definition Documentation

5.75.1.1 #define CLP_VERSION "trunk"

Definition at line 8 of file config_clp_default.h.

5.75.1.2 #define CLP_VERSION_MAJOR 9999

Definition at line 11 of file config_clp_default.h.

5.75.1.3 #define CLP_VERSION_MINOR 9999

Definition at line 14 of file config_clp_default.h.

5.75.1.4 #define CLP_VERSION_RELEASE 9999

Definition at line 17 of file config_clp_default.h.

5.76 src/config_default.h File Reference

```
#include "configall_system.h"
#include "config_clp_default.h"
```

Macros

- `#define COIN_CLP_CHECKLEVEL 0`
- `#define COIN_CLP_VERBOSITY 0`
- `#define COIN_HAS_COINUTILS 1`
- `#define COIN_HAS_CLP 1`

5.76.1 Macro Definition Documentation

5.76.1.1 `#define COIN_CLP_CHECKLEVEL 0`

Definition at line 14 of file `config_default.h`.

5.76.1.2 `#define COIN_CLP_VERBOSITY 0`

Definition at line 17 of file `config_default.h`.

5.76.1.3 `#define COIN_HAS_COINUTILS 1`

Definition at line 20 of file `config_default.h`.

5.76.1.4 `#define COIN_HAS_CLP 1`

Definition at line 23 of file `config_default.h`.

5.77 `src/ldiot.hpp` File Reference

```
#include "ClpSimplex.hpp"
```

Classes

- struct `IdiotResult`
for use internally
- class `Idiot`
This class implements a very silly algorithm.

Macros

- `#define OsiSolverInterface ClpSimplex`

5.77.1 Macro Definition Documentation

5.77.1.1 `#define OsiSolverInterface ClpSimplex`

Definition at line 14 of file `ldiot.hpp`.

5.78 `src/MyEventHandler.hpp` File Reference

```
#include "ClpEventHandler.hpp"
```


Classes

- class [MyEventHandler](#)

This is so user can trap events and do useful stuff.

5.79 src/MyMessageHandler.hpp File Reference

```
#include <deque>
#include "CoinPragma.hpp"
#include <stdio.h>
#include "CoinMessageHandler.hpp"
```

Classes

- class [MyMessageHandler](#)

Typedefs

- typedef std::vector< double > [StdVectorDouble](#)

5.79.1 Typedef Documentation

5.79.1.1 typedef std::vector<double> StdVectorDouble

Definition at line 23 of file MyMessageHandler.hpp.

5.80 src/OsiClp/OsiClpSolverInterface.hpp File Reference

```
#include <string>
#include <cfloat>
#include <map>
#include "ClpSimplex.hpp"
#include "ClpLinearObjective.hpp"
#include "CoinPackedMatrix.hpp"
#include "OsiSolverInterface.hpp"
#include "CoinWarmStartBasis.hpp"
#include "ClpEventHandler.hpp"
#include "ClpNode.hpp"
#include "CoinIndexedVector.hpp"
#include "CoinFinite.hpp"
```

Classes

- class [OsiClpSolverInterface](#)
Clp Solver Interface.
- class [OsiClpDisasterHandler](#)

Functions

- bool [OsiClpHasNDEBUG](#) ()
- void [OsiClpSolverInterfaceUnitTest](#) (const std::string &mpsDir, const std::string &netlibDir)

A function that tests the methods in the [OsiClpSolverInterface](#) class.

Variables

- static const double [OsiClpInfinity](#) = COIN_DBL_MAX

5.80.1 Function Documentation

5.80.1.1 bool [OsiClpHasNDEBUG](#) ()

5.80.1.2 void [OsiClpSolverInterfaceUnitTest](#) (const std::string & *mpsDir*, const std::string & *netlibDir*)

A function that tests the methods in the [OsiClpSolverInterface](#) class.

5.80.2 Variable Documentation

5.80.2.1 const double [OsiClpInfinity](#) = COIN_DBL_MAX [static]

Definition at line 28 of file [OsiClpSolverInterface.hpp](#).

Index

- ~AbcDualRowDantzig
 - AbcDualRowDantzig, [17](#)
- ~AbcDualRowPivot
 - AbcDualRowPivot, [20](#)
- ~AbcDualRowSteepest
 - AbcDualRowSteepest, [24](#)
- ~AbcMatrix
 - AbcMatrix, [31](#)
- ~AbcMatrix2
 - AbcMatrix2, [42](#)
- ~AbcMatrix3
 - AbcMatrix3, [45](#)
- ~AbcNonLinearCost
 - AbcNonLinearCost, [48](#)
- ~AbcPrimalColumnDantzig
 - AbcPrimalColumnDantzig, [52](#)
- ~AbcPrimalColumnPivot
 - AbcPrimalColumnPivot, [54](#)
- ~AbcPrimalColumnSteepest
 - AbcPrimalColumnSteepest, [59](#)
- ~AbcSimplex
 - AbcSimplex, [75](#)
- ~AbcSimplexFactorization
 - AbcSimplexFactorization, [112](#)
- ~AbcTolerancesEtc
 - AbcTolerancesEtc, [124](#)
- ~AbcWarmStart
 - AbcWarmStart, [128](#)
- ~AbcWarmStartOrganizer
 - AbcWarmStartOrganizer, [132](#)
- ~CbcOrClpParam
 - CbcOrClpParam, [143](#)
- ~ClpCholeskyBase
 - ClpCholeskyBase, [150](#)
- ~ClpCholeskyDense
 - ClpCholeskyDense, [158](#)
- ~ClpCholeskyMumps
 - ClpCholeskyMumps, [162](#)
- ~ClpCholeskyTaucs
 - ClpCholeskyTaucs, [164](#)
- ~ClpCholeskyUfl
 - ClpCholeskyUfl, [166](#)
- ~ClpCholeskyWssmp
 - ClpCholeskyWssmp, [167](#)
- ~ClpCholeskyWssmpKKT
 - ClpCholeskyWssmpKKT, [169](#)
- ~ClpConstraint
 - ClpConstraint, [172](#)
- ~ClpConstraintLinear
 - ClpConstraintLinear, [176](#)
- ~ClpConstraintQuadratic
 - ClpConstraintQuadratic, [179](#)
- ~ClpDataSave
 - ClpDataSave, [181](#)
- ~ClpDisasterHandler
 - ClpDisasterHandler, [184](#)
- ~ClpDualRowDantzig
 - ClpDualRowDantzig, [186](#)
- ~ClpDualRowPivot
 - ClpDualRowPivot, [188](#)
- ~ClpDualRowSteepest
 - ClpDualRowSteepest, [192](#)
- ~ClpDummyMatrix
 - ClpDummyMatrix, [196](#)
- ~ClpDynamicExampleMatrix
 - ClpDynamicExampleMatrix, [203](#)
- ~ClpDynamicMatrix
 - ClpDynamicMatrix, [211](#)
- ~ClpEventHandler
 - ClpEventHandler, [224](#)
- ~ClpFactorization
 - ClpFactorization, [228](#)
- ~ClpGubDynamicMatrix
 - ClpGubDynamicMatrix, [237](#)
- ~ClpGubMatrix
 - ClpGubMatrix, [247](#)
- ~ClpHashValue
 - ClpHashValue, [257](#)
- ~ClpInterior
 - ClpInterior, [265](#)
- ~ClpLinearObjective
 - ClpLinearObjective, [281](#)
- ~ClpLsqr
 - ClpLsqr, [284](#)
- ~ClpMatrixBase
 - ClpMatrixBase, [291](#)
- ~ClpModel
 - ClpModel, [315](#)
- ~ClpNetworkBasis
 - ClpNetworkBasis, [344](#)
- ~ClpNetworkMatrix
 - ClpNetworkMatrix, [348](#)
- ~ClpNode
 - ClpNode, [356](#)
- ~ClpNodeStuff
 - ClpNodeStuff, [362](#)
- ~ClpNonLinearCost
 - ClpNonLinearCost, [368](#)
- ~ClpObjective
 - ClpObjective, [373](#)
- ~ClpPackedMatrix
 - ClpPackedMatrix, [380](#)

- ~ClpPackedMatrix2
 - ClpPackedMatrix2, [391](#)
- ~ClpPackedMatrix3
 - ClpPackedMatrix3, [393](#)
- ~ClpPdcoBase
 - ClpPdcoBase, [398](#)
- ~ClpPlusMinusOneMatrix
 - ClpPlusMinusOneMatrix, [402](#)
- ~ClpPresolve
 - ClpPresolve, [414](#)
- ~ClpPrimalColumnDantzig
 - ClpPrimalColumnDantzig, [419](#)
- ~ClpPrimalColumnPivot
 - ClpPrimalColumnPivot, [422](#)
- ~ClpPrimalColumnSteepest
 - ClpPrimalColumnSteepest, [427](#)
- ~ClpPrimalQuadraticDantzig
 - ClpPrimalQuadraticDantzig, [431](#)
- ~ClpQuadraticObjective
 - ClpQuadraticObjective, [433](#)
- ~ClpSimplex
 - ClpSimplex, [453](#)
- ~ClpSimplexProgress
 - ClpSimplexProgress, [507](#)
- ~ClpSolve
 - ClpSolve, [513](#)
- ~CoinAbcAnyFactorization
 - CoinAbcAnyFactorization, [521](#)
- ~CoinAbcDenseFactorization
 - CoinAbcDenseFactorization, [533](#)
- ~CoinAbcTypeFactorization
 - CoinAbcTypeFactorization, [551](#)
- ~Idiot
 - Idiot, [582](#)
- ~MyEventHandler
 - MyEventHandler, [588](#)
- ~MyMessageHandler
 - MyMessageHandler, [590](#)
- ~OsiClpDisasterHandler
 - OsiClpDisasterHandler, [594](#)
- ~OsiClpSolverInterface
 - OsiClpSolverInterface, [608](#)
- a
 - ClpCholeskyDenseC, [160](#)
- ABC_DENSE_CODE
 - CoinAbcCommon.hpp, [705](#)
- ABC_EXPONENT
 - CoinAbcCommon.hpp, [705](#)
- ABC_INLINE
 - CoinAbcCommon.hpp, [703](#)
- ABC_INSTRUMENT
 - CoinAbcCommon.hpp, [703](#)
- ABC_INTEL
 - CoinAbcCommon.hpp, [704](#)
- ABC_NUMBER_USEFUL
 - AbcSimplex.hpp, [646](#)
- ABC_PARALLEL
 - CoinAbcCommon.hpp, [703](#)
- ABC_SMALL
 - CoinAbcFactorization.hpp, [710](#)
- ADD_A_BIT
 - AbcSimplex.hpp, [645](#)
- ALL_SAME
 - ClpModel.hpp, [690](#)
- ALL_STATUS_OK
 - AbcSimplex.hpp, [645](#)
- aMatrix
 - ClpCholeskyDense, [159](#)
- AbcDualRowSteepest
 - keep, [24](#)
 - normal, [24](#)
- AbcPrimalColumnSteepest
 - keep, [59](#)
 - normal, [59](#)
- AbcSimplex
 - atLowerBound, [74](#)
 - atUpperBound, [74](#)
 - basic, [74](#)
 - bothFake, [74](#)
 - isFixed, [74](#)
 - isFree, [74](#)
 - lowerFake, [74](#)
 - noFake, [74](#)
 - superBasic, [74](#)
 - upperFake, [74](#)
- abc_assert
 - CoinAbcHelperFunctions.hpp, [714](#)
- abcBaseModel_
 - AbcSimplex, [100](#)
- abcCost_
 - AbcSimplex, [98](#)
- abcDj_
 - AbcSimplex, [98](#)
- AbcDualRowDantzig, [15](#)
 - ~AbcDualRowDantzig, [17](#)
 - AbcDualRowDantzig, [17](#)
 - AbcDualRowDantzig, [17](#)
 - clone, [18](#)
 - operator=, [18](#)
 - pivotRow, [17](#)
 - recomputeInfeasibilities, [18](#)
 - saveWeights, [18](#)
 - updatePrimalSolution, [17](#)
 - updateWeights, [17](#)
 - updateWeights1, [17](#)
 - updateWeights2, [17](#)
 - updateWeightsOnly, [17](#)

- AbcDualRowPivot, 18
 - ~AbcDualRowPivot, 20
 - AbcDualRowPivot, 20
 - AbcDualRowPivot, 20
 - checkAccuracy, 21
 - clearArrays, 21
 - clone, 21
 - looksOptimal, 21
 - model, 21
 - model_, 22
 - operator=, 21
 - pivotRow, 20
 - recomputeInfeasibilities, 21
 - saveWeights, 21
 - setModel, 21
 - type, 21
 - type_, 22
 - updatePrimalSolution, 20
 - updatePrimalSolutionAndWeights, 20
 - updateWeights, 20
 - updateWeights1, 20
 - updateWeights2, 20
 - updateWeightsOnly, 20
- abcDualRowPivot_
 - AbcSimplex, 100
- AbcDualRowSteepest, 22
 - ~AbcDualRowSteepest, 24
 - AbcDualRowSteepest, 24
 - AbcDualRowSteepest, 24
 - clearArrays, 25
 - clone, 25
 - fill, 25
 - infeasible, 26
 - looksOptimal, 25
 - mode, 26
 - model, 26
 - operator=, 25
 - Persistence, 24
 - persistence, 26
 - pivotRow, 24
 - recomputeInfeasibilities, 25
 - saveWeights, 25
 - setPersistence, 26
 - updatePrimalSolution, 25
 - updatePrimalSolutionAndWeights, 25
 - updateWeights, 24
 - updateWeights1, 24
 - updateWeights2, 25
 - updateWeightsOnly, 24
 - weights, 26
- AbcDualRowSteepest.hpp
 - DEVEX_ADD_ONE, 639
 - DEVEX_TRY_NORM, 639
- abcFactorization_
 - AbcSimplex, 100
- abcLower_
 - AbcSimplex, 98
- AbcMatrix, 26
 - ~AbcMatrix, 31
 - AbcMatrix, 31
 - AbcMatrix, 31
 - add, 34
 - blockStart, 38, 39
 - blockStart_, 40
 - chooseBestDj, 36
 - column_, 39
 - copy, 39
 - countBasis, 33
 - createRowCopy, 33
 - currentWanted, 38
 - currentWanted_, 40
 - dualColumn1, 35
 - dualColumn1 Part, 36
 - dualColumn1 Row, 35
 - dualColumn1 Row1, 35
 - dualColumn1 Row2, 35
 - dualColumn1 RowFew, 35
 - element_, 39
 - endFraction, 38
 - endFraction_, 40
 - fillBasis, 33
 - getElements, 32
 - getIndices, 32
 - getMutableElements, 32
 - getMutableIndices, 32
 - getMutableVectorLengths, 33
 - getMutableVectorStarts, 33
 - getNumCols, 32
 - getNumElements, 32
 - getNumRows, 32
 - getPackedMatrix, 31
 - getVectorLengths, 33
 - getVectorStarts, 32
 - gotRowCopy, 39
 - inOutUseful, 34
 - isColOrdered, 32
 - makeAllUseful, 34
 - matrix, 37
 - matrix_, 39
 - minimumGoodReducedCosts, 37
 - minimumGoodReducedCosts_, 41
 - minimumObjectsScan, 37
 - minimumObjectsScan_, 41
 - model_, 39
 - moveLargestToStart, 34
 - numberColumnBlocks, 39
 - numberColumnBlocks_, 40
 - numberRowBlocks, 39

- numberRowBlocks_, 40
- operator=, 39
- originalWanted, 38
- originalWanted_, 40
- partialPricing, 37
- pivotColumnDantzig, 36
- primalColumnDouble, 36
- primalColumnRow, 36
- primalColumnRowAndDjs, 36
- primalColumnSparseDouble, 36
- primalColumnSubset, 37
- putIntoUseful, 34
- rebalance, 36
- reverseOrderedCopy, 33
- rowColumns, 33
- rowElements, 33
- rowEnd, 33
- rowStart, 33
- rowStart_, 39
- savedBestDj, 38
- savedBestDj_, 40
- savedBestSequence, 38
- savedBestSequence_, 40
- scale, 33
- setCurrentWanted, 38
- setEndFraction, 38
- setMinimumGoodReducedCosts, 37
- setMinimumObjectsScan, 37
- setModel, 32
- setOriginalWanted, 38
- setSavedBestDj, 38
- setSavedBestSequence, 38
- setStartFraction, 37
- sortUseful, 34
- startColumnBlock, 38
- startColumnBlock_, 40
- startFraction, 37
- startFraction_, 40
- subsetTransposeTimes, 37
- takeOutOfUseful, 33
- timesIncludingSlacks, 34
- timesModifyExcludingSlacks, 34
- timesModifyIncludingSlacks, 34
- transposeTimes, 37
- transposeTimesAll, 35
- transposeTimesBasic, 35
- transposeTimesNonBasic, 34, 35
- unpack, 34
- abcMatrix
 - AbcSimplex, 79
- AbcMatrix.hpp
 - NUMBER_ROW_BLOCKS, 640
- AbcMatrix2, 41
 - ~AbcMatrix2, 42
- AbcMatrix2, 42
 - AbcMatrix2, 42
 - column_, 43
 - count_, 43
 - numberBlocks_, 43
 - numberRows_, 43
 - offset_, 43
 - operator=, 42
 - rowStart_, 43
 - transposeTimes, 42
 - usefulInfo, 42
 - work_, 43
- AbcMatrix3, 43
 - ~AbcMatrix3, 45
 - AbcMatrix3, 45
 - AbcMatrix3, 45
 - block_, 46
 - column_, 45
 - element_, 46
 - numberBlocks_, 45
 - numberColumns_, 45
 - operator=, 45
 - row_, 46
 - sortBlocks, 45
 - start_, 46
 - swapOne, 45
 - transposeTimes, 45
 - transposeTimes2, 45
- abcMatrix_
 - AbcSimplex, 98
- AbcNonLinearCost, 46
 - ~AbcNonLinearCost, 48
 - AbcNonLinearCost, 48
 - AbcNonLinearCost, 48
 - averageTheta, 50
 - changeDownInCost, 50
 - changeInCost, 49, 50
 - changeUpInCost, 49
 - checkChanged, 48
 - checkInfeasibilities, 48
 - feasibleBounds, 49
 - feasibleCost, 50
 - feasibleReportCost, 50
 - getCurrentStatus, 51
 - goBack, 49
 - goBackAll, 49
 - goThru, 48
 - largestInfeasibility, 50
 - nearest, 49
 - numberInfeasibilities, 50
 - operator=, 48
 - refresh, 49
 - refreshCosts, 49
 - refreshFromPerturbed, 49

- setAverageTheta, 50
 - setChangeInCost, 50
 - setOne, 49
 - setOneBasic, 49
 - setOneOutgoing, 49
 - statusArray, 50
 - sumInfeasibilities, 50
 - validate, 51
 - zapCosts, 49
- abcNonLinearCost
 - AbcSimplex, 78
- AbcNonLinearCost.hpp
 - CLP_ABOVE_UPPER, 641
 - CLP_BELOW_LOWER, 641
 - CLP_FEASIBLE, 641
 - CLP_SAME, 641
 - currentStatus, 641
 - originalStatus, 641
 - setCurrentStatus, 642
 - setInitialStatus, 642
 - setOriginalStatus, 642
 - setSameStatus, 642
- abcNonLinearCost_
 - AbcSimplex, 100
- abcPerturbation
 - AbcSimplex, 85
- abcPerturbation_
 - AbcSimplex, 98
- abcPivotVariable_
 - AbcSimplex, 100
- AbcPrimalColumnDantzig, 51
 - ~AbcPrimalColumnDantzig, 52
 - AbcPrimalColumnDantzig, 52
 - AbcPrimalColumnDantzig, 52
 - clone, 52
 - operator=, 52
 - pivotColumn, 52
 - saveWeights, 52
- AbcPrimalColumnPivot, 53
 - ~AbcPrimalColumnPivot, 54
 - AbcPrimalColumnPivot, 54
 - AbcPrimalColumnPivot, 54
 - clearArrays, 55
 - clone, 55
 - looksOptimal, 55
 - looksOptimal_, 56
 - maximumPivotsChanged, 56
 - model, 56
 - model_, 56
 - numberSprintColumns, 56
 - operator=, 55
 - pivotColumn, 54
 - pivotRow, 55
 - saveWeights, 55
 - setLooksOptimal, 55
 - setModel, 56
 - switchOffSprint, 56
 - type, 56
 - type_, 56
 - updateWeights, 55
- abcPrimalColumnPivot_
 - AbcSimplex, 100
- AbcPrimalColumnSteepest, 57
 - ~AbcPrimalColumnSteepest, 59
 - AbcPrimalColumnSteepest, 59
 - AbcPrimalColumnSteepest, 59
 - checkAccuracy, 60
 - clearArrays, 60
 - clone, 61
 - djsAndDevex, 59
 - djsAndDevex2, 59
 - doSteepestWork, 60
 - initializeWeights, 60
 - justDevex, 59
 - justDjs, 59
 - looksOptimal, 60
 - maximumPivotsChanged, 60
 - mode, 60
 - operator=, 60
 - partialPricing, 59
 - Persistence, 58
 - persistence, 61
 - pivotColumn, 59
 - reference, 61
 - saveWeights, 60
 - setPersistence, 61
 - setReference, 61
 - unrollWeights, 60
 - updateWeights, 60
- abcProgress
 - AbcSimplex, 88
- abcProgress_
 - AbcSimplex, 100
- AbcSimplex, 61
 - ~AbcSimplex, 75
 - abcBaseModel_, 100
 - abcCost_, 98
 - abcDj_, 98
 - abcDualRowPivot_, 100
 - abcFactorization_, 100
 - abcLower_, 98
 - abcMatrix, 79
 - abcMatrix_, 98
 - abcNonLinearCost, 78
 - abcNonLinearCost_, 100
 - abcPerturbation, 85
 - abcPerturbation_, 98
 - abcPivotVariable_, 100

abcPrimalColumnPivot_, 100
 abcProgress, 88
 abcProgress_, 100
 AbcSimplex, 74, 75
 AbcSimplexUnitTest, 92
 abcSolution_, 98
 abcUpper_, 98
 AbcSimplex, 74, 75
 AbcWarmStart.hpp, 648
 acceptablePivot, 78
 active, 90
 allSlackBasis, 90
 arrayForBtran, 83
 arrayForBtran_, 101
 arrayForDualColumn, 83
 arrayForDualColumn_, 101
 arrayForFlipBounds, 83
 arrayForFlipBounds_, 101
 arrayForFlipRhs, 83
 arrayForFlipRhs_, 101
 arrayForFtran, 83
 arrayForFtran_, 101
 arrayForReplaceColumn, 83
 arrayForReplaceColumn_, 101
 arrayForTableauRow, 83
 arrayForTableauRow_, 101
 atFakeBound, 89
 baseModel, 75
 btranAlpha_, 94
 checkArrays, 81
 checkBothSolutions, 80
 checkConsistentPivots, 90
 checkDjs, 81
 checkDualSolution, 80
 checkDualSolutionPlusFake, 80
 checkMoveBack, 81
 checkPrimalSolution, 80
 checkSolutionBasic, 81
 cleanFactorization, 81
 cleanStatus, 79
 clearActive, 90
 clearArrays, 88
 clearArraysPublic, 88
 clearFlagged, 89
 clearPivoted, 89
 clpModel_, 100
 clpObjectiveValue, 82
 columnScale2, 83
 columnUseScale_, 97
 computeDuals, 80
 computeInternalObjectiveValue, 91
 computeObjective, 80
 computeObjectiveValue, 90
 computePrimals, 80
 copyFromSaved, 81
 cost, 88
 costAddress, 88
 costBasic, 85
 costBasic_, 99
 costRegion, 84, 85
 costSaved_, 99
 crash, 90
 createStatus, 90
 currentAcceptablePivot, 87
 currentAcceptablePivot_, 94
 currentDualBound, 78
 currentDualBound_, 93
 currentDualTolerance, 78
 currentDualTolerance_, 93
 defaultFactorizationFrequency, 76
 deleteBaseModel, 75
 djBasic, 85
 djBasic_, 99
 djRegion, 84, 85
 djSaved_, 99
 doAbcDual, 76
 doAbcPrimal, 76
 dual, 76
 dualRowPivot, 78
 factorization, 76
 factorizationFrequency, 76
 FakeBound, 74
 fakeDjs, 85
 fakeSuperBasic, 87
 fillPerturbation, 81
 firstFree, 87
 flagged, 90
 freeSequenceIn, 87
 freeSequenceIn_, 95
 ftAlpha_, 94
 getAvailableArray, 88
 getAvailableArrayPublic, 88
 getBasis, 76
 getColSolution, 83
 getEmptyFactorization, 76
 getFakeBound, 89
 getInternalColumnStatus, 86
 getInternalStatus, 85
 getReducedCost, 83
 getRowActivity, 83
 getRowPrice, 83
 getSolution, 77
 gutsOfCopy, 84
 gutsOfDelete, 84
 gutsOfInitialize, 84
 gutsOfPrimalSolution, 80
 gutsOfResize, 84
 gutsOfSolution, 80, 84

housekeeping, 80
initialDenseFactorization, 86
initialNumberInfeasibilities_, 97
initialSumInfeasibilities_, 94
internalFactorize, 79
internalStatus, 85
internalStatus_, 97
internalStatusSaved_, 98
inverseColumnScale2, 82
inverseColumnUseScale_, 97
inverseRowScale2, 82
isColumn, 86
isObjectiveLimitTestValid, 77
largestGap_, 93
lastCleaned_, 96
lastDualBound_, 93
lastDualError_, 94
lastFirstFree, 87
lastFirstFree_, 95
lastPivotRow, 86
lastPivotRow_, 97
lastPrimalError_, 94
lower, 87
lowerAddress, 87
lowerBasic, 85
lowerBasic_, 99
lowerRegion, 84, 85
lowerSaved_, 99
makeBaseModel, 75
maximumAbcNumberColumns_, 95
maximumAbcNumberRows, 77
maximumAbcNumberRows_, 95
maximumNumberTotal, 77
maximumNumberTotal_, 95
maximumTotal, 77
minimizationObjectiveValue, 78
minimumThetaMovement_, 94
moveInfo, 91
moveStatusFromClp, 82
moveStatusToClp, 82
moveToBasic, 84
movement_, 94
multipleSequenceIn_, 101
normalDualColumnIteration_, 95
numberDisasters_, 101
numberFlagged_, 95
numberFlipped_, 101
numberFreeNonBasic_, 96
numberOrdinary, 78
numberOrdinary_, 96
numberTotal, 77
numberTotal_, 95
numberTotalWithoutFixed, 77
numberTotalWithoutFixed_, 95
objectiveChange_, 94
objectiveOffset_, 93
offset_, 97
offsetRhs_, 97
operator=, 75
ordinaryVariables, 78
ordinaryVariables_, 96
originalLower, 88
originalModel, 75
originalUpper, 88
permuteBasis, 79
permuteIn, 79
permuteOut, 79
perturbationBasic_, 98
perturbationFactor_, 93
perturbationSaved, 78
perturbationSaved_, 98
pivotVariable, 82
pivoted, 89
primal, 76
primalColumnPivot, 79
printStuff, 90
putBackSolution, 75
putStuffInBasis, 90
rawObjectiveValue, 90
rawObjectiveValue_, 93
reducedCost, 87
reducedCostAddress, 77
refreshCosts, 81
refreshLower, 81
refreshUpper, 81
resize, 92
restoreData, 79
restoreGoodStatus, 81
reversePivotVariable_, 100
rowScale2, 82
saveData, 79
saveData_, 101
saveGoodStatus, 81
scaleFromExternal, 82
scaleFromExternal_, 97
scaleToExternal, 82
scaleToExternal_, 97
sequenceIn, 86
sequenceOut, 86
sequenceWithin, 86
setActive, 90
setAvailableArray, 89
setClpSimplexObjectiveValue, 77
setColBounds, 92
setColLower, 91
setColSetBounds, 92
setColUpper, 91
setColumnBounds, 91

- setColumnLower, [91](#)
- setColumnSetBounds, [91](#)
- setColumnUpper, [91](#)
- setCurrentDualTolerance, [78](#)
- setDualRowPivotAlgorithm, [76](#)
- setFactorization, [76](#)
- setFactorizationFrequency, [77](#)
- setFakeBound, [89](#)
- setFlagged, [89](#)
- setInitialDenseFactorization, [86](#)
- setInternalColumnStatus, [86](#)
- setInternalStatus, [86](#)
- setMultipleSequenceIn, [80](#)
- setNumberOrdinary, [78](#)
- setObjCoeff, [91](#)
- setObjectiveCoefficient, [91](#)
- setPivoted, [89](#)
- setPrimalColumnPivotAlgorithm, [76](#)
- setRowBounds, [92](#)
- setRowLower, [92](#)
- setRowSetBounds, [92](#)
- setRowUpper, [92](#)
- setSequenceIn, [86](#)
- setSequenceOut, [86](#)
- setStateOfProblem, [82](#)
- setToBaseModel, [75](#)
- setUsedArray, [89](#)
- setValuesPassAction, [81](#)
- setupDualValuesPass, [77](#)
- setupPointers, [81](#)
- solution, [87](#)
- solutionAddress, [87](#)
- solutionBasic, [85](#)
- solutionBasic_, [99](#)
- solutionRegion, [84](#)
- solutionSaved_, [99](#)
- startAtLowerOther_, [96](#)
- startAtUpperNoOther_, [96](#)
- startAtUpperOther_, [96](#)
- startFixed_, [96](#)
- startOther_, [96](#)
- startup, [90](#)
- stateDualColumn_, [95](#)
- stateOfIteration_, [94](#)
- stateOfProblem, [82](#)
- stateOfProblem_, [96](#)
- Status, [74](#)
- sumFakeInfeasibilities_, [93](#)
- sumNonBasicCosts_, [93](#)
- swap, [89](#)
- swapDualStuff, [89](#)
- swapFactorization, [76](#)
- swapPrimalStuff, [89](#)
- swappedAlgorithm_, [97](#)
- tempArray_, [97](#)
- tightenPrimalBounds, [76](#)
- translate, [84](#)
- unpack, [80](#)
- upper, [88](#)
- upperAddress, [88](#)
- upperBasic, [85](#)
- upperBasic_, [99](#)
- upperRegion, [84](#), [85](#)
- upperSaved_, [99](#)
- upperTheta, [83](#)
- upperTheta_, [101](#)
- usefulArray, [77](#)
- usefulArray_, [100](#)
- valueIncomingDual, [83](#)
- AbcSimplex.hpp
 - ABC_NUMBER_USEFUL, [646](#)
 - ADD_A_BIT, [645](#)
 - ALL_STATUS_OK, [645](#)
 - AbcSimplexUnitTest, [646](#)
 - COLUMN_DUAL_OK, [645](#)
 - COLUMN_PRIMAL_OK, [645](#)
 - DO_JUST_BOUNDS, [646](#)
 - DO_SOLUTION, [646](#)
 - DO_STATUS, [646](#)
 - FAKE_SUPERBASIC, [646](#)
 - HEAVY_PERTURBATION, [644](#)
 - inverseRowUseScale_, [645](#)
 - NEED_BASIS_SORT, [646](#)
 - NUMBER_THREADS, [645](#)
 - PAN, [644](#)
 - PESSIMISTIC, [645](#)
 - ROW_DUAL_OK, [645](#)
 - ROW_PRIMAL_OK, [645](#)
 - rowUseScale_, [644](#)
 - startAtLowerNoOther_, [645](#)
 - TRY_ABC_GUS, [644](#)
 - VALUES_PASS, [646](#)
 - VALUES_PASS2, [646](#)
- AbcSimplexDual, [102](#)
 - bounceTolerances, [108](#)
 - changeBound, [107](#)
 - changeBounds, [107](#)
 - checkCutoff, [108](#)
 - checkPossibleCleanup, [107](#)
 - checkReplace, [106](#)
 - checkReplacePart1, [106](#)
 - checkReplacePart1a, [106](#)
 - checkReplacePart1b, [106](#)
 - checkUnbounded, [107](#)
 - cleanupAfterStrongBranching, [105](#)
 - createDualPricingVectorSerial, [106](#)
 - dual, [104](#)
 - dualColumn1, [106](#)

- dualColumn1A, [106](#)
- dualColumn1B, [106](#)
- dualColumn2, [107](#)
- dualColumn2First, [107](#)
- dualColumn2Most, [107](#)
- dualPivotColumn, [106](#)
- dualPivotRow, [107](#)
- fastDual, [108](#)
- finishSolve, [108](#)
- flipBack, [106](#)
- flipBounds, [106](#)
- getTableauColumnFlipAndStartReplaceSerial, [106](#)
- getTableauColumnPart1Serial, [106](#)
- getTableauColumnPart2, [106](#)
- gutsOfDual, [108](#)
- makeNonFreeVariablesDualFeasible, [108](#)
- nextSuperBasic, [108](#)
- noPivotColumn, [106](#)
- noPivotRow, [106](#)
- numberAtFakeBound, [108](#)
- originalBound, [107](#)
- perturb, [108](#)
- perturbB, [108](#)
- pivotResultPart1, [108](#)
- replaceColumnPart3, [106](#)
- resetFakeBounds, [108](#)
- setupForStrongBranching, [105](#)
- startupSolve, [108](#)
- statusOfProblemInDual, [107](#)
- strongBranching, [105](#)
- updateDualsInDual, [106](#)
- updatePrimalSolution, [106](#)
- whatNext, [107](#)
- whileIterating2, [105](#)
- whileIterating3, [106](#)
- whileIteratingParallel, [105](#)
- whileIteratingSerial, [105](#)
- AbcSimplexFactorization, [109](#)
 - ~AbcSimplexFactorization, [112](#)
 - AbcSimplexFactorization, [112](#)
 - AbcSimplexFactorization, [112](#)
 - almostDestructor, [116](#)
 - areaFactor, [115](#)
 - checkMarkArrays, [117](#)
 - checkReplacePart1, [112](#)
 - checkReplacePart1a, [112](#)
 - checkReplacePart1b, [112](#)
 - checkReplacePart2, [113](#)
 - clearArrays, [115](#)
 - factorization, [118](#)
 - factorize, [112](#)
 - forceOtherFactorization, [116](#)
 - getDenseThreshold, [116](#)
 - goDenseOrSmall, [116](#)
 - goDenseThreshold, [117](#)
 - goLongThreshold, [117](#)
 - goSmallThreshold, [117](#)
 - goSparse, [117](#)
 - maximumPivots, [114](#)
 - minimumPivotTolerance, [116](#)
 - needToReorder, [117](#)
 - numberDense, [115](#)
 - numberElements, [114](#)
 - numberRows, [116](#)
 - numberSlacks, [116](#)
 - operator=, [112](#)
 - pivotRegion, [116](#)
 - pivotTolerance, [116](#)
 - pivots, [114](#)
 - replaceColumnPart3, [113](#)
 - saferTolerances, [115](#)
 - setDenseThreshold, [116](#)
 - setFactorization, [112](#)
 - setGoDenseThreshold, [117](#)
 - setGoLongThreshold, [117](#)
 - setGoSmallThreshold, [117](#)
 - setPivots, [115](#)
 - setStatus, [115](#)
 - status, [115](#)
 - synchronize, [117](#)
 - timeToRefactorize, [115](#)
 - typeOfFactorization, [117](#)
 - updateColumn, [113](#)
 - updateColumnCpu, [114](#)
 - updateColumnFT, [113](#)
 - updateColumnFTPart1, [113](#)
 - updateColumnFTPart2, [113](#)
 - updateColumnTranspose, [113](#)
 - updateColumnTransposeCpu, [114](#)
 - updateFullColumn, [114](#)
 - updateFullColumnTranspose, [114](#)
 - updateTwoColumnsFT, [113](#)
 - updateWeights, [114](#)
 - usingFT, [114](#)
 - zeroTolerance, [115](#)
- AbcSimplexPrimal, [118](#)
 - alwaysOptimal, [121](#)
 - checkUnbounded, [122](#)
 - clearAll, [123](#)
 - createUpdateDuals, [122](#)
 - doFTUpdate, [122](#)
 - exactOutgoing, [121](#)
 - lexSolve, [123](#)
 - nextSuperBasic, [123](#)
 - perturb, [123](#)
 - pivotResult, [121](#)
 - pivotResult4, [121](#)
 - primal, [120](#)

- primalColumn, 122
- primalRay, 123
- primalRow, 122
- statusOfProblemInPrimal, 122
- unPerturb, 123
- unflag, 123
- updateMinorCandidate, 122
- updatePartialUpdate, 122
- updatePrimalsInPrimal, 121, 122
- whileIterating, 121
- AbcSimplexPrimal::pivotStruct, 636
 - alpha_, 637
 - directionIn_, 637
 - directionOut_, 638
 - dualIn_, 637
 - dualOut_, 637
 - lowerIn_, 637
 - lowerOut_, 637
 - pivotRow_, 638
 - saveDualIn_, 637
 - sequenceIn_, 637
 - sequenceOut_, 637
 - theta_, 637
 - upperIn_, 637
 - upperOut_, 637
 - valueIn_, 637
 - valueOut_, 637
 - valuesPass_, 638
- AbcSimplexUnitTest
 - AbcSimplex, 92
 - AbcSimplex.hpp, 646
- abcSolution_
 - AbcSimplex, 98
- abcState
 - ClpSimplex, 453
- abcState_
 - ClpSimplex, 488
- AbcTolerancesEtc, 123
 - ~AbcTolerancesEtc, 124
 - AbcTolerancesEtc, 124
 - AbcTolerancesEtc, 124
 - allowedInfeasibility_, 126
 - alphaAccuracy_, 125
 - baseIteration_, 126
 - dontFactorizePivots_, 126
 - dualBound_, 125
 - dualTolerance_, 125
 - forceFactorization_, 126
 - incomingInfeasibility_, 125
 - infeasibilityCost_, 125
 - largeValue_, 125
 - maximumPivots_, 126
 - numberRefinements_, 126
 - operator=, 125
 - perturbation_, 126
 - primalTolerance_, 125
 - primalToleranceToGetOptimal_, 125
 - zeroTolerance_, 125
- abcUpper_
 - AbcSimplex, 98
- AbcWarmStart, 126
 - ~AbcWarmStart, 128
 - AbcWarmStart, 128
 - AbcWarmStart, 128
 - assignBasisStatus, 130
 - clone, 130
 - compressRows, 129
 - createBasis0, 130
 - createBasis12, 130
 - createBasis34, 130
 - deleteColumns, 129
 - deleteRows, 129
 - extraInformation_, 131
 - lengthExtraInformation_, 130
 - model, 130
 - model_, 131
 - nextBasis_, 131
 - numberValidRows_, 131
 - operator=, 130
 - organizer_, 131
 - previousBasis_, 131
 - resize, 129
 - setModel, 129
 - setSize, 129
 - stamp_, 131
 - typeExtraInformation_, 130
- AbcWarmStart.hpp
 - AbcSimplex, 648
 - CLP_WARMSTART, 648
- AbcWarmStartOrganizer, 131
 - ~AbcWarmStartOrganizer, 132
 - AbcWarmStartOrganizer, 132
 - AbcWarmStartOrganizer, 132
 - createBasis0, 133
 - createBasis12, 133
 - createBasis34, 133
 - deleteBasis, 133
 - firstBasis_, 133
 - lastBasis_, 133
 - model_, 133
 - numberBases_, 133
 - operator=, 133
 - sizeBases_, 133
- acceptableMaxTheta
 - ClpSimplexOther::parametricsData, 635
- acceptablePivot
 - AbcSimplex, 78
- acceptablePivot_

- ClpDataSave, 182
- ClpSimplex, 482
- activated
 - ClpObjective, 375
- activated_
 - ClpObjective, 375
- active
 - AbcSimplex, 90
 - ClpSimplex, 474
- actualDualStep_
 - ClpInterior, 276
- actualPrimalStep_
 - ClpInterior, 276
- add
 - AbcMatrix, 34
 - ClpDummyMatrix, 198
 - ClpGubMatrix, 249
 - ClpMatrixBase, 294, 295
 - ClpNetworkMatrix, 351
 - ClpPackedMatrix, 384
 - ClpPlusMinusOneMatrix, 406
 - ClpSimplex, 465
- addCol
 - OsiClpSolverInterface, 620
- addCols
 - OsiClpSolverInterface, 620
- addColumn
 - ClpDynamicMatrix, 213
 - ClpModel, 318
- addColumnns
 - ClpModel, 318
- addHelp
 - CbcOrClpParam, 143
- addLink
 - CoinAbcTypeFactorization, 561
- addRow
 - ClpModel, 317
 - OsiClpSolverInterface, 620
- addRows
 - ClpModel, 317
 - OsiClpSolverInterface, 620, 621
- addValue
 - ClpHashValue, 257
- adjustedAreaFactor
 - ClpFactorization, 232
 - CoinAbcTypeFactorization, 553
- affineProduct
 - ClpPredictorCorrector, 412
- afterCrunch
 - ClpSimplexOther, 500
- afterPivot
 - CoinAbcTypeFactorization, 561
- algorithm
 - ClpInterior, 267
 - ClpSimplex, 461
- algorithm_
 - ClpInterior, 279
 - ClpSimplex, 486
- allElementsInRange
 - ClpMatrixBase, 293
 - ClpPackedMatrix, 383
- allSlackBasis
 - AbcSimplex, 90
 - ClpSimplex, 475
- allowedInfeasibility_
 - AbcTolerancesEtc, 126
 - ClpSimplex, 488
- almostDestructor
 - AbcSimplexFactorization, 116
 - ClpFactorization, 232
 - CoinAbcTypeFactorization, 551
- alpha
 - ClpSimplex, 468
- alpha_
 - AbcSimplexPrimal::pivotStruct, 637
 - ClpSimplex, 481
- alphaAccuracy
 - ClpSimplex, 466
- alphaAccuracy_
 - AbcTolerancesEtc, 125
 - ClpSimplex, 481
- alwaysOptimal
 - AbcSimplexPrimal, 121
 - ClpSimplexPrimal, 503
- ampl_info, 134
 - arguments, 137
 - branchDirection, 136
 - buffer, 137
 - columnLower, 135
 - columnStatus, 136
 - columnUpper, 136
 - cut, 137
 - direction, 135
 - dualSolution, 136
 - elements, 136
 - logLevel, 137
 - nonLinear, 137
 - numberArguments, 135
 - numberBinary, 135
 - numberColumns, 135
 - numberElements, 135
 - numberIntegers, 135
 - numberRows, 135
 - numberSos, 135
 - objValue, 135
 - objective, 135
 - offset, 135
 - primalSolution, 136

- priorities, 136
- problemStatus, 135
- pseudoDown, 136
- pseudoUp, 136
- rowLower, 135
- rowStatus, 136
- rowUpper, 135
- rows, 136
- sosIndices, 137
- sosPriority, 136
- sosReference, 137
- sosStart, 137
- sosType, 136
- special, 137
- starts, 136
- ampl_obj_prec
 - Clp_ampl.h, 657
- append
 - CbcOrClpParam, 143
- appendCols
 - ClpMatrixBase, 292
 - ClpNetworkMatrix, 349
 - ClpPackedMatrix, 382
 - ClpPlusMinusOneMatrix, 404
- appendMatrix
 - ClpMatrixBase, 292
 - ClpNetworkMatrix, 349
 - ClpPackedMatrix, 382
 - ClpPlusMinusOneMatrix, 404
- appendRows
 - ClpMatrixBase, 292
 - ClpNetworkMatrix, 349
 - ClpPackedMatrix, 382
 - ClpPlusMinusOneMatrix, 404
- applyColCut
 - OsiClpSolverInterface, 627
- applyCuts
 - OsiClpSolverInterface, 621
- applyNode
 - ClpNode, 356
- applyRowCut
 - OsiClpSolverInterface, 627
- applyRowCuts
 - OsiClpSolverInterface, 621
- areaFactor
 - AbcSimplexFactorization, 115
 - ClpFactorization, 229, 230
 - CoinAbcAnyFactorization, 523
- areaFactor_
 - CoinAbcAnyFactorization, 528
- arguments
 - ampl_info, 137
- arrayForBtran_
 - AbcSimplex, 101
- arrayForDualColumn
 - AbcSimplex, 83
- arrayForDualColumn_
 - AbcSimplex, 101
- arrayForFlipBounds
 - AbcSimplex, 83
- arrayForFlipBounds_
 - AbcSimplex, 101
- arrayForFlipRhs
 - AbcSimplex, 83
- arrayForFlipRhs_
 - AbcSimplex, 101
- arrayForFtran
 - AbcSimplex, 83
- arrayForFtran_
 - AbcSimplex, 101
- arrayForReplaceColumn
 - AbcSimplex, 83
- arrayForReplaceColumn_
 - AbcSimplex, 101
- arrayForTableauRow
 - AbcSimplex, 83
- arrayForTableauRow_
 - AbcSimplex, 101
- assignBasisStatus
 - AbcWarmStart, 130
- assignProblem
 - OsiClpSolverInterface, 622
- atLowerBound
 - AbcSimplex, 74
 - ClpDynamicMatrix, 211
 - ClpGubDynamicMatrix, 237
 - ClpSimplex, 452
- atUpperBound
 - AbcSimplex, 74
 - ClpDynamicMatrix, 211
 - ClpGubDynamicMatrix, 237
 - ClpSimplex, 452
- atFakeBound
 - AbcSimplex, 89
- atolmin
 - Info, 587
- atolnew
 - Outfo, 634
- atolold
 - Outfo, 634
- automatic
 - ClpSolve, 513
- automaticScale_
 - ClpSimplex, 488
- automaticScaling
 - ClpSimplex, 467

- averageAfterL_
 - CoinAbcStatistics, [538](#)
- averageAfterR_
 - CoinAbcStatistics, [538](#)
- averageAfterU_
 - CoinAbcStatistics, [538](#)
- averageTheta
 - AbcNonLinearCost, [50](#)
 - ClpNonLinearCost, [370](#)
- BASIS_SAME
 - ClpModel.hpp, [690](#)
- BLOCKING8
 - CoinAbcCommonFactorization.hpp, [708](#)
- BLOCKING8X8
 - CoinAbcCommonFactorization.hpp, [708](#)
- bNumber
 - ClpCholeskyDense, [159](#)
- backToBasics
 - ClpMatrixBase, [298](#)
- backToPivotRow_
 - ClpDynamicMatrix, [218](#)
 - ClpGubMatrix, [254](#)
- backward
 - ClpGubMatrix, [253](#)
- backward_
 - ClpGubMatrix, [254](#)
- backwardBasic
 - ClpSimplexOther::parametricsData, [635](#)
- badTimes
 - ClpSimplexProgress, [509](#)
- barrier
 - ClpSimplex, [456](#)
- baseliteration
 - ClpSimplex, [476](#)
- baseliteration_
 - AbcTolerancesEtc, [126](#)
 - ClpSimplex, [480](#)
- baseL
 - CoinAbcTypeFactorization, [553](#)
- baseL_
 - CoinAbcTypeFactorization, [570](#)
- baseMatrix_
 - ClpModel, [342](#)
- baseModel
 - AbcSimplex, [75](#)
 - ClpSimplex, [453](#)
- baseModel_
 - ClpSimplex, [488](#)
 - OsiClpSolverInterface, [632](#)
- baseObjectiveNorm_
 - ClpInterior, [276](#)
- baseRowCopy_
 - ClpModel, [342](#)
- basic
 - AbcSimplex, [74](#)
 - ClpSimplex, [452](#)
- basis_
 - OsiClpSolverInterface, [631](#)
- basisIsAvailable
 - OsiClpSolverInterface, [609](#)
- beforeStatusOfProblemInDual
 - ClpEventHandler, [223](#)
- beforeStatusOfProblemInPrimal
 - ClpEventHandler, [223](#)
- bestEverPivot
 - dualColumnResult, [579](#)
- bestObjectiveValue_
 - ClpSimplex, [480](#)
- bestPivot
 - ClpSimplexOther, [499](#)
- bestPossibleImprovement
 - ClpSimplex, [473](#)
- bestPossibleImprovement_
 - ClpSimplex, [479](#)
- block
 - dualColumnResult, [580](#)
- block_
 - AbcMatrix3, [46](#)
 - ClpPackedMatrix3, [394](#)
- blockStart
 - AbcMatrix, [38](#), [39](#)
- blockStart_
 - AbcMatrix, [40](#)
- blockStruct, [137](#)
 - numberElements_, [138](#)
 - numberInBlock_, [138](#)
 - numberPrice_, [138](#)
 - startElements_, [138](#)
 - startIndices_, [138](#)
- blockStruct3, [138](#)
 - numberElements_, [139](#)
 - numberInBlock_, [139](#)
 - numberPrice_, [139](#)
 - startElements_, [138](#)
 - startIndices_, [138](#)
- borrowDiag1
 - ClpLsq, [285](#)
- borrowModel
 - ClpInterior, [266](#)
 - ClpModel, [319](#)
 - ClpSimplex, [454](#), [455](#)
- bothFake
 - AbcSimplex, [74](#)
 - ClpSimplex, [452](#)
- bounceTolerances
 - AbcSimplexDual, [108](#)
- branch

ClpNode::branchState, [139](#)
 branchAndBound
 OsiClpSolverInterface, [609](#)
 branchDirection
 ampl_info, [136](#)
 branchState_
 ClpNode, [359](#)
 branchingValue
 ClpNode, [357](#)
 branchingValue_
 ClpNode, [358](#)
 btranAlpha_
 AbcSimplex, [94](#)
 btranAverageAfterL_
 CoinAbcTypeFactorization, [576](#)
 btranAverageAfterR_
 CoinAbcTypeFactorization, [576](#)
 btranAverageAfterU_
 CoinAbcTypeFactorization, [576](#)
 btranCountAfterL_
 CoinAbcTypeFactorization, [576](#)
 btranCountAfterR_
 CoinAbcTypeFactorization, [575](#)
 btranCountAfterU_
 CoinAbcTypeFactorization, [575](#)
 btranCountInput_
 CoinAbcTypeFactorization, [575](#)
 btranFullAverageAfterL_
 CoinAbcTypeFactorization, [577](#)
 btranFullAverageAfterR_
 CoinAbcTypeFactorization, [577](#)
 btranFullAverageAfterU_
 CoinAbcTypeFactorization, [577](#)
 btranFullCountAfterL_
 CoinAbcTypeFactorization, [577](#)
 btranFullCountAfterR_
 CoinAbcTypeFactorization, [577](#)
 btranFullCountAfterU_
 CoinAbcTypeFactorization, [577](#)
 btranFullCountInput_
 CoinAbcTypeFactorization, [577](#)
 buffer
 ampl_info, [137](#)

 CBC_PARAM_ACTION_BAB
 CbcOrClpParam.hpp, [656](#)
 CBC_PARAM_ACTION_DIRMIPLIB
 CbcOrClpParam.hpp, [655](#)
 CBC_PARAM_ACTION_DOHEURISTIC
 CbcOrClpParam.hpp, [656](#)
 CBC_PARAM_ACTION_MIPLIB
 CbcOrClpParam.hpp, [656](#)
 CBC_PARAM_ACTION_MIPSTART
 CbcOrClpParam.hpp, [656](#)

CBC_PARAM_ACTION_PRIORITYIN
 CbcOrClpParam.hpp, [656](#)
 CBC_PARAM_ACTION_STRENGTHEN
 CbcOrClpParam.hpp, [656](#)
 CBC_PARAM_ACTION_USERCBC
 CbcOrClpParam.hpp, [656](#)
 CBC_PARAM_DBL_ALLOWABLEGAP
 CbcOrClpParam.hpp, [651](#)
 CBC_PARAM_DBL_ARTIFICIALCOST
 CbcOrClpParam.hpp, [652](#)
 CBC_PARAM_DBL_CUTOFF
 CbcOrClpParam.hpp, [651](#)
 CBC_PARAM_DBL_DEXTRA3
 CbcOrClpParam.hpp, [652](#)
 CBC_PARAM_DBL_DEXTRA4
 CbcOrClpParam.hpp, [652](#)
 CBC_PARAM_DBL_DEXTRA5
 CbcOrClpParam.hpp, [652](#)
 CBC_PARAM_DBL_DJFIX
 CbcOrClpParam.hpp, [652](#)
 CBC_PARAM_DBL_FAKECUTOFF
 CbcOrClpParam.hpp, [652](#)
 CBC_PARAM_DBL_FAKEINCREMENT
 CbcOrClpParam.hpp, [652](#)
 CBC_PARAM_DBL_GAPRATIO
 CbcOrClpParam.hpp, [652](#)
 CBC_PARAM_DBL_INCREMENT
 CbcOrClpParam.hpp, [651](#)
 CBC_PARAM_DBL_INFEASIBILITYWEIGHT
 CbcOrClpParam.hpp, [651](#)
 CBC_PARAM_DBL_INTEGERTOLERANCE
 CbcOrClpParam.hpp, [651](#)
 CBC_PARAM_DBL_SMALLBAB
 CbcOrClpParam.hpp, [652](#)
 CBC_PARAM_DBL_TIGHTENFACTOR
 CbcOrClpParam.hpp, [652](#)
 CBC_PARAM_DBL_TIMELIMIT_BAB
 CbcOrClpParam.hpp, [651](#)
 CBC_PARAM_FULLGENERALQUERY
 CbcOrClpParam.hpp, [651](#)
 CBC_PARAM_GENERALQUERY
 CbcOrClpParam.hpp, [651](#)
 CBC_PARAM_INT_CUTDEPTH
 CbcOrClpParam.hpp, [652](#)
 CBC_PARAM_INT_CUTLENGTH
 CbcOrClpParam.hpp, [653](#)
 CBC_PARAM_INT_CUTPASS
 CbcOrClpParam.hpp, [653](#)
 CBC_PARAM_INT_CUTPASSINTREE
 CbcOrClpParam.hpp, [653](#)
 CBC_PARAM_INT_DENSE
 CbcOrClpParam.hpp, [653](#)
 CBC_PARAM_INT_DEPTHMINIBAB
 CbcOrClpParam.hpp, [653](#)

- CBC_PARAM_INT_DIVEOPT
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_INT_EXPERIMENT
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_INT_EXTRA1
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_INT_EXTRA2
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_INT_EXTRA3
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_INT_EXTRA4
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_INT_EXTRA_VARIABLES
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_INT_FPUMPITS
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_INT_FPUMPTUNE
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_INT_FPUMPTUNE2
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_INT_HOPTIONS
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_INT_MAX_SLOW_CUTS
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_INT_MAXHOTITS
 CbcOrClpParam.hpp, [652](#)
- CBC_PARAM_INT_MAXNODES
 CbcOrClpParam.hpp, [652](#)
- CBC_PARAM_INT_MAXSAVEDSOLS
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_INT_MAXSOLS
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_INT_MIOPTIONS
 CbcOrClpParam.hpp, [652](#)
- CBC_PARAM_INT_MOREMIOPTIONS
 CbcOrClpParam.hpp, [652](#)
- CBC_PARAM_INT_MOREMOREMIOPTIONS
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_INT_MULTIPLEROOTS
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_INT_NUMBERANALYZE
 CbcOrClpParam.hpp, [652](#)
- CBC_PARAM_INT_NUMBERBEFORE
 CbcOrClpParam.hpp, [652](#)
- CBC_PARAM_INT_RANDOMSEED
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_INT_SMALLFACT
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_INT_STRATEGY
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_INT_STRONG_STRATEGY
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_INT_STRONGBRANCHING
 CbcOrClpParam.hpp, [652](#)
- CBC_PARAM_INT_TESTOSI
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_INT_THREADS
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_INT_VUBTRY
 CbcOrClpParam.hpp, [653](#)
- CBC_PARAM_NOTUSED_CBCSTUFF
 CbcOrClpParam.hpp, [656](#)
- CBC_PARAM_NOTUSED_INVALID
 CbcOrClpParam.hpp, [656](#)
- CBC_PARAM_NOTUSED_OSLSTUFF
 CbcOrClpParam.hpp, [656](#)
- CBC_PARAM_STR_BRANCHSTRATEGY
 CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_CLIQUECUTS
 CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_COMBINE
 CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_COSTSTRATEGY
 CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_CPX
 CbcOrClpParam.hpp, [655](#)
- CBC_PARAM_STR_CROSSOVER2
 CbcOrClpParam.hpp, [655](#)
- CBC_PARAM_STR_CUTOFF_CONSTRAINT
 CbcOrClpParam.hpp, [655](#)
- CBC_PARAM_STR_CUTSSTRATEGY
 CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_DINS
 CbcOrClpParam.hpp, [655](#)
- CBC_PARAM_STR_DIVINGC
 CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_DIVINGF
 CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_DIVINGG
 CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_DIVINGL
 CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_DIVINGP
 CbcOrClpParam.hpp, [655](#)
- CBC_PARAM_STR_DIVINGS
 CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_DIVINGV
 CbcOrClpParam.hpp, [655](#)
- CBC_PARAM_STR_DW
 CbcOrClpParam.hpp, [655](#)
- CBC_PARAM_STR_FLOWCUTS
 CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_FPUMP
 CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_GMICUTS
 CbcOrClpParam.hpp, [655](#)
- CBC_PARAM_STR_GOMORYCUTS
 CbcOrClpParam.hpp, [654](#)

- CBC_PARAM_STR_GREEDY
 - CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_HEURISTICSTRATEGY
 - CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_KNAPSACKCUTS
 - CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_LAGOMORYCUTS
 - CbcOrClpParam.hpp, [655](#)
- CBC_PARAM_STR_LANDPCUTS
 - CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_LATWOMIRCUTS
 - CbcOrClpParam.hpp, [655](#)
- CBC_PARAM_STR_LOCALTREE
 - CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_MIXEDCUTS
 - CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_NAIVE
 - CbcOrClpParam.hpp, [655](#)
- CBC_PARAM_STR_NODESTRATEGY
 - CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_PIVOTANDCOMPLEMENT
 - CbcOrClpParam.hpp, [655](#)
- CBC_PARAM_STR_PIVOTANDFIX
 - CbcOrClpParam.hpp, [655](#)
- CBC_PARAM_STR_PREPROCESS
 - CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_PROBINGCUTS
 - CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_PROXIMITY
 - CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_RANDROUND
 - CbcOrClpParam.hpp, [655](#)
- CBC_PARAM_STR_REDSPLIT2CUTS
 - CbcOrClpParam.hpp, [655](#)
- CBC_PARAM_STR_REDSPLITCUTS
 - CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_RENS
 - CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_RESIDCUTS
 - CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_RINS
 - CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_ROUNDING
 - CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_SOLVER
 - CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_SOS
 - CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_TWOMIRCUTS
 - CbcOrClpParam.hpp, [654](#)
- CBC_PARAM_STR_VND
 - CbcOrClpParam.hpp, [655](#)
- CBC_PARAM_STR_ZEROHALFCUTS
 - CbcOrClpParam.hpp, [655](#)
- CLP_BAD_BOUNDS
 - ClpMessage.hpp, [687](#)
- CLP_BAD_MATRIX
 - ClpMessage.hpp, [687](#)
- CLP_BAD_STRING_VALUES
 - ClpMessage.hpp, [688](#)
- CLP_BARRIER_ABS_DROPPED
 - ClpMessage.hpp, [688](#)
- CLP_BARRIER_ABS_ERROR
 - ClpMessage.hpp, [688](#)
- CLP_BARRIER_ACCURACY
 - ClpMessage.hpp, [687](#)
- CLP_BARRIER_CLOSE_TO_OPTIMAL
 - ClpMessage.hpp, [687](#)
- CLP_BARRIER_COMPLEMENTARITY
 - ClpMessage.hpp, [687](#)
- CLP_BARRIER_DIAGONAL
 - ClpMessage.hpp, [688](#)
- CLP_BARRIER_DUALINF
 - ClpMessage.hpp, [688](#)
- CLP_BARRIER_END
 - ClpMessage.hpp, [687](#)
- CLP_BARRIER_EXIT
 - ClpMessage.hpp, [687](#)
- CLP_BARRIER_EXIT2
 - ClpMessage.hpp, [687](#)
- CLP_BARRIER_FEASIBLE
 - ClpMessage.hpp, [688](#)
- CLP_BARRIER_GONE_INFEASIBLE
 - ClpMessage.hpp, [687](#)
- CLP_BARRIER_INFO
 - ClpMessage.hpp, [687](#)
- CLP_BARRIER_ITERATION
 - ClpMessage.hpp, [687](#)
- CLP_BARRIER_KILLED
 - ClpMessage.hpp, [688](#)
- CLP_BARRIER_KKT
 - ClpMessage.hpp, [688](#)
- CLP_BARRIER_MU
 - ClpMessage.hpp, [687](#)
- CLP_BARRIER_NEGATIVE_GAPS
 - ClpMessage.hpp, [687](#)
- CLP_BARRIER_OBJECTIVE_GAP
 - ClpMessage.hpp, [687](#)
- CLP_BARRIER_REDUCING
 - ClpMessage.hpp, [688](#)
- CLP_BARRIER_SAFE
 - ClpMessage.hpp, [687](#)
- CLP_BARRIER_SCALING
 - ClpMessage.hpp, [687](#)
- CLP_BARRIER_SLACKS
 - ClpMessage.hpp, [688](#)
- CLP_BARRIER_STEP
 - ClpMessage.hpp, [688](#)

- CLP_BARRIER_STOPPING
 - ClpMessage.hpp, [687](#)
- CLP_COMPLICATED_MODEL
 - ClpMessage.hpp, [688](#)
- CLP_CRASH
 - ClpMessage.hpp, [687](#)
- CLP_CRUNCH_STATS
 - ClpMessage.hpp, [688](#)
- CLP_DUAL_BOUNDS
 - ClpMessage.hpp, [686](#)
- CLP_DUAL_CHECK
 - ClpMessage.hpp, [687](#)
- CLP_DUAL_CHECKB
 - ClpMessage.hpp, [686](#)
- CLP_DUAL_ORIGINAL
 - ClpMessage.hpp, [686](#)
- CLP_DUMMY_END
 - ClpMessage.hpp, [688](#)
- CLP_DUPLICATEELEMENTS
 - ClpMessage.hpp, [686](#)
- CLP_EMPTY_PROBLEM
 - ClpMessage.hpp, [687](#)
- CLP_END_VALUES_PASS
 - ClpMessage.hpp, [687](#)
- CLP_FATHOM_FINISH
 - ClpMessage.hpp, [688](#)
- CLP_FATHOM_SOLUTION
 - ClpMessage.hpp, [688](#)
- CLP_FATHOM_STATUS
 - ClpMessage.hpp, [688](#)
- CLP_GENERAL
 - ClpMessage.hpp, [688](#)
- CLP_GENERAL2
 - ClpMessage.hpp, [688](#)
- CLP_GENERAL_WARNING
 - ClpMessage.hpp, [688](#)
- CLP_IDIOT_ITERATION
 - ClpMessage.hpp, [687](#)
- CLP_IMPORT_ERRORS
 - ClpMessage.hpp, [687](#)
- CLP_IMPORT_RESULT
 - ClpMessage.hpp, [687](#)
- CLP_INFEASIBLE
 - ClpMessage.hpp, [687](#)
- CLP_INITIALIZE_STEEP
 - ClpMessage.hpp, [687](#)
- CLP_INTERVAL_TIMING
 - ClpMessage.hpp, [687](#)
- CLP_LOOP
 - ClpMessage.hpp, [687](#)
- CLP_MATRIX_CHANGE
 - ClpMessage.hpp, [687](#)
- CLP_MODIFIEDBOUNDS
 - ClpMessage.hpp, [686](#)
- CLP_PACKEDSCALE_FINAL
 - ClpMessage.hpp, [687](#)
- CLP_PACKEDSCALE_FORGET
 - ClpMessage.hpp, [687](#)
- CLP_PACKEDSCALE_INITIAL
 - ClpMessage.hpp, [687](#)
- CLP_PACKEDSCALE_WHILE
 - ClpMessage.hpp, [687](#)
- CLP_PARAM_ACTION_ALLSLACK
 - CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_BARRIER
 - CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_BASISIN
 - CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_BASISOUT
 - CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_CLEARCUTS
 - CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_CSVSTATISTICS
 - CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_DEBUG
 - CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_DIRECTORY
 - CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_DIRNETLIB
 - CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_DIRSAMPLE
 - CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_DUALSIMPLEX
 - CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_DUMMY
 - CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_EITHERSIMPLEX
 - CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_ENVIRONMENT
 - CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_EXIT
 - CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_EXPORT
 - CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_FAKEBOUND
 - CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_GMPL_SOLUTION
 - CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_HELP
 - CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_IMPORT
 - CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_MAXIMIZE
 - CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_MINIMIZE
 - CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_MODELIN
 - CbcOrClpParam.hpp, [656](#)

- CLP_PARAM_ACTION_NETLIB_BARRIER
CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_NETLIB_DUAL
CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_NETLIB_EITHER
CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_NETLIB_PRIMAL
CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_NETLIB_TUNE
CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_NETWORK
CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_NEXTBESTSOLUTION
CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_OUTDUPROWS
CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_PARAMETRICS
CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_PLUSMINUS
CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_PRIMALSIMPLEX
CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_PRINTMASK
CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_REALLY_SCALE
CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_RESTORE
CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_REVERSE
CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_SAVE
CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_SAVESOL
CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_SOLUTION
CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_SOLVECONTINUOUS
CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_STATISTICS
CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_STDIN
CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_STOREDFILE
CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_TIGHTEN
CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_UNITTEST
CbcOrClpParam.hpp, [655](#)
- CLP_PARAM_ACTION_USERCLP
CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_ACTION_VERSION
CbcOrClpParam.hpp, [656](#)
- CLP_PARAM_DBL_DUALTOLERANCE
CbcOrClpParam.hpp, [651](#)
- CLP_PARAM_DBL_OBJSCALE
CbcOrClpParam.hpp, [651](#)
- CLP_PARAM_DBL_OBJSCALE2
CbcOrClpParam.hpp, [652](#)
- CLP_PARAM_DBL_PRESOLVETOLERANCE
CbcOrClpParam.hpp, [652](#)
- CLP_PARAM_DBL_PRIMALTOLERANCE
CbcOrClpParam.hpp, [651](#)
- CLP_PARAM_DBL_PRIMALWEIGHT
CbcOrClpParam.hpp, [651](#)
- CLP_PARAM_DBL_RHSSCALE
CbcOrClpParam.hpp, [651](#)
- CLP_PARAM_DBL_TIMELIMIT
CbcOrClpParam.hpp, [651](#)
- CLP_PARAM_DBL_ZEROTOLERANCE
CbcOrClpParam.hpp, [651](#)
- CLP_PARAM_INT_CPP
CbcOrClpParam.hpp, [652](#)
- CLP_PARAM_INT_DECOMPOSE_BLOCKS
CbcOrClpParam.hpp, [652](#)
- CLP_PARAM_INT_DUALIZE
CbcOrClpParam.hpp, [652](#)
- CLP_PARAM_INT_IDIOT
CbcOrClpParam.hpp, [652](#)
- CLP_PARAM_INT_LOGLEVEL
CbcOrClpParam.hpp, [652](#)
- CLP_PARAM_INT_MAXFACTOR
CbcOrClpParam.hpp, [652](#)
- CLP_PARAM_INT_MAXITERATION
CbcOrClpParam.hpp, [652](#)
- CLP_PARAM_INT_MORESPECIALOPTIONS
CbcOrClpParam.hpp, [652](#)
- CLP_PARAM_INT_OUTPUTFORMAT
CbcOrClpParam.hpp, [652](#)
- CLP_PARAM_INT_PERTVALUE
CbcOrClpParam.hpp, [652](#)
- CLP_PARAM_INT_PRESOLVEOPTIONS
CbcOrClpParam.hpp, [652](#)
- CLP_PARAM_INT_PRESOLVEPASS
CbcOrClpParam.hpp, [652](#)
- CLP_PARAM_INT_PRINTOPTIONS
CbcOrClpParam.hpp, [652](#)
- CLP_PARAM_INT_PROCESSTUNE
CbcOrClpParam.hpp, [652](#)
- CLP_PARAM_INT_RANDOMSEED
CbcOrClpParam.hpp, [652](#)
- CLP_PARAM_INT_SLPVALUE
CbcOrClpParam.hpp, [652](#)
- CLP_PARAM_INT_SOLVERLOGLEVEL
CbcOrClpParam.hpp, [652](#)
- CLP_PARAM_INT_SPECIALOPTIONS
CbcOrClpParam.hpp, [652](#)

CLP_PARAM_INT_SPRINT
 CbcOrClpParam.hpp, 652

CLP_PARAM_INT_SUBSTITUTION
 CbcOrClpParam.hpp, 652

CLP_PARAM_INT_USESOLUTION
 CbcOrClpParam.hpp, 652

CLP_PARAM_INT_VERBOSE
 CbcOrClpParam.hpp, 652

CLP_PARAM_STR_ABCWANTED
 CbcOrClpParam.hpp, 654

CLP_PARAM_STR_ALLCOMMANDS
 CbcOrClpParam.hpp, 654

CLP_PARAM_STR_AUTOSCALE
 CbcOrClpParam.hpp, 653

CLP_PARAM_STR_BARRIERSCALE
 CbcOrClpParam.hpp, 654

CLP_PARAM_STR_BIASLU
 CbcOrClpParam.hpp, 653

CLP_PARAM_STR_CHOLESKY
 CbcOrClpParam.hpp, 653

CLP_PARAM_STR_CRASH
 CbcOrClpParam.hpp, 653

CLP_PARAM_STR_CROSSOVER
 CbcOrClpParam.hpp, 654

CLP_PARAM_STR_DIRECTION
 CbcOrClpParam.hpp, 653

CLP_PARAM_STR_DUALPIVOT
 CbcOrClpParam.hpp, 653

CLP_PARAM_STR_ERRORALLOWED
 CbcOrClpParam.hpp, 653

CLP_PARAM_STR_FACTORIZATION
 CbcOrClpParam.hpp, 654

CLP_PARAM_STR_GAMMA
 CbcOrClpParam.hpp, 654

CLP_PARAM_STR_INTPRINT
 CbcOrClpParam.hpp, 654

CLP_PARAM_STR_KEEPNAMES
 CbcOrClpParam.hpp, 653

CLP_PARAM_STR_KKT
 CbcOrClpParam.hpp, 654

CLP_PARAM_STR_MESSAGES
 CbcOrClpParam.hpp, 653

CLP_PARAM_STR_PERTURBATION
 CbcOrClpParam.hpp, 653

CLP_PARAM_STR_PFI
 CbcOrClpParam.hpp, 654

CLP_PARAM_STR_PREOLVE
 CbcOrClpParam.hpp, 653

CLP_PARAM_STR_PRIMALPIVOT
 CbcOrClpParam.hpp, 653

CLP_PARAM_STR_SCALING
 CbcOrClpParam.hpp, 653

CLP_PARAM_STR_SPARSEFACTOR
 CbcOrClpParam.hpp, 653

CLP_PARAM_STR_TIME_MODE
 CbcOrClpParam.hpp, 654

CLP_PARAM_STR_VECTOR
 CbcOrClpParam.hpp, 654

CLP_PARAMETRICS_STATS
 ClpMessage.hpp, 688

CLP_PARAMETRICS_STATS2
 ClpMessage.hpp, 688

CLP_POSSIBLELOOP
 ClpMessage.hpp, 686

CLP_PRIMAL_DJ
 ClpMessage.hpp, 687

CLP_PRIMAL_OPTIMAL
 ClpMessage.hpp, 686

CLP_PRIMAL_ORIGINAL
 ClpMessage.hpp, 686

CLP_PRIMAL_WEIGHT
 ClpMessage.hpp, 686

CLP_QUADRATIC_BOTH
 ClpMessage.hpp, 687

CLP_QUADRATIC_PRIMAL_DETAILS
 ClpMessage.hpp, 687

CLP_RIM_SCALE
 ClpMessage.hpp, 688

CLP_RIMSTATISTICS1
 ClpMessage.hpp, 686

CLP_RIMSTATISTICS2
 ClpMessage.hpp, 686

CLP_RIMSTATISTICS3
 ClpMessage.hpp, 686

CLP_SIMPLEX_ACCURACY
 ClpMessage.hpp, 686

CLP_SIMPLEX_BADFACTOR
 ClpMessage.hpp, 686

CLP_SIMPLEX_BOUNDTIGHTEN
 ClpMessage.hpp, 686

CLP_SIMPLEX_ERROR
 ClpMessage.hpp, 686

CLP_SIMPLEX_FINISHED
 ClpMessage.hpp, 686

CLP_SIMPLEX_FLAG
 ClpMessage.hpp, 686

CLP_SIMPLEX_FREEIN
 ClpMessage.hpp, 687

CLP_SIMPLEX_GIVINGUP
 ClpMessage.hpp, 686

CLP_SIMPLEX_HOUSE1
 ClpMessage.hpp, 686

CLP_SIMPLEX_HOUSE2
 ClpMessage.hpp, 686

CLP_SIMPLEX_INFEASIBILITIES
 ClpMessage.hpp, 686

CLP_SIMPLEX_INFEASIBLE
 ClpMessage.hpp, 686

- CLP_SIMPLEX_INTERRUPT
 - ClpMessage.hpp, 686
- CLP_SIMPLEX_NONLINEAR
 - ClpMessage.hpp, 687
- CLP_SIMPLEX_PERTURB
 - ClpMessage.hpp, 686
- CLP_SIMPLEX_PIVOTROW
 - ClpMessage.hpp, 687
- CLP_SIMPLEX_STATUS
 - ClpMessage.hpp, 686
- CLP_SIMPLEX_STOPPED
 - ClpMessage.hpp, 686
- CLP_SIMPLEX_UNBOUNDED
 - ClpMessage.hpp, 686
- CLP_SINGULARITIES
 - ClpMessage.hpp, 686
- CLP_SLP_ITER
 - ClpMessage.hpp, 688
- CLP_SMALLELEMENTS
 - ClpMessage.hpp, 686
- CLP_SPRINT
 - ClpMessage.hpp, 687
- CLP_TIMING
 - ClpMessage.hpp, 687
- CLP_UNABLE_OPEN
 - ClpMessage.hpp, 687
- CBCMAXPARAMETERS
 - CbcOrClpParam.hpp, 651
- CHOL_SMALL_VALUE
 - ClpCholeskyBase.hpp, 675
- CLP_ABOVE_UPPER
 - AbcNonLinearCost.hpp, 641
 - ClpNonLinearCost.hpp, 692
- CLP_BELOW_LOWER
 - AbcNonLinearCost.hpp, 641
 - ClpNonLinearCost.hpp, 692
- CLP_CYCLE
 - ClpSolve.hpp, 701
- CLP_FEASIBLE
 - AbcNonLinearCost.hpp, 641
 - ClpNonLinearCost.hpp, 692
- CLP_LONG_CHOLESKY
 - ClpCholeskyBase.hpp, 675
- CLP_METHOD1
 - ClpNonLinearCost.hpp, 692
- CLP_METHOD2
 - ClpNonLinearCost.hpp, 692
- CLP_Message
 - ClpMessage.hpp, 686
- CLP_PROGRESS
 - ClpSolve.hpp, 701
- CLP_SAME
 - AbcNonLinearCost.hpp, 641
 - ClpNonLinearCost.hpp, 692
- CLP_VERSION
 - config_clp_default.h, 719
- CLP_VERSION_MAJOR
 - config_clp_default.h, 719
- CLP_VERSION_MINOR
 - config_clp_default.h, 719
- CLP_WARMSTART
 - AbcWarmStart.hpp, 648
- COIN_BIG_DOUBLE
 - CoinAbcFactorization.hpp, 710
- COIN_CBC_USING_CLP
 - ClpModel.hpp, 689
- COIN_CLP_VERBOSITY
 - config_default.h, 720
- COIN_FAC_NEW
 - CoinAbcCommon.hpp, 703
- COIN_FAST_CODE
 - ClpFactorization.hpp, 681
 - ClpNetworkBasis.hpp, 691
- COIN_HAS_CLP
 - config_default.h, 720
- COIN_HAS_COINUTILS
 - config_default.h, 720
- COIN_RESTRICT
 - ClpPackedMatrix.hpp, 694
- COLUMN_DUAL_OK
 - AbcSimplex.hpp, 645
- COLUMN_LOWER_SAME
 - ClpModel.hpp, 690
- COLUMN_PRIMAL_OK
 - AbcSimplex.hpp, 645
- COLUMN_UPPER_SAME
 - ClpModel.hpp, 690
- CONVERTROW
 - CoinAbcBaseFactorization.hpp, 701
- canCombine
 - ClpMatrixBase, 297
 - ClpPackedMatrix, 387
 - ClpPlusMinusOneMatrix, 407
- canDoPartialPricing
 - ClpMatrixBase, 295
 - ClpNetworkMatrix, 351
 - ClpPackedMatrix, 384
 - ClpPlusMinusOneMatrix, 408
- canDoSimplexInterface
 - OsiClpSolverInterface, 609
- canGetRowCopy
 - ClpMatrixBase, 293
- CbcOrClpParam.hpp
 - CBC_PARAM_ACTION_BAB, 656
 - CBC_PARAM_ACTION_DIRMIPLIB, 655
 - CBC_PARAM_ACTION_DOHEURISTIC, 656
 - CBC_PARAM_ACTION_MIPLIB, 656
 - CBC_PARAM_ACTION_MIPSTART, 656

CBC_PARAM_ACTION_PRIORITYIN, 656
CBC_PARAM_ACTION_STRENGTHEN, 656
CBC_PARAM_ACTION_USERCBC, 656
CBC_PARAM_DBL_ALLOWABLEGAP, 651
CBC_PARAM_DBL_ARTIFICIALCOST, 652
CBC_PARAM_DBL_CUTOFF, 651
CBC_PARAM_DBL_DEXTRA3, 652
CBC_PARAM_DBL_DEXTRA4, 652
CBC_PARAM_DBL_DEXTRA5, 652
CBC_PARAM_DBL_DJFIX, 652
CBC_PARAM_DBL_FAKECUTOFF, 652
CBC_PARAM_DBL_FAKEINCREMENT, 652
CBC_PARAM_DBL_GAPRATIO, 652
CBC_PARAM_DBL_INCREMENT, 651
CBC_PARAM_DBL_INFEASIBILITYWEIGHT, 651
CBC_PARAM_DBL_INTEGERTOLERANCE, 651
CBC_PARAM_DBL_SMALLBAB, 652
CBC_PARAM_DBL_TIGHTENFACTOR, 652
CBC_PARAM_DBL_TIMELIMIT_BAB, 651
CBC_PARAM_FULLGENERALQUERY, 651
CBC_PARAM_GENERALQUERY, 651
CBC_PARAM_INT_CUTDEPTH, 652
CBC_PARAM_INT_CUTLENGTH, 653
CBC_PARAM_INT_CUTPASS, 653
CBC_PARAM_INT_CUTPASSINTREE, 653
CBC_PARAM_INT_DENSE, 653
CBC_PARAM_INT_DEPTHMINIBAB, 653
CBC_PARAM_INT_DIVEOPT, 653
CBC_PARAM_INT_EXPERIMENT, 653
CBC_PARAM_INT_EXTRA1, 653
CBC_PARAM_INT_EXTRA2, 653
CBC_PARAM_INT_EXTRA3, 653
CBC_PARAM_INT_EXTRA4, 653
CBC_PARAM_INT_EXTRA_VARIABLES, 653
CBC_PARAM_INT_FPUMPITS, 653
CBC_PARAM_INT_FPUMPTUNE, 653
CBC_PARAM_INT_FPUMPTUNE2, 653
CBC_PARAM_INT_HOPTIONS, 653
CBC_PARAM_INT_MAX_SLOW_CUTS, 653
CBC_PARAM_INT_MAXHOTITS, 652
CBC_PARAM_INT_MAXNODES, 652
CBC_PARAM_INT_MAXSAVEDSOLS, 653
CBC_PARAM_INT_MAXSOLS, 653
CBC_PARAM_INT_MIPOPTIONS, 652
CBC_PARAM_INT_MOREMIPOPTIONS, 652
CBC_PARAM_INT_MOREMOREMIPOPTIONS, 653
CBC_PARAM_INT_MULTIPLEROOTS, 653
CBC_PARAM_INT_NUMBERANALYZE, 652
CBC_PARAM_INT_NUMBERBEFORE, 652
CBC_PARAM_INT_RANDOMSEED, 653
CBC_PARAM_INT_SMALLFACT, 653
CBC_PARAM_INT_STRATEGY, 653
CBC_PARAM_INT_STRONG_STRATEGY, 653
CBC_PARAM_INT_STRONGBRANCHING, 652
CBC_PARAM_INT_TESTOSI, 653
CBC_PARAM_INT_THREADS, 653
CBC_PARAM_INT_VUBTRY, 653
CBC_PARAM_NOTUSED_CBCSTUFF, 656
CBC_PARAM_NOTUSED_INVALID, 656
CBC_PARAM_NOTUSED_OSLSTUFF, 656
CBC_PARAM_STR_BRANCHSTRATEGY, 654
CBC_PARAM_STR_CLIQUECUTS, 654
CBC_PARAM_STR_COMBINE, 654
CBC_PARAM_STR_COSTSTRATEGY, 654
CBC_PARAM_STR_CPX, 655
CBC_PARAM_STR_CROSSOVER2, 655
CBC_PARAM_STR_CUTOFF_CONSTRAINT, 655
CBC_PARAM_STR_CUTSSTRATEGY, 654
CBC_PARAM_STR_DINS, 655
CBC_PARAM_STR_DIVINGC, 654
CBC_PARAM_STR_DIVINGF, 654
CBC_PARAM_STR_DIVINGG, 654
CBC_PARAM_STR_DIVINGL, 654
CBC_PARAM_STR_DIVINGP, 655
CBC_PARAM_STR_DIVINGS, 654
CBC_PARAM_STR_DIVINGV, 655
CBC_PARAM_STR_DW, 655
CBC_PARAM_STR_FLOWCUTS, 654
CBC_PARAM_STR_FPUMP, 654
CBC_PARAM_STR_GMICUTS, 655
CBC_PARAM_STR_GOMORYCUTS, 654
CBC_PARAM_STR_GREEDY, 654
CBC_PARAM_STR_HEURISTICSTRATEGY, 654
CBC_PARAM_STR_KNAPSACKCUTS, 654
CBC_PARAM_STR_LAGOMORYCUTS, 655
CBC_PARAM_STR_LANDPCUTS, 654
CBC_PARAM_STR_LATWOMIRCUTS, 655
CBC_PARAM_STR_LOCALTREE, 654
CBC_PARAM_STR_MIXEDCUTS, 654
CBC_PARAM_STR_NAIVE, 655
CBC_PARAM_STR_NODESTRATEGY, 654
CBC_PARAM_STR_PIVOTANDCOMPLEMENT, 655
CBC_PARAM_STR_PIVOTANDFIX, 655
CBC_PARAM_STR_PREPROCESS, 654
CBC_PARAM_STR_PROBINGCUTS, 654
CBC_PARAM_STR_PROXIMITY, 654
CBC_PARAM_STR_RANDROUND, 655
CBC_PARAM_STR_REDSPLIT2CUTS, 655
CBC_PARAM_STR_REDSPLITCUTS, 654
CBC_PARAM_STR_RENS, 654
CBC_PARAM_STR_RESIDCUTS, 654
CBC_PARAM_STR_RINS, 654
CBC_PARAM_STR_ROUNDING, 654
CBC_PARAM_STR_SOLVER, 654
CBC_PARAM_STR_SOS, 654
CBC_PARAM_STR_TWOMIRCUTS, 654
CBC_PARAM_STR_VND, 655

- CBC_PARAM_STR_ZEROHALFCUTS, 655
- CLP_PARAM_ACTION_ALLSLACK, 656
- CLP_PARAM_ACTION_BARRIER, 656
- CLP_PARAM_ACTION_BASISIN, 656
- CLP_PARAM_ACTION_BASISOUT, 656
- CLP_PARAM_ACTION_CLEARCUTS, 656
- CLP_PARAM_ACTION_CSVSTATISTICS, 656
- CLP_PARAM_ACTION_DEBUG, 656
- CLP_PARAM_ACTION_DIRECTORY, 655
- CLP_PARAM_ACTION_DIRNETLIB, 655
- CLP_PARAM_ACTION_DIRSAMPLE, 655
- CLP_PARAM_ACTION_DUALSIMPLEX, 655
- CLP_PARAM_ACTION_DUMMY, 656
- CLP_PARAM_ACTION_EITHERSIMPLEX, 655
- CLP_PARAM_ACTION_ENVIRONMENT, 656
- CLP_PARAM_ACTION_EXIT, 655
- CLP_PARAM_ACTION_EXPORT, 655
- CLP_PARAM_ACTION_FAKEBOUND, 655
- CLP_PARAM_ACTION_GMPL_SOLUTION, 656
- CLP_PARAM_ACTION_HELP, 655
- CLP_PARAM_ACTION_IMPORT, 655
- CLP_PARAM_ACTION_MAXIMIZE, 655
- CLP_PARAM_ACTION_MINIMIZE, 655
- CLP_PARAM_ACTION_MODELIN, 656
- CLP_PARAM_ACTION_NETLIB_BARRIER, 656
- CLP_PARAM_ACTION_NETLIB_DUAL, 655
- CLP_PARAM_ACTION_NETLIB_EITHER, 655
- CLP_PARAM_ACTION_NETLIB_PRIMAL, 655
- CLP_PARAM_ACTION_NETLIB_TUNE, 656
- CLP_PARAM_ACTION_NETWORK, 656
- CLP_PARAM_ACTION_NEXTBESTSOLUTION, 656
- CLP_PARAM_ACTION_OUTDUPROWS, 656
- CLP_PARAM_ACTION_PARAMETRICS, 656
- CLP_PARAM_ACTION_PLUSMINUS, 655
- CLP_PARAM_ACTION_PRIMALSIMPLEX, 655
- CLP_PARAM_ACTION_PRINTMASK, 656
- CLP_PARAM_ACTION_REALLY_SCALE, 656
- CLP_PARAM_ACTION_RESTORE, 655
- CLP_PARAM_ACTION_REVERSE, 656
- CLP_PARAM_ACTION_SAVE, 655
- CLP_PARAM_ACTION_SAVESOL, 655
- CLP_PARAM_ACTION_SOLUTION, 655
- CLP_PARAM_ACTION_SOLVECONTINUOUS, 656
- CLP_PARAM_ACTION_STATISTICS, 656
- CLP_PARAM_ACTION_STDIN, 655
- CLP_PARAM_ACTION_STOREDFILE, 656
- CLP_PARAM_ACTION_TIGHTEN, 655
- CLP_PARAM_ACTION_UNITTEST, 655
- CLP_PARAM_ACTION_USERCLP, 656
- CLP_PARAM_ACTION_VERSION, 656
- CLP_PARAM_DBL_DUALBOUND, 651
- CLP_PARAM_DBL_DUALTOLERANCE, 651
- CLP_PARAM_DBL_OBJSCALE, 651
- CLP_PARAM_DBL_OBJSCALE2, 652
- CLP_PARAM_DBL_PRESVOLUTOLERANCE, 652
- CLP_PARAM_DBL_PRIMALTOLERANCE, 651
- CLP_PARAM_DBL_PRIMALWEIGHT, 651
- CLP_PARAM_DBL_RHSSCALE, 651
- CLP_PARAM_DBL_TIMELIMIT, 651
- CLP_PARAM_DBL_ZEROTOLERANCE, 651
- CLP_PARAM_INT_CPP, 652
- CLP_PARAM_INT_DECOMPOSE_BLOCKS, 652
- CLP_PARAM_INT_DUALIZE, 652
- CLP_PARAM_INT_IDIOT, 652
- CLP_PARAM_INT_LOGLEVEL, 652
- CLP_PARAM_INT_MAXFACTOR, 652
- CLP_PARAM_INT_MAXITERATION, 652
- CLP_PARAM_INT_MORESPECIALOPTIONS, 652
- CLP_PARAM_INT_OUTPUTFORMAT, 652
- CLP_PARAM_INT_PERTVALUE, 652
- CLP_PARAM_INT_PREOLVEOPTIONS, 652
- CLP_PARAM_INT_PREOLVEPASS, 652
- CLP_PARAM_INT_PRINTOPTIONS, 652
- CLP_PARAM_INT_PROCESSTUNE, 652
- CLP_PARAM_INT_RANDOMSEED, 652
- CLP_PARAM_INT_SLPVALUE, 652
- CLP_PARAM_INT_SOLVERLOGLEVEL, 652
- CLP_PARAM_INT_SPECIALOPTIONS, 652
- CLP_PARAM_INT_SPRINT, 652
- CLP_PARAM_INT_SUBSTITUTION, 652
- CLP_PARAM_INT_USESOLUTION, 652
- CLP_PARAM_INT_VERBOSE, 652
- CLP_PARAM_STR_ABCWANTED, 654
- CLP_PARAM_STR_ALLCOMMANDS, 654
- CLP_PARAM_STR_AUTOSCALE, 653
- CLP_PARAM_STR_BARRIERSCALE, 654
- CLP_PARAM_STR_BIASLU, 653
- CLP_PARAM_STR_CHOLESKY, 653
- CLP_PARAM_STR_CRASH, 653
- CLP_PARAM_STR_CROSSOVER, 654
- CLP_PARAM_STR_DIRECTION, 653
- CLP_PARAM_STR_DUALPIVOT, 653
- CLP_PARAM_STR_ERRORALLOWED, 653
- CLP_PARAM_STR_FACTORIZATION, 654
- CLP_PARAM_STR_GAMMA, 654
- CLP_PARAM_STR_INTPRINT, 654
- CLP_PARAM_STR_KEEPNAMES, 653
- CLP_PARAM_STR_KKT, 654
- CLP_PARAM_STR_MESSAGES, 653
- CLP_PARAM_STR_PERTURBATION, 653
- CLP_PARAM_STR_PFI, 654
- CLP_PARAM_STR_PREOLVE, 653
- CLP_PARAM_STR_PRIMALPIVOT, 653
- CLP_PARAM_STR_SCALING, 653
- CLP_PARAM_STR_SPARSEFACTOR, 653
- CLP_PARAM_STR_TIME_MODE, 654
- CLP_PARAM_STR_VECTOR, 654

- CbcOrClpParam, 140
 - ~CbcOrClpParam, 143
 - addHelp, 143
 - append, 143
 - CbcOrClpParam, 142
 - CbcOrClpParam, 142
 - checkDoubleParameter, 144
 - currentOption, 145
 - currentOptionAsInteger, 145
 - displayThis, 146
 - doubleParameter, 143, 144
 - doubleValue, 145
 - fakeKeyWord, 146
 - intParameter, 143, 144
 - intValue, 145
 - lengthMatchName, 144
 - matchName, 144
 - matches, 146
 - name, 143
 - operator=, 143
 - parameterOption, 145
 - printLongHelp, 146
 - printOptions, 145
 - printString, 146
 - setCurrentOption, 145
 - setCurrentOptionWithMessage, 145
 - setDoubleParameter, 143, 144
 - setDoubleParameterWithMessage, 143, 144
 - setDoubleValue, 145
 - setFakeKeyWord, 146
 - setIntParameter, 143, 144
 - setIntParameterWithMessage, 143, 144
 - setIntValue, 145
 - setLonghelp, 146
 - setStringValue, 145
 - shortHelp, 143
 - stringValue, 145
 - type, 146
 - whereUsed, 146
- CbcOrClpParam.hpp
 - CBCMAXPARAMETERS, 651
 - CbcOrClpParameterType, 651
 - CoinReadGetCommand, 656
 - CoinReadGetDoubleField, 657
 - CoinReadGetIntField, 657
 - CoinReadGetString, 657
 - CoinReadNextField, 656
 - CoinReadPrintit, 657
 - establishParams, 657
 - saveSolution, 657
 - setCbcOrClpPrinting, 657
 - whichParam, 657
- CbcOrClpParameterType
 - CbcOrClpParam.hpp, 651
- changeBound
 - AbcSimplexDual, 107
 - ClpSimplexDual, 493
- changeBounds
 - AbcSimplexDual, 107
 - ClpSimplexDual, 493
- changeCost_
 - ClpGubMatrix, 254
- changeDownInCost
 - AbcNonLinearCost, 50
 - ClpNonLinearCost, 369
- changeInCost
 - AbcNonLinearCost, 49, 50
 - ClpNonLinearCost, 369, 370
- changeMade_
 - ClpSimplex, 486
- changeState
 - ClpNode, 357
- changeUpInCost
 - AbcNonLinearCost, 49
 - ClpNonLinearCost, 369
- check
 - ClpDisasterHandler, 184
 - OsiClpDisasterHandler, 595
- checkAccuracy
 - AbcDualRowPivot, 21
 - AbcPrimalColumnSteepest, 60
 - ClpDualRowPivot, 189
 - ClpPrimalColumnSteepest, 428
- checkArrays
 - AbcSimplex, 81
- checkBothSolutions
 - AbcSimplex, 80
 - ClpSimplex, 465
- checkChanged
 - AbcNonLinearCost, 48
 - ClpNonLinearCost, 368
- checkConsistency
 - CoinAbcTypeFactorization, 561
- checkConsistentPivots
 - AbcSimplex, 90
- checkCutoff
 - AbcSimplexDual, 108
- checkDjs
 - AbcSimplex, 81
- checkDoubleParameter
 - CbcOrClpParam, 144
- checkDualSolution
 - AbcSimplex, 80
 - ClpSimplex, 465
- checkDualSolutionPlusFake
 - AbcSimplex, 80
- checkFeasible
 - ClpGubDynamicMatrix, 239

- ClpMatrixBase, [296](#)
- checkFlags
 - ClpPackedMatrix, [389](#)
- checkGaps
 - ClpPackedMatrix, [388](#)
- checkGoodMove
 - ClpPredictorCorrector, [411](#)
- checkGoodMove2
 - ClpPredictorCorrector, [411](#)
- checkInfeasibilities
 - AbcNonLinearCost, [48](#)
 - ClpNonLinearCost, [368](#)
- checkLinks
 - CoinAbcBaseFactorization.hpp, [701](#)
- checkMarkArrays
 - AbcSimplexFactorization, [117](#)
 - CoinAbcAnyFactorization, [524](#)
 - CoinAbcTypeFactorization, [559](#)
- checkMoveBack
 - AbcSimplex, [81](#)
- checkPivot
 - CoinAbcDenseFactorization, [536](#)
 - CoinAbcTypeFactorization, [564](#)
- checkPossibleCleanup
 - AbcSimplexDual, [107](#)
 - ClpSimplexDual, [493](#)
- checkPossibleValuesMove
 - ClpSimplexDual, [493](#)
- checkPrimalSolution
 - AbcSimplex, [80](#)
 - ClpSimplex, [465](#)
- checkReplace
 - AbcSimplexDual, [106](#)
- checkReplacePart1
 - AbcSimplexDual, [106](#)
 - AbcSimplexFactorization, [112](#)
 - CoinAbcAnyFactorization, [526](#)
 - CoinAbcTypeFactorization, [556](#)
- checkReplacePart1a
 - AbcSimplexDual, [106](#)
 - AbcSimplexFactorization, [112](#)
 - CoinAbcAnyFactorization, [526](#)
- checkReplacePart1b
 - AbcSimplexDual, [106](#)
 - AbcSimplexFactorization, [112](#)
 - CoinAbcAnyFactorization, [526](#)
- checkReplacePart2
 - AbcSimplexFactorization, [113](#)
 - CoinAbcAnyFactorization, [526](#)
 - CoinAbcDenseFactorization, [534](#)
 - CoinAbcTypeFactorization, [557](#)
- checkSolution
 - ClpInterior, [270](#)
 - ClpSimplex, [463](#)
- checkSolutionBasic
 - AbcSimplex, [81](#)
- checkSolutionInternal
 - ClpSimplex, [463](#)
- checkSparse
 - CoinAbcTypeFactorization, [559](#)
- checkUnbounded
 - AbcSimplexDual, [107](#)
 - AbcSimplexPrimal, [122](#)
 - ClpSimplexDual, [493](#)
 - ClpSimplexPrimal, [504](#)
- checkUnscaledSolution
 - ClpSimplex, [463](#)
- checkValid
 - ClpPlusMinusOneMatrix, [406](#)
- chgColumnLower
 - ClpModel, [318](#)
- chgColumnUpper
 - ClpModel, [318](#)
- chgObjCoefficients
 - ClpModel, [318](#)
- chgRowLower
 - ClpModel, [318](#)
- chgRowUpper
 - ClpModel, [318](#)
- cholesky_
 - ClpInterior, [279](#)
- choleskyCondition
 - ClpCholeskyBase, [151](#)
- choleskyCondition_
 - ClpCholeskyBase, [154](#)
- choleskyRow_
 - ClpCholeskyBase, [155](#)
- choleskyStart_
 - ClpCholeskyBase, [155](#)
- cholmod_common
 - ClpCholeskyUfl.hpp, [678](#)
- cholmod_factor
 - ClpCholeskyUfl.hpp, [678](#)
- chooseBestDj
 - AbcMatrix, [36](#)
- chooseVariable
 - ClpNode, [356](#)
- cilk_for
 - CoinAbcCommon.hpp, [703](#)
- cilk_spawn
 - CoinAbcCommon.hpp, [703](#)
- cilk_sync
 - CoinAbcCommon.hpp, [703](#)
- cleanData
 - ClpGubDynamicMatrix, [239](#)
- cleanFactorization
 - AbcSimplex, [81](#)
 - ClpSimplex, [466](#)

- cleanMatrix
 - ClpModel, [319](#)
- cleanStatus
 - AbcSimplex, [79](#)
 - ClpSimplex, [464](#)
- cleanUp
 - ClpFactorization, [233](#)
- cleanUpForCrunch
 - ClpNode, [356](#)
- cleanup
 - ClpSimplex, [456](#)
 - CoinAbcTypeFactorization, [562](#)
- cleanupAfterPostsolve
 - ClpSimplexOther, [500](#)
- cleanupAfterStrongBranching
 - AbcSimplexDual, [105](#)
 - ClpSimplexDual, [492](#)
- cleanupScaling
 - OsiClpSolverInterface, [625](#)
- cleanupScaling_
 - OsiClpSolverInterface, [632](#)
- clearActive
 - AbcSimplex, [90](#)
 - ClpSimplex, [474](#)
- clearAll
 - AbcSimplexPrimal, [123](#)
 - ClpSimplexPrimal, [505](#)
- clearArrays
 - AbcDualRowPivot, [21](#)
 - AbcDualRowSteepest, [25](#)
 - AbcPrimalColumnPivot, [55](#)
 - AbcPrimalColumnSteepest, [60](#)
 - AbcSimplex, [88](#)
 - AbcSimplexFactorization, [115](#)
 - ClpDualRowPivot, [189](#)
 - ClpDualRowSteepest, [193](#)
 - ClpFactorization, [231](#)
 - ClpPrimalColumnPivot, [422](#)
 - ClpPrimalColumnSteepest, [428](#)
 - CoinAbcAnyFactorization, [525](#)
 - CoinAbcDenseFactorization, [535](#)
 - CoinAbcTypeFactorization, [559](#)
- clearArraysPublic
 - AbcSimplex, [88](#)
- clearBadTimes
 - ClpSimplexProgress, [509](#)
- clearFakeLower
 - ClpInterior, [271](#)
- clearFakeUpper
 - ClpInterior, [272](#)
- clearFeasibleExtremePoints
 - MyMessageHandler, [590](#)
- clearFixed
 - ClpInterior, [270](#)
- clearFixedOrFree
 - ClpInterior, [271](#)
- clearFlagged
 - AbcSimplex, [89](#)
 - ClpGubMatrix, [252](#)
 - ClpInterior, [270](#)
 - ClpSimplex, [474](#)
- clearIterationNumbers
 - ClpSimplexProgress, [508](#)
- clearLowerBound
 - ClpInterior, [271](#)
- clearOddState
 - ClpSimplexProgress, [508](#)
- clearPivoted
 - AbcSimplex, [89](#)
 - ClpSimplex, [474](#)
- clearTimesFlagged
 - ClpSimplexProgress, [509](#)
- clearUpperBound
 - ClpInterior, [271](#)
- clique_
 - ClpCholeskyBase, [156](#)
- clone
 - AbcDualRowDantzig, [18](#)
 - AbcDualRowPivot, [21](#)
 - AbcDualRowSteepest, [25](#)
 - AbcPrimalColumnDantzig, [52](#)
 - AbcPrimalColumnPivot, [55](#)
 - AbcPrimalColumnSteepest, [61](#)
 - AbcWarmStart, [130](#)
 - ClpCholeskyBase, [153](#)
 - ClpCholeskyDense, [160](#)
 - ClpCholeskyMumps, [162](#)
 - ClpCholeskyTaucs, [164](#)
 - ClpCholeskyUfl, [166](#)
 - ClpCholeskyWssmp, [168](#)
 - ClpCholeskyWssmpKKT, [170](#)
 - ClpConstraint, [173](#)
 - ClpConstraintLinear, [177](#)
 - ClpConstraintQuadratic, [180](#)
 - ClpDisasterHandler, [184](#)
 - ClpDualRowDantzig, [186](#)
 - ClpDualRowPivot, [190](#)
 - ClpDualRowSteepest, [193](#)
 - ClpDummyMatrix, [200](#)
 - ClpDynamicExampleMatrix, [204](#)
 - ClpDynamicMatrix, [214](#)
 - ClpEventHandler, [224](#)
 - ClpGubDynamicMatrix, [239](#)
 - ClpGubMatrix, [251](#)
 - ClpLinearObjective, [282](#)
 - ClpMatrixBase, [298](#)
 - ClpNetworkMatrix, [352](#)
 - ClpObjective, [374](#)

- ClpPackedMatrix, 388
- ClpPdcoBase, 398
- ClpPlusMinusOneMatrix, 408
- ClpPrimalColumnDantzig, 420
- ClpPrimalColumnPivot, 423
- ClpPrimalColumnSteepest, 429
- ClpPrimalQuadraticDantzig, 431
- ClpQuadraticObjective, 435
- CoinAbcAnyFactorization, 522
- CoinAbcDenseFactorization, 533
- CoinAbcTypeFactorization, 551
- MyEventHandler, 588
- MyMessageHandler, 591
- OsiClpDisasterHandler, 595
- OsiClpSolverInterface, 627
- ClpDualObjectiveLimit
 - ClpParameters.hpp, 695
- ClpDualRowSteepest
 - keep, 192
 - normal, 192
- ClpDualTolerance
 - ClpParameters.hpp, 695
- ClpDynamicMatrix
 - atLowerBound, 211
 - atUpperBound, 211
 - inSmall, 211
 - soloKey, 211
- ClpEventHandler
 - beforeStatusOfProblemInDual, 223
 - beforeStatusOfProblemInPrimal, 223
 - complicatedPivotIn, 223
 - complicatedPivotOut, 223
 - endInDual, 223
 - endInPrimal, 223
 - endOfCreateRim, 223
 - endOfFactorization, 223
 - endOfIteration, 223
 - endOfValuesPass, 223
 - goodFactorization, 223
 - looksEndInDual, 223
 - looksEndInPrimal, 223
 - modifyMatrixInMiniPostsolve, 223
 - modifyMatrixInMiniPresolve, 223
 - moreMiniPresolve, 223
 - noCandidateInDual, 223
 - noCandidateInPrimal, 223
 - noTheta, 223
 - node, 223
 - pivotRow, 223
 - presolveAfterFirstSolve, 223
 - presolveAfterSolve, 223
 - presolveBeforeSolve, 223
 - presolveEnd, 223
 - presolveInfeasible, 223
 - presolveSize, 223
 - presolveStart, 223
 - slightlyInfeasible, 223
 - solution, 223
 - startOfIterationInDual, 223
 - startOfStatusOfProblemInDual, 223
 - startOfStatusOfProblemInPrimal, 223
 - theta, 223
 - treeStatus, 223
 - updateDualsInDual, 223
- ClpGubDynamicMatrix
 - atLowerBound, 237
 - atUpperBound, 237
 - inSmall, 237
- ClpLastDblParam
 - ClpParameters.hpp, 695
- ClpLastIntParam
 - ClpParameters.hpp, 695
- ClpLastStrParam
 - ClpParameters.hpp, 695
- ClpMaxNumIteration
 - ClpParameters.hpp, 694
- ClpMaxNumIterationHotStart
 - ClpParameters.hpp, 694
- ClpMaxSeconds
 - ClpParameters.hpp, 695
- ClpMessage.hpp
 - CLP_BAD_BOUNDS, 687
 - CLP_BAD_MATRIX, 687
 - CLP_BAD_STRING_VALUES, 688
 - CLP_BARRIER_ABS_DROPPED, 688
 - CLP_BARRIER_ABS_ERROR, 688
 - CLP_BARRIER_ACCURACY, 687
 - CLP_BARRIER_CLOSE_TO_OPTIMAL, 687
 - CLP_BARRIER_COMPLEMENTARITY, 687
 - CLP_BARRIER_DIAGONAL, 688
 - CLP_BARRIER_DUALINF, 688
 - CLP_BARRIER_END, 687
 - CLP_BARRIER_EXIT, 687
 - CLP_BARRIER_EXIT2, 687
 - CLP_BARRIER_FEASIBLE, 688
 - CLP_BARRIER_GONE_INFEASIBLE, 687
 - CLP_BARRIER_INFO, 687
 - CLP_BARRIER_ITERATION, 687
 - CLP_BARRIER_KILLED, 688
 - CLP_BARRIER_KKT, 688
 - CLP_BARRIER_MU, 687
 - CLP_BARRIER_NEGATIVE_GAPS, 687
 - CLP_BARRIER_OBJECTIVE_GAP, 687
 - CLP_BARRIER_REDUCING, 688
 - CLP_BARRIER_SAFE, 687
 - CLP_BARRIER_SCALING, 687
 - CLP_BARRIER_SLACKS, 688
 - CLP_BARRIER_STEP, 688

- CLP_BARRIER_STOPPING, [687](#)
- CLP_COMPLICATED_MODEL, [688](#)
- CLP_CRASH, [687](#)
- CLP_CRUNCH_STATS, [688](#)
- CLP_DUAL_BOUNDS, [686](#)
- CLP_DUAL_CHECK, [687](#)
- CLP_DUAL_CHECKB, [686](#)
- CLP_DUAL_ORIGINAL, [686](#)
- CLP_DUMMY_END, [688](#)
- CLP_DUPLICATEELEMENTS, [686](#)
- CLP_EMPTY_PROBLEM, [687](#)
- CLP_END_VALUES_PASS, [687](#)
- CLP_FATHOM_FINISH, [688](#)
- CLP_FATHOM_SOLUTION, [688](#)
- CLP_FATHOM_STATUS, [688](#)
- CLP_GENERAL, [688](#)
- CLP_GENERAL2, [688](#)
- CLP_GENERAL_WARNING, [688](#)
- CLP_IDIOT_ITERATION, [687](#)
- CLP_IMPORT_ERRORS, [687](#)
- CLP_IMPORT_RESULT, [687](#)
- CLP_INFEASIBLE, [687](#)
- CLP_INITIALIZE_STEEP, [687](#)
- CLP_INTERVAL_TIMING, [687](#)
- CLP_LOOP, [687](#)
- CLP_MATRIX_CHANGE, [687](#)
- CLP_MODIFIEDBOUNDS, [686](#)
- CLP_PACKEDSCALE_FINAL, [687](#)
- CLP_PACKEDSCALE_FORGET, [687](#)
- CLP_PACKEDSCALE_INITIAL, [687](#)
- CLP_PACKEDSCALE_WHILE, [687](#)
- CLP_PARAMETRICS_STATS, [688](#)
- CLP_PARAMETRICS_STATS2, [688](#)
- CLP_POSSIBLELOOP, [686](#)
- CLP_PRIMAL_DJ, [687](#)
- CLP_PRIMAL_OPTIMAL, [686](#)
- CLP_PRIMAL_ORIGINAL, [686](#)
- CLP_PRIMAL_WEIGHT, [686](#)
- CLP_QUADRATIC_BOTH, [687](#)
- CLP_QUADRATIC_PRIMAL_DETAILS, [687](#)
- CLP_RIM_SCALE, [688](#)
- CLP_RIMSTATISTICS1, [686](#)
- CLP_RIMSTATISTICS2, [686](#)
- CLP_RIMSTATISTICS3, [686](#)
- CLP_SIMPLEX_ACCURACY, [686](#)
- CLP_SIMPLEX_BADFACTOR, [686](#)
- CLP_SIMPLEX_BOUNDTIGHTEN, [686](#)
- CLP_SIMPLEX_ERROR, [686](#)
- CLP_SIMPLEX_FINISHED, [686](#)
- CLP_SIMPLEX_FLAG, [686](#)
- CLP_SIMPLEX_FREEIN, [687](#)
- CLP_SIMPLEX_GIVINGUP, [686](#)
- CLP_SIMPLEX_HOUSE1, [686](#)
- CLP_SIMPLEX_HOUSE2, [686](#)
- CLP_SIMPLEX_INFEASIBILITIES, [686](#)
- CLP_SIMPLEX_INFEASIBLE, [686](#)
- CLP_SIMPLEX_INTERRUPT, [686](#)
- CLP_SIMPLEX_NONLINEAR, [687](#)
- CLP_SIMPLEX_PERTURB, [686](#)
- CLP_SIMPLEX_PIVOTROW, [687](#)
- CLP_SIMPLEX_STATUS, [686](#)
- CLP_SIMPLEX_STOPPED, [686](#)
- CLP_SIMPLEX_UNBOUNDED, [686](#)
- CLP_SINGULARITIES, [686](#)
- CLP_SLP_ITER, [688](#)
- CLP_SMALLELEMENTS, [686](#)
- CLP_SPRINT, [687](#)
- CLP_TIMING, [687](#)
- CLP_UNABLE_OPEN, [687](#)
- ClpNameDiscipline
 - ClpParameters.hpp, [694](#)
- ClpObjOffset
 - ClpParameters.hpp, [695](#)
- ClpParameters.hpp
 - ClpDualObjectiveLimit, [695](#)
 - ClpDualTolerance, [695](#)
 - ClpLastDbIParam, [695](#)
 - ClpLastIntParam, [695](#)
 - ClpLastStrParam, [695](#)
 - ClpMaxNumIteration, [694](#)
 - ClpMaxNumIterationHotStart, [694](#)
 - ClpMaxSeconds, [695](#)
 - ClpNameDiscipline, [694](#)
 - ClpObjOffset, [695](#)
 - ClpPresolveTolerance, [695](#)
 - ClpPrimalObjectiveLimit, [695](#)
 - ClpPrimalTolerance, [695](#)
 - ClpProbName, [695](#)
- ClpPresolveTolerance
 - ClpParameters.hpp, [695](#)
- ClpPrimalColumnSteepest
 - keep, [426](#)
 - normal, [426](#)
- ClpPrimalObjectiveLimit
 - ClpParameters.hpp, [695](#)
- ClpPrimalTolerance
 - ClpParameters.hpp, [695](#)
- ClpProbName
 - ClpParameters.hpp, [695](#)
- ClpSimplex
 - atLowerBound, [452](#)
 - atUpperBound, [452](#)
 - basic, [452](#)
 - bothFake, [452](#)
 - isFixed, [452](#)
 - isFree, [452](#)
 - lowerFake, [452](#)
 - noFake, [452](#)

- superBasic, [452](#)
- upperFake, [452](#)
- ClpSolve
 - automatic, [513](#)
 - notImplemented, [513](#)
 - presolveNumber, [513](#)
 - presolveNumberCost, [513](#)
 - presolveOff, [513](#)
 - presolveOn, [513](#)
 - useBarrier, [513](#)
 - useBarrierNoCross, [513](#)
 - useDual, [513](#)
 - usePrimal, [513](#)
 - usePrimalorSprint, [513](#)
- Clp_C_Interface.h
 - Clp_Solve, [664](#)
 - Clp_addColumns, [665](#)
 - Clp_addRows, [665](#)
 - Clp_algorithm, [671](#)
 - Clp_checkSolution, [671](#)
 - Clp_chgColumnLower, [665](#)
 - Clp_chgColumnUpper, [665](#)
 - Clp_chgObjCoefficients, [666](#)
 - Clp_chgRowLower, [665](#)
 - Clp_chgRowUpper, [665](#)
 - Clp_clearCallBack, [669](#)
 - Clp_columnLower, [668](#)
 - Clp_columnName, [669](#)
 - Clp_columnUpper, [668](#)
 - Clp_copyInIntegerInformation, [665](#)
 - Clp_copyNames, [666](#)
 - Clp_copyinStatus, [668](#)
 - Clp_crash, [670](#)
 - Clp_deleteColumns, [665](#)
 - Clp_deleteIntegerInformation, [665](#)
 - Clp_deleteModel, [664](#)
 - Clp_deleteRows, [665](#)
 - Clp_dropNames, [666](#)
 - Clp_dual, [670](#)
 - Clp_dualBound, [670](#)
 - Clp_dualColumnSolution, [667](#)
 - Clp_dualFeasible, [670](#)
 - Clp_dualObjectiveLimit, [666](#)
 - Clp_dualRowSolution, [667](#)
 - Clp_dualTolerance, [666](#)
 - Clp_getColLower, [673](#)
 - Clp_getColSolution, [672](#)
 - Clp_getColUpper, [673](#)
 - Clp_getColumnStatus, [668](#)
 - Clp_getElements, [668](#)
 - Clp_getIndices, [668](#)
 - Clp_getIterationCount, [671](#)
 - Clp_getNumCols, [671](#)
 - Clp_getNumElements, [668](#)
 - Clp_getNumRows, [671](#)
 - Clp_getObjCoefficients, [673](#)
 - Clp_getObjSense, [672](#)
 - Clp_getObjValue, [673](#)
 - Clp_getReducedCost, [672](#)
 - Clp_getRowActivity, [672](#)
 - Clp_getRowLower, [672](#)
 - Clp_getRowPrice, [672](#)
 - Clp_getRowStatus, [669](#)
 - Clp_getRowUpper, [673](#)
 - Clp_getSmallElementValue, [673](#)
 - Clp_getUserPointer, [669](#)
 - Clp_getVectorLengths, [668](#)
 - Clp_getVectorStarts, [668](#)
 - Clp_hitMaximumIterations, [667](#)
 - Clp_idiot, [670](#)
 - Clp_infeasibilityCost, [670](#)
 - Clp_infeasibilityRay, [668](#)
 - Clp_initialBarrierNoCrossSolve, [670](#)
 - Clp_initialBarrierSolve, [670](#)
 - Clp_initialDualSolve, [669](#)
 - Clp_initialPrimalSolve, [669](#)
 - Clp_initialSolve, [669](#)
 - Clp_initialSolveWithOptions, [669](#)
 - Clp_integerInformation, [668](#)
 - Clp_isAbandoned, [672](#)
 - Clp_isDualObjectiveLimitReached, [672](#)
 - Clp_isIterationLimitReached, [672](#)
 - Clp_isPrimalObjectiveLimitReached, [672](#)
 - Clp_isProvenDualInfeasible, [672](#)
 - Clp_isProvenOptimal, [672](#)
 - Clp_isProvenPrimalInfeasible, [672](#)
 - Clp_lengthNames, [669](#)
 - Clp_loadProblem, [664](#)
 - Clp_loadQuadraticObjective, [665](#)
 - Clp_logLevel, [669](#)
 - Clp_maximumSeconds, [667](#)
 - Clp_newModel, [664](#)
 - Clp_numberColumns, [666](#)
 - Clp_numberDualInfeasibilities, [671](#)
 - Clp_numberIterations, [666](#)
 - Clp_numberPrimalInfeasibilities, [671](#)
 - Clp_numberRows, [666](#)
 - Clp_objective, [668](#)
 - Clp_objectiveOffset, [666](#)
 - Clp_objectiveValue, [668](#)
 - Clp_optimizationDirection, [667](#)
 - Clp_perturbation, [671](#)
 - Clp_primal, [670](#)
 - Clp_primalColumnSolution, [667](#)
 - Clp_primalFeasible, [670](#)
 - Clp_primalRowSolution, [667](#)
 - Clp_primalTolerance, [666](#)
 - Clp_printModel, [673](#)

- Clp_problemName, [666](#)
- Clp_readMps, [665](#)
- Clp_registerCallBack, [669](#)
- Clp_resize, [665](#)
- Clp_restoreModel, [671](#)
- Clp_rowLower, [667](#)
- Clp_rowName, [669](#)
- Clp_rowUpper, [668](#)
- Clp_saveModel, [671](#)
- Clp_scaling, [670](#)
- Clp_scalingFlag, [670](#)
- Clp_secondaryStatus, [667](#)
- Clp_setAlgorithm, [671](#)
- Clp_setColSolution, [672](#)
- Clp_setColumnStatus, [669](#)
- Clp_setDualBound, [670](#)
- Clp_setDualObjectiveLimit, [666](#)
- Clp_setDualTolerance, [666](#)
- Clp_setInfeasibilityCost, [670](#)
- Clp_setLogLevel, [669](#)
- Clp_setMaximumIterations, [667](#)
- Clp_setMaximumSeconds, [667](#)
- Clp_setNumberIterations, [666](#)
- Clp_setObjSense, [672](#)
- Clp_setObjectiveOffset, [666](#)
- Clp_setOptimizationDirection, [667](#)
- Clp_setPerturbation, [671](#)
- Clp_setPrimalTolerance, [666](#)
- Clp_setProblemName, [666](#)
- Clp_setProblemStatus, [667](#)
- Clp_setRowStatus, [669](#)
- Clp_setSecondaryStatus, [667](#)
- Clp_setSmallElementValue, [673](#)
- Clp_setUserPointer, [669](#)
- Clp_status, [667](#)
- Clp_statusArray, [668](#)
- Clp_statusExists, [668](#)
- Clp_sumDualInfeasibilities, [671](#)
- Clp_sumPrimalInfeasibilities, [671](#)
- Clp_unboundedRay, [668](#)
- ClpSolve_delete, [664](#)
- ClpSolve_doDoubleton, [674](#)
- ClpSolve_doDual, [673](#)
- ClpSolve_doDupcol, [674](#)
- ClpSolve_doDuprow, [674](#)
- ClpSolve_doForcing, [674](#)
- ClpSolve_doImpliedFree, [674](#)
- ClpSolve_doSingleton, [674](#)
- ClpSolve_doSingletonColumn, [674](#)
- ClpSolve_doTighten, [674](#)
- ClpSolve_doTripleton, [674](#)
- ClpSolve_getExtraInfo, [673](#)
- ClpSolve_getPresolvePasses, [673](#)
- ClpSolve_getPresolveType, [673](#)
- ClpSolve_getSolveType, [673](#)
- ClpSolve_getSpecialOption, [673](#)
- ClpSolve_infeasibleReturn, [673](#)
- ClpSolve_new, [664](#)
- ClpSolve_presolveActions, [674](#)
- ClpSolve_setDoDoubleton, [674](#)
- ClpSolve_setDoDual, [674](#)
- ClpSolve_setDoDupcol, [674](#)
- ClpSolve_setDoDuprow, [674](#)
- ClpSolve_setDoForcing, [674](#)
- ClpSolve_setDoImpliedFree, [674](#)
- ClpSolve_setDoSingleton, [674](#)
- ClpSolve_setDoSingletonColumn, [674](#)
- ClpSolve_setDoTighten, [674](#)
- ClpSolve_setDoTripleton, [674](#)
- ClpSolve_setInfeasibleReturn, [673](#)
- ClpSolve_setPresolveActions, [674](#)
- ClpSolve_setPresolveType, [673](#)
- ClpSolve_setSolveType, [673](#)
- ClpSolve_setSpecialOption, [673](#)
- ClpSolve_setSubstitution, [674](#)
- ClpSolve_substitution, [674](#)
- maximumIterations, [667](#)
- Clp_Solve
 - Clp_C_Interface.h, [664](#)
- Clp_addColumns
 - Clp_C_Interface.h, [665](#)
- Clp_addRows
 - Clp_C_Interface.h, [665](#)
- Clp_algorithm
 - Clp_C_Interface.h, [671](#)
- Clp_ampl.h
 - ampl_obj_prec, [657](#)
 - freeArgs, [657](#)
 - freeArrays1, [657](#)
 - freeArrays2, [657](#)
 - readAmpl, [657](#)
 - writeAmpl, [657](#)
- Clp_checkSolution
 - Clp_C_Interface.h, [671](#)
- Clp_chgColumnLower
 - Clp_C_Interface.h, [665](#)
- Clp_chgColumnUpper
 - Clp_C_Interface.h, [665](#)
- Clp_chgObjCoefficients
 - Clp_C_Interface.h, [666](#)
- Clp_chgRowLower
 - Clp_C_Interface.h, [665](#)
- Clp_chgRowUpper
 - Clp_C_Interface.h, [665](#)
- Clp_clearCallBack
 - Clp_C_Interface.h, [669](#)
- Clp_columnLower
 - Clp_C_Interface.h, [668](#)

Clp_columnName
 Clp_C_Interface.h, 669

Clp_columnUpper
 Clp_C_Interface.h, 668

Clp_copyInIntegerInformation
 Clp_C_Interface.h, 665

Clp_copyNames
 Clp_C_Interface.h, 666

Clp_copyinStatus
 Clp_C_Interface.h, 668

Clp_crash
 Clp_C_Interface.h, 670

Clp_deleteColumns
 Clp_C_Interface.h, 665

Clp_deleteIntegerInformation
 Clp_C_Interface.h, 665

Clp_deleteModel
 Clp_C_Interface.h, 664

Clp_deleteRows
 Clp_C_Interface.h, 665

Clp_dropNames
 Clp_C_Interface.h, 666

Clp_dual
 Clp_C_Interface.h, 670

Clp_dualBound
 Clp_C_Interface.h, 670

Clp_dualColumnSolution
 Clp_C_Interface.h, 667

Clp_dualFeasible
 Clp_C_Interface.h, 670

Clp_dualObjectiveLimit
 Clp_C_Interface.h, 666

Clp_dualRowSolution
 Clp_C_Interface.h, 667

Clp_dualTolerance
 Clp_C_Interface.h, 666

Clp_getColLower
 Clp_C_Interface.h, 673

Clp_getColSolution
 Clp_C_Interface.h, 672

Clp_getColUpper
 Clp_C_Interface.h, 673

Clp_getColumnStatus
 Clp_C_Interface.h, 668

Clp_getElements
 Clp_C_Interface.h, 668

Clp_getIndices
 Clp_C_Interface.h, 668

Clp_getIterationCount
 Clp_C_Interface.h, 671

Clp_getNumCols
 Clp_C_Interface.h, 671

Clp_getNumElements
 Clp_C_Interface.h, 668

Clp_getNumRows
 Clp_C_Interface.h, 671

Clp_getObjCoefficients
 Clp_C_Interface.h, 673

Clp_getObjSense
 Clp_C_Interface.h, 672

Clp_getObjValue
 Clp_C_Interface.h, 673

Clp_getReducedCost
 Clp_C_Interface.h, 672

Clp_getRowActivity
 Clp_C_Interface.h, 672

Clp_getRowLower
 Clp_C_Interface.h, 672

Clp_getRowPrice
 Clp_C_Interface.h, 672

Clp_getRowStatus
 Clp_C_Interface.h, 669

Clp_getRowUpper
 Clp_C_Interface.h, 673

Clp_getSmallElementValue
 Clp_C_Interface.h, 673

Clp_getUserPointer
 Clp_C_Interface.h, 669

Clp_getVectorLengths
 Clp_C_Interface.h, 668

Clp_getVectorStarts
 Clp_C_Interface.h, 668

Clp_hitMaximumIterations
 Clp_C_Interface.h, 667

Clp_idiot
 Clp_C_Interface.h, 670

Clp_infeasibilityCost
 Clp_C_Interface.h, 670

Clp_infeasibilityRay
 Clp_C_Interface.h, 668

Clp_initialBarrierNoCrossSolve
 Clp_C_Interface.h, 670

Clp_initialBarrierSolve
 Clp_C_Interface.h, 670

Clp_initialDualSolve
 Clp_C_Interface.h, 669

Clp_initialPrimalSolve
 Clp_C_Interface.h, 669

Clp_initialSolve
 Clp_C_Interface.h, 669

Clp_initialSolveWithOptions
 Clp_C_Interface.h, 669

Clp_integerInformation
 Clp_C_Interface.h, 668

Clp_isAbandoned
 Clp_C_Interface.h, 672

Clp_isDualObjectiveLimitReached
 Clp_C_Interface.h, 672

- Clp_isIterationLimitReached
 - Clp_C_Interface.h, [672](#)
- Clp_isPrimalObjectiveLimitReached
 - Clp_C_Interface.h, [672](#)
- Clp_isProvenDualInfeasible
 - Clp_C_Interface.h, [672](#)
- Clp_isProvenOptimal
 - Clp_C_Interface.h, [672](#)
- Clp_isProvenPrimalInfeasible
 - Clp_C_Interface.h, [672](#)
- Clp_lengthNames
 - Clp_C_Interface.h, [669](#)
- Clp_loadProblem
 - Clp_C_Interface.h, [664](#)
- Clp_loadQuadraticObjective
 - Clp_C_Interface.h, [665](#)
- Clp_logLevel
 - Clp_C_Interface.h, [669](#)
- Clp_maximumSeconds
 - Clp_C_Interface.h, [667](#)
- Clp_newModel
 - Clp_C_Interface.h, [664](#)
- Clp_numberColumns
 - Clp_C_Interface.h, [666](#)
- Clp_numberDualInfeasibilities
 - Clp_C_Interface.h, [671](#)
- Clp_numberIterations
 - Clp_C_Interface.h, [666](#)
- Clp_numberPrimalInfeasibilities
 - Clp_C_Interface.h, [671](#)
- Clp_numberRows
 - Clp_C_Interface.h, [666](#)
- Clp_objective
 - Clp_C_Interface.h, [668](#)
- Clp_objectiveOffset
 - Clp_C_Interface.h, [666](#)
- Clp_objectiveValue
 - Clp_C_Interface.h, [668](#)
- Clp_optimizationDirection
 - Clp_C_Interface.h, [667](#)
- Clp_perturbation
 - Clp_C_Interface.h, [671](#)
- Clp_primal
 - Clp_C_Interface.h, [670](#)
- Clp_primalColumnSolution
 - Clp_C_Interface.h, [667](#)
- Clp_primalFeasible
 - Clp_C_Interface.h, [670](#)
- Clp_primalRowSolution
 - Clp_C_Interface.h, [667](#)
- Clp_primalTolerance
 - Clp_C_Interface.h, [666](#)
- Clp_printModel
 - Clp_C_Interface.h, [673](#)
- Clp_problemName
 - Clp_C_Interface.h, [666](#)
- Clp_readMps
 - Clp_C_Interface.h, [665](#)
- Clp_registerCallBack
 - Clp_C_Interface.h, [669](#)
- Clp_resize
 - Clp_C_Interface.h, [665](#)
- Clp_restoreModel
 - Clp_C_Interface.h, [671](#)
- Clp_rowLower
 - Clp_C_Interface.h, [667](#)
- Clp_rowName
 - Clp_C_Interface.h, [669](#)
- Clp_rowUpper
 - Clp_C_Interface.h, [668](#)
- Clp_saveModel
 - Clp_C_Interface.h, [671](#)
- Clp_scaling
 - Clp_C_Interface.h, [670](#)
- Clp_scalingFlag
 - Clp_C_Interface.h, [670](#)
- Clp_secondaryStatus
 - Clp_C_Interface.h, [667](#)
- Clp_setAlgorithm
 - Clp_C_Interface.h, [671](#)
- Clp_setColSolution
 - Clp_C_Interface.h, [672](#)
- Clp_setColumnStatus
 - Clp_C_Interface.h, [669](#)
- Clp_setDualBound
 - Clp_C_Interface.h, [670](#)
- Clp_setDualObjectiveLimit
 - Clp_C_Interface.h, [666](#)
- Clp_setDualTolerance
 - Clp_C_Interface.h, [666](#)
- Clp_setInfeasibilityCost
 - Clp_C_Interface.h, [670](#)
- Clp_setLogLevel
 - Clp_C_Interface.h, [669](#)
- Clp_setMaximumIterations
 - Clp_C_Interface.h, [667](#)
- Clp_setMaximumSeconds
 - Clp_C_Interface.h, [667](#)
- Clp_setNumberIterations
 - Clp_C_Interface.h, [666](#)
- Clp_setObjSense
 - Clp_C_Interface.h, [672](#)
- Clp_setObjectiveOffset
 - Clp_C_Interface.h, [666](#)
- Clp_setOptimizationDirection
 - Clp_C_Interface.h, [667](#)
- Clp_setPerturbation
 - Clp_C_Interface.h, [671](#)

- Clp_setPrimalTolerance
 - Clp_C_Interface.h, 666
- Clp_setProblemName
 - Clp_C_Interface.h, 666
- Clp_setProblemStatus
 - Clp_C_Interface.h, 667
- Clp_setRowStatus
 - Clp_C_Interface.h, 669
- Clp_setSecondaryStatus
 - Clp_C_Interface.h, 667
- Clp_setSmallElementValue
 - Clp_C_Interface.h, 673
- Clp_setUserPointer
 - Clp_C_Interface.h, 669
- Clp_status
 - Clp_C_Interface.h, 667
- Clp_statusArray
 - Clp_C_Interface.h, 668
- Clp_statusExists
 - Clp_C_Interface.h, 668
- Clp_sumDualInfeasibilities
 - Clp_C_Interface.h, 671
- Clp_sumPrimalInfeasibilities
 - Clp_C_Interface.h, 671
- Clp_unboundedRay
 - Clp_C_Interface.h, 668
- ClpCholeskyBase, 147
 - ~ClpCholeskyBase, 150
 - choleskyCondition, 151
 - choleskyCondition_, 154
 - choleskyRow_, 155
 - choleskyStart_, 155
 - clique_, 156
 - clone, 153
 - ClpCholeskyBase, 150
 - ClpCholeskyBase, 150
 - dense_, 157
 - denseColumn_, 156
 - denseThreshold_, 157
 - diagonal, 152
 - diagonal_, 155
 - doKKT_, 154
 - doubleParameters_, 156
 - factorize, 151
 - factorizePart2, 153
 - firstDense_, 156
 - getDoubleParameter, 153
 - getIntegerParameter, 152
 - goDense, 151
 - goDense_, 154
 - indexStart_, 155
 - integerParameters_, 156
 - kkt, 152
 - link_, 156
 - model_, 154
 - numberRows, 152
 - numberRows_, 154
 - numberRowsDropped, 151
 - numberRowsDropped_, 155
 - numberTrials_, 154
 - operator=, 153
 - order, 150
 - permute_, 155
 - permuteInverse_, 155
 - preOrder, 154
 - rank, 152
 - resetRowsDropped, 151
 - rowCopy_, 156
 - rowsDropped, 151
 - rowsDropped_, 155
 - setDoubleParameter, 153
 - setGoDense, 151
 - setIntegerParameter, 152
 - setKKT, 152
 - setModel, 153
 - setType, 153
 - size, 152
 - sizeFactor_, 156
 - sizeIndex_, 156
 - solve, 151, 153
 - solveKKT, 151
 - sparseFactor, 152
 - sparseFactor_, 155
 - status, 151
 - status_, 154
 - symbolic, 150
 - symbolic1, 153
 - symbolic2, 153
 - type, 153
 - type_, 154
 - updateDense, 154
 - whichDense_, 156
 - workDouble, 152
 - workDouble_, 155
 - workInteger_, 156
- ClpCholeskyBase.hpp
 - CHOL_SMALL_VALUE, 675
 - CLP_LONG_CHOLESKY, 675
 - longDouble, 675
- ClpCholeskyCfactor
 - ClpCholeskyDense.hpp, 676
- ClpCholeskyCfactorLeaf
 - ClpCholeskyDense.hpp, 676
- ClpCholeskyCrecRec
 - ClpCholeskyDense.hpp, 676
- ClpCholeskyCrecRecLeaf
 - ClpCholeskyDense.hpp, 676
- ClpCholeskyCrecTri

- ClpCholeskyDense.hpp, 676
- ClpCholeskyCrecTriLeaf
 - ClpCholeskyDense.hpp, 676
- ClpCholeskyCtriRec
 - ClpCholeskyDense.hpp, 676
- ClpCholeskyCtriRecLeaf
 - ClpCholeskyDense.hpp, 676
- ClpCholeskyDense, 157
 - ~ClpCholeskyDense, 158
 - aMatrix, 159
 - bNumber, 159
 - clone, 160
 - ClpCholeskyDense, 158
 - ClpCholeskyDense, 158
 - diagonal, 159
 - factorize, 159
 - factorizePart2, 159
 - factorizePart3, 159
 - operator=, 160
 - order, 158
 - reserveSpace, 159
 - solve, 159
 - solveB1, 159
 - solveB2, 159
 - solveF1, 159
 - solveF2, 159
 - space, 159
 - symbolic, 158
- ClpCholeskyDense.hpp
 - ClpCholeskyCfactor, 676
 - ClpCholeskyCfactorLeaf, 676
 - ClpCholeskyCrecRec, 676
 - ClpCholeskyCrecRecLeaf, 676
 - ClpCholeskyCrecTri, 676
 - ClpCholeskyCrecTriLeaf, 676
 - ClpCholeskyCtriRec, 676
 - ClpCholeskyCtriRecLeaf, 676
 - ClpCholeskySpawn, 676
- ClpCholeskyDenseC, 160
 - a, 160
 - diagonal_, 160
 - doubleParameters_, 161
 - integerParameters_, 161
 - n, 161
 - numberBlocks, 161
 - rowsDropped, 160
 - work, 160
- ClpCholeskyMumps, 161
 - ~ClpCholeskyMumps, 162
 - clone, 162
 - ClpCholeskyMumps, 162
 - ClpCholeskyMumps, 162
 - factorize, 162
 - order, 162
 - solve, 162
 - symbolic, 162
- ClpCholeskyMumps.hpp
 - DMUMPS_STRUC_C, 677
- ClpCholeskySpawn
 - ClpCholeskyDense.hpp, 676
- ClpCholeskyTaucs, 163
 - ~ClpCholeskyTaucs, 164
 - clone, 164
 - ClpCholeskyTaucs, 164
 - ClpCholeskyTaucs, 164
 - factorize, 164
 - operator=, 164
 - order, 164
 - solve, 164
 - symbolic, 164
- ClpCholeskyUfl, 165
 - ~ClpCholeskyUfl, 166
 - clone, 166
 - ClpCholeskyUfl, 166
 - ClpCholeskyUfl, 166
 - factorize, 166
 - order, 166
 - solve, 166
 - symbolic, 166
- ClpCholeskyUfl.hpp
 - cholmod_common, 678
 - cholmod_factor, 678
- ClpCholeskyWssmp, 167
 - ~ClpCholeskyWssmp, 167
 - clone, 168
 - ClpCholeskyWssmp, 167, 168
 - ClpCholeskyWssmp, 167, 168
 - factorize, 168
 - operator=, 168
 - order, 168
 - solve, 168
 - symbolic, 168
- ClpCholeskyWssmpKKT, 168
 - ~ClpCholeskyWssmpKKT, 169
 - clone, 170
 - ClpCholeskyWssmpKKT, 169
 - ClpCholeskyWssmpKKT, 169
 - factorize, 170
 - operator=, 170
 - order, 170
 - solve, 170
 - solveKKT, 170
 - symbolic, 170
- ClpConstraint, 170
 - ~ClpConstraint, 172
 - clone, 173
 - ClpConstraint, 172
 - ClpConstraint, 172

- deleteSome, 173
- functionValue, 172, 173
- functionValue_, 174
- gradient, 172
- lastGradient_, 174
- markNonlinear, 173
- markNonzero, 173
- newXValues, 174
- numberCoefficients, 173
- offset, 174
- offset_, 174
- operator=, 173
- reallyScale, 173
- resize, 172
- rowNumber, 173
- rowNumber_, 174
- type, 173
- type_, 174
- ClpConstraintLinear, 174
 - ~ClpConstraintLinear, 176
 - clone, 177
 - ClpConstraintLinear, 176
 - ClpConstraintLinear, 176
 - coefficient, 177
 - column, 177
 - deleteSome, 176
 - gradient, 176
 - markNonlinear, 176
 - markNonzero, 177
 - numberCoefficients, 177
 - numberColumns, 177
 - operator=, 177
 - reallyScale, 176
 - resize, 176
- ClpConstraintQuadratic, 177
 - ~ClpConstraintQuadratic, 179
 - clone, 180
 - ClpConstraintQuadratic, 179
 - ClpConstraintQuadratic, 179
 - coefficient, 180
 - column, 180
 - deleteSome, 179
 - gradient, 179
 - markNonlinear, 179
 - markNonzero, 180
 - numberCoefficients, 180
 - numberColumns, 180
 - operator=, 180
 - reallyScale, 179
 - resize, 179
 - start, 180
- ClpCopyOfArray
 - ClpParameters.hpp, 695, 696
- ClpDataSave, 181
 - ~ClpDataSave, 181
 - acceptablePivot_, 182
 - ClpDataSave, 181
 - ClpDataSave, 181
 - dualBound_, 182
 - forceFactorization_, 182
 - infeasibilityCost_, 182
 - objectiveScale_, 182
 - operator=, 182
 - perturbation_, 182
 - pivotTolerance_, 182
 - scalingFlag_, 182
 - sparseThreshold_, 182
 - specialOptions_, 182
 - zeroFactorizationTolerance_, 182
 - zeroSimplexTolerance_, 182
- ClpDbiParam
 - ClpParameters.hpp, 695
- ClpDisasterHandler, 183
 - ~ClpDisasterHandler, 184
 - check, 184
 - clone, 184
 - ClpDisasterHandler, 184
 - ClpDisasterHandler, 184
 - intoSimplex, 184
 - model_, 185
 - operator=, 184
 - saveInfo, 184
 - setSimplex, 184
 - simplex, 184
 - typeOfDisaster, 184
- ClpDisjointCopyN
 - ClpParameters.hpp, 695
- ClpDualRowDantzig, 185
 - ~ClpDualRowDantzig, 186
 - clone, 186
 - ClpDualRowDantzig, 186
 - ClpDualRowDantzig, 186
 - operator=, 186
 - pivotRow, 186
 - updatePrimalSolution, 186
 - updateWeights, 186
- ClpDualRowPivot, 187
 - ~ClpDualRowPivot, 188
 - checkAccuracy, 189
 - clearArrays, 189
 - clone, 190
 - ClpDualRowPivot, 188
 - ClpDualRowPivot, 188
 - looksOptimal, 189
 - maximumPivotsChanged, 189
 - model, 190
 - model_, 190
 - operator=, 190

- pivotRow, 189
 - saveWeights, 189
 - setModel, 190
 - type, 190
 - type_, 190
 - unrollWeights, 189
 - updatePrimalSolution, 189
 - updateWeights, 189
- ClpDualRowSteepest, 190
 - ~ClpDualRowSteepest, 192
 - clearArrays, 193
 - clone, 193
 - ClpDualRowSteepest, 192
 - ClpDualRowSteepest, 192
 - fill, 193
 - looksOptimal, 193
 - maximumPivotsChanged, 193
 - mode, 194
 - operator=, 193
 - Persistence, 192
 - persistence, 194
 - pivotRow, 192
 - saveWeights, 193
 - setPersistence, 194
 - unrollWeights, 193
 - updatePrimalSolution, 193
 - updateWeights, 192
- ClpDummyMatrix, 194
 - ~ClpDummyMatrix, 196
 - add, 198
 - clone, 200
 - ClpDummyMatrix, 196
 - ClpDummyMatrix, 196
 - countBasis, 198
 - deleteCols, 198
 - deleteRows, 198
 - fillBasis, 198
 - getElements, 197
 - getIndices, 197
 - getNumCols, 197
 - getNumElements, 197
 - getNumRows, 197
 - getPackedMatrix, 197
 - getVectorLengths, 197
 - getVectorStarts, 197
 - isColOrdered, 197
 - numberColumns_, 200
 - numberElements_, 200
 - numberRows_, 200
 - operator=, 200
 - releasePackedMatrix, 199
 - reverseOrderedCopy, 198
 - subsetTransposeTimes, 199
 - times, 199
 - transposeTimes, 199
 - unpack, 198
 - unpackPacked, 198
- ClpDynamicExampleMatrix, 200
 - ~ClpDynamicExampleMatrix, 203
 - clone, 204
 - ClpDynamicExampleMatrix, 203
 - ClpDynamicExampleMatrix, 203
 - columnLowerGen, 204
 - columnLowerGen_, 206
 - columnUpperGen, 204
 - columnUpperGen_, 206
 - costGen, 204
 - costGen_, 205
 - createVariable, 203
 - dynamicStatusGen_, 205
 - elementGen, 204
 - elementGen_, 205
 - flaggedGen, 205
 - fullStartGen, 204
 - fullStartGen_, 205
 - getDynamicStatusGen, 205
 - idGen, 204
 - idGen_, 206
 - numberColumns, 204
 - numberColumns_, 205
 - operator=, 203
 - packDown, 203
 - partialPricing, 203
 - rowGen, 204
 - rowGen_, 205
 - setDynamicStatusGen, 204
 - setFlaggedGen, 205
 - startColumnGen, 204
 - startColumnGen_, 205
 - unsetFlagged, 205
- ClpDynamicMatrix, 206
 - ~ClpDynamicMatrix, 211
 - addColumn, 213
 - backToPivotRow_, 218
 - clone, 214
 - ClpDynamicMatrix, 211
 - ClpDynamicMatrix, 211
 - columnLower, 213, 215
 - columnLower_, 221
 - columnUpper, 214, 216
 - columnUpper_, 221
 - cost, 215
 - cost_, 220
 - createVariable, 213
 - dualExpanded, 212
 - DynamicStatus, 211
 - dynamicStatus, 217
 - dynamicStatus_, 221

- element, 215
- element_, 220
- firstAvailable, 216
- firstAvailable_, 219
- firstAvailableBefore_, 219
- firstDynamic, 216
- firstDynamic_, 219
- flagged, 215
- flaggedSlack, 214
- fromIndex_, 218
- generalExpanded, 212
- getDynamicStatus, 215
- getStatus, 214
- gubCrash, 213
- gubRowStatus, 217
- id, 215
- id_, 220
- infeasibilityWeight_, 219
- initialProblem, 213
- keyValue, 212
- keyVariable, 216
- keyVariable_, 218
- lastDynamic, 216
- lastDynamic_, 219
- lowerSet, 216
- lowerSet_, 218
- maximumElements_, 220
- maximumGubColumns_, 220
- model_, 218
- modifyOffset, 212
- next_, 220
- noCheck_, 219
- numberActiveSets_, 218
- numberDualInfeasibilities_, 219
- numberElements, 216
- numberElements_, 219
- numberGubColumns, 216
- numberGubColumns_, 220
- numberGubEntries, 214
- numberPrimalInfeasibilities_, 219
- numberSets, 214
- numberSets_, 218
- numberStaticRows, 216
- numberStaticRows_, 219
- objectiveOffset, 215
- objectiveOffset_, 218
- operator=, 214
- packDown, 213
- partialPricing, 212
- reducedCost, 213
- refresh, 213
- rhsOffset, 212
- row, 215
- row_, 220
- savedBestGubDual_, 217
- savedBestSet_, 217
- setDynamicStatus, 215
- setFlagged, 215
- setFlaggedSlack, 214
- setStatus, 214
- startColumn, 215
- startColumn_, 220
- startSet_, 220
- startSets, 214
- status_, 218
- sumDualInfeasibilities_, 217
- sumOfRelaxedDualInfeasibilities_, 217
- sumOfRelaxedPrimalInfeasibilities_, 217
- sumPrimalInfeasibilities_, 217
- switchOffCheck, 217
- times, 212
- toIndex_, 218
- unsetFlagged, 215
- unsetFlaggedSlack, 214
- updatePivot, 212
- upperSet, 216
- upperSet_, 218
- whichSet, 217
- writeMps, 213
- ClpEventHandler, 221
 - ~ClpEventHandler, 224
 - clone, 224
 - ClpEventHandler, 224
 - ClpEventHandler, 224
 - Event, 223
 - event, 224
 - eventWithInfo, 224
 - model_, 224
 - operator=, 224
 - setSimplex, 224
 - simplex, 224
- ClpFactorization, 225
 - ~ClpFactorization, 228
 - adjustedAreaFactor, 232
 - almostDestructor, 232
 - areaFactor, 229, 230
 - cleanUp, 233
 - clearArrays, 231
 - ClpFactorization, 228
 - ClpFactorization, 228
 - denseThreshold, 231
 - factorize, 228
 - forceOtherFactorization, 232
 - getWeights, 233
 - goDenseOrSmall, 233
 - goDenseThreshold, 233
 - goOslThreshold, 232
 - goSmallThreshold, 233

- goSparse, [233](#)
- isDenseOrSmall, [233](#)
- maximumPivots, [229](#)
- messageLevel, [231](#)
- needToReorder, [233](#)
- networkBasis, [233](#)
- numberDense, [230](#)
- numberElements, [229](#)
- numberElementsL, [230](#)
- numberElementsR, [231](#)
- numberElementsU, [230](#)
- numberOfRows, [231](#)
- operator=, [228](#)
- permute, [229](#)
- persistenceFlag, [232](#)
- pivotColumn, [229](#)
- pivotTolerance, [231](#)
- pivots, [229](#)
- relaxAccuracyCheck, [232](#)
- replaceColumn, [228](#)
- saferTolerances, [230](#)
- setBiasLU, [232](#)
- setDefaultValues, [232](#)
- setDenseThreshold, [231](#)
- setFactorization, [233](#)
- setForrestTomlin, [232](#)
- setGoDenseThreshold, [233](#)
- setGoOsiThreshold, [232](#)
- setGoSmallThreshold, [233](#)
- setPersistenceFlag, [232](#)
- setStatus, [230](#)
- sparseThreshold, [230](#)
- status, [230](#)
- timeToRefactorize, [231](#)
- updateColumn, [228](#)
- updateColumnFT, [228](#)
- updateColumnForDebug, [229](#)
- updateColumnTranspose, [229](#)
- updateTwoColumnsFT, [229](#)
- zeroTolerance, [230](#)
- ClpFactorization.hpp
 - COIN_FAST_CODE, [681](#)
- ClpFillN
 - ClpParameters.hpp, [695](#)
- ClpGubDynamicMatrix, [234](#)
 - ~ClpGubDynamicMatrix, [237](#)
 - checkFeasible, [239](#)
 - cleanData, [239](#)
 - clone, [239](#)
 - ClpGubDynamicMatrix, [237](#)
 - ClpGubDynamicMatrix, [237](#)
 - cost, [240](#)
 - cost_, [242](#)
 - DynamicStatus, [237](#)
 - dynamicStatus, [241](#)
 - dynamicStatus_, [242](#)
 - element, [240](#)
 - element_, [242](#)
 - firstAvailable, [240](#)
 - firstAvailable_, [243](#)
 - firstDynamic, [241](#)
 - firstDynamic_, [243](#)
 - flagged, [239](#)
 - fullStart, [240](#)
 - fullStart_, [242](#)
 - getDynamicStatus, [239](#)
 - gubRowStatus, [241](#)
 - id, [240](#)
 - id_, [242](#)
 - insertNonBasic, [238](#)
 - lastDynamic, [241](#)
 - lastDynamic_, [243](#)
 - lowerColumn, [240](#)
 - lowerColumn_, [242](#)
 - lowerSet, [240](#)
 - lowerSet_, [242](#)
 - numberElements, [241](#)
 - numberElements_, [243](#)
 - numberGubColumns, [240](#)
 - numberGubColumns_, [242](#)
 - objectiveOffset, [239](#)
 - objectiveOffset_, [241](#)
 - operator=, [239](#)
 - partialPricing, [238](#)
 - rhsOffset, [238](#)
 - row, [239](#)
 - row_, [241](#)
 - savedFirstAvailable_, [243](#)
 - setDynamicStatus, [239](#)
 - setFirstAvailable, [241](#)
 - setFlagged, [239](#)
 - startColumn, [239](#)
 - startColumn_, [241](#)
 - synchronize, [238](#)
 - times, [238](#)
 - unsetFlagged, [239](#)
 - updatePivot, [238](#)
 - upperColumn, [240](#)
 - upperColumn_, [242](#)
 - upperSet, [240](#)
 - upperSet_, [242](#)
 - useEffectiveRhs, [238](#)
 - whichSet, [241](#)
- ClpGubMatrix, [243](#)
 - ~ClpGubMatrix, [247](#)
 - add, [249](#)
 - backToPivotRow_, [254](#)
 - backward, [253](#)

- backward_, 254
- changeCost_, 254
- clearFlagged, 252
- clone, 251
- ClpGubMatrix, 247, 248
- ClpGubMatrix, 247, 248
- correctSequence, 251
- countBasis, 248
- dualExpanded, 250
- end, 252
- end_, 253
- extendUpdated, 250
- fillBasis, 248
- firstGub_, 256
- flagged, 252
- fromIndex_, 255
- generalExpanded, 250
- getStatus, 251
- gubSlackIn_, 255
- gubType_, 256
- hiddenRows, 249
- infeasibilityWeight_, 253
- keyVariable, 253
- keyVariable_, 254
- lastGub_, 256
- lower, 252
- lower_, 254
- model_, 255
- next_, 254
- noCheck_, 255
- numberDualInfeasibilities_, 255
- numberPrimalInfeasibilities_, 255
- numberSets, 253
- numberSets_, 255
- operator=, 251
- partialPricing, 249
- possiblePivotKey_, 255
- primalExpanded, 250
- redoSet, 251
- reverseOrderedCopy, 248
- rhsOffset, 251
- saveNumber_, 255
- saveStatus_, 254
- savedKeyVariable_, 254
- setAbove, 252
- setBelow, 252
- setFeasible, 252
- setFlagged, 252
- setStatus, 251
- start, 252
- start_, 253
- status_, 254
- subsetClone, 251
- subsetTransposeTimes, 249
- sumDualInfeasibilities_, 253
- sumOfRelaxedDualInfeasibilities_, 253
- sumOfRelaxedPrimalInfeasibilities_, 253
- sumPrimalInfeasibilities_, 253
- switchOffCheck, 253
- synchronize, 251
- toIndex_, 255
- transposeTimes, 249
- transposeTimesByRow, 249
- unpack, 248
- unpackPacked, 248
- updatePivot, 250
- upper, 252
- upper_, 254
- useEffectiveRhs, 250
- weight, 252
- ClpHashValue, 256
 - ~ClpHashValue, 257
 - addValue, 257
 - ClpHashValue, 257
 - ClpHashValue, 257
 - hash_, 258
 - index, 257
 - lastUsed_, 258
 - maxHash_, 258
 - numberEntries, 258
 - numberHash_, 258
 - operator=, 258
- ClpHashValue::CoinHashLink, 578
 - index, 578
 - next, 578
 - value, 578
- ClpHelperFunctions.hpp
 - ClpTraceDebug, 682
 - ClpTracePrint, 682
 - CoinSqrt, 682
 - getNorms, 682
 - innerProduct, 682
 - maximumAbsElement, 682
 - multiplyAdd, 682
 - setElements, 682
- ClpIntParam
 - ClpParameters.hpp, 694
- ClpInterior, 258
 - ~ClpInterior, 265
 - actualDualStep_, 276
 - actualPrimalStep_, 276
 - algorithm, 267
 - algorithm_, 279
 - baseObjectiveNorm_, 276
 - borrowModel, 266
 - checkSolution, 270
 - cholesky_, 279
 - clearFakeLower, 271

clearFakeUpper, 272
clearFixed, 270
clearFixedOrFree, 271
clearFlagged, 270
clearLowerBound, 271
clearUpperBound, 271
ClpInterior, 265
ClpInteriorUnitTest, 272
ClpInterior, 265
columnLowerWork_, 273
columnUpperWork_, 273
complementarityGap, 268
complementarityGap_, 276
cost_, 273
createWorkingData, 269
deleteWorkingData, 269
delta, 268
delta_, 275
deltaSL_, 277
deltaSU_, 277
deltaW_, 277
deltaX_, 277
deltaY_, 277
deltaZ_, 277
diagonal_, 277
diagonalNorm, 267
diagonalNorm_, 274
diagonalPerturbation, 268
diagonalPerturbation_, 275
diagonalScaleFactor_, 275
dj_, 273
dualFeasible, 267
dualObjective, 267
dualObjective_, 274
dualR, 269
dualR_, 278
errorRegion_, 276
fakeLower, 271
fakeUpper, 272
fixFixed, 269
fixed, 270
fixedOrFree, 271
flagged, 270
gamma, 268
gamma_, 275
goneDualFeasible_, 279
gonePrimalFeasible_, 279
gutsOfCopy, 269
gutsOfDelete, 269
historyInfeasibility_, 276
housekeeping, 269
isColumn, 270
largestDualError, 268
largestDualError_, 272
largestPrimalError, 268
largestPrimalError_, 272
linearPerturbation, 267
linearPerturbation_, 275
loadProblem, 266
lower_, 273
lowerBound, 271
lowerSlack_, 277
lsqrObject_, 274
maximumBarrierIterations, 269
maximumBarrierIterations_, 279
maximumBoundInfeasibility_, 275
maximumDualError_, 275
maximumRHSCheck_, 276
maximumRHSError_, 275
mu_, 274
numberComplementarityItems_, 279
numberComplementarityPairs_, 279
numberFixed, 269
objectiveNorm_, 274
operator=, 266
pdco, 266
pdcoStuff_, 274
primalDual, 267
primalFeasible, 267
primalObjective, 267
primalObjective_, 274
primalR, 269
primalR_, 278
projectionTolerance, 268
projectionTolerance_, 275
quadraticDjs, 270
rawObjectiveValue, 270
readMps, 266
returnModel, 266
rhs_, 273
rhsB_, 278
rhsC_, 278
rhsFixRegion_, 276
rhsL_, 278
rhsNorm_, 274
rhsU_, 278
rhsW_, 278
rhsZ_, 278
rowLowerWork_, 273
rowUpperWork_, 273
sanityCheck, 269
scaleFactor_, 275
sequenceWithin, 270
setAlgorithm, 267
setCholesky, 269
setDelta, 268
setDiagonalPerturbation, 268
setFakeLower, 271

- setFakeUpper, 271
- setFixed, 270
- setFixedOrFree, 270
- setFlagged, 270
- setGamma, 268
- setLinearPerturbation, 268
- setLowerBound, 271
- setMaximumBarrierIterations, 269
- setProjectionTolerance, 268
- setUpperBound, 271
- smallestInfeasibility_, 276
- solution_, 277
- solutionNorm_, 274
- stepLength_, 274
- sumDualInfeasibilities, 267
- sumDualInfeasibilities_, 272
- sumPrimalInfeasibilities, 267
- sumPrimalInfeasibilities_, 272
- targetGap_, 275
- upper_, 273
- upperBound, 271
- upperSlack_, 276
- wVec_, 278
- workArray_, 277
- worstComplementarity_, 272
- worstDirectionAccuracy_, 276
- x_, 273
- xsize_, 272
- y_, 273
- zVec_, 278
- zsize_, 272
- ClpInterior.hpp
 - ClpInteriorUnitTest, 684
 - LENGTH_HISTORY, 683
- ClpInteriorUnitTest
 - ClpInterior, 272
 - ClpInterior.hpp, 684
- ClpLinearObjective, 279
 - ~ClpLinearObjective, 281
 - clone, 282
 - ClpLinearObjective, 281
 - ClpLinearObjective, 281
 - deleteSome, 282
 - gradient, 281
 - objectiveValue, 281
 - operator=, 282
 - reallyScale, 282
 - reducedGradient, 281
 - resize, 281
 - stepLength, 281
 - subsetClone, 282
- ClpLsq, 282
 - ~ClpLsq, 284
 - borrowDiag1, 285
 - ClpLsq, 284
 - ClpLsq, 284
 - diag1_, 285
 - diag2_, 285
 - do_Lsq, 285
 - matVecMult, 285
 - model_, 285
 - ncols_, 285
 - nrows_, 285
 - operator=, 284
 - setParam, 285
- clpMatrix
 - ClpModel, 330
- ClpMatrixBase, 286
 - ~ClpMatrixBase, 291
 - add, 294, 295
 - allElementsInRange, 293
 - appendCols, 292
 - appendMatrix, 292
 - appendRows, 292
 - backToBasics, 298
 - canCombine, 297
 - canDoPartialPricing, 295
 - canGetRowCopy, 293
 - checkFeasible, 296
 - clone, 298
 - ClpMatrixBase, 291
 - ClpMatrixBase, 291
 - correctSequence, 296
 - countBasis, 293
 - createVariable, 296
 - currentWanted, 300
 - currentWanted_, 301
 - deleteCols, 292
 - deleteRows, 292
 - dualExpanded, 295
 - dubiousWeights, 294
 - endFraction, 300
 - endFraction_, 301
 - extendUpdated, 295
 - fillBasis, 293
 - generalExpanded, 296
 - getElements, 291
 - getIndices, 291
 - getNumCols, 291
 - getNumElements, 291
 - getNumRows, 291
 - getPackedMatrix, 291
 - getVectorLength, 292
 - getVectorLengths, 292
 - getVectorStarts, 291
 - hiddenRows, 295
 - isColOrdered, 291
 - lastRefresh, 299

lastRefresh_, 301
listTransposeTimes, 298
minimumGoodReducedCosts, 299
minimumGoodReducedCosts_, 302
minimumObjectsScan, 299
minimumObjectsScan_, 302
modifyCoefficient, 292
operator=, 300
originalWanted, 300
originalWanted_, 301
partialPricing, 295
primalExpanded, 295
rangeOfElements, 294
reallyScale, 294
reducedCost, 296
refresh, 294
refreshFrequency, 299
refreshFrequency_, 301
releasePackedMatrix, 295
reverseOrderedCopy, 292
rhsOffset, 299
rhsOffset_, 301
savedBestDj, 300
savedBestDj_, 301
savedBestSequence, 300
savedBestSequence_, 301
scale, 293
scaleRowCopy, 293
scaledColumnCopy, 293
setCurrentWanted, 300
setDimensions, 294
setEndFraction, 300
setMinimumGoodReducedCosts, 299
setMinimumObjectsScan, 299
setOriginalWanted, 300
setRefreshFrequency, 299
setSavedBestDj, 300
setSavedBestSequence, 300
setSkipDualCheck, 299
setStartFraction, 300
setType, 298
skipDualCheck, 299
skipDualCheck_, 302
startFraction, 299
startFraction_, 301
subsetClone, 298
subsetTimes2, 298
subsetTransposeTimes, 297
times, 296, 297
transposeTimes, 297
transposeTimes2, 298
trueSequenceIn_, 302
trueSequenceOut_, 302
type, 298
type_, 301
unpack, 294
unpackPacked, 294
updatePivot, 296
useEffectiveRhs, 298
ClpMatrixBase.hpp
 FREE_ACCEPT, 685
 FREE_BIAS, 685
ClpMessage, 302
 ClpMessage, 303
 ClpMessage, 303
ClpMessage.hpp
 CLP_Message, 686
ClpModel, 303
 ~ClpModel, 315
 addColumn, 318
 addColumnns, 318
 addRow, 317
 addRows, 317
 baseMatrix_, 342
 baseRowCopy_, 342
 borrowModel, 319
 chgColumnLower, 318
 chgColumnUpper, 318
 chgObjCoefficients, 318
 chgRowLower, 318
 chgRowUpper, 318
 cleanMatrix, 319
 clpMatrix, 330
 ClpModel, 315
 clpScaledMatrix, 330
 ClpModel, 315
 coinMessages, 333
 coinMessages_, 342
 coinMessagesPointer, 333
 columnActivity_, 338
 columnLower, 329
 columnLower_, 338
 columnName, 334
 columnNames, 334
 columnNames_, 341
 columnNamesAsChar, 337
 columnScale, 327
 columnScale_, 339
 columnUpper, 329
 columnUpper_, 339
 copy, 319
 copyColumnNames, 319
 copyInIntegerInformation, 316
 copyNames, 319
 copyRowNames, 319
 copyInStatus, 331
 createCoinModel, 320
 createEmptyMatrix, 319

dblParam_, 337
 defaultHandler, 333
 defaultHandler_, 341
 deleteColumns, 317
 deleteIntegerInformation, 316
 deleteNamesAsChar, 337
 deleteQuadraticObjective, 316
 deleteRay, 331
 deleteRows, 317
 deleteRowsAndColumns, 317
 dropNames, 319
 dual_, 338
 dualColumnSolution, 324
 dualObjectiveLimit, 321
 dualRowSolution, 324
 dualTolerance, 321
 emptyProblem, 335
 eventHandler, 333
 eventHandler_, 341
 findNetwork, 320
 generateCpp, 335
 getColLower, 329
 getColSolution, 324
 getColUpper, 329
 getColumnName, 334
 getDbIParam, 335
 getIntParam, 335
 getIterationCount, 321
 getNumCols, 320
 getNumElements, 329
 getNumRows, 320
 getObjCoefficients, 329
 getObjSense, 323
 getObjValue, 330
 getReducedCost, 324
 getRowActivity, 323
 getRowBound, 336
 getRowLower, 324
 getRowName, 334
 getRowObjCoefficients, 329
 getRowPrice, 324
 getRowUpper, 324
 getSmallElementValue, 329
 getStrParam, 335
 getTrustedUserPointer, 332
 getUserPointer, 332
 gutsOfCopy, 336
 gutsOfDelete, 336
 gutsOfLoadModel, 336
 gutsOfScaling, 336
 handler_, 341
 hitMaximumIterations, 322
 inCbcBranchAndBound, 336
 infeasibilityRay, 331
 intParam_, 340
 integerInformation, 330
 integerType_, 340
 internalRay, 331
 inverseColumnScale, 327
 inverseColumnScale_, 339
 inverseRowScale, 327
 inverseRowScale_, 339
 isAbandoned, 323
 isDualObjectiveLimitReached, 323
 isInteger, 317
 isIterationLimitReached, 323
 isPrimalObjectiveLimitReached, 323
 isProvenDualInfeasible, 323
 isProvenOptimal, 323
 isProvenPrimalInfeasible, 323
 lengthNames, 334
 lengthNames_, 341
 loadProblem, 315, 316
 loadQuadraticObjective, 316
 logLevel, 333
 matrix, 329
 matrix_, 339
 maximumColumns_, 342
 maximumInternalColumns_, 342
 maximumInternalRows_, 342
 maximumIterations, 322
 maximumRows_, 342
 maximumSeconds, 322
 messageHandler, 332
 messages, 333
 messages_, 341
 messagesPointer, 333
 modifyCoefficient, 318
 mutableColumnScale, 327
 mutableInverseColumnScale, 328
 mutableInverseRowScale, 327
 mutableRandomNumberGenerator, 333
 mutableRowScale, 327
 newLanguage, 332
 numberColumns, 320
 numberColumns_, 338
 numberIterations, 321
 numberIterations_, 340
 numberRows, 320
 numberRows_, 337
 numberThreads, 332
 numberThreads_, 341
 objective, 328
 objective_, 338
 objectiveAsObject, 334
 objectiveOffset, 321
 objectiveScale, 328
 objectiveScale_, 337

objectiveValue, 330
objectiveValue_, 337
onStopped, 337
operator=, 315
optimizationDirection, 323
optimizationDirection_, 337
passInEventHandler, 333
passInMessageHandler, 332
permanentArrays, 336
popMessageHandler, 332
presolveTolerance, 321
primalColumnSolution, 324
primalObjectiveLimit, 321
primalRowSolution, 323
primalTolerance, 320
problemName, 321
problemStatus, 322
problemStatus_, 340
pushMessageHandler, 332
randomNumberGenerator, 333
randomNumberGenerator_, 341
rawObjectiveValue, 336
ray, 331
ray_, 339
rayExists, 331
readGMPL, 316
readMps, 316
reducedCost_, 338
replaceMatrix, 330
resize, 317
returnModel, 319
rhsScale, 328
rhsScale_, 337
rowActivity_, 338
rowCopy, 329
rowCopy_, 339
rowLower, 324
rowLower_, 338
rowName, 334
rowNames, 334
rowNames_, 341
rowNamesAsChar, 337
rowObjective, 329
rowObjective_, 338
rowScale, 327
rowScale_, 339
rowUpper, 324
rowUpper_, 338
savedColumnScale_, 342
savedRowScale_, 342
scaledMatrix_, 339
scaling, 328
scalingFlag, 328
scalingFlag_, 339
secondaryStatus, 322
secondaryStatus_, 340
setClpScaledMatrix, 330
setColBounds, 325
setColLower, 325
setColSetBounds, 325
setColSolution, 324
setColUpper, 325
setColumnBounds, 325
setColumnLower, 325
setColumnName, 320
setColumnScale, 328
setColumnSetBounds, 325
setColumnUpper, 325
setContinuous, 316
setDbIParam, 335
setDefaultMessageHandler, 332
setDualObjectiveLimit, 321
setDualTolerance, 321
setIntParam, 335
setInteger, 317
setLanguage, 332
setLengthNames, 334
setLogLevel, 333
setMaximumIterations, 322
setMaximumSeconds, 322
setNewRowCopy, 330
setNumberIterations, 321
setNumberThreads, 332
setObjCoeff, 325
setObjective, 334
setObjectiveCoefficient, 324
setObjectiveOffset, 321
setObjectivePointer, 334
setObjectiveScale, 328
setObjectiveValue, 330
setOptimizationDirection, 323
setPrimalObjectiveLimit, 321
setPrimalTolerance, 320
setProblemStatus, 322
setRandomSeed, 334
setRhsScale, 328
setRowBounds, 327
setRowLower, 327
setRowName, 319
setRowObjective, 316
setRowScale, 328
setRowSetBounds, 327
setRowUpper, 327
setSecondaryStatus, 322
setSmallElementValue, 329
setSolveType, 322
setSpecialOptions, 336
setStrParam, 335

- setTrustedUserPointer, 332
- setUserPointer, 331
- setWhatsChanged, 332
- smallElement_, 337
- solveType, 321
- solveType_, 340
- specialOptions, 335
- specialOptions_, 341
- startPermanentArrays, 336
- status, 322
- status_, 340
- statusArray, 331
- statusCopy, 331
- statusExists, 331
- stopPermanentArrays, 336
- strParam_, 342
- swapRowScale, 328
- swapScaledMatrix, 330
- times, 335
- transposeTimes, 335
- trustedUserPointer_, 340
- unboundedRay, 331
- unscale, 328
- userPointer_, 340
- whatsChanged, 332
- whatsChanged_, 340
- writeMps, 320
- ClpModel.hpp
 - ALL_SAME, 690
 - BASIS_SAME, 690
 - COLUMN_LOWER_SAME, 690
 - COLUMN_UPPER_SAME, 690
 - MATRIX_SAME, 689
 - OBJECTIVE_SAME, 690
 - ROW_LOWER_SAME, 690
 - ROW_UPPER_SAME, 690
- clpModel_
 - AbcSimplex, 100
- ClpNetworkBasis, 343
 - ~ClpNetworkBasis, 344
 - ClpNetworkBasis, 344
 - ClpNetworkBasis, 344
 - factorize, 344
 - operator=, 344
 - replaceColumn, 344
 - updateColumn, 344
 - updateColumnTranspose, 344, 345
- ClpNetworkBasis.hpp
 - COIN_FAST_CODE, 691
- ClpNetworkMatrix, 345
 - ~ClpNetworkMatrix, 348
 - add, 351
 - appendCols, 349
 - appendMatrix, 349
 - appendRows, 349
 - canDoPartialPricing, 351
 - clone, 352
 - ClpNetworkMatrix, 348
 - ClpNetworkMatrix, 348
 - countBasis, 350
 - deleteCols, 349
 - deleteRows, 349
 - dubiousWeights, 350
 - fillBasis, 350
 - getElements, 349
 - getIndices, 349
 - getNumCols, 348
 - getNumElements, 348
 - getNumRows, 348
 - getPackedMatrix, 348
 - getVectorLengths, 349
 - getVectorStarts, 349
 - indices_, 353
 - isColOrdered, 348
 - lengths_, 352
 - matrix_, 352
 - numberColumns_, 353
 - numberRows_, 353
 - operator=, 352
 - partialPricing, 351
 - rangeOfElements, 350
 - releasePackedMatrix, 351
 - reverseOrderedCopy, 350
 - subsetClone, 352
 - subsetTransposeTimes, 352
 - times, 351
 - transposeTimes, 351, 352
 - trueNetwork, 352
 - trueNetwork_, 353
 - unpack, 350
 - unpackPacked, 350
- ClpNode, 353
 - ~ClpNode, 356
 - applyNode, 356
 - branchState_, 359
 - branchingValue, 357
 - branchingValue_, 358
 - changeState, 357
 - chooseVariable, 356
 - cleanUpForCrunch, 356
 - ClpNode, 356
 - ClpNode, 356
 - createArrays, 356
 - depth, 357
 - depth_, 359
 - dualSolution, 356
 - dualSolution_, 359
 - estimatedSolution, 357

- estimatedSolution_, 358
- factorization_, 358
- fathomed, 357
- fixOnReducedCosts, 356
- fixed_, 359
- flags_, 360
- gutsOfConstructor, 358
- lower_, 359
- maximumColumns_, 360
- maximumFixed_, 360
- maximumIntegers_, 360
- maximumRows_, 360
- numberFixed_, 359
- numberInfeasibilities, 357
- numberInfeasibilities_, 359
- objectiveValue, 356
- objectiveValue_, 358
- oddArraysExist, 357
- operator=, 358
- pivotVariables_, 359
- primalSolution, 356
- primalSolution_, 358
- sequence, 357
- sequence_, 359
- setObjectiveValue, 356
- status_, 358
- statusArray, 357
- sumInfeasibilities, 357
- sumInfeasibilities_, 358
- upper_, 359
- way, 357
- weights_, 358
- ClpNode::branchState, 139
 - branch, 139
 - firstBranch, 139
 - spare, 139
- ClpNodeStuff, 360
 - ~ClpNodeStuff, 362
 - ClpNodeStuff, 362
 - ClpNodeStuff, 362
 - downPseudo_, 363
 - fillPseudoCosts, 362
 - handler_, 364
 - integerIncrement_, 363
 - integerTolerance_, 363
 - large_, 364
 - maximumNodes, 363
 - maximumNodes_, 365
 - maximumSpace, 363
 - nBound_, 364
 - nDepth_, 365
 - nNodes_, 365
 - nodeCalled_, 366
 - nodeInfo_, 364
 - numberBeforeTrust_, 365
 - numberDown_, 363
 - numberDownInfeasible_, 364
 - numberIterations_, 365
 - numberNodesExplored_, 365
 - numberUp_, 363
 - numberUpInfeasible_, 364
 - operator=, 362
 - presolveType_, 365
 - priority_, 363
 - saveCosts_, 364
 - saveOptions_, 364
 - smallChange_, 363
 - solverOptions_, 365
 - startingDepth_, 365
 - stateOfSearch_, 365
 - upPseudo_, 363
 - update, 363
 - whichColumn_, 364
 - whichRow_, 364
 - zap, 362
- ClpNonLinearCost, 366
 - ~ClpNonLinearCost, 368
 - averageTheta, 370
 - changeDownInCost, 369
 - changeInCost, 369, 370
 - changeUpInCost, 369
 - checkChanged, 368
 - checkInfeasibilities, 368
 - ClpNonLinearCost, 368
 - ClpNonLinearCost, 368
 - cost, 370
 - feasibleBounds, 369
 - feasibleCost, 370
 - feasibleReportCost, 370
 - goBack, 368
 - goBackAll, 369
 - goThru, 368
 - infeasible, 371
 - largestInfeasibility, 370
 - lookBothWays, 371
 - lower, 370
 - nearest, 369
 - numberInfeasibilities, 370
 - operator=, 368
 - refresh, 369
 - refreshCosts, 369
 - setAverageTheta, 371
 - setChangeInCost, 371
 - setInfeasible, 371
 - setMethod, 371
 - setOne, 369
 - setOneOutgoing, 369
 - statusArray, 371

- sumInfeasibilities, 370
- upper, 370
- validate, 371
- zapCosts, 369
- ClpNonLinearCost.hpp
 - CLP_ABOVE_UPPER, 692
 - CLP_BELOW_LOWER, 692
 - CLP_FEASIBLE, 692
 - CLP_METHOD1, 692
 - CLP_METHOD2, 692
 - CLP_SAME, 692
 - currentStatus, 693
 - originalStatus, 693
 - setCurrentStatus, 693
 - setInitialStatus, 693
 - setOriginalStatus, 693
 - setSameStatus, 693
- ClpObjective, 371
 - ~ClpObjective, 373
 - activated, 375
 - activated_, 375
 - clone, 374
 - ClpObjective, 373
 - ClpObjective, 373
 - deleteSome, 374
 - gradient, 373
 - markNonlinear, 374
 - newXValues, 374
 - nonlinearOffset, 375
 - objectiveValue, 374
 - offset_, 375
 - operator=, 374
 - reallyScale, 374
 - reducedGradient, 373
 - resize, 374
 - setActivated, 375
 - setType, 375
 - stepLength, 373
 - subsetClone, 374
 - type, 374
 - type_, 375
- clpObjectiveValue
 - AbcSimplex, 82
- ClpPackedMatrix, 375
 - ~ClpPackedMatrix, 380
 - add, 384
 - allElementsInRange, 383
 - appendCols, 382
 - appendMatrix, 382
 - appendRows, 382
 - canCombine, 387
 - canDoPartialPricing, 384
 - checkFlags, 389
 - checkGaps, 388
 - clone, 388
 - ClpPackedMatrix, 380
 - ClpPackedMatrix, 380
 - columnCopy_, 389
 - copy, 388
 - correctSequence, 388
 - countBasis, 383
 - createScaledMatrix, 383
 - deleteCols, 382
 - deleteRows, 382
 - dubiousWeights, 384
 - fillBasis, 383
 - flags, 388
 - flags_, 389
 - getElements, 381
 - getIndices, 381
 - getMutableElements, 381
 - getNumCols, 381
 - getNumElements, 380
 - getNumRows, 381
 - getPackedMatrix, 380
 - getVectorLength, 381
 - getVectorLengths, 381
 - getVectorStarts, 381
 - isColOrdered, 380
 - makeSpecialColumnCopy, 387
 - matrix, 387
 - matrix_, 389
 - modifyCoefficient, 382
 - numberActiveColumns, 388
 - numberActiveColumns_, 389
 - operator=, 388
 - partialPricing, 385
 - rangeOfElements, 383
 - reallyScale, 385
 - refresh, 385
 - releasePackedMatrix, 384
 - releaseSpecialColumnCopy, 387
 - replaceVector, 382
 - reverseOrderedCopy, 383
 - rowCopy_, 389
 - scale, 383
 - scaleRowCopy, 383
 - scaledColumnCopy, 383
 - setDimensions, 385
 - setMatrixNull, 387
 - setNumberActiveColumns, 388
 - specialColumnCopy, 388
 - specialRowCopy, 388
 - subsetClone, 388
 - subsetTimes2, 387
 - subsetTransposeTimes, 386
 - times, 385
 - transposeTimes, 385, 386

- transposeTimes2, [387](#)
- transposeTimesByColumn, [386](#)
- transposeTimesByRow, [386](#)
- transposeTimesSubset, [386](#)
- unpack, [384](#)
- unpackPacked, [384](#)
- useEffectiveRhs, [387](#)
- wantsSpecialColumnCopy, [387](#)
- zeros, [387](#)
- ClpPackedMatrix.hpp
 - COIN_RESTRICT, [694](#)
- ClpPackedMatrix2, [389](#)
 - ~ClpPackedMatrix2, [391](#)
 - ClpPackedMatrix2, [390](#), [391](#)
 - ClpPackedMatrix2, [390](#), [391](#)
 - column_, [392](#)
 - count_, [391](#)
 - numberBlocks_, [391](#)
 - numberRows_, [391](#)
 - offset_, [391](#)
 - operator=, [391](#)
 - rowStart_, [391](#)
 - transposeTimes, [391](#)
 - usefulInfo, [391](#)
 - work_, [392](#)
- ClpPackedMatrix3, [392](#)
 - ~ClpPackedMatrix3, [393](#)
 - block_, [394](#)
 - ClpPackedMatrix3, [393](#)
 - ClpPackedMatrix3, [393](#)
 - column_, [394](#)
 - element_, [394](#)
 - numberBlocks_, [394](#)
 - numberColumns_, [394](#)
 - operator=, [394](#)
 - row_, [394](#)
 - sortBlocks, [394](#)
 - start_, [394](#)
 - swapOne, [394](#)
 - transposeTimes, [393](#)
 - transposeTimes2, [393](#)
- ClpParameters.hpp
 - ClpCopyOfArray, [695](#), [696](#)
 - ClpDbiParam, [695](#)
 - ClpDisjointCopyN, [695](#)
 - ClpFillN, [695](#)
 - ClpIntParam, [694](#)
 - ClpStrParam, [695](#)
- ClpPdco, [395](#)
 - getBoundTypes, [396](#)
 - getGrad, [396](#)
 - getHessian, [396](#)
 - getObj, [396](#)
 - lsqr, [396](#)
 - matPrecon, [396](#)
 - matVecMult, [396](#)
 - pdco, [396](#)
- ClpPdcoBase, [396](#)
 - ~ClpPdcoBase, [398](#)
 - clone, [398](#)
 - ClpPdcoBase, [398](#)
 - ClpPdcoBase, [398](#)
 - d1_, [399](#)
 - d2_, [399](#)
 - getD1, [398](#)
 - getD2, [399](#)
 - getGrad, [398](#)
 - getHessian, [398](#)
 - getObj, [398](#)
 - matPrecon, [398](#)
 - matVecMult, [398](#)
 - operator=, [399](#)
 - setType, [398](#)
 - sizeD1, [398](#)
 - sizeD2, [398](#)
 - type, [398](#)
 - type_, [399](#)
- ClpPlusMinusOneMatrix, [399](#)
 - ~ClpPlusMinusOneMatrix, [402](#)
 - add, [406](#)
 - appendCols, [404](#)
 - appendMatrix, [404](#)
 - appendRows, [404](#)
 - canCombine, [407](#)
 - canDoPartialPricing, [408](#)
 - checkValid, [406](#)
 - clone, [408](#)
 - ClpPlusMinusOneMatrix, [402](#), [403](#)
 - ClpPlusMinusOneMatrix, [402](#), [403](#)
 - columnOrdered_, [409](#)
 - countBasis, [405](#)
 - deleteCols, [404](#)
 - deleteRows, [404](#)
 - dubiousWeights, [405](#)
 - fillBasis, [405](#)
 - getElements, [403](#)
 - getIndices, [404](#)
 - getMutableIndices, [404](#)
 - getNumCols, [403](#)
 - getNumElements, [403](#)
 - getNumRows, [403](#)
 - getPackedMatrix, [403](#)
 - getVectorLengths, [404](#)
 - getVectorStarts, [404](#)
 - indices_, [409](#)
 - isColOrdered, [403](#)
 - lengths_, [409](#)
 - matrix_, [408](#)

- numberColumns_, 409
 - numberRows_, 409
 - operator=, 408
 - partialPricing, 408
 - passInCopy, 408
 - rangeOfElements, 405
 - releasePackedMatrix, 406
 - reverseOrderedCopy, 405
 - setDimensions, 406
 - startNegative, 408
 - startNegative_, 409
 - startPositive, 408
 - startPositive_, 409
 - subsetClone, 408
 - subsetTimes2, 407
 - subsetTransposeTimes, 407
 - times, 406
 - transposeTimes, 406, 407
 - transposeTimes2, 407
 - transposeTimesByRow, 407
 - unpack, 405
 - unpackPacked, 405
- ClpPredictorCorrector, 409
 - affineProduct, 412
 - checkGoodMove, 411
 - checkGoodMove2, 411
 - complementarityGap, 411
 - createSolution, 411
 - debugMove, 412
 - findDirectionVector, 411
 - findStepLength, 411
 - setupForSolve, 411
 - solve, 411
 - solveSystem, 411
 - updateSolution, 411
- ClpPresolve, 412
 - ~ClpPresolve, 414
 - ClpPresolve, 414
 - ClpPresolve, 414
 - destroyPresolve, 418
 - doDoubleton, 415
 - doDual, 415
 - doDupcol, 416
 - doDuprow, 416
 - doForcing, 416
 - doGubrow, 417
 - doImpliedFree, 416
 - doIntersection, 417
 - doSingleton, 415
 - doSingletonColumn, 417
 - doTighten, 416
 - doTripletion, 416
 - doTwoxTwo, 417
 - gutsOfPresolvedModel, 418
 - model, 414
 - nonLinearValue, 415
 - originalColumns, 415
 - originalModel, 415
 - originalRows, 415
 - postsolve, 418
 - presolve, 418
 - presolveActions, 417
 - presolveStatus, 418
 - presolvedModel, 414
 - presolvedModelToFile, 414
 - setDoDoubleton, 415
 - setDoDual, 415
 - setDoDupcol, 416
 - setDoDuprow, 416
 - setDoForcing, 416
 - setDoGubrow, 417
 - setDoImpliedFree, 416
 - setDoIntersection, 417
 - setDoSingleton, 415
 - setDoSingletonColumn, 417
 - setDoTighten, 416
 - setDoTripletion, 416
 - setDoTwoxTwo, 417
 - setNonLinearValue, 415
 - setOriginalModel, 415
 - setPresolveActions, 417
 - setSubstitution, 417
 - statistics, 417
- ClpPrimalColumnDantzig, 418
 - ~ClpPrimalColumnDantzig, 419
 - clone, 420
 - ClpPrimalColumnDantzig, 419
 - ClpPrimalColumnDantzig, 419
 - operator=, 420
 - pivotColumn, 419
 - saveWeights, 419
- ClpPrimalColumnPivot, 420
 - ~ClpPrimalColumnPivot, 422
 - clearArrays, 422
 - clone, 423
 - ClpPrimalColumnPivot, 422
 - ClpPrimalColumnPivot, 422
 - looksOptimal, 423
 - looksOptimal_, 424
 - maximumPivotsChanged, 423
 - model, 423
 - model_, 424
 - numberSprintColumns, 423
 - operator=, 423
 - pivotColumn, 422
 - pivotRow, 422
 - saveWeights, 422
 - setLooksOptimal, 423

- setModel, [423](#)
- switchOffSprint, [423](#)
- type, [423](#)
- type_, [424](#)
- updateWeights, [422](#)
- ClpPrimalColumnSteepest, [424](#)
 - ~ClpPrimalColumnSteepest, [427](#)
 - checkAccuracy, [428](#)
 - clearArrays, [428](#)
 - clone, [429](#)
 - ClpPrimalColumnSteepest, [427](#)
 - ClpPrimalColumnSteepest, [427](#)
 - djsAndDevex, [427](#)
 - djsAndDevex2, [427](#)
 - djsAndSteepest, [427](#)
 - djsAndSteepest2, [428](#)
 - initializeWeights, [428](#)
 - justDevex, [428](#)
 - justDjs, [427](#)
 - justSteepest, [428](#)
 - looksOptimal, [428](#)
 - maximumPivotsChanged, [429](#)
 - mode, [429](#)
 - numberSprintColumns, [429](#)
 - operator=, [429](#)
 - partialPricing, [427](#)
 - Persistence, [426](#)
 - persistence, [429](#)
 - pivotColumn, [427](#)
 - pivotColumnOldMethod, [427](#)
 - reference, [429](#)
 - saveWeights, [428](#)
 - setPersistence, [429](#)
 - setReference, [429](#)
 - switchOffSprint, [429](#)
 - transposeTimes2, [428](#)
 - unrollWeights, [428](#)
 - updateWeights, [428](#)
- ClpPrimalQuadraticDantzig, [430](#)
 - ~ClpPrimalQuadraticDantzig, [431](#)
 - clone, [431](#)
 - ClpPrimalQuadraticDantzig, [431](#)
 - ClpPrimalQuadraticDantzig, [431](#)
 - operator=, [431](#)
 - pivotColumn, [431](#)
 - saveWeights, [431](#)
- ClpQuadraticObjective, [431](#)
 - ~ClpQuadraticObjective, [433](#)
 - clone, [435](#)
 - ClpQuadraticObjective, [433](#)
 - ClpQuadraticObjective, [433](#)
 - deleteQuadraticObjective, [435](#)
 - deleteSome, [434](#)
 - fullMatrix, [435](#)
 - gradient, [434](#)
 - linearObjective, [435](#)
 - loadQuadraticObjective, [435](#)
 - markNonlinear, [434](#)
 - numberColumns, [435](#)
 - numberExtendedColumns, [435](#)
 - objectiveValue, [434](#)
 - operator=, [434](#)
 - quadraticObjective, [435](#)
 - reallyScale, [434](#)
 - reducedGradient, [434](#)
 - resize, [434](#)
 - stepLength, [434](#)
 - subsetClone, [435](#)
- clpScaledMatrix
 - ClpModel, [330](#)
- ClpSimplex, [436](#)
 - ~ClpSimplex, [453](#)
 - abcState, [453](#)
 - abcState_, [488](#)
 - acceptablePivot_, [482](#)
 - active, [474](#)
 - add, [465](#)
 - algorithm, [461](#)
 - algorithm_, [486](#)
 - allSlackBasis, [475](#)
 - allowedInfeasibility_, [488](#)
 - alpha, [468](#)
 - alpha_, [481](#)
 - alphaAccuracy, [466](#)
 - alphaAccuracy_, [481](#)
 - automaticScale_, [488](#)
 - automaticScaling, [467](#)
 - barrier, [456](#)
 - baseliteration, [476](#)
 - baseliteration_, [480](#)
 - baseModel, [453](#)
 - baseModel_, [488](#)
 - bestObjectiveValue_, [480](#)
 - bestPossibleImprovement, [473](#)
 - bestPossibleImprovement_, [479](#)
 - borrowModel, [454](#), [455](#)
 - changeMade_, [486](#)
 - checkBothSolutions, [465](#)
 - checkDualSolution, [465](#)
 - checkPrimalSolution, [465](#)
 - checkSolution, [463](#)
 - checkSolutionInternal, [463](#)
 - checkUnscaledSolution, [463](#)
 - cleanFactorization, [466](#)
 - cleanStatus, [464](#)
 - cleanup, [456](#)
 - clearActive, [474](#)
 - clearFlagged, [474](#)

clearPivoted, 474
ClpSimplex, 452, 453
ClpSimplexUnitTest, 479
ClpSimplex, 452, 453
columnActivityWork_, 485
columnArray, 463
columnArray_, 484
columnLowerWork_, 483
columnPrimalSequence_, 480
columnUpperWork_, 483
computeDuals, 464
computeInternalObjectiveValue, 475
computeObjectiveValue, 475
computePrimals, 464
copyEnabledStuff, 460
copyFactorization, 458
cost, 472
cost_, 483
costAddress, 472
costRegion, 469, 470
crash, 458
createPiecewiseLinearCosts, 463
createRim, 468
createRim1, 469
createRim4, 469
createRim5, 469
createStatus, 474
currentDualTolerance, 467
currentPrimalTolerance, 467
defaultFactorizationFrequency, 460
deleteBaseModel, 453
deleteRim, 469
directionIn, 470
directionIn_, 484
directionOut, 471
directionOut_, 484
disasterArea_, 486
disasterHandler, 466
dj_, 484
djRegion, 469, 470
dontFactorizePivots_, 487
doubleCheck, 459
dual, 455
dualBound, 461
dualBound_, 481
dualDebug, 455
dualFeasible, 460
dualIn, 468
dualIn_, 481
dualOut, 471
dualOut_, 482
dualPivotResultPart1, 460
dualRanging, 456
dualRowPivot, 464
dualRowPivot_, 485
dualTolerance_, 482
factorization, 461
factorization_, 486
factorizationFrequency, 461
factorize, 464
FakeBound, 452
fastCrunch, 459
fastDual2, 459
fathom, 459
fathomMany, 459
finish, 460
firstFree_, 487
flagged, 474
forceFactorization, 475
forceFactorization_, 486
generateCpp, 476
getBInvACol, 476
getBInvARow, 476
getBInvCol, 476
getBInvRow, 476
getBasics, 476
getBasis, 458
getColumnStatus, 474
getEmptyFactorization, 476
getFakeBound, 473
getRowStatus, 474
getSolution, 463
getStatus, 470
getbackSolution, 455
goodAccuracy, 464
gutsOfCopy, 468
gutsOfDelete, 468
gutsOfSolution, 468
housekeeping, 465
incomingInfeasibility_, 487
infeasibilityCost, 461
infeasibilityCost_, 482
infeasibilityRay, 475
initialBarrierNoCrossSolve, 455
initialBarrierSolve, 455
initialDenseFactorization, 470
initialDualSolve, 455
initialPrimalSolve, 455
initialSolve, 455
internalFactorize, 464
isColumn, 471
isObjectiveLimitTestValid, 461
largeValue, 466
largeValue_, 480
largestDualError, 466
largestDualError_, 480
largestPrimalError, 466
largestPrimalError_, 480

lastBadIteration, 475
lastBadIteration_, 487
lastFlaggedIteration_, 487
lastGoodIteration_, 484
loadNonLinear, 455
loadProblem, 454, 456
lower, 472
lower_, 483
lowerAddress, 472
lowerIn_, 481
lowerOut_, 481
lowerRegion, 469, 470
makeBaseModel, 453
maximumBasic, 476
maximumBasic_, 487
maximumPerturbationSize_, 488
miniPostsolve, 458
miniPresolve, 457
modifyCoefficientsAndPivot, 457
moreSpecialOptions, 473
moreSpecialOptions_, 480
moveInfo, 476
moveTowardsPrimalFeasible, 457
nonLinearCost, 473
nonLinearCost_, 486
nonlinearSLP, 456
numberChanged_, 487
numberDualInfeasibilities, 462
numberDualInfeasibilities_, 485
numberDualInfeasibilitiesWithoutFree, 462
numberDualInfeasibilitiesWithoutFree_, 485
numberExtraRows, 476
numberExtraRows_, 487
numberFake_, 487
numberPrimalInfeasibilities, 462
numberPrimalInfeasibilities_, 485
numberRefinements, 467
numberRefinements_, 485
numberTimesOptimal_, 486
objectiveWork_, 483
operator=, 453
originalLower, 472
originalModel, 453
originalUpper, 473
OsiClpSolverInterface, 479
outDuplicateRows, 457
passInEventHandler, 455
perturbation, 461
perturbation_, 486
perturbationArray_, 488
pivot, 459
pivotResultPart2, 460
pivotRow, 468
pivotRow_, 484
pivotVariable, 467
pivotVariable_, 485
pivoted, 474
primal, 455
primalColumnPivot, 464
primalColumnPivot_, 485
primalFeasible, 460
primalPivotResult, 460
primalRanging, 457
primalTolerance_, 482
primalToleranceToGetOptimal_, 480
progress, 475
progress_, 488
progressFlag, 475
progressFlag_, 487
rawObjectiveValue, 475
readBasis, 458
readGMPL, 454
readLp, 454
readMps, 454
reducedCost, 472
reducedCostAddress, 472
reducedCostWork_, 484
reducedGradient, 456
removeSuperBasicSlacks, 457
resize, 479
restoreData, 464
restoreModel, 463
returnModel, 464
rowActivityWork_, 485
rowArray, 463
rowArray_, 483
rowLowerWork_, 483
rowObjectiveWork_, 483
rowPrimalSequence_, 480
rowReducedCost_, 484
rowUpperWork_, 483
sanityCheck, 469
saveData, 464
saveModel, 463
saveStatus_, 486
savedSolution_, 486
scaleObjective, 465
scaleRealObjective, 458
scaleRealRhs, 458
sequenceIn, 470
sequenceIn_, 484
sequenceOut, 470
sequenceOut_, 484
sequenceWithin, 472
setAbcState, 453
setActive, 474
setAlgorithm, 461
setAlpha, 468

- setAlphaAccuracy, [466](#)
- setAutomaticScaling, [467](#)
- setColBounds, [477](#)
- setColLower, [477](#)
- setColSetBounds, [477](#)
- setColUpper, [477](#)
- setColumnBounds, [477](#)
- setColumnLower, [477](#)
- setColumnSetBounds, [477](#)
- setColumnStatus, [474](#)
- setColumnUpper, [477](#)
- setCurrentDualTolerance, [467](#)
- setCurrentPrimalTolerance, [467](#)
- setDirectionIn, [471](#)
- setDirectionOut, [471](#)
- setDisasterHandler, [466](#)
- setDualBound, [461](#)
- setDualIn, [468](#)
- setDualOut, [471](#)
- setDualRowPivotAlgorithm, [459](#)
- setEmptyFactorization, [476](#)
- setFactorization, [458](#)
- setFactorizationFrequency, [461](#)
- setFakeBound, [473](#)
- setFlagged, [474](#)
- setInfeasibilityCost, [461](#)
- setInitialDenseFactorization, [470](#)
- setLargeValue, [466](#)
- setLargestDualError, [467](#)
- setLargestPrimalError, [466](#)
- setLastBadIteration, [475](#)
- setLowerOut, [471](#)
- setMoreSpecialOptions, [473](#)
- setNumberDualInfeasibilities, [462](#)
- setNumberPrimalInfeasibilities, [463](#)
- setNumberRefinements, [468](#)
- setObjCoeff, [477](#)
- setObjectiveCoefficient, [476](#)
- setPersistenceFlag, [453](#)
- setPerturbation, [461](#)
- setPivotRow, [468](#)
- setPivoted, [474](#)
- setPrimalColumnPivotAlgorithm, [459](#)
- setRowBounds, [479](#)
- setRowLower, [479](#)
- setRowSetBounds, [479](#)
- setRowStatus, [474](#)
- setRowUpper, [479](#)
- setSequenceIn, [470](#)
- setSequenceOut, [470](#)
- setSparseFactorization, [461](#)
- setStatus, [470](#)
- setSumDualInfeasibilities, [462](#)
- setSumOfRelaxedDualInfeasibilities, [462](#)
- setSumOfRelaxedPrimalInfeasibilities, [462](#)
- setSumPrimalInfeasibilities, [462](#)
- setTheta, [471](#)
- setToBaseModel, [453](#)
- setUpperOut, [471](#)
- setValueOut, [471](#)
- setValuesPassAction, [465](#)
- setZeroTolerance, [467](#)
- solution, [472](#)
- solution_, [485](#)
- solutionAddress, [472](#)
- solutionRegion, [469](#)
- solve, [456](#)
- solveBenders, [465](#)
- solveDW, [465](#)
- spareDoubleArray_, [488](#)
- spareIntArray_, [488](#)
- sparseFactorization, [461](#)
- startFastDual2, [459](#)
- startPermanentArrays, [470](#)
- startup, [460](#)
- Status, [452](#)
- statusOfProblem, [460](#)
- stopFastDual2, [459](#)
- strongBranching, [459](#)
- sumDualInfeasibilities, [462](#)
- sumDualInfeasibilities_, [482](#)
- sumOfRelaxedDualInfeasibilities, [462](#)
- sumOfRelaxedDualInfeasibilities_, [482](#)
- sumOfRelaxedPrimalInfeasibilities, [462](#)
- sumOfRelaxedPrimalInfeasibilities_, [482](#)
- sumPrimalInfeasibilities, [462](#)
- sumPrimalInfeasibilities_, [482](#)
- swapFactorization, [458](#)
- theta, [473](#)
- theta_, [481](#)
- tightenPrimalBounds, [458](#)
- unpack, [465](#)
- unpackPacked, [465](#)
- upper, [472](#)
- upper_, [483](#)
- upperAddress, [472](#)
- upperIn_, [481](#)
- upperOut_, [482](#)
- upperRegion, [469](#), [470](#)
- valueIn_, [481](#)
- valueIncomingDual, [468](#)
- valueOut, [471](#)
- valueOut_, [481](#)
- writeBasis, [458](#)
- zeroTolerance, [467](#)
- zeroTolerance_, [480](#)
- ClpSimplex.hpp
- ClpSimplexUnitTest, [699](#)

- DEVEX_ADD_ONE, 699
- DEVEX_TRY_NORM, 699
- ClpSimplexDual, 489
 - changeBound, 493
 - changeBounds, 493
 - checkPossibleCleanup, 493
 - checkPossibleValuesMove, 493
 - checkUnbounded, 493
 - cleanupAfterStrongBranching, 492
 - doEasyOnesInValuesPass, 493
 - dual, 490
 - dualColumn, 492
 - dualColumn0, 492
 - dualRow, 493
 - fastDual, 494
 - finishSolve, 494
 - flipBounds, 492
 - gutsOfDual, 494
 - nextSuperBasic, 494
 - numberAtFakeBound, 494
 - originalBound, 493
 - perturb, 494
 - pivotResultPart1, 494
 - resetFakeBounds, 494
 - setupForStrongBranching, 492
 - startupSolve, 494
 - statusOfProblemInDual, 493
 - strongBranching, 492
 - updateDualsInDual, 492
 - updateDualsInValuesPass, 492
 - whileIterating, 492
- ClpSimplexNonlinear, 494
 - directionVector, 496
 - pivotColumn, 496
 - pivotNonlinearResult, 496
 - primal, 495
 - primalSLP, 496
 - statusOfProblemInPrimal, 496
 - whileIterating, 496
- ClpSimplexOther, 497
 - afterCrunch, 500
 - bestPivot, 499
 - cleanupAfterPostsolve, 500
 - crunch, 500
 - dualOfModel, 500
 - dualRanging, 498
 - expandKnapsack, 500
 - getGubBasis, 500
 - gubVersion, 500
 - metrics, 499
 - metricsObj, 499
 - primalRanging, 498
 - readBasis, 500
 - restoreFromDual, 500
 - setGubBasis, 500
 - tightenIntegerBounds, 500
 - writeBasis, 499
- ClpSimplexOther::metricsData, 634
 - acceptableMaxTheta, 635
 - backwardBasic, 635
 - endingTheta, 635
 - firstIteration, 636
 - lowerActive, 635
 - lowerChange, 635
 - lowerCoefficient, 636
 - lowerGap, 635
 - lowerList, 635
 - markDone, 635
 - maxTheta, 635
 - startingTheta, 635
 - unscaledChangesOffset, 636
 - upperActive, 636
 - upperChange, 635
 - upperCoefficient, 636
 - upperGap, 636
 - upperList, 635
- ClpSimplexPrimal, 501
 - alwaysOptimal, 503
 - checkUnbounded, 504
 - clearAll, 505
 - exactOutgoing, 503, 504
 - lexSolve, 505
 - nextSuperBasic, 505
 - perturb, 505
 - pivotResult, 504
 - primal, 502
 - primalColumn, 504
 - primalRay, 505
 - primalRow, 504
 - statusOfProblemInPrimal, 504
 - unPerturb, 505
 - unflag, 505
 - updatePrimalsInPrimal, 504
 - whileIterating, 504
- ClpSimplexProgress, 505
 - ~ClpSimplexProgress, 507
 - badTimes, 509
 - clearBadTimes, 509
 - clearIterationNumbers, 508
 - clearOddState, 508
 - clearTimesFlagged, 509
 - ClpSimplexProgress, 507
 - ClpSimplexProgress, 507
 - cycle, 508
 - endOddState, 508
 - fillFromModel, 508
 - in_, 510
 - incrementReallyBadTimes, 509

- incrementTimesFlagged, 509
- infeasibility_, 509
- initialWeight_, 509
- iterationNumber_, 510
- lastInfeasibility, 508
- lastIterationNumber, 508
- lastObjective, 508
- looping, 508
- model_, 510
- modifyObjective, 508
- newOddState, 508
- numberBadTimes_, 510
- numberInfeasibilities_, 510
- numberReallyBadTimes_, 510
- numberTimes_, 510
- numberTimesFlagged_, 510
- objective_, 509
- oddState, 508
- oddState_, 510
- operator=, 507
- out_, 510
- realInfeasibility_, 509
- reallyBadTimes, 509
- reset, 507
- setInfeasibility, 508
- startCheck, 508
- timesFlagged, 509
- way_, 510
- ClpSimplexUnitTest
 - ClpSimplex, 479
 - ClpSimplex.hpp, 699
- ClpSolve, 511
 - ~ClpSolve, 513
 - ClpSolve, 513
 - ClpSolve, 513
 - doDoubleton, 515
 - doDual, 514
 - doDupcol, 515
 - doDuprow, 516
 - doForcing, 515
 - doImpliedFree, 515
 - doKillSmall, 516
 - doSingleton, 514
 - doSingletonColumn, 516
 - doTighten, 515
 - doTripletion, 515
 - generateCpp, 513
 - getExtraInfo, 514
 - getPresolvePasses, 514
 - getPresolveType, 514
 - getSolveType, 514
 - getSpecialOption, 514
 - infeasibleReturn, 514
 - operator=, 513
 - presolveActions, 516
 - PresolveType, 513
 - setDoDoubleton, 515
 - setDoDual, 514
 - setDoDupcol, 516
 - setDoDuprow, 516
 - setDoForcing, 515
 - setDoImpliedFree, 515
 - setDoKillSmall, 516
 - setDoSingleton, 515
 - setDoSingletonColumn, 516
 - setDoTighten, 515
 - setDoTripletion, 515
 - setInfeasibleReturn, 514
 - setPresolveActions, 516
 - setPresolveType, 514
 - setSolveType, 514
 - setSpecialOption, 514
 - setSubstitution, 516
 - SolveType, 513
 - substitution, 516
- ClpSolve.hpp
 - CLP_CYCLE, 701
 - CLP_PROGRESS, 701
- ClpSolve_delete
 - Clp_C_Interface.h, 664
- ClpSolve_doDoubleton
 - Clp_C_Interface.h, 674
- ClpSolve_doDual
 - Clp_C_Interface.h, 673
- ClpSolve_doDupcol
 - Clp_C_Interface.h, 674
- ClpSolve_doDuprow
 - Clp_C_Interface.h, 674
- ClpSolve_doForcing
 - Clp_C_Interface.h, 674
- ClpSolve_doImpliedFree
 - Clp_C_Interface.h, 674
- ClpSolve_doSingleton
 - Clp_C_Interface.h, 674
- ClpSolve_doSingletonColumn
 - Clp_C_Interface.h, 674
- ClpSolve_doTighten
 - Clp_C_Interface.h, 674
- ClpSolve_doTripletion
 - Clp_C_Interface.h, 674
- ClpSolve_getExtraInfo
 - Clp_C_Interface.h, 673
- ClpSolve_getPresolvePasses
 - Clp_C_Interface.h, 673
- ClpSolve_getPresolveType
 - Clp_C_Interface.h, 673
- ClpSolve_getSolveType
 - Clp_C_Interface.h, 673

- ClpSolve_getSpecialOption
 - Clp_C_Interface.h, [673](#)
- ClpSolve_infeasibleReturn
 - Clp_C_Interface.h, [673](#)
- ClpSolve_new
 - Clp_C_Interface.h, [664](#)
- ClpSolve_presolveActions
 - Clp_C_Interface.h, [674](#)
- ClpSolve_setDoDoubleton
 - Clp_C_Interface.h, [674](#)
- ClpSolve_setDoDual
 - Clp_C_Interface.h, [674](#)
- ClpSolve_setDoDupcol
 - Clp_C_Interface.h, [674](#)
- ClpSolve_setDoDuprow
 - Clp_C_Interface.h, [674](#)
- ClpSolve_setDoForcing
 - Clp_C_Interface.h, [674](#)
- ClpSolve_setDoImpliedFree
 - Clp_C_Interface.h, [674](#)
- ClpSolve_setDoSingleton
 - Clp_C_Interface.h, [674](#)
- ClpSolve_setDoSingletonColumn
 - Clp_C_Interface.h, [674](#)
- ClpSolve_setDoTighten
 - Clp_C_Interface.h, [674](#)
- ClpSolve_setDoTripleton
 - Clp_C_Interface.h, [674](#)
- ClpSolve_setInfeasibleReturn
 - Clp_C_Interface.h, [673](#)
- ClpSolve_setPresolveActions
 - Clp_C_Interface.h, [674](#)
- ClpSolve_setPresolveType
 - Clp_C_Interface.h, [673](#)
- ClpSolve_setSolveType
 - Clp_C_Interface.h, [673](#)
- ClpSolve_setSpecialOption
 - Clp_C_Interface.h, [673](#)
- ClpSolve_setSubstitution
 - Clp_C_Interface.h, [674](#)
- ClpSolve_substitution
 - Clp_C_Interface.h, [674](#)
- ClpStrParam
 - ClpParameters.hpp, [695](#)
- ClpTraceDebug
 - ClpHelperFunctions.hpp, [682](#)
- ClpTracePrint
 - ClpHelperFunctions.hpp, [682](#)
- ClpTrustedData, [517](#)
 - data, [517](#)
 - typeCall, [517](#)
 - typeStruct, [517](#)
- coefficient
 - ClpConstraintLinear, [177](#)
 - ClpConstraintQuadratic, [180](#)
- coin_prefetch
 - CoinAbcHelperFunctions.hpp, [714](#)
- coin_prefetch_const
 - CoinAbcHelperFunctions.hpp, [714](#)
- CoinAbcAnyFactorization, [517](#)
 - ~CoinAbcAnyFactorization, [521](#)
 - areaFactor, [523](#)
 - areaFactor_, [528](#)
 - checkMarkArrays, [524](#)
 - checkReplacePart1, [526](#)
 - checkReplacePart1a, [526](#)
 - checkReplacePart1b, [526](#)
 - checkReplacePart2, [526](#)
 - clearArrays, [525](#)
 - clone, [522](#)
 - CoinAbcAnyFactorization, [521](#)
 - CoinAbcAnyFactorization, [521](#)
 - elements, [524](#)
 - elements_, [530](#)
 - factor, [526](#)
 - factorElements_, [529](#)
 - getAccuracyCheck, [523](#)
 - getAreas, [525](#)
 - goSparse, [524](#)
 - indices, [525](#)
 - intWorkArea, [524](#)
 - makeNonSingular, [526](#)
 - maximumPivots, [523](#)
 - maximumPivots_, [529](#)
 - maximumRows_, [529](#)
 - minimumPivotTolerance, [523](#)
 - minimumPivotTolerance_, [528](#)
 - numberDense, [522](#)
 - numberDense_, [529](#)
 - numberElements, [525](#)
 - numberGoodColumns, [523](#)
 - numberGoodU_, [529](#)
 - numberInColumn, [524](#)
 - numberInRow, [524](#)
 - numberPivots_, [529](#)
 - numberRows, [522](#)
 - numberRows_, [529](#)
 - numberSlacks, [522](#)
 - numberSlacks_, [529](#)
 - operator=, [522](#)
 - permute, [525](#)
 - permuteBack, [524](#)
 - pivotColumn, [525](#)
 - pivotRegion, [523](#)
 - pivotRow, [524](#)
 - pivotRow_, [529](#)
 - pivotTolerance, [523](#)
 - pivotTolerance_, [528](#)

- pivots, [522](#)
- postProcess, [526](#)
- preProcess, [526](#)
- relaxAccuracyCheck, [523](#)
- relaxCheck_, [528](#)
- replaceColumnPart3, [527](#)
- setNumberRows, [522](#)
- setNumberSlacks, [522](#)
- setPivots, [522](#)
- setSolveMode, [525](#)
- setStatus, [522](#)
- setUsefulInformation, [525](#)
- solveMode, [525](#)
- solveMode_, [530](#)
- starts, [524](#)
- status, [522](#)
- status_, [529](#)
- updateColumn, [527](#)
- updateColumnCpu, [528](#)
- updateColumnFT, [527](#)
- updateColumnFTPart1, [527](#)
- updateColumnFTPart2, [527](#)
- updateColumnTranspose, [527](#)
- updateColumnTransposeCpu, [528](#)
- updateFullColumn, [527](#)
- updateFullColumnTranspose, [528](#)
- updateTwoColumnsFT, [527](#)
- updateWeights, [528](#)
- wantsTableauColumn, [525](#)
- workArea, [524](#)
- workArea_, [530](#)
- zeroTolerance, [524](#)
- zeroTolerance_, [528](#)
- CoinAbcBaseFactorization.hpp
 - CONVERTROW, [701](#)
 - checkLinks, [701](#)
 - FACTOR_CPU, [701](#)
 - LARGE_SET, [701](#)
 - LARGE_UNSET, [701](#)
- CoinAbcCommon.hpp
 - ABC_DENSE_CODE, [705](#)
 - ABC_EXPONENT, [705](#)
 - ABC_INLINE, [703](#)
 - ABC_INSTRUMENT, [703](#)
 - ABC_INTEL, [704](#)
 - ABC_PARALLEL, [703](#)
 - COIN_FAC_NEW, [703](#)
 - cilk_for, [703](#)
 - cilk_spawn, [703](#)
 - cilk_sync, [703](#)
 - CoinAbcMemcpy, [706](#)
 - CoinAbcMemset0, [706](#)
 - CoinCheckZero, [706](#)
 - CoinExponent, [705](#)
 - CoinFabs, [704](#)
 - CoinSimplexDouble, [705](#)
 - CoinSimplexInt, [705](#)
 - CoinSimplexUnsignedInt, [705](#)
 - instrument_add, [704](#)
 - instrument_do, [704](#)
 - instrument_end, [704](#)
 - instrument_end_and_adjust, [704](#)
 - instrument_start, [704](#)
 - SLACK_VALUE, [703](#)
 - TEST_INT_NONZERO, [704](#)
- CoinAbcCommonFactorization.hpp
 - BLOCKING8, [708](#)
 - BLOCKING8X8, [708](#)
 - CoinAbcDgetrf, [708](#)
 - CoinAbcDgetrs, [708](#)
 - factorizationStatistics, [707](#)
 - INITIAL_AVERAGE, [707](#)
 - INITIAL_AVERAGE2, [707](#)
 - SWAP_FACTOR, [708](#)
 - setStatistics, [707](#)
 - twiddleBtranFactor1, [707](#)
 - twiddleBtranFactor2, [708](#)
 - twiddleBtranFullFactor1, [708](#)
 - twiddleFactor1S, [707](#)
 - twiddleFactor2S, [707](#)
 - twiddleFtranFTFactor1, [707](#)
 - twiddleFtranFTFactor2, [708](#)
 - twiddleFtranFactor1, [707](#)
 - twiddleFtranFactor2, [707](#)
- CoinAbcCompact
 - CoinAbcHelperFunctions.hpp, [719](#)
- CoinAbcDenseFactorization, [530](#)
 - ~CoinAbcDenseFactorization, [533](#)
 - checkPivot, [536](#)
 - checkReplacePart2, [534](#)
 - clearArrays, [535](#)
 - clone, [533](#)
 - CoinAbcDenseFactorization, [533](#)
 - CoinAbcDenseFactorizationUnitTest, [536](#)
 - CoinAbcDenseFactorization, [533](#)
 - factor, [533](#)
 - getAreas, [533](#)
 - gutsOfCopy, [536](#)
 - gutsOfDestructor, [536](#)
 - gutsOfInitialize, [536](#)
 - indices, [536](#)
 - makeNonSingular, [533](#)
 - maximumCoefficient, [534](#)
 - maximumRowsAdjusted_, [536](#)
 - maximumSpace_, [536](#)
 - numberElements, [533](#)
 - operator=, [533](#)
 - permute, [536](#)

- postProcess, [533](#)
- preProcess, [533](#)
- replaceColumn, [534](#)
- replaceColumnPart3, [534](#)
- updateColumn, [535](#)
- updateColumnFT, [534](#), [535](#)
- updateColumnFTPart1, [534](#)
- updateColumnFTPart2, [534](#)
- updateColumnTranspose, [535](#)
- updateFullColumn, [535](#)
- updateFullColumnTranspose, [535](#)
- updateTwoColumnsFT, [535](#)
- updateWeights, [535](#)
- CoinAbcDenseFactorization.hpp
 - slackValue2_, [709](#)
- CoinAbcDenseFactorizationUnitTest
 - CoinAbcDenseFactorization, [536](#)
- CoinAbcDgetrf
 - CoinAbcCommonFactorization.hpp, [708](#)
- CoinAbcDgetrs
 - CoinAbcCommonFactorization.hpp, [708](#)
- CoinAbcFactorization.hpp
 - ABC_SMALL, [710](#)
 - COIN_BIG_DOUBLE, [710](#)
 - CoinAbcTypeFactorization, [710](#)
- CoinAbcFactorizationUnitTest
 - CoinAbcTypeFactorization, [566](#)
- CoinAbcGatherFrom
 - CoinAbcHelperFunctions.hpp, [718](#)
- CoinAbcGatherUpdate
 - CoinAbcHelperFunctions.hpp, [717](#)
- CoinAbcGetNorms
 - CoinAbcHelperFunctions.hpp, [718](#)
- CoinAbcHelperFunctions.hpp
 - abc_assert, [714](#)
 - coin_prefetch, [714](#)
 - coin_prefetch_const, [714](#)
 - CoinAbcCompact, [719](#)
 - CoinAbcGatherFrom, [718](#)
 - CoinAbcGatherUpdate, [717](#)
 - CoinAbcGetNorms, [718](#)
 - CoinAbcInnerProduct, [718](#)
 - CoinAbcInverseSqrts, [718](#)
 - CoinAbcMaximumAbsElement, [718](#)
 - CoinAbcMaximumAbsElementAndScale, [718](#)
 - CoinAbcMemcpyLong, [718](#)
 - CoinAbcMemmove, [719](#)
 - CoinAbcMemmoveAndZero, [719](#)
 - CoinAbcMemset0Long, [718](#), [719](#)
 - CoinAbcMinMaxAbsElement, [718](#)
 - CoinAbcMinMaxAbsNormalValues, [718](#)
 - CoinAbcMultiplyAdd, [718](#)
 - CoinAbcMultiplyIndexed, [717](#)
 - CoinAbcReciprocal, [718](#)
 - CoinAbcScale, [718](#)
 - CoinAbcScaleNormalValues, [718](#)
 - CoinAbcScatterTo, [718](#)
 - CoinAbcScatterToList, [718](#)
 - CoinAbcScatterUpdate, [715](#)
 - CoinAbcScatterUpdate0, [715](#)
 - CoinAbcScatterUpdate1, [715](#)
 - CoinAbcScatterUpdate1Add, [717](#)
 - CoinAbcScatterUpdate1Subtract, [716](#)
 - CoinAbcScatterUpdate2, [715](#)
 - CoinAbcScatterUpdate2Add, [717](#)
 - CoinAbcScatterUpdate2Subtract, [716](#)
 - CoinAbcScatterUpdate3, [716](#)
 - CoinAbcScatterUpdate3Add, [717](#)
 - CoinAbcScatterUpdate3Subtract, [716](#)
 - CoinAbcScatterUpdate4, [716](#)
 - CoinAbcScatterUpdate4Add, [717](#)
 - CoinAbcScatterUpdate4N, [716](#)
 - CoinAbcScatterUpdate4NAdd, [717](#)
 - CoinAbcScatterUpdate4NPlus1, [716](#)
 - CoinAbcScatterUpdate4NPlus1Add, [717](#)
 - CoinAbcScatterUpdate4NPlus1Subtract, [717](#)
 - CoinAbcScatterUpdate4NPlus2, [716](#)
 - CoinAbcScatterUpdate4NPlus2Add, [717](#)
 - CoinAbcScatterUpdate4NPlus2Subtract, [717](#)
 - CoinAbcScatterUpdate4NPlus3, [716](#)
 - CoinAbcScatterUpdate4NPlus3Add, [717](#)
 - CoinAbcScatterUpdate4NPlus3Subtract, [717](#)
 - CoinAbcScatterUpdate4NSubtract, [716](#)
 - CoinAbcScatterUpdate4Subtract, [716](#)
 - CoinAbcScatterUpdate5, [716](#)
 - CoinAbcScatterUpdate5Add, [717](#)
 - CoinAbcScatterUpdate5Subtract, [716](#)
 - CoinAbcScatterUpdate6, [716](#)
 - CoinAbcScatterUpdate6Add, [717](#)
 - CoinAbcScatterUpdate6Subtract, [716](#)
 - CoinAbcScatterUpdate7, [716](#)
 - CoinAbcScatterUpdate7Add, [717](#)
 - CoinAbcScatterUpdate7Subtract, [716](#)
 - CoinAbcScatterUpdate8, [716](#)
 - CoinAbcScatterUpdate8Add, [717](#)
 - CoinAbcScatterUpdate8Subtract, [716](#)
 - CoinAbcScatterZeroTo, [718](#)
 - CoinAbcSetElements, [718](#)
 - INLINE_GATHER, [715](#)
 - INLINE_SCATTER, [714](#)
 - NEW_CHUNK_SIZE, [714](#)
 - scatterUpdate, [715](#)
 - UNROLL_GATHER, [715](#)
 - UNROLL_SCATTER, [714](#)
- CoinAbcInnerProduct
 - CoinAbcHelperFunctions.hpp, [718](#)
- CoinAbcInverseSqrts
 - CoinAbcHelperFunctions.hpp, [718](#)

- CoinAbcMaximumAbsElement
 - CoinAbcHelperFunctions.hpp, 718
- CoinAbcMaximumAbsElementAndScale
 - CoinAbcHelperFunctions.hpp, 718
- CoinAbcMemcpy
 - CoinAbcCommon.hpp, 706
- CoinAbcMemcpyLong
 - CoinAbcHelperFunctions.hpp, 718
- CoinAbcMemmove
 - CoinAbcHelperFunctions.hpp, 719
- CoinAbcMemmoveAndZero
 - CoinAbcHelperFunctions.hpp, 719
- CoinAbcMemset0
 - CoinAbcCommon.hpp, 706
- CoinAbcMemset0Long
 - CoinAbcHelperFunctions.hpp, 718, 719
- CoinAbcMinMaxAbsElement
 - CoinAbcHelperFunctions.hpp, 718
- CoinAbcMinMaxAbsNormalValues
 - CoinAbcHelperFunctions.hpp, 718
- CoinAbcMultiplyAdd
 - CoinAbcHelperFunctions.hpp, 718
- CoinAbcMultiplyIndexed
 - CoinAbcHelperFunctions.hpp, 717
- CoinAbcReciprocal
 - CoinAbcHelperFunctions.hpp, 718
- CoinAbcScale
 - CoinAbcHelperFunctions.hpp, 718
- CoinAbcScaleNormalValues
 - CoinAbcHelperFunctions.hpp, 718
- CoinAbcScatterTo
 - CoinAbcHelperFunctions.hpp, 718
- CoinAbcScatterToList
 - CoinAbcHelperFunctions.hpp, 718
- CoinAbcScatterUpdate
 - CoinAbcHelperFunctions.hpp, 715
- CoinAbcScatterUpdate0
 - CoinAbcHelperFunctions.hpp, 715
- CoinAbcScatterUpdate1
 - CoinAbcHelperFunctions.hpp, 715
- CoinAbcScatterUpdate1Add
 - CoinAbcHelperFunctions.hpp, 717
- CoinAbcScatterUpdate1Subtract
 - CoinAbcHelperFunctions.hpp, 716
- CoinAbcScatterUpdate2
 - CoinAbcHelperFunctions.hpp, 715
- CoinAbcScatterUpdate2Add
 - CoinAbcHelperFunctions.hpp, 717
- CoinAbcScatterUpdate2Subtract
 - CoinAbcHelperFunctions.hpp, 716
- CoinAbcScatterUpdate3
 - CoinAbcHelperFunctions.hpp, 716
- CoinAbcScatterUpdate3Add
 - CoinAbcHelperFunctions.hpp, 717
- CoinAbcScatterUpdate3Subtract
 - CoinAbcHelperFunctions.hpp, 716
- CoinAbcScatterUpdate4
 - CoinAbcHelperFunctions.hpp, 716
- CoinAbcScatterUpdate4Add
 - CoinAbcHelperFunctions.hpp, 717
- CoinAbcScatterUpdate4N
 - CoinAbcHelperFunctions.hpp, 716
- CoinAbcScatterUpdate4NAdd
 - CoinAbcHelperFunctions.hpp, 717
- CoinAbcScatterUpdate4NPlus1
 - CoinAbcHelperFunctions.hpp, 716
- CoinAbcScatterUpdate4NPlus1Add
 - CoinAbcHelperFunctions.hpp, 717
- CoinAbcScatterUpdate4NPlus1Subtract
 - CoinAbcHelperFunctions.hpp, 717
- CoinAbcScatterUpdate4NPlus2
 - CoinAbcHelperFunctions.hpp, 716
- CoinAbcScatterUpdate4NPlus2Add
 - CoinAbcHelperFunctions.hpp, 717
- CoinAbcScatterUpdate4NPlus2Subtract
 - CoinAbcHelperFunctions.hpp, 717
- CoinAbcScatterUpdate4NPlus3
 - CoinAbcHelperFunctions.hpp, 716
- CoinAbcScatterUpdate4NPlus3Add
 - CoinAbcHelperFunctions.hpp, 717
- CoinAbcScatterUpdate4NPlus3Subtract
 - CoinAbcHelperFunctions.hpp, 717
- CoinAbcScatterUpdate4NSubtract
 - CoinAbcHelperFunctions.hpp, 716
- CoinAbcScatterUpdate4Subtract
 - CoinAbcHelperFunctions.hpp, 716
- CoinAbcScatterUpdate5
 - CoinAbcHelperFunctions.hpp, 716
- CoinAbcScatterUpdate5Add
 - CoinAbcHelperFunctions.hpp, 717
- CoinAbcScatterUpdate5Subtract
 - CoinAbcHelperFunctions.hpp, 716
- CoinAbcScatterUpdate6
 - CoinAbcHelperFunctions.hpp, 716
- CoinAbcScatterUpdate6Add
 - CoinAbcHelperFunctions.hpp, 717
- CoinAbcScatterUpdate6Subtract
 - CoinAbcHelperFunctions.hpp, 716
- CoinAbcScatterUpdate7
 - CoinAbcHelperFunctions.hpp, 716
- CoinAbcScatterUpdate7Add
 - CoinAbcHelperFunctions.hpp, 717
- CoinAbcScatterUpdate7Subtract
 - CoinAbcHelperFunctions.hpp, 716
- CoinAbcScatterUpdate8
 - CoinAbcHelperFunctions.hpp, 716
- CoinAbcScatterUpdate8Add
 - CoinAbcHelperFunctions.hpp, 717

- CoinAbcScatterUpdate8Subtract
 - CoinAbcHelperFunctions.hpp, 716
- CoinAbcScatterZeroTo
 - CoinAbcHelperFunctions.hpp, 718
- CoinAbcSetElements
 - CoinAbcHelperFunctions.hpp, 718
- CoinAbcStack, 537
 - next, 537
 - stack, 537
 - start, 537
- CoinAbcStatistics, 537
 - averageAfterL_, 538
 - averageAfterR_, 538
 - averageAfterU_, 538
 - countAfterL_, 538
 - countAfterR_, 538
 - countAfterU_, 538
 - countInput_, 538
 - numberCounts_, 538
- CoinAbcThreadInfo, 538
 - result, 538
 - status, 539
 - stuff, 539
- CoinAbcTypeFactorization, 539
 - ~CoinAbcTypeFactorization, 551
 - addLink, 561
 - adjustedAreaFactor, 553
 - afterPivot, 561
 - almostDestructor, 551
 - baseL, 553
 - baseL_, 570
 - btranAverageAfterL_, 576
 - btranAverageAfterR_, 576
 - btranAverageAfterU_, 576
 - btranCountAfterL_, 576
 - btranCountAfterR_, 575
 - btranCountAfterU_, 575
 - btranCountInput_, 575
 - btranFullAverageAfterL_, 577
 - btranFullAverageAfterR_, 577
 - btranFullAverageAfterU_, 577
 - btranFullCountAfterL_, 577
 - btranFullCountAfterR_, 577
 - btranFullCountAfterU_, 577
 - btranFullCountInput_, 577
 - checkConsistency, 561
 - checkMarkArrays, 559
 - checkPivot, 564
 - checkReplacePart1, 556
 - checkReplacePart2, 557
 - checkSparse, 559
 - cleanup, 562
 - clearArrays, 559
 - clone, 551
 - CoinAbcFactorizationUnitTest, 566
 - CoinAbcTypeFactorization, 551
 - CoinAbcFactorization.hpp, 710
 - CoinAbcTypeFactorization, 551
 - conditionNumber, 551
 - convertColumnToRowU_, 573
 - convertColumnToRowUAddress_, 566
 - convertRowToColumnU_, 572
 - convertRowToColumnUAddress_, 566
 - deleteLink, 562
 - denseArea_, 573
 - denseAreaAddress_, 568
 - denseThreshold, 554
 - denseThreshold_, 575
 - denseVector, 556
 - doAddresses, 562
 - elementByRowL, 552
 - elementByRowL_, 574
 - elementByRowLAddress_, 568
 - elementL_, 573
 - elementLAddress_, 567
 - elementRAddress_, 568
 - elementRowU_, 573
 - elementRowUAddress_, 566
 - elementU, 555
 - elementU_, 572
 - elementUAddress_, 566
 - elements, 555
 - factor, 560
 - factorDense, 560
 - factorSparse, 560
 - firstCount, 552
 - firstCount_, 571
 - firstCountAddress_, 567
 - firstZeroed_, 569
 - fromLongArray, 556
 - ftranAverageAfterL_, 574
 - ftranAverageAfterR_, 574
 - ftranAverageAfterU_, 574
 - ftranCountAfterL_, 574
 - ftranCountAfterR_, 574
 - ftranCountAfterU_, 574
 - ftranCountInput_, 574
 - ftranFTAveragAfterL_, 575
 - ftranFTAveragAfterR_, 575
 - ftranFTAveragAfterU_, 575
 - ftranFTCountAfterL_, 575
 - ftranFTCountAfterR_, 575
 - ftranFTCountAfterU_, 575
 - ftranFTCountInput_, 575
 - ftranFullAverageAfterL_, 576
 - ftranFullAverageAfterR_, 576
 - ftranFullAverageAfterU_, 576
 - ftranFullCountAfterL_, 576

ftranFullCountAfterR_, 576
 ftranFullCountAfterU_, 576
 ftranFullCountInput_, 576
 getAreas, 560
 getColumnSpace, 561
 getColumnSpaceltrate, 561
 getColumnSpaceltrateR, 561
 getRowSpace, 561
 getRowSpaceltrate, 561
 goSparse, 559
 goSparse2, 559
 gotLCopy, 565
 gotRCopy, 565
 gotSparse, 565
 gotUCopy, 565
 gutsOfCopy, 559
 gutsOfDestructor, 559
 gutsOfInitialize, 559
 indexColumnL, 552
 indexColumnL_, 573
 indexColumnLAddress_, 568
 indexColumnU_, 572
 indexColumnUAddress_, 566
 indexRowL, 552
 indexRowL_, 573
 indexRowLAddress_, 567
 indexRowRAddress_, 568
 indexRowU, 555
 indexRowU_, 572
 indexRowUAddress_, 566
 indices, 551
 initialNumberRows_, 577
 lastColumn_, 571
 lastColumnAddress_, 567
 lastCount, 553
 lastCountAddress_, 567
 lastEntryByColumnU_, 570
 lastEntryByRowU_, 570
 lastRow_, 572
 lastRowAddress_, 567
 lastSlack_, 574
 leadingDimension_, 571
 lengthAreaL, 554
 lengthAreaL_, 570
 lengthAreaR_, 569
 lengthAreaU, 554
 lengthAreaU_, 570
 lengthL_, 570
 lengthR_, 569
 lengthU_, 570
 listAddress_, 567
 makeNonSingular, 560
 markRow_, 572
 markRowAddress_, 567
 maximumCoefficient, 554
 maximumMaximumPivots_, 576
 maximumPivots, 554
 maximumRows_, 575
 maximumRowsExtra, 553
 maximumRowsExtra_, 569
 maximumU_, 570
 messageLevel, 553, 554
 messageLevel_, 574
 modifyLink, 562
 nextColumn_, 571
 nextColumnAddress_, 567
 nextCount, 553
 nextCountAddress_, 567
 nextRow_, 572
 nextRowAddress_, 567
 numberBtranCounts_, 576
 numberBtranFullCounts_, 577
 numberCompressions, 555
 numberCompressions_, 574
 numberElements, 553
 numberElementsL, 554
 numberElementsR, 555
 numberElementsU, 554
 numberForrestTomlin, 553
 numberFtranCounts_, 575
 numberFtranFTCounts_, 575
 numberFtranFullCounts_, 576
 numberGoodL_, 569
 numberInColumn, 555
 numberInColumn_, 571
 numberInColumnAddress_, 566
 numberInColumnPlus_, 571
 numberInColumnPlusAddress_, 566
 numberInRow, 555
 numberInRow_, 571
 numberInRowAddress_, 566
 numberL, 553
 numberL_, 569
 numberR_, 569
 numberRowsExtra, 553
 numberRowsExtra_, 568
 numberRowsLeft_, 569
 numberRowsSmall_, 569
 numberTrials_, 570
 numberU_, 570
 operator=, 551
 pack, 559
 permute, 551
 permute_, 571
 permuteAddress_, 566
 pivot, 564, 565
 pivotColumn, 552
 pivotColumn_, 571

`pivotColumnAddress_`, 566
`pivotColumnSingleton`, 560
`pivotLOrder`, 552
`pivotLOrderAddress_`, 568
`pivotLinkedBackwards`, 552
`pivotLinkedBackwardsAddress_`, 568
`pivotLinkedForwards`, 552
`pivotLinkedForwardsAddress_`, 568
`pivotOneOtherRow`, 560
`pivotRegion`, 552
`pivotRegion_`, 572
`pivotRegionAddress_`, 566
`pivotRowSingleton`, 560
`postProcess`, 560
`preProcess`, 560
`preProcess3`, 560
`preProcess4`, 560
`printRegion`, 559
`reorderU`, 561
`replaceColumnPFI`, 564
`replaceColumnPart3`, 557
`replaceColumnU`, 557
`resetStatistics`, 559
`saveColumn_`, 572
`saveColumnAddress_`, 567
`scan`, 556
`separateLinks`, 562
`setDenseThreshold`, 554
`setNoGotLCopy`, 565
`setNoGotRCopy`, 565
`setNoGotSparse`, 565
`setNoGotUCopy`, 565
`setNumberElementsU`, 554
`setYesGotLCopy`, 565
`setYesGotRCopy`, 565
`setYesGotSparse`, 565
`setYesGotUCopy`, 565
`show_self`, 551
`sizeSparseArray_`, 577
`sort`, 551
`spaceForForrestTomlin`, 554
`sparse_`, 574
`sparseAddress_`, 568
`sparseThreshold`, 559
`sparseThreshold_`, 569
`startColumnL`, 552
`startColumnL_`, 573
`startColumnLAddress_`, 567
`startColumnR`, 555
`startColumnRAddress_`, 568
`startColumnU`, 556
`startColumnU_`, 572
`startColumnUAddress_`, 566
`startRowL`, 552
`startRowL_`, 573
`startRowLAddress_`, 567
`startRowU`, 571
`startRowUAddress_`, 566
`starts`, 555
`state_`, 577
`storeFT`, 562
`toLongArray`, 556
`totalElements_`, 569
`unpack`, 559
`updateColumn`, 558
`updateColumnCpu`, 558
`updateColumnFT`, 557, 558
`updateColumnFTPart1`, 557
`updateColumnFTPart2`, 557
`updateColumnL`, 562
`updateColumnLDense`, 562
`updateColumnLDensish`, 562
`updateColumnLSparse`, 562
`updateColumnPFI`, 563
`updateColumnR`, 562
`updateColumnTranspose`, 558
`updateColumnTransposeCpu`, 558
`updateColumnTransposeL`, 564
`updateColumnTransposeLByRow`, 564
`updateColumnTransposeLDensish`, 564
`updateColumnTransposeLSparse`, 564
`updateColumnTransposePFI`, 563
`updateColumnTransposeR`, 564
`updateColumnTransposeRDensish`, 564
`updateColumnTransposeRSparse`, 564
`updateColumnTransposeU`, 563
`updateColumnTransposeUByColumn`, 563
`updateColumnTransposeUDensish`, 563
`updateColumnTransposeUSparse`, 563
`updateColumnU`, 562
`updateColumnUDense`, 563
`updateColumnUDensish`, 563
`updateColumnUSparse`, 563
`updateFullColumn`, 558
`updateFullColumnTranspose`, 558
`updatePartialUpdate`, 557
`updateTwoColumnsFT`, 558
`updateTwoColumnsUDensish`, 563
`updateWeights`, 558
`wantToGoDense`, 561
`wantsTableauColumn`, 557
`workArea2_`, 573
`workArea2Address_`, 568
`workArea_`, 573
`workAreaAddress_`, 568
`CoinCheckZero`
 `CoinAbcCommon.hpp`, 706
`CoinExponent`

- CoinAbcCommon.hpp, 705
- CoinFabs
 - CoinAbcCommon.hpp, 704
- coinMessages
 - ClpModel, 333
- coinMessages_
 - ClpModel, 342
- coinMessagesPointer
 - ClpModel, 333
- CoinReadGetCommand
 - CbcOrClpParam.hpp, 656
- CoinReadGetDoubleField
 - CbcOrClpParam.hpp, 657
- CoinReadGetIntField
 - CbcOrClpParam.hpp, 657
- CoinReadGetString
 - CbcOrClpParam.hpp, 657
- CoinReadNextField
 - CbcOrClpParam.hpp, 656
- CoinReadPrintit
 - CbcOrClpParam.hpp, 657
- CoinSimplexDouble
 - CoinAbcCommon.hpp, 705
- CoinSimplexInt
 - CoinAbcCommon.hpp, 705
- CoinSimplexUnsignedInt
 - CoinAbcCommon.hpp, 705
- CoinSqrt
 - ClpHelperFunctions.hpp, 682
- column
 - ClpConstraintLinear, 177
 - ClpConstraintQuadratic, 180
- column_
 - AbcMatrix, 39
 - AbcMatrix2, 43
 - AbcMatrix3, 45
 - ClpPackedMatrix2, 392
 - ClpPackedMatrix3, 394
- columnActivity_
 - ClpModel, 338
 - OsiClpSolverInterface, 630
- columnActivityWork_
 - ClpSimplex, 485
- columnArray
 - ClpSimplex, 463
- columnArray_
 - ClpSimplex, 484
- columnCopy_
 - ClpPackedMatrix, 389
- columnLower
 - ampl_info, 135
 - ClpDynamicMatrix, 213, 215
 - ClpModel, 329
- columnLower_
 - ClpDynamicMatrix, 221
 - ClpModel, 338
- columnLowerGen
 - ClpDynamicExampleMatrix, 204
- columnLowerGen_
 - ClpDynamicExampleMatrix, 206
- columnLowerWork_
 - ClpInterior, 273
 - ClpSimplex, 483
- columnName
 - ClpModel, 334
- columnNames
 - ClpModel, 334
- columnNames_
 - ClpModel, 341
- columnNamesAsChar
 - ClpModel, 337
- columnOrdered_
 - ClpPlusMinusOneMatrix, 409
- columnPrimalSequence_
 - ClpSimplex, 480
- columnScale
 - ClpModel, 327
- columnScale2
 - AbcSimplex, 83
- columnScale_
 - ClpModel, 339
 - OsiClpSolverInterface, 633
- columnStatus
 - ampl_info, 136
- columnUpper
 - ampl_info, 136
 - ClpDynamicMatrix, 214, 216
 - ClpModel, 329
- columnUpper_
 - ClpDynamicMatrix, 221
 - ClpModel, 339
- columnUpperGen
 - ClpDynamicExampleMatrix, 204
- columnUpperGen_
 - ClpDynamicExampleMatrix, 206
- columnUpperWork_
 - ClpInterior, 273
 - ClpSimplex, 483
- columnUseScale_
 - AbcSimplex, 97
- complementarityGap
 - ClpInterior, 268
 - ClpPredictorCorrector, 411
- complementarityGap_
 - ClpInterior, 276
- complicatedPivotIn
 - ClpEventHandler, 223
- complicatedPivotOut

- ClpEventHandler, 223
- compressRows
 - AbcWarmStart, 129
- computeDuals
 - AbcSimplex, 80
 - ClpSimplex, 464
- computeInternalObjectiveValue
 - AbcSimplex, 91
 - ClpSimplex, 475
- computeLargestAway
 - OsiClpSolverInterface, 629
- computeObjective
 - AbcSimplex, 80
- computeObjectiveValue
 - AbcSimplex, 90
 - ClpSimplex, 475
- computePrimals
 - AbcSimplex, 80
 - ClpSimplex, 464
- conditionNumber
 - CoinAbcTypeFactorization, 551
- config_clp_default.h
 - CLP_VERSION, 719
- config_default.h
 - COIN_HAS_CLP, 720
- continuousModel_
 - OsiClpSolverInterface, 633
- convertColumnToRowU_
 - CoinAbcTypeFactorization, 573
- convertColumnToRowUAddress_
 - CoinAbcTypeFactorization, 566
- convertRowToColumnU_
 - CoinAbcTypeFactorization, 572
- convertRowToColumnUAddress_
 - CoinAbcTypeFactorization, 566
- copy
 - AbcMatrix, 39
 - ClpModel, 319
 - ClpPackedMatrix, 388
- copyColumnNames
 - ClpModel, 319
- copyEnabledStuff
 - ClpSimplex, 460
 - OsiClpSolverInterface, 611
- copyEnabledSuff
 - OsiClpSolverInterface, 611
- copyFactorization
 - ClpSimplex, 458
- copyFromSaved
 - AbcSimplex, 81
- copyInIntegerInformation
 - ClpModel, 316
- copyNames
 - ClpModel, 319
- copyRowNames
 - ClpModel, 319
- copyinStatus
 - ClpModel, 331
- correctSequence
 - ClpGubMatrix, 251
 - ClpMatrixBase, 296
 - ClpPackedMatrix, 388
- cost
 - AbcSimplex, 88
 - ClpDynamicMatrix, 215
 - ClpGubDynamicMatrix, 240
 - ClpNonLinearCost, 370
 - ClpSimplex, 472
- cost_
 - ClpDynamicMatrix, 220
 - ClpGubDynamicMatrix, 242
 - ClpInterior, 273
 - ClpSimplex, 483
- costAddress
 - AbcSimplex, 88
 - ClpSimplex, 472
- costBasic
 - AbcSimplex, 85
- costBasic_
 - AbcSimplex, 99
- costGen
 - ClpDynamicExampleMatrix, 204
- costGen_
 - ClpDynamicExampleMatrix, 205
- costRegion
 - AbcSimplex, 84, 85
 - ClpSimplex, 469, 470
- costSaved_
 - AbcSimplex, 99
- count_
 - AbcMatrix2, 43
 - ClpPackedMatrix2, 391
- countAfterL_
 - CoinAbcStatistics, 538
- countAfterR_
 - CoinAbcStatistics, 538
- countAfterU_
 - CoinAbcStatistics, 538
- countBasis
 - AbcMatrix, 33
 - ClpDummyMatrix, 198
 - ClpGubMatrix, 248
 - ClpMatrixBase, 293
 - ClpNetworkMatrix, 350
 - ClpPackedMatrix, 383
 - ClpPlusMinusOneMatrix, 405
- countInput_
 - CoinAbcStatistics, 538

- crash
 - AbcSimplex, 90
 - ClpSimplex, 458
 - Idiot, 582
- createArrays
 - ClpNode, 356
- createBasis0
 - AbcWarmStart, 130
 - AbcWarmStartOrganizer, 133
- createBasis12
 - AbcWarmStart, 130
 - AbcWarmStartOrganizer, 133
- createBasis34
 - AbcWarmStart, 130
 - AbcWarmStartOrganizer, 133
- createCoinModel
 - ClpModel, 320
- createDualPricingVectorSerial
 - AbcSimplexDual, 106
- createEmptyMatrix
 - ClpModel, 319
- createPiecewiseLinearCosts
 - ClpSimplex, 463
- createRim
 - ClpSimplex, 468
- createRim1
 - ClpSimplex, 469
- createRim4
 - ClpSimplex, 469
- createRim5
 - ClpSimplex, 469
- createRowCopy
 - AbcMatrix, 33
- createScaledMatrix
 - ClpPackedMatrix, 383
- createSolution
 - ClpPredictorCorrector, 411
- createStatus
 - AbcSimplex, 90
 - ClpSimplex, 474
- createUpdateDuals
 - AbcSimplexPrimal, 122
- createVariable
 - ClpDynamicExampleMatrix, 203
 - ClpDynamicMatrix, 213
 - ClpMatrixBase, 296
- createWorkingData
 - ClpInterior, 269
- crossOver
 - Idiot, 582
- crossover
 - OsiClpSolverInterface, 609
- crunch
 - ClpSimplexOther, 500
- OsiClpSolverInterface, 628
- currentAcceptablePivot
 - AbcSimplex, 87
- currentAcceptablePivot_
 - AbcSimplex, 94
- currentDualBound
 - AbcSimplex, 78
- currentDualBound_
 - AbcSimplex, 93
- currentDualTolerance
 - AbcSimplex, 78
 - ClpSimplex, 467
- currentDualTolerance_
 - AbcSimplex, 93
- currentOption
 - CbcOrClpParam, 145
- currentOptionAsInteger
 - CbcOrClpParam, 145
- currentPrimalTolerance
 - ClpSimplex, 467
- currentStatus
 - AbcNonLinearCost.hpp, 641
 - ClpNonLinearCost.hpp, 693
- currentWanted
 - AbcMatrix, 38
 - ClpMatrixBase, 300
- currentWanted_
 - AbcMatrix, 40
 - ClpMatrixBase, 301
- cut
 - ampl_info, 137
- cycle
 - ClpSimplexProgress, 508
- d1_
 - ClpPdcoBase, 399
- d2_
 - ClpPdcoBase, 399
- DEVEX_ADD_ONE
 - AbcDualRowSteepest.hpp, 639
 - ClpSimplex.hpp, 699
- DEVEX_TRY_NORM
 - AbcDualRowSteepest.hpp, 639
 - ClpSimplex.hpp, 699
- DMUMPS_STRUC_C
 - ClpCholeskyMumps.hpp, 677
- DO_BASIS_AND_ORDER
 - AbcSimplex.hpp, 645
- DO_JUST_BOUNDS
 - AbcSimplex.hpp, 646
- DO_SOLUTION
 - AbcSimplex.hpp, 646
- DO_STATUS
 - AbcSimplex.hpp, 646

- data
 - ClpTrustedData, [517](#)
- dblParam_
 - ClpModel, [337](#)
- debugMove
 - ClpPredictorCorrector, [412](#)
- defaultFactorizationFrequency
 - AbcSimplex, [76](#)
 - ClpSimplex, [460](#)
- defaultHandler
 - ClpModel, [333](#)
- defaultHandler_
 - ClpModel, [341](#)
- deleteBaseModel
 - AbcSimplex, [75](#)
 - ClpSimplex, [453](#)
- deleteBasis
 - AbcWarmStartOrganizer, [133](#)
- deleteCols
 - ClpDummyMatrix, [198](#)
 - ClpMatrixBase, [292](#)
 - ClpNetworkMatrix, [349](#)
 - ClpPackedMatrix, [382](#)
 - ClpPlusMinusOneMatrix, [404](#)
 - OsiClpSolverInterface, [620](#)
- deleteColumns
 - AbcWarmStart, [129](#)
 - ClpModel, [317](#)
- deleteIntegerInformation
 - ClpModel, [316](#)
- deleteLink
 - CoinAbcTypeFactorization, [562](#)
- deleteNamesAsChar
 - ClpModel, [337](#)
- deleteQuadraticObjective
 - ClpModel, [316](#)
 - ClpQuadraticObjective, [435](#)
- deleteRay
 - ClpModel, [331](#)
- deleteRim
 - ClpSimplex, [469](#)
- deleteRows
 - AbcWarmStart, [129](#)
 - ClpDummyMatrix, [198](#)
 - ClpMatrixBase, [292](#)
 - ClpModel, [317](#)
 - ClpNetworkMatrix, [349](#)
 - ClpPackedMatrix, [382](#)
 - ClpPlusMinusOneMatrix, [404](#)
 - OsiClpSolverInterface, [621](#)
- deleteRowsAndColumns
 - ClpModel, [317](#)
- deleteScaleFactors
 - OsiClpSolverInterface, [628](#)
- deleteSome
 - ClpConstraint, [173](#)
 - ClpConstraintLinear, [176](#)
 - ClpConstraintQuadratic, [179](#)
 - ClpLinearObjective, [282](#)
 - ClpObjective, [374](#)
 - ClpQuadraticObjective, [434](#)
- deleteWorkingData
 - ClpInterior, [269](#)
- delta
 - ClpInterior, [268](#)
 - Options, [592](#)
- delta_
 - ClpInterior, [275](#)
- deltaSL_
 - ClpInterior, [277](#)
- deltaSU_
 - ClpInterior, [277](#)
- deltaW_
 - ClpInterior, [277](#)
- deltaX_
 - ClpInterior, [277](#)
- deltaY_
 - ClpInterior, [277](#)
- deltaZ_
 - ClpInterior, [277](#)
- deltay
 - Info, [587](#)
- dense_
 - ClpCholeskyBase, [157](#)
- denseArea_
 - CoinAbcTypeFactorization, [573](#)
- denseAreaAddress_
 - CoinAbcTypeFactorization, [568](#)
- denseColumn_
 - ClpCholeskyBase, [156](#)
- denseThreshold
 - ClpFactorization, [231](#)
 - CoinAbcTypeFactorization, [554](#)
- denseThreshold_
 - ClpCholeskyBase, [157](#)
 - CoinAbcTypeFactorization, [575](#)
- denseVector
 - CoinAbcTypeFactorization, [556](#)
- depth
 - ClpNode, [357](#)
- depth_
 - ClpNode, [359](#)
- destroyPresolve
 - ClpPresolve, [418](#)
- diag1_
 - ClpLsqr, [285](#)
- diag2_
 - ClpLsqr, [285](#)

- diagonal
 - ClpCholeskyBase, [152](#)
 - ClpCholeskyDense, [159](#)
- diagonal_
 - ClpCholeskyBase, [155](#)
 - ClpCholeskyDenseC, [160](#)
 - ClpInterior, [277](#)
- diagonalNorm
 - ClpInterior, [267](#)
- diagonalNorm_
 - ClpInterior, [274](#)
- diagonalPerturbation
 - ClpInterior, [268](#)
- diagonalPerturbation_
 - ClpInterior, [275](#)
- diagonalScaleFactor_
 - ClpInterior, [275](#)
- direction
 - ampl_info, [135](#)
- directionIn
 - ClpSimplex, [470](#)
- directionIn_
 - AbcSimplexPrimal::pivotStruct, [637](#)
 - ClpSimplex, [484](#)
- directionOut
 - ClpSimplex, [471](#)
- directionOut_
 - AbcSimplexPrimal::pivotStruct, [638](#)
 - ClpSimplex, [484](#)
- directionVector
 - ClpSimplexNonlinear, [496](#)
- disableFactorization
 - OsiClpSolverInterface, [609](#)
- disableSimplexInterface
 - OsiClpSolverInterface, [611](#)
- disasterArea_
 - ClpSimplex, [486](#)
- disasterHandler
 - ClpSimplex, [466](#)
 - OsiClpSolverInterface, [626](#)
- disasterHandler_
 - OsiClpSolverInterface, [633](#)
- displayThis
 - CbcOrClpParam, [146](#)
- dj_
 - ClpInterior, [273](#)
 - ClpSimplex, [484](#)
- djAtBeginning
 - IdiotResult, [586](#)
- djAtEnd
 - IdiotResult, [586](#)
- djBasic
 - AbcSimplex, [85](#)
- djBasic_
 - AbcSimplex, [99](#)
- djRegion
 - AbcSimplex, [84](#), [85](#)
 - ClpSimplex, [469](#), [470](#)
- djSaved_
 - AbcSimplex, [99](#)
- djsAndDevex
 - AbcPrimalColumnSteepest, [59](#)
 - ClpPrimalColumnSteepest, [427](#)
- djsAndDevex2
 - AbcPrimalColumnSteepest, [59](#)
 - ClpPrimalColumnSteepest, [427](#)
- djsAndSteepest
 - ClpPrimalColumnSteepest, [427](#)
- djsAndSteepest2
 - ClpPrimalColumnSteepest, [428](#)
- do_lsqr
 - ClpLsqr, [285](#)
- doAbcDual
 - AbcSimplex, [76](#)
- doAbcPrimal
 - AbcSimplex, [76](#)
- doAddresses
 - CoinAbcTypeFactorization, [562](#)
- doDoubleton
 - ClpPresolve, [415](#)
 - ClpSolve, [515](#)
- doDual
 - ClpPresolve, [415](#)
 - ClpSolve, [514](#)
- doDupcol
 - ClpPresolve, [416](#)
 - ClpSolve, [515](#)
- doDuprow
 - ClpPresolve, [416](#)
 - ClpSolve, [516](#)
- doEasyOnesInValuesPass
 - ClpSimplexDual, [493](#)
- doFTUpdate
 - AbcSimplexPrimal, [122](#)
- doForcing
 - ClpPresolve, [416](#)
 - ClpSolve, [515](#)
- doGubrow
 - ClpPresolve, [417](#)
- doImpliedFree
 - ClpPresolve, [416](#)
 - ClpSolve, [515](#)
- doIntersection
 - ClpPresolve, [417](#)
- doKKT_
 - ClpCholeskyBase, [154](#)
- doKillSmall
 - ClpSolve, [516](#)

- doSingleton
 - ClpPresolve, 415
 - ClpSolve, 514
- doSingletonColumn
 - ClpPresolve, 417
 - ClpSolve, 516
- doSteepestWork
 - AbcPrimalColumnSteepest, 60
- doTighten
 - ClpPresolve, 416
 - ClpSolve, 515
- doTripleton
 - ClpPresolve, 416
 - ClpSolve, 515
- doTwoxTwo
 - ClpPresolve, 417
- dontFactorizePivots_
 - AbcTolerancesEtc, 126
 - ClpSimplex, 487
- doubleCheck
 - ClpSimplex, 459
- doubleParameter
 - CbcOrClpParam, 143, 144
- doubleParameters_
 - ClpCholeskyBase, 156
 - ClpCholeskyDenseC, 161
- doubleValue
 - CbcOrClpParam, 145
- downPseudo_
 - ClpNodeStuff, 363
- downRange
 - OsiClpSolverInterface, 628
- dropNames
 - ClpModel, 319
- dropThis
 - IdiotResult, 586
- dual
 - AbcSimplex, 76
 - AbcSimplexDual, 104
 - ClpSimplex, 455
 - ClpSimplexDual, 490
- dual_
 - ClpModel, 338
- dualBound
 - ClpSimplex, 461
- dualBound_
 - AbcTolerancesEtc, 125
 - ClpDataSave, 182
 - ClpSimplex, 481
- dualColumn
 - ClpSimplexDual, 492
- dualColumn0
 - ClpSimplexDual, 492
- dualColumn1
 - AbcMatrix, 35
 - AbcSimplexDual, 106
- dualColumn1A
 - AbcSimplexDual, 106
- dualColumn1B
 - AbcSimplexDual, 106
- dualColumn1Part
 - AbcMatrix, 36
- dualColumn1Row
 - AbcMatrix, 35
- dualColumn1Row1
 - AbcMatrix, 35
- dualColumn1Row2
 - AbcMatrix, 35
- dualColumn1RowFew
 - AbcMatrix, 35
- dualColumn2
 - AbcSimplexDual, 107
- dualColumn2First
 - AbcSimplexDual, 107
- dualColumn2Most
 - AbcSimplexDual, 107
- dualColumnResult, 578
 - bestEverPivot, 579
 - block, 580
 - increaseInObjective, 579
 - increaseInThis, 579
 - lastPivotValue, 579
 - lastSequence, 579
 - modifyCosts, 580
 - numberLastSwapped, 580
 - numberRemaining, 580
 - numberSwapped, 580
 - sequence, 580
 - tentativeTheta, 579
 - theta, 579
 - thisPivotValue, 579
 - thruThis, 579
 - totalThru, 579
 - useThru, 579
- dualColumnSolution
 - ClpModel, 324
- dualDebug
 - ClpSimplex, 455
- dualExpanded
 - ClpDynamicMatrix, 212
 - ClpGubMatrix, 250
 - ClpMatrixBase, 295
- dualFeasible
 - ClpInterior, 267
 - ClpSimplex, 460
- dualIn
 - ClpSimplex, 468
- dualIn_

- ClpModel, 335
- enableFactorization
 - OsiClpSolverInterface, 609
- enableSimplexInterface
 - OsiClpSolverInterface, 611
- end
 - ClpGubMatrix, 252
- endInDual
 - ClpEventHandler, 223
- endInPrimal
 - ClpEventHandler, 223
- endOfCreateRim
 - ClpEventHandler, 223
- endOfFactorization
 - ClpEventHandler, 223
- endOfIteration
 - ClpEventHandler, 223
- endOfValuesPass
 - ClpEventHandler, 223
- end_
 - ClpGubMatrix, 253
- endFraction
 - AbcMatrix, 38
 - ClpMatrixBase, 300
- endFraction_
 - AbcMatrix, 40
 - ClpMatrixBase, 301
- endOddState
 - ClpSimplexProgress, 508
- endingTheta
 - ClpSimplexOther::parametricsData, 635
- errorRegion_
 - ClpInterior, 276
- establishParams
 - CbcOrClpParam.hpp, 657
- estimatedSolution
 - ClpNode, 357
- estimatedSolution_
 - ClpNode, 358
- Event
 - ClpEventHandler, 223
- event
 - ClpEventHandler, 224
 - MyEventHandler, 588
- eventHandler
 - ClpModel, 333
- eventHandler_
 - ClpModel, 341
- eventWithInfo
 - ClpEventHandler, 224
- exactOutgoing
 - AbcSimplexPrimal, 121
 - ClpSimplexPrimal, 503, 504
- expandKnapsack
 - ClpSimplexOther, 500
- extendUpdated
 - ClpGubMatrix, 250
 - ClpMatrixBase, 295
- extraInformation_
 - AbcWarmStart, 131
- extractSenseRhsRange
 - OsiClpSolverInterface, 627
- FACTOR_CPU
 - CoinAbcBaseFactorization.hpp, 701
- FAKE_SUPERBASIC
 - AbcSimplex.hpp, 646
- FREE_ACCEPT
 - ClpMatrixBase.hpp, 685
- FREE_BIAS
 - ClpMatrixBase.hpp, 685
- factor
 - CoinAbcAnyFactorization, 526
 - CoinAbcDenseFactorization, 533
 - CoinAbcTypeFactorization, 560
- factorDense
 - CoinAbcTypeFactorization, 560
- factorElements_
 - CoinAbcAnyFactorization, 529
- factorSparse
 - CoinAbcTypeFactorization, 560
- factorization
 - AbcSimplex, 76
 - AbcSimplexFactorization, 118
 - ClpSimplex, 461
- factorization_
 - ClpNode, 358
 - ClpSimplex, 486
 - OsiClpSolverInterface, 630
- factorizationFrequency
 - AbcSimplex, 76
 - ClpSimplex, 461
- factorizationStatistics
 - CoinAbcCommonFactorization.hpp, 707
- factorize
 - AbcSimplexFactorization, 112
 - ClpCholeskyBase, 151
 - ClpCholeskyDense, 159
 - ClpCholeskyMumps, 162
 - ClpCholeskyTaucs, 164
 - ClpCholeskyUfl, 166
 - ClpCholeskyWssmp, 168
 - ClpCholeskyWssmpKKT, 170
 - ClpFactorization, 228
 - ClpNetworkBasis, 344
 - ClpSimplex, 464
- factorizePart2
 - ClpCholeskyBase, 153

- ClpCholeskyDense, 159
- factorizePart3
 - ClpCholeskyDense, 159
- FakeBound
 - AbcSimplex, 74
 - ClpSimplex, 452
- fakeDjs
 - AbcSimplex, 85
- fakeKeyword
 - CbcOrClpParam, 146
- fakeLower
 - ClpInterior, 271
- fakeMinInSimplex_
 - OsiClpSolverInterface, 632
- fakeObjective
 - OsiClpSolverInterface, 626
- fakeObjective_
 - OsiClpSolverInterface, 633
- fakeSuperBasic
 - AbcSimplex, 87
- fakeUpper
 - ClpInterior, 272
- fastCrunch
 - ClpSimplex, 459
- fastDual
 - AbcSimplexDual, 108
 - ClpSimplexDual, 494
- fastDual2
 - ClpSimplex, 459
- fathom
 - ClpSimplex, 459
- fathomMany
 - ClpSimplex, 459
- fathomed
 - ClpNode, 357
- FeaTol
 - Options, 592
- feasibleBounds
 - AbcNonLinearCost, 49
 - ClpNonLinearCost, 369
- feasibleCost
 - AbcNonLinearCost, 50
 - ClpNonLinearCost, 370
- feasibleExtremePoints_
 - MyMessageHandler, 591
- feasibleReportCost
 - AbcNonLinearCost, 50
 - ClpNonLinearCost, 370
- fill
 - AbcDualRowSteepest, 25
 - ClpDualRowSteepest, 193
- fillBasis
 - AbcMatrix, 33
 - ClpDummyMatrix, 198
- ClpGubMatrix, 248
- ClpMatrixBase, 293
- ClpNetworkMatrix, 350
- ClpPackedMatrix, 383
- ClpPlusMinusOneMatrix, 405
- fillFromModel
 - ClpSimplexProgress, 508
- fillParamMaps
 - OsiClpSolverInterface, 627
- fillPerturbation
 - AbcSimplex, 81
- fillPseudoCosts
 - ClpNodeStuff, 362
- findDirectionVector
 - ClpPredictorCorrector, 411
- findIntegersAndSOS
 - OsiClpSolverInterface, 619
- findNetwork
 - ClpModel, 320
- findStepLength
 - ClpPredictorCorrector, 411
- finish
 - ClpSimplex, 460
- finishSolve
 - AbcSimplexDual, 108
 - ClpSimplexDual, 494
- firstAvailable
 - ClpDynamicMatrix, 216
 - ClpGubDynamicMatrix, 240
- firstAvailable_
 - ClpDynamicMatrix, 219
 - ClpGubDynamicMatrix, 243
- firstAvailableBefore_
 - ClpDynamicMatrix, 219
- firstBasis_
 - AbcWarmStartOrganizer, 133
- firstBranch
 - ClpNode::branchState, 139
- firstCount
 - CoinAbcTypeFactorization, 552
- firstCount_
 - CoinAbcTypeFactorization, 571
- firstCountAddress_
 - CoinAbcTypeFactorization, 567
- firstDense_
 - ClpCholeskyBase, 156
- firstDynamic
 - ClpDynamicMatrix, 216
 - ClpGubDynamicMatrix, 241
- firstDynamic_
 - ClpDynamicMatrix, 219
 - ClpGubDynamicMatrix, 243
- firstFree
 - AbcSimplex, 87

- firstFree_
 - ClpSimplex, [487](#)
- firstGub_
 - ClpGubMatrix, [256](#)
- firstIteration
 - ClpSimplexOther::parametricsData, [636](#)
- firstZeroed_
 - CoinAbcTypeFactorization, [569](#)
- fixFixed
 - ClpInterior, [269](#)
- fixOnReducedCosts
 - ClpNode, [356](#)
- fixed
 - ClpInterior, [270](#)
- fixed_
 - ClpNode, [359](#)
- fixedOrFree
 - ClpInterior, [271](#)
- flagged
 - AbcSimplex, [90](#)
 - ClpDynamicMatrix, [215](#)
 - ClpGubDynamicMatrix, [239](#)
 - ClpGubMatrix, [252](#)
 - ClpInterior, [270](#)
 - ClpSimplex, [474](#)
- flaggedGen
 - ClpDynamicExampleMatrix, [205](#)
- flaggedSlack
 - ClpDynamicMatrix, [214](#)
- flags
 - ClpPackedMatrix, [388](#)
- flags_
 - ClpNode, [360](#)
 - ClpPackedMatrix, [389](#)
- flipBack
 - AbcSimplexDual, [106](#)
- flipBounds
 - AbcSimplexDual, [106](#)
 - ClpSimplexDual, [492](#)
- forceFactorization
 - ClpSimplex, [475](#)
- forceFactorization_
 - AbcTolerancesEtc, [126](#)
 - ClpDataSave, [182](#)
 - ClpSimplex, [486](#)
- forceOtherFactorization
 - AbcSimplexFactorization, [116](#)
 - ClpFactorization, [232](#)
- freeArgs
 - Clp_ampl.h, [657](#)
- freeArrays1
 - Clp_ampl.h, [657](#)
- freeArrays2
 - Clp_ampl.h, [657](#)
- freeCachedResults
 - OsiClpSolverInterface, [627](#)
- freeCachedResults0
 - OsiClpSolverInterface, [627](#)
- freeCachedResults1
 - OsiClpSolverInterface, [627](#)
- freeSequenceIn
 - AbcSimplex, [87](#)
- freeSequenceIn_
 - AbcSimplex, [95](#)
- fromIndex_
 - ClpDynamicMatrix, [218](#)
 - ClpGubMatrix, [255](#)
- fromLongArray
 - CoinAbcTypeFactorization, [556](#)
- ftAlpha_
 - AbcSimplex, [94](#)
- ftranAverageAfterL_
 - CoinAbcTypeFactorization, [574](#)
- ftranAverageAfterR_
 - CoinAbcTypeFactorization, [574](#)
- ftranAverageAfterU_
 - CoinAbcTypeFactorization, [574](#)
- ftranCountAfterL_
 - CoinAbcTypeFactorization, [574](#)
- ftranCountAfterR_
 - CoinAbcTypeFactorization, [574](#)
- ftranCountAfterU_
 - CoinAbcTypeFactorization, [574](#)
- ftranCountInput_
 - CoinAbcTypeFactorization, [574](#)
- ftranFTAverageAfterL_
 - CoinAbcTypeFactorization, [575](#)
- ftranFTAverageAfterR_
 - CoinAbcTypeFactorization, [575](#)
- ftranFTAverageAfterU_
 - CoinAbcTypeFactorization, [575](#)
- ftranFTCountAfterL_
 - CoinAbcTypeFactorization, [575](#)
- ftranFTCountAfterR_
 - CoinAbcTypeFactorization, [575](#)
- ftranFTCountAfterU_
 - CoinAbcTypeFactorization, [575](#)
- ftranFTCountInput_
 - CoinAbcTypeFactorization, [575](#)
- ftranFullAverageAfterL_
 - CoinAbcTypeFactorization, [576](#)
- ftranFullAverageAfterR_
 - CoinAbcTypeFactorization, [576](#)
- ftranFullAverageAfterU_
 - CoinAbcTypeFactorization, [576](#)
- ftranFullCountAfterL_
 - CoinAbcTypeFactorization, [576](#)
- ftranFullCountAfterR_
 - CoinAbcTypeFactorization, [576](#)

- CoinAbcTypeFactorization, 576
- ftranFullCountAfterU_
 - CoinAbcTypeFactorization, 576
- ftranFullCountInput_
 - CoinAbcTypeFactorization, 576
- fullMatrix
 - ClpQuadraticObjective, 435
- fullStart
 - ClpGubDynamicMatrix, 240
- fullStart_
 - ClpGubDynamicMatrix, 242
- fullStartGen
 - ClpDynamicExampleMatrix, 204
- fullStartGen_
 - ClpDynamicExampleMatrix, 205
- functionPointer
 - scatterStruct, 638
- functionValue
 - ClpConstraint, 172, 173
- functionValue_
 - ClpConstraint, 174
- gamma
 - ClpInterior, 268
 - Options, 592
- gamma_
 - ClpInterior, 275
- generalExpanded
 - ClpDynamicMatrix, 212
 - ClpGubMatrix, 250
 - ClpMatrixBase, 296
- generateCpp
 - ClpModel, 335
 - ClpSimplex, 476
 - ClpSolve, 513
 - OsiClpSolverInterface, 624
- getAccuracyCheck
 - CoinAbcAnyFactorization, 523
- getAreas
 - CoinAbcAnyFactorization, 525
 - CoinAbcDenseFactorization, 533
 - CoinAbcTypeFactorization, 560
- getAvailableArray
 - AbcSimplex, 88
- getAvailableArrayPublic
 - AbcSimplex, 88
- getBlncCol
 - ClpSimplex, 476
 - OsiClpSolverInterface, 610
- getBlncRow
 - ClpSimplex, 476
 - OsiClpSolverInterface, 610
- getBlncCol
 - ClpSimplex, 476
- OsiClpSolverInterface, 610
- getBlncRow
 - ClpSimplex, 476
 - OsiClpSolverInterface, 610
- getBasics
 - ClpSimplex, 476
 - OsiClpSolverInterface, 610
- getBasis
 - AbcSimplex, 76
 - ClpSimplex, 458
 - OsiClpSolverInterface, 627, 628
- getBasisDiff
 - OsiClpSolverInterface, 628
- getBasisStatus
 - OsiClpSolverInterface, 609
- getBoundTypes
 - ClpPdco, 396
- getColLower
 - ClpModel, 329
 - OsiClpSolverInterface, 614
- getColName
 - OsiClpSolverInterface, 614
- getColSolution
 - AbcSimplex, 83
 - ClpModel, 324
 - OsiClpSolverInterface, 616
- getColType
 - OsiClpSolverInterface, 615
- getColUpper
 - ClpModel, 329
 - OsiClpSolverInterface, 614
- getColumnName
 - ClpModel, 334
- getColumnSpace
 - CoinAbcTypeFactorization, 561
- getColumnSpaceIterate
 - CoinAbcTypeFactorization, 561
- getColumnSpaceIterateR
 - CoinAbcTypeFactorization, 561
- getColumnStatus
 - ClpSimplex, 474
- getConstPointerToWarmStart
 - OsiClpSolverInterface, 612
- getCurrentStatus
 - AbcNonLinearCost, 51
- getD1
 - ClpPdcoBase, 398
- getD2
 - ClpPdcoBase, 399
- getDbParam
 - ClpModel, 335
 - OsiClpSolverInterface, 611
- getDenseThreshold
 - AbcSimplexFactorization, 116

- getDoubleParameter
 - ClpCholeskyBase, 153
- getDropEnoughFeasibility
 - Idiot, 585
- getDropEnoughWeighted
 - Idiot, 585
- getDualRays
 - OsiClpSolverInterface, 616
- getDynamicStatus
 - ClpDynamicMatrix, 215
 - ClpGubDynamicMatrix, 239
- getDynamicStatusGen
 - ClpDynamicExampleMatrix, 205
- getElements
 - AbcMatrix, 32
 - ClpDummyMatrix, 197
 - ClpMatrixBase, 291
 - ClpNetworkMatrix, 349
 - ClpPackedMatrix, 381
 - ClpPlusMinusOneMatrix, 403
- getEmptyFactorization
 - AbcSimplex, 76
 - ClpSimplex, 476
- getEmptyWarmStart
 - OsiClpSolverInterface, 612
- getExitInfeasibility
 - Idiot, 583
- getExtraInfo
 - ClpSolve, 514
- getFakeBound
 - AbcSimplex, 89
 - ClpSimplex, 473
- getFeasibilityTolerance
 - Idiot, 583
- getFeasibleExtremePoints
 - MyMessageHandler, 590
- getGrad
 - ClpPdco, 396
 - ClpPdcoBase, 398
- getGubBasis
 - ClpSimplexOther, 500
- getHessian
 - ClpPdco, 396
 - ClpPdcoBase, 398
- getIndices
 - AbcMatrix, 32
 - ClpDummyMatrix, 197
 - ClpMatrixBase, 291
 - ClpNetworkMatrix, 349
 - ClpPackedMatrix, 381
 - ClpPlusMinusOneMatrix, 404
- getInfinity
 - OsiClpSolverInterface, 616
- getIntParam
 - ClpModel, 335
 - OsiClpSolverInterface, 611
- getIntegerParameter
 - ClpCholeskyBase, 152
- getInternalColumnStatus
 - AbcSimplex, 86
- getInternalStatus
 - AbcSimplex, 85
- getIterationCount
 - ClpModel, 321
 - OsiClpSolverInterface, 616
- getLightweight
 - Idiot, 584
- getLogLevel
 - Idiot, 584
- getMajorIterations
 - Idiot, 583
- getMatrixByCol
 - OsiClpSolverInterface, 616
- getMatrixByRow
 - OsiClpSolverInterface, 615
- getMinorIterations
 - Idiot, 584
- getMinorIterations0
 - Idiot, 584
- getModelPtr
 - OsiClpSolverInterface, 624
- getMutableElements
 - AbcMatrix, 32
 - ClpPackedMatrix, 381
- getMutableIndices
 - AbcMatrix, 32
 - ClpPlusMinusOneMatrix, 404
- getMutableMatrixByCol
 - OsiClpSolverInterface, 616
- getMutableVectorLengths
 - AbcMatrix, 33
- getMutableVectorStarts
 - AbcMatrix, 33
- getNorms
 - ClpHelperFunctions.hpp, 682
- getNumCols
 - AbcMatrix, 32
 - ClpDummyMatrix, 197
 - ClpMatrixBase, 291
 - ClpModel, 320
 - ClpNetworkMatrix, 348
 - ClpPackedMatrix, 381
 - ClpPlusMinusOneMatrix, 403
 - OsiClpSolverInterface, 613
- getNumElements
 - AbcMatrix, 32
 - ClpDummyMatrix, 197
 - ClpMatrixBase, 291

- ClpModel, [329](#)
- ClpNetworkMatrix, [348](#)
- ClpPackedMatrix, [380](#)
- ClpPlusMinusOneMatrix, [403](#)
- OsiClpSolverInterface, [613](#)
- getNumRows
 - AbcMatrix, [32](#)
 - ClpDummyMatrix, [197](#)
 - ClpMatrixBase, [291](#)
 - ClpModel, [320](#)
 - ClpNetworkMatrix, [348](#)
 - ClpPackedMatrix, [381](#)
 - ClpPlusMinusOneMatrix, [403](#)
 - OsiClpSolverInterface, [613](#)
- getObj
 - ClpPdco, [396](#)
 - ClpPdcoBase, [398](#)
- getObjCoefficients
 - ClpModel, [329](#)
 - OsiClpSolverInterface, [615](#)
- getObjSense
 - ClpModel, [323](#)
 - OsiClpSolverInterface, [615](#)
- getObjValue
 - ClpModel, [330](#)
 - OsiClpSolverInterface, [616](#)
- getPackedMatrix
 - AbcMatrix, [31](#)
 - ClpDummyMatrix, [197](#)
 - ClpMatrixBase, [291](#)
 - ClpNetworkMatrix, [348](#)
 - ClpPackedMatrix, [380](#)
 - ClpPlusMinusOneMatrix, [403](#)
- getPointerToWarmStart
 - OsiClpSolverInterface, [612](#), [613](#)
- getPresolvePasses
 - ClpSolve, [514](#)
- getPresolveType
 - ClpSolve, [514](#)
- getPrimalRays
 - OsiClpSolverInterface, [617](#)
- getReasonablyFeasible
 - Idiot, [583](#)
- getReducelIterations
 - Idiot, [584](#)
- getReducedCost
 - AbcSimplex, [83](#)
 - ClpModel, [324](#)
 - OsiClpSolverInterface, [616](#)
- getReducedGradient
 - OsiClpSolverInterface, [610](#)
- getRightHandSide
 - OsiClpSolverInterface, [614](#)
- getRowActivity
 - AbcSimplex, [83](#)
 - ClpModel, [323](#)
 - OsiClpSolverInterface, [616](#)
- getRowBound
 - ClpModel, [336](#)
- getRowLower
 - ClpModel, [324](#)
 - OsiClpSolverInterface, [614](#)
- getRowName
 - ClpModel, [334](#)
 - OsiClpSolverInterface, [613](#)
- getRowObjCoefficients
 - ClpModel, [329](#)
- getRowPrice
 - AbcSimplex, [83](#)
 - ClpModel, [324](#)
 - OsiClpSolverInterface, [616](#)
- getRowRange
 - OsiClpSolverInterface, [614](#)
- getRowSense
 - OsiClpSolverInterface, [614](#)
- getRowSpace
 - CoinAbcTypeFactorization, [561](#)
- getRowSpacelIterate
 - CoinAbcTypeFactorization, [561](#)
- getRowStatus
 - ClpSimplex, [474](#)
- getRowUpper
 - ClpModel, [324](#)
 - OsiClpSolverInterface, [615](#)
- getSizeL
 - OsiClpSolverInterface, [626](#)
- getSizeU
 - OsiClpSolverInterface, [626](#)
- getSmallElementValue
 - ClpModel, [329](#)
- getSolution
 - AbcSimplex, [77](#)
 - ClpSimplex, [463](#)
- getSolveType
 - ClpSolve, [514](#)
- getSpecialOption
 - ClpSolve, [514](#)
- getStartingWeight
 - Idiot, [582](#)
- getStatus
 - ClpDynamicMatrix, [214](#)
 - ClpGubMatrix, [251](#)
 - ClpSimplex, [470](#)
- getStrParam
 - ClpModel, [335](#)
 - OsiClpSolverInterface, [611](#)
- getStrategy
 - Idiot, [584](#)

- getTableauColumnFlipAndStartReplaceSerial
 - AbcSimplexDual, [106](#)
- getTableauColumnPart1Serial
 - AbcSimplexDual, [106](#)
- getTableauColumnPart2
 - AbcSimplexDual, [106](#)
- getTrustedUserPointer
 - ClpModel, [332](#)
- getUserPointer
 - ClpModel, [332](#)
- getVectorLength
 - ClpMatrixBase, [292](#)
 - ClpPackedMatrix, [381](#)
- getVectorLengths
 - AbcMatrix, [33](#)
 - ClpDummyMatrix, [197](#)
 - ClpMatrixBase, [292](#)
 - ClpNetworkMatrix, [349](#)
 - ClpPackedMatrix, [381](#)
 - ClpPlusMinusOneMatrix, [404](#)
- getVectorStarts
 - AbcMatrix, [32](#)
 - ClpDummyMatrix, [197](#)
 - ClpMatrixBase, [291](#)
 - ClpNetworkMatrix, [349](#)
 - ClpPackedMatrix, [381](#)
 - ClpPlusMinusOneMatrix, [404](#)
- getWarmStart
 - OsiClpSolverInterface, [612](#)
- getWeightFactor
 - Idiot, [583](#)
- getWeights
 - ClpFactorization, [233](#)
- getbackSolution
 - ClpSimplex, [455](#)
- goBack
 - AbcNonLinearCost, [49](#)
 - ClpNonLinearCost, [368](#)
- goBackAll
 - AbcNonLinearCost, [49](#)
 - ClpNonLinearCost, [369](#)
- goDense
 - ClpCholeskyBase, [151](#)
- goDense_
 - ClpCholeskyBase, [154](#)
- goDenseOrSmall
 - AbcSimplexFactorization, [116](#)
 - ClpFactorization, [233](#)
- goDenseThreshold
 - AbcSimplexFactorization, [117](#)
 - ClpFactorization, [233](#)
- goLongThreshold
 - AbcSimplexFactorization, [117](#)
- goOsiThreshold
 - ClpFactorization, [232](#)
- goSmallThreshold
 - AbcSimplexFactorization, [117](#)
 - ClpFactorization, [233](#)
- goSparse
 - AbcSimplexFactorization, [117](#)
 - ClpFactorization, [233](#)
 - CoinAbcAnyFactorization, [524](#)
 - CoinAbcTypeFactorization, [559](#)
- goSparse2
 - CoinAbcTypeFactorization, [559](#)
- goThru
 - AbcNonLinearCost, [48](#)
 - ClpNonLinearCost, [368](#)
- goneDualFeasible_
 - ClpInterior, [279](#)
- gonePrimalFeasible_
 - ClpInterior, [279](#)
- goodFactorization
 - ClpEventHandler, [223](#)
- goodAccuracy
 - ClpSimplex, [464](#)
- gotLCopy
 - CoinAbcTypeFactorization, [565](#)
- gotRCopy
 - CoinAbcTypeFactorization, [565](#)
- gotRowCopy
 - AbcMatrix, [39](#)
- gotSparse
 - CoinAbcTypeFactorization, [565](#)
- gotUCopy
 - CoinAbcTypeFactorization, [565](#)
- gradient
 - ClpConstraint, [172](#)
 - ClpConstraintLinear, [176](#)
 - ClpConstraintQuadratic, [179](#)
 - ClpLinearObjective, [281](#)
 - ClpObjective, [373](#)
 - ClpQuadraticObjective, [434](#)
- gubCrash
 - ClpDynamicMatrix, [213](#)
- gubRowStatus
 - ClpDynamicMatrix, [217](#)
 - ClpGubDynamicMatrix, [241](#)
- gubSlackIn_
 - ClpGubMatrix, [255](#)
- gubType_
 - ClpGubMatrix, [256](#)
- gubVersion
 - ClpSimplexOther, [500](#)
- gutsOfConstructor
 - ClpNode, [358](#)
- gutsOfCopy
 - AbcSimplex, [84](#)

- ClpInterior, 269
- ClpModel, 336
- ClpSimplex, 468
- CoinAbcDenseFactorization, 536
- CoinAbcTypeFactorization, 559
- gutsOfDelete
 - AbcSimplex, 84
 - ClpInterior, 269
 - ClpModel, 336
 - ClpSimplex, 468
- gutsOfDestructor
 - CoinAbcDenseFactorization, 536
 - CoinAbcTypeFactorization, 559
 - OsiClpSolverInterface, 627
- gutsOfDual
 - AbcSimplexDual, 108
 - ClpSimplexDual, 494
- gutsOfInitialize
 - AbcSimplex, 84
 - CoinAbcDenseFactorization, 536
 - CoinAbcTypeFactorization, 559
- gutsOfLoadModel
 - ClpModel, 336
- gutsOfPresolvedModel
 - ClpPresolve, 418
- gutsOfPrimalSolution
 - AbcSimplex, 80
- gutsOfResize
 - AbcSimplex, 84
- gutsOfScaling
 - ClpModel, 336
- gutsOfSolution
 - AbcSimplex, 80, 84
 - ClpSimplex, 468
- HEAVY_PERTURBATION
 - AbcSimplex.hpp, 644
- handler_
 - ClpModel, 341
 - ClpNodeStuff, 364
- hash_
 - ClpHashValue, 258
- hiddenRows
 - ClpGubMatrix, 249
 - ClpMatrixBase, 295
- historyInfeasibility_
 - ClpInterior, 276
- hitMaximumIterations
 - ClpModel, 322
- housekeeping
 - AbcSimplex, 80
 - ClpInterior, 269
 - ClpSimplex, 465

INITIAL_AVERAGE

- CoinAbcCommonFactorization.hpp, 707
- INITIAL_AVERAGE2
 - CoinAbcCommonFactorization.hpp, 707
- INLINE_GATHER
 - CoinAbcHelperFunctions.hpp, 715
- INLINE_SCATTER
 - CoinAbcHelperFunctions.hpp, 714
- id
 - ClpDynamicMatrix, 215
 - ClpGubDynamicMatrix, 240
- id_
 - ClpDynamicMatrix, 220
 - ClpGubDynamicMatrix, 242
- idGen
 - ClpDynamicExampleMatrix, 204
- idGen_
 - ClpDynamicExampleMatrix, 206
- Idiot, 580
 - ~Idiot, 582
 - crash, 582
 - crossOver, 582
 - getDropEnoughFeasibility, 585
 - getDropEnoughWeighted, 585
 - getExitInfeasibility, 583
 - getFeasibilityTolerance, 583
 - getLightweight, 584
 - getLogLevel, 584
 - getMajorIterations, 583
 - getMinorIterations, 584
 - getMinorIterations0, 584
 - getReasonablyFeasible, 583
 - getReduceIterations, 584
 - getStartingWeight, 582
 - getStrategy, 584
 - getWeightFactor, 583
 - Idiot, 582
 - operator=, 582
 - setDropEnoughFeasibility, 585
 - setDropEnoughWeighted, 585
 - setExitInfeasibility, 583
 - setFeasibilityTolerance, 583
 - setLightweight, 584
 - setLogLevel, 584
 - setMajorIterations, 584
 - setMinorIterations, 584
 - setMinorIterations0, 584
 - setReasonablyFeasible, 583
 - setReduceIterations, 584
 - setStartingWeight, 583
 - setStrategy, 585
 - setWeightFactor, 583
 - solve, 582
 - solve2, 585
- Idiot.hpp

- OsiSolverInterface, 720
- IdiotResult, 585
 - djAtBeginning, 586
 - djAtEnd, 586
 - dropThis, 586
 - infeas, 586
 - iteration, 586
 - objval, 586
 - sumSquared, 586
 - weighted, 586
- inSmall
 - ClpDynamicMatrix, 211
 - ClpGubDynamicMatrix, 237
- in_
 - ClpSimplexProgress, 510
- inCbcBranchAndBound
 - ClpModel, 336
- inOutUseful
 - AbcMatrix, 34
- inTrouble
 - OsiClpDisasterHandler, 596
- inTrouble_
 - OsiClpDisasterHandler, 596
- incomingInfeasibility_
 - AbcTolerancesEtc, 125
 - ClpSimplex, 487
- increaseInObjective
 - dualColumnResult, 579
- increaseInThis
 - dualColumnResult, 579
- incrementReallyBadTimes
 - ClpSimplexProgress, 509
- incrementTimesFlagged
 - ClpSimplexProgress, 509
- index
 - ClpHashValue, 257
 - ClpHashValue::CoinHashLink, 578
- indexColumnL
 - CoinAbcTypeFactorization, 552
- indexColumnL_
 - CoinAbcTypeFactorization, 573
- indexColumnLAddress_
 - CoinAbcTypeFactorization, 568
- indexColumnU_
 - CoinAbcTypeFactorization, 572
- indexColumnUAddress_
 - CoinAbcTypeFactorization, 566
- indexRowL
 - CoinAbcTypeFactorization, 552
- indexRowL_
 - CoinAbcTypeFactorization, 573
- indexRowLAddress_
 - CoinAbcTypeFactorization, 567
- indexRowRAddress_
 - CoinAbcTypeFactorization, 568
- indexRowU
 - CoinAbcTypeFactorization, 555
- indexRowU_
 - CoinAbcTypeFactorization, 572
- indexRowUAddress_
 - CoinAbcTypeFactorization, 566
- indexStart_
 - ClpCholeskyBase, 155
- indices
 - CoinAbcAnyFactorization, 525
 - CoinAbcDenseFactorization, 536
 - CoinAbcTypeFactorization, 551
- indices_
 - ClpNetworkMatrix, 353
 - ClpPlusMinusOneMatrix, 409
- infeas
 - IdiotResult, 586
- infeasibility_
 - ClpSimplexProgress, 509
- infeasibilityCost
 - ClpSimplex, 461
- infeasibilityCost_
 - AbcTolerancesEtc, 125
 - ClpDataSave, 182
 - ClpSimplex, 482
- infeasibilityRay
 - ClpModel, 331
 - ClpSimplex, 475
- infeasibilityWeight_
 - ClpDynamicMatrix, 219
 - ClpGubMatrix, 253
- infeasible
 - AbcDualRowSteepest, 26
 - ClpNonLinearCost, 371
- infeasibleReturn
 - ClpSolve, 514
- Info, 586
 - atolmin, 587
 - deltay, 587
 - LSdamp, 587
 - r3norm, 587
- initialBarrierNoCrossSolve
 - ClpSimplex, 455
- initialBarrierSolve
 - ClpSimplex, 455
- initialDenseFactorization
 - AbcSimplex, 86
 - ClpSimplex, 470
- initialDualSolve
 - ClpSimplex, 455
- initialNumberInfeasibilities_
 - AbcSimplex, 97
- initialNumberRows_

- CoinAbcTypeFactorization, [577](#)
- initialPrimalSolve
 - ClpSimplex, [455](#)
- initialProblem
 - ClpDynamicMatrix, [213](#)
- initialSolve
 - ClpSimplex, [455](#)
 - OsiClpSolverInterface, [609](#)
- initialSumInfeasibilities_
 - AbcSimplex, [94](#)
- initialWeight_
 - ClpSimplexProgress, [509](#)
- initializeWeights
 - AbcPrimalColumnSteepest, [60](#)
 - ClpPrimalColumnSteepest, [428](#)
- innerProduct
 - ClpHelperFunctions.hpp, [682](#)
- insertNonBasic
 - ClpGubDynamicMatrix, [238](#)
- instrument_add
 - CoinAbcCommon.hpp, [704](#)
- instrument_do
 - CoinAbcCommon.hpp, [704](#)
- instrument_end
 - CoinAbcCommon.hpp, [704](#)
- instrument_end_and_adjust
 - CoinAbcCommon.hpp, [704](#)
- instrument_start
 - CoinAbcCommon.hpp, [704](#)
- intParam_
 - ClpModel, [340](#)
- intParameter
 - CbcOrClpParam, [143](#), [144](#)
- intValue
 - CbcOrClpParam, [145](#)
- intWorkArea
 - CoinAbcAnyFactorization, [524](#)
- integerIncrement_
 - ClpNodeStuff, [363](#)
- integerInformation
 - ClpModel, [330](#)
- integerInformation_
 - OsiClpSolverInterface, [631](#)
- integerParameters_
 - ClpCholeskyBase, [156](#)
 - ClpCholeskyDenseC, [161](#)
- integerTolerance_
 - ClpNodeStuff, [363](#)
- integerType_
 - ClpModel, [340](#)
- internalFactorize
 - AbcSimplex, [79](#)
 - ClpSimplex, [464](#)
- internalRay
 - ClpModel, [331](#)
- internalStatus
 - AbcSimplex, [85](#)
- internalStatus_
 - AbcSimplex, [97](#)
- internalStatusSaved_
 - AbcSimplex, [98](#)
- intoSimplex
 - ClpDisasterHandler, [184](#)
 - OsiClpDisasterHandler, [595](#)
- inverseColumnScale
 - ClpModel, [327](#)
- inverseColumnScale2
 - AbcSimplex, [82](#)
- inverseColumnScale_
 - ClpModel, [339](#)
- inverseColumnUseScale_
 - AbcSimplex, [97](#)
- inverseRowScale
 - ClpModel, [327](#)
- inverseRowScale2
 - AbcSimplex, [82](#)
- inverseRowScale_
 - ClpModel, [339](#)
- inverseRowUseScale_
 - AbcSimplex.hpp, [645](#)
- isFixed
 - AbcSimplex, [74](#)
 - ClpSimplex, [452](#)
- isFree
 - AbcSimplex, [74](#)
 - ClpSimplex, [452](#)
- isAbandoned
 - ClpModel, [323](#)
 - OsiClpSolverInterface, [612](#)
- isBinary
 - OsiClpSolverInterface, [615](#)
- isColOrdered
 - AbcMatrix, [32](#)
 - ClpDummyMatrix, [197](#)
 - ClpMatrixBase, [291](#)
 - ClpNetworkMatrix, [348](#)
 - ClpPackedMatrix, [380](#)
 - ClpPlusMinusOneMatrix, [403](#)
- isColumn
 - AbcSimplex, [86](#)
 - ClpInterior, [270](#)
 - ClpSimplex, [471](#)
- isContinuous
 - OsiClpSolverInterface, [615](#)
- isDenseOrSmall
 - ClpFactorization, [233](#)
- isDualObjectiveLimitReached
 - ClpModel, [323](#)

- OsiClpSolverInterface, [612](#)
- isFreeBinary
 - OsiClpSolverInterface, [615](#)
- isInteger
 - ClpModel, [317](#)
 - OsiClpSolverInterface, [615](#)
- isIntegerNonBinary
 - OsiClpSolverInterface, [615](#)
- isIterationLimitReached
 - ClpModel, [323](#)
 - OsiClpSolverInterface, [612](#)
- isObjectiveLimitTestValid
 - AbcSimplex, [77](#)
 - ClpSimplex, [461](#)
- isOptionalInteger
 - OsiClpSolverInterface, [615](#)
- isPrimalObjectiveLimitReached
 - ClpModel, [323](#)
 - OsiClpSolverInterface, [612](#)
- isProvenDualInfeasible
 - ClpModel, [323](#)
 - OsiClpSolverInterface, [612](#)
- isProvenOptimal
 - ClpModel, [323](#)
 - OsiClpSolverInterface, [612](#)
- isProvenPrimalInfeasible
 - ClpModel, [323](#)
 - OsiClpSolverInterface, [612](#)
- istop
 - Outfo, [634](#)
- iteration
 - IdiotResult, [586](#)
- iterationNumber_
 - ClpSimplexProgress, [510](#)
 - MyMessageHandler, [591](#)
- itlimOrig_
 - OsiClpSolverInterface, [631](#)
- itnccg
 - Outfo, [634](#)
- justDevex
 - AbcPrimalColumnSteepest, [59](#)
 - ClpPrimalColumnSteepest, [428](#)
- justDjs
 - AbcPrimalColumnSteepest, [59](#)
 - ClpPrimalColumnSteepest, [427](#)
- justSteepest
 - ClpPrimalColumnSteepest, [428](#)
- keep
 - AbcDualRowSteepest, [24](#)
 - AbcPrimalColumnSteepest, [59](#)
 - ClpDualRowSteepest, [192](#)
 - ClpPrimalColumnSteepest, [426](#)
- keyValue
 - ClpDynamicMatrix, [212](#)
- keyVariable
 - ClpDynamicMatrix, [216](#)
 - ClpGubMatrix, [253](#)
- keyVariable_
 - ClpDynamicMatrix, [218](#)
 - ClpGubMatrix, [254](#)
- kkt
 - ClpCholeskyBase, [152](#)
- LARGE_SET
 - CoinAbcBaseFactorization.hpp, [701](#)
- LARGE_UNSET
 - CoinAbcBaseFactorization.hpp, [701](#)
- LENGTH_HISTORY
 - ClpInterior.hpp, [683](#)
- LSQRMaxIter
 - Options, [592](#)
- LSQRatol1
 - Options, [592](#)
- LSQRatol2
 - Options, [593](#)
- LSQRconlim
 - Options, [593](#)
- LSdamp
 - Info, [587](#)
- LSmethod
 - Options, [592](#)
- LSproblem
 - Options, [592](#)
- large_
 - ClpNodeStuff, [364](#)
- largeValue
 - ClpSimplex, [466](#)
- largeValue_
 - AbcTolerancesEtc, [125](#)
 - ClpSimplex, [480](#)
- largestAway
 - OsiClpSolverInterface, [629](#)
- largestAway_
 - OsiClpSolverInterface, [630](#)
- largestDualError
 - ClpInterior, [268](#)
 - ClpSimplex, [466](#)
- largestDualError_
 - ClpInterior, [272](#)
 - ClpSimplex, [480](#)
- largestGap_
 - AbcSimplex, [93](#)
- largestInfeasibility
 - AbcNonLinearCost, [50](#)
 - ClpNonLinearCost, [370](#)
- largestPrimalError
 - ClpInterior, [268](#)

- ClpSimplex, 466
- largestPrimalError_
 - ClpInterior, 272
 - ClpSimplex, 480
- lastAlgorithm
 - OsiClpSolverInterface, 624
- lastAlgorithm_
 - OsiClpSolverInterface, 631
- lastBadIteration
 - ClpSimplex, 475
- lastBadIteration_
 - ClpSimplex, 487
- lastBasis_
 - AbcWarmStartOrganizer, 133
- lastCleaned_
 - AbcSimplex, 96
- lastColumn_
 - CoinAbcTypeFactorization, 571
- lastColumnAddress_
 - CoinAbcTypeFactorization, 567
- lastCount
 - CoinAbcTypeFactorization, 553
- lastCountAddress_
 - CoinAbcTypeFactorization, 567
- lastDualBound_
 - AbcSimplex, 93
- lastDualError_
 - AbcSimplex, 94
- lastDynamic
 - ClpDynamicMatrix, 216
 - ClpGubDynamicMatrix, 241
- lastDynamic_
 - ClpDynamicMatrix, 219
 - ClpGubDynamicMatrix, 243
- lastEntryByColumnU_
 - CoinAbcTypeFactorization, 570
- lastEntryByRowU_
 - CoinAbcTypeFactorization, 570
- lastFirstFree
 - AbcSimplex, 87
- lastFirstFree_
 - AbcSimplex, 95
- lastFlaggedIteration_
 - ClpSimplex, 487
- lastGoodIteration_
 - ClpSimplex, 484
- lastGradient_
 - ClpConstraint, 174
- lastGub_
 - ClpGubMatrix, 256
- lastInfeasibility
 - ClpSimplexProgress, 508
- lastIterationNumber
 - ClpSimplexProgress, 508
- lastNumberRows_
 - OsiClpSolverInterface, 633
- lastObjective
 - ClpSimplexProgress, 508
- lastPivotRow
 - AbcSimplex, 86
- lastPivotRow_
 - AbcSimplex, 97
- lastPivotValue
 - dualColumnResult, 579
- lastPrimalError_
 - AbcSimplex, 94
- lastRefresh
 - ClpMatrixBase, 299
- lastRefresh_
 - ClpMatrixBase, 301
- lastRow_
 - CoinAbcTypeFactorization, 572
- lastRowAddress_
 - CoinAbcTypeFactorization, 567
- lastSequence
 - dualColumnResult, 579
- lastSlack_
 - CoinAbcTypeFactorization, 574
- lastUsed_
 - ClpHashValue, 258
- leadingDimension_
 - CoinAbcTypeFactorization, 571
- lengthAreaL
 - CoinAbcTypeFactorization, 554
- lengthAreaL_
 - CoinAbcTypeFactorization, 570
- lengthAreaR_
 - CoinAbcTypeFactorization, 569
- lengthAreaU
 - CoinAbcTypeFactorization, 554
- lengthAreaU_
 - CoinAbcTypeFactorization, 570
- lengthExtraInformation_
 - AbcWarmStart, 130
- lengthL_
 - CoinAbcTypeFactorization, 570
- lengthMatchName
 - CbcOrClpParam, 144
- lengthNames
 - ClpModel, 334
- lengthNames_
 - ClpModel, 341
- lengthR_
 - CoinAbcTypeFactorization, 569
- lengthU_
 - CoinAbcTypeFactorization, 570
- lengths_
 - ClpNetworkMatrix, 352

- ClpPlusMinusOneMatrix, 409
- lexSolve
 - AbcSimplexPrimal, 123
 - ClpSimplexPrimal, 505
 - OsiClpSolverInterface, 629
- linearObjective
 - ClpQuadraticObjective, 435
- linearObjective_
 - OsiClpSolverInterface, 632
- linearPerturbation
 - ClpInterior, 267
- linearPerturbation_
 - ClpInterior, 275
- link_
 - ClpCholeskyBase, 156
- listAddress_
 - CoinAbcTypeFactorization, 567
- listTransposeTimes
 - ClpMatrixBase, 298
- loadFromCoinModel
 - OsiClpSolverInterface, 623
- loadNonLinear
 - ClpSimplex, 455
- loadProblem
 - ClpInterior, 266
 - ClpModel, 315, 316
 - ClpSimplex, 454, 456
 - OsiClpSolverInterface, 621, 622
- loadQuadraticObjective
 - ClpModel, 316
 - ClpQuadraticObjective, 435
- logLevel
 - ampl_info, 137
 - ClpModel, 333
- longDouble
 - ClpCholeskyBase.hpp, 675
- lookBothWays
 - ClpNonLinearCost, 371
- looksEndInDual
 - ClpEventHandler, 223
- looksEndInPrimal
 - ClpEventHandler, 223
- looksOptimal
 - AbcDualRowPivot, 21
 - AbcDualRowSteepest, 25
 - AbcPrimalColumnPivot, 55
 - AbcPrimalColumnSteepest, 60
 - ClpDualRowPivot, 189
 - ClpDualRowSteepest, 193
 - ClpPrimalColumnPivot, 423
 - ClpPrimalColumnSteepest, 428
- looksOptimal_
 - AbcPrimalColumnPivot, 56
 - ClpPrimalColumnPivot, 424
- looping
 - ClpSimplexProgress, 508
- lower
 - AbcSimplex, 87
 - ClpGubMatrix, 252
 - ClpNonLinearCost, 370
 - ClpSimplex, 472
- lowerFake
 - AbcSimplex, 74
 - ClpSimplex, 452
- lower_
 - ClpGubMatrix, 254
 - ClpInterior, 273
 - ClpNode, 359
 - ClpSimplex, 483
- lowerActive
 - ClpSimplexOther::parametricsData, 635
- lowerAddress
 - AbcSimplex, 87
 - ClpSimplex, 472
- lowerBasic
 - AbcSimplex, 85
- lowerBasic_
 - AbcSimplex, 99
- lowerBound
 - ClpInterior, 271
- lowerChange
 - ClpSimplexOther::parametricsData, 635
- lowerCoefficient
 - ClpSimplexOther::parametricsData, 636
- lowerColumn
 - ClpGubDynamicMatrix, 240
- lowerColumn_
 - ClpGubDynamicMatrix, 242
- lowerGap
 - ClpSimplexOther::parametricsData, 635
- lowerIn_
 - AbcSimplexPrimal::pivotStruct, 637
 - ClpSimplex, 481
- lowerList
 - ClpSimplexOther::parametricsData, 635
- lowerOut_
 - AbcSimplexPrimal::pivotStruct, 637
 - ClpSimplex, 481
- lowerRegion
 - AbcSimplex, 84, 85
 - ClpSimplex, 469, 470
- lowerSaved_
 - AbcSimplex, 99
- lowerSet
 - ClpDynamicMatrix, 216
 - ClpGubDynamicMatrix, 240
- lowerSet_
 - ClpDynamicMatrix, 218

- ClpGubDynamicMatrix, [242](#)
- lowerSlack_
 - ClpInterior, [277](#)
- lsqr
 - ClpPdco, [396](#)
- lsqrObject_
 - ClpInterior, [274](#)
- MATRIX_SAME
 - ClpModel.hpp, [689](#)
- makeAllUseful
 - AbcMatrix, [34](#)
- makeBaseModel
 - AbcSimplex, [75](#)
 - ClpSimplex, [453](#)
- makeNonFreeVariablesDualFeasible
 - AbcSimplexDual, [108](#)
- makeNonSingular
 - CoinAbcAnyFactorization, [526](#)
 - CoinAbcDenseFactorization, [533](#)
 - CoinAbcTypeFactorization, [560](#)
- makeSpecialColumnCopy
 - ClpPackedMatrix, [387](#)
- markDone
 - ClpSimplexOther::parametricsData, [635](#)
- markHotStart
 - OsiClpSolverInterface, [613](#)
- markNonlinear
 - ClpConstraint, [173](#)
 - ClpConstraintLinear, [176](#)
 - ClpConstraintQuadratic, [179](#)
 - ClpObjective, [374](#)
 - ClpQuadraticObjective, [434](#)
- markNonzero
 - ClpConstraint, [173](#)
 - ClpConstraintLinear, [177](#)
 - ClpConstraintQuadratic, [180](#)
- markRow_
 - CoinAbcTypeFactorization, [572](#)
- markRowAddress_
 - CoinAbcTypeFactorization, [567](#)
- matPrecon
 - ClpPdco, [396](#)
 - ClpPdcoBase, [398](#)
- matVecMult
 - ClpLsqr, [285](#)
 - ClpPdco, [396](#)
 - ClpPdcoBase, [398](#)
- matchName
 - CbcOrClpParam, [144](#)
- matches
 - CbcOrClpParam, [146](#)
- matrix
 - AbcMatrix, [37](#)
- ClpModel, [329](#)
- ClpPackedMatrix, [387](#)
- matrix_
 - AbcMatrix, [39](#)
 - ClpModel, [339](#)
 - ClpNetworkMatrix, [352](#)
 - ClpPackedMatrix, [389](#)
 - ClpPlusMinusOneMatrix, [408](#)
- matrixByRow_
 - OsiClpSolverInterface, [631](#)
- matrixByRowAtContinuous_
 - OsiClpSolverInterface, [631](#)
- maxHash_
 - ClpHashValue, [258](#)
- MaxIter
 - Options, [592](#)
- maxTheta
 - ClpSimplexOther::parametricsData, [635](#)
- maximumAbcNumberColumns_
 - AbcSimplex, [95](#)
- maximumAbcNumberRows
 - AbcSimplex, [77](#)
- maximumAbcNumberRows_
 - AbcSimplex, [95](#)
- maximumAbsElement
 - ClpHelperFunctions.hpp, [682](#)
- maximumBarrierIterations
 - ClpInterior, [269](#)
- maximumBarrierIterations_
 - ClpInterior, [279](#)
- maximumBasic
 - ClpSimplex, [476](#)
- maximumBasic_
 - ClpSimplex, [487](#)
- maximumBoundInfeasibility_
 - ClpInterior, [275](#)
- maximumCoefficient
 - CoinAbcDenseFactorization, [534](#)
 - CoinAbcTypeFactorization, [554](#)
- maximumColumns_
 - ClpModel, [342](#)
 - ClpNode, [360](#)
- maximumDualError_
 - ClpInterior, [275](#)
- maximumElements_
 - ClpDynamicMatrix, [220](#)
- maximumFixed_
 - ClpNode, [360](#)
- maximumGubColumns_
 - ClpDynamicMatrix, [220](#)
- maximumIntegers_
 - ClpNode, [360](#)
- maximumInternalColumns_
 - ClpModel, [342](#)

- maximumInternalRows_
 - ClpModel, [342](#)
- maximumIterations
 - Clp_C_Interface.h, [667](#)
 - ClpModel, [322](#)
- maximumMaximumPivots_
 - CoinAbcTypeFactorization, [576](#)
- maximumNodes
 - ClpNodeStuff, [363](#)
- maximumNodes_
 - ClpNodeStuff, [365](#)
- maximumNumberTotal
 - AbcSimplex, [77](#)
- maximumNumberTotal_
 - AbcSimplex, [95](#)
- maximumPerturbationSize_
 - ClpSimplex, [488](#)
- maximumPivots
 - AbcSimplexFactorization, [114](#)
 - ClpFactorization, [229](#)
 - CoinAbcAnyFactorization, [523](#)
 - CoinAbcTypeFactorization, [554](#)
- maximumPivots_
 - AbcTolerancesEtc, [126](#)
 - CoinAbcAnyFactorization, [529](#)
- maximumPivotsChanged
 - AbcPrimalColumnPivot, [56](#)
 - AbcPrimalColumnSteepest, [60](#)
 - ClpDualRowPivot, [189](#)
 - ClpDualRowSteepest, [193](#)
 - ClpPrimalColumnPivot, [423](#)
 - ClpPrimalColumnSteepest, [429](#)
- maximumRHSCheck_
 - ClpInterior, [276](#)
- maximumRHSError_
 - ClpInterior, [275](#)
- maximumRows_
 - ClpModel, [342](#)
 - ClpNode, [360](#)
 - CoinAbcAnyFactorization, [529](#)
 - CoinAbcTypeFactorization, [575](#)
- maximumRowsAdjusted_
 - CoinAbcDenseFactorization, [536](#)
- maximumRowsExtra
 - CoinAbcTypeFactorization, [553](#)
- maximumRowsExtra_
 - CoinAbcTypeFactorization, [569](#)
- maximumSeconds
 - ClpModel, [322](#)
- maximumSpace
 - ClpNodeStuff, [363](#)
- maximumSpace_
 - CoinAbcDenseFactorization, [536](#)
- maximumTotal
 - AbcSimplex, [77](#)
- maximumU_
 - CoinAbcTypeFactorization, [570](#)
- messageHandler
 - ClpModel, [332](#)
- messageLevel
 - ClpFactorization, [231](#)
 - CoinAbcTypeFactorization, [553](#), [554](#)
- messageLevel_
 - CoinAbcTypeFactorization, [574](#)
- messages
 - ClpModel, [333](#)
- messages_
 - ClpModel, [341](#)
- messagesPointer
 - ClpModel, [333](#)
- miniPostsolve
 - ClpSimplex, [458](#)
- miniPresolve
 - ClpSimplex, [457](#)
- minimizationObjectiveValue
 - AbcSimplex, [78](#)
- minimumGoodReducedCosts
 - AbcMatrix, [37](#)
 - ClpMatrixBase, [299](#)
- minimumGoodReducedCosts_
 - AbcMatrix, [41](#)
 - ClpMatrixBase, [302](#)
- minimumObjectsScan
 - AbcMatrix, [37](#)
 - ClpMatrixBase, [299](#)
- minimumObjectsScan_
 - AbcMatrix, [41](#)
 - ClpMatrixBase, [302](#)
- minimumPivotTolerance
 - AbcSimplexFactorization, [116](#)
 - CoinAbcAnyFactorization, [523](#)
- minimumPivotTolerance_
 - CoinAbcAnyFactorization, [528](#)
- minimumThetaMovement_
 - AbcSimplex, [94](#)
- mode
 - AbcDualRowSteepest, [26](#)
 - AbcPrimalColumnSteepest, [60](#)
 - ClpDualRowSteepest, [194](#)
 - ClpPrimalColumnSteepest, [429](#)
- model
 - AbcDualRowPivot, [21](#)
 - AbcDualRowSteepest, [26](#)
 - AbcPrimalColumnPivot, [56](#)
 - AbcWarmStart, [130](#)
 - ClpDualRowPivot, [190](#)
 - ClpPresolve, [414](#)
 - ClpPrimalColumnPivot, [423](#)

- MyMessageHandler, 590
- model_
 - AbcDualRowPivot, 22
 - AbcMatrix, 39
 - AbcPrimalColumnPivot, 56
 - AbcWarmStart, 131
 - AbcWarmStartOrganizer, 133
 - ClpCholeskyBase, 154
 - ClpDisasterHandler, 185
 - ClpDualRowPivot, 190
 - ClpDynamicMatrix, 218
 - ClpEventHandler, 224
 - ClpGubMatrix, 255
 - ClpLsqr, 285
 - ClpPrimalColumnPivot, 424
 - ClpSimplexProgress, 510
 - MyMessageHandler, 591
- modelPtr_
 - OsiClpSolverInterface, 629
- modifyMatrixInMiniPostsolve
 - ClpEventHandler, 223
- modifyMatrixInMiniPresolve
 - ClpEventHandler, 223
- modifyCoefficient
 - ClpMatrixBase, 292
 - ClpModel, 318
 - ClpPackedMatrix, 382
 - OsiClpSolverInterface, 621
- modifyCoefficientsAndPivot
 - ClpSimplex, 457
- modifyCosts
 - dualColumnResult, 580
- modifyLink
 - CoinAbcTypeFactorization, 562
- modifyObjective
 - ClpSimplexProgress, 508
- modifyOffset
 - ClpDynamicMatrix, 212
- moreMiniPresolve
 - ClpEventHandler, 223
- moreSpecialOptions
 - ClpSimplex, 473
- moreSpecialOptions_
 - ClpSimplex, 480
- moveInfo
 - AbcSimplex, 91
 - ClpSimplex, 476
- moveLargestToStart
 - AbcMatrix, 34
- moveStatusFromClp
 - AbcSimplex, 82
- moveStatusToClp
 - AbcSimplex, 82
- moveToBasic
 - AbcSimplex, 84
- moveTowardsPrimalFeasible
 - ClpSimplex, 457
- movement_
 - AbcSimplex, 94
- mu0
 - Options, 592
- mu_
 - ClpInterior, 274
- multipleSequenceIn_
 - AbcSimplex, 101
- multiplyAdd
 - ClpHelperFunctions.hpp, 682
- mutableColumnScale
 - ClpModel, 327
- mutableInverseColumnScale
 - ClpModel, 328
- mutableInverseRowScale
 - ClpModel, 327
- mutableRandomNumberGenerator
 - ClpModel, 333
- mutableRowScale
 - ClpModel, 327
- MyEventHandler, 587
 - ~MyEventHandler, 588
 - clone, 588
 - event, 588
 - MyEventHandler, 588
 - MyEventHandler, 588
 - operator=, 588
- MyMessageHandler, 589
 - ~MyMessageHandler, 590
 - clearFeasibleExtremePoints, 590
 - clone, 591
 - feasibleExtremePoints_, 591
 - getFeasibleExtremePoints, 590
 - iterationNumber_, 591
 - model, 590
 - model_, 591
 - MyMessageHandler, 590
 - MyMessageHandler, 590
 - operator=, 590
 - print, 590
 - setModel, 590
- MyMessageHandler.hpp
 - StdVectorDouble, 721
- n
 - ClpCholeskyDenseC, 161
- nBound_
 - ClpNodeStuff, 364
- nDepth_
 - ClpNodeStuff, 365
- NEED_BASIS_SORT

- AbcSimplex.hpp, 646
- NEW_CHUNK_SIZE
 - CoinAbcHelperFunctions.hpp, 714
- nNodes_
 - ClpNodeStuff, 365
- NUMBER_ROW_BLOCKS
 - AbcMatrix.hpp, 640
- NUMBER_THREADS
 - AbcSimplex.hpp, 645
- name
 - CbcOrClpParam, 143
- ncols_
 - ClpLsq, 285
- nearest
 - AbcNonLinearCost, 49
 - ClpNonLinearCost, 369
- needToReorder
 - AbcSimplexFactorization, 117
 - ClpFactorization, 233
- networkBasis
 - ClpFactorization, 233
- newLanguage
 - ClpModel, 332
 - OsiClpSolverInterface, 624
- newOddState
 - ClpSimplexProgress, 508
- newXValues
 - ClpConstraint, 174
 - ClpObjective, 374
- next
 - ClpHashValue::CoinHashLink, 578
 - CoinAbcStack, 537
- next_
 - ClpDynamicMatrix, 220
 - ClpGubMatrix, 254
- nextBasis_
 - AbcWarmStart, 131
- nextColumn_
 - CoinAbcTypeFactorization, 571
- nextColumnAddress_
 - CoinAbcTypeFactorization, 567
- nextCount
 - CoinAbcTypeFactorization, 553
- nextCountAddress_
 - CoinAbcTypeFactorization, 567
- nextRow_
 - CoinAbcTypeFactorization, 572
- nextRowAddress_
 - CoinAbcTypeFactorization, 567
- nextSuperBasic
 - AbcSimplexDual, 108
 - AbcSimplexPrimal, 123
 - ClpSimplexDual, 494
 - ClpSimplexPrimal, 505
- noCandidateInDual
 - ClpEventHandler, 223
- noCandidateInPrimal
 - ClpEventHandler, 223
- noFake
 - AbcSimplex, 74
 - ClpSimplex, 452
- noTheta
 - ClpEventHandler, 223
- noCheck_
 - ClpDynamicMatrix, 219
 - ClpGubMatrix, 255
- noPivotColumn
 - AbcSimplexDual, 106
- noPivotRow
 - AbcSimplexDual, 106
- node
 - ClpEventHandler, 223
- nodeCalled_
 - ClpNodeStuff, 366
- nodeInfo_
 - ClpNodeStuff, 364
- nonLinear
 - ampl_info, 137
- nonLinearCost
 - ClpSimplex, 473
- nonLinearCost_
 - ClpSimplex, 486
- nonLinearValue
 - ClpPresolve, 415
- nonlinearOffset
 - ClpObjective, 375
- nonlinearSLP
 - ClpSimplex, 456
- normal
 - AbcDualRowSteepest, 24
 - AbcPrimalColumnSteepest, 59
 - ClpDualRowSteepest, 192
 - ClpPrimalColumnSteepest, 426
- normalDualColumnIteration_
 - AbcSimplex, 95
- notImplemented
 - ClpSolve, 513
- notOwned_
 - OsiClpSolverInterface, 631
- nrows_
 - ClpLsq, 285
- number
 - scatterStruct, 638
- numberActiveColumns
 - ClpPackedMatrix, 388
- numberActiveColumns_
 - ClpPackedMatrix, 389
- numberActiveSets_

- ClpDynamicMatrix, 218
- numberArguments
 - ampl_info, 135
- numberAtFakeBound
 - AbcSimplexDual, 108
 - ClpSimplexDual, 494
- numberBadTimes_
 - ClpSimplexProgress, 510
- numberBases_
 - AbcWarmStartOrganizer, 133
- numberBeforeTrust_
 - ClpNodeStuff, 365
- numberBinary
 - ampl_info, 135
- numberBlocks
 - ClpCholeskyDenseC, 161
- numberBlocks_
 - AbcMatrix2, 43
 - AbcMatrix3, 45
 - ClpPackedMatrix2, 391
 - ClpPackedMatrix3, 394
- numberBtranCounts_
 - CoinAbcTypeFactorization, 576
- numberBtranFullCounts_
 - CoinAbcTypeFactorization, 577
- numberChanged_
 - ClpSimplex, 487
- numberCoefficients
 - ClpConstraint, 173
 - ClpConstraintLinear, 177
 - ClpConstraintQuadratic, 180
- numberColumnBlocks
 - AbcMatrix, 39
- numberColumnBlocks_
 - AbcMatrix, 40
- numberColumns
 - ampl_info, 135
 - ClpConstraintLinear, 177
 - ClpConstraintQuadratic, 180
 - ClpDynamicExampleMatrix, 204
 - ClpModel, 320
 - ClpQuadraticObjective, 435
- numberColumns_
 - AbcMatrix3, 45
 - ClpDummyMatrix, 200
 - ClpDynamicExampleMatrix, 205
 - ClpModel, 338
 - ClpNetworkMatrix, 353
 - ClpPackedMatrix3, 394
 - ClpPlusMinusOneMatrix, 409
- numberComplementarityItems_
 - ClpInterior, 279
- numberComplementarityPairs_
 - ClpInterior, 279
- numberCompressions
 - CoinAbcTypeFactorization, 555
- numberCompressions_
 - CoinAbcTypeFactorization, 574
- numberCounts_
 - CoinAbcStatistics, 538
- numberDense
 - AbcSimplexFactorization, 115
 - ClpFactorization, 230
 - CoinAbcAnyFactorization, 522
- numberDense_
 - CoinAbcAnyFactorization, 529
- numberDisasters_
 - AbcSimplex, 101
- numberDown_
 - ClpNodeStuff, 363
- numberDownInfeasible_
 - ClpNodeStuff, 364
- numberDualInfeasibilities
 - ClpSimplex, 462
- numberDualInfeasibilities_
 - ClpDynamicMatrix, 219
 - ClpGubMatrix, 255
 - ClpSimplex, 485
- numberDualInfeasibilitiesWithoutFree
 - ClpSimplex, 462
- numberDualInfeasibilitiesWithoutFree_
 - ClpSimplex, 485
- numberElements
 - AbcSimplexFactorization, 114
 - ampl_info, 135
 - ClpDynamicMatrix, 216
 - ClpFactorization, 229
 - ClpGubDynamicMatrix, 241
 - CoinAbcAnyFactorization, 525
 - CoinAbcDenseFactorization, 533
 - CoinAbcTypeFactorization, 553
- numberElements_
 - blockStruct, 138
 - blockStruct3, 139
 - ClpDummyMatrix, 200
 - ClpDynamicMatrix, 219
 - ClpGubDynamicMatrix, 243
- numberElementsL
 - ClpFactorization, 230
 - CoinAbcTypeFactorization, 554
- numberElementsR
 - ClpFactorization, 231
 - CoinAbcTypeFactorization, 555
- numberElementsU
 - ClpFactorization, 230
 - CoinAbcTypeFactorization, 554
- numberEntries
 - ClpHashValue, 258

- numberExtendedColumns
 - ClpQuadraticObjective, [435](#)
- numberExtraRows
 - ClpSimplex, [476](#)
- numberExtraRows_
 - ClpSimplex, [487](#)
- numberFake_
 - ClpSimplex, [487](#)
- numberFixed
 - ClpInterior, [269](#)
- numberFixed_
 - ClpNode, [359](#)
- numberFlagged_
 - AbcSimplex, [95](#)
- numberFlipped_
 - AbcSimplex, [101](#)
- numberForrestTomlin
 - CoinAbcTypeFactorization, [553](#)
- numberFreeNonBasic_
 - AbcSimplex, [96](#)
- numberFtranCounts_
 - CoinAbcTypeFactorization, [575](#)
- numberFtranFTCounts_
 - CoinAbcTypeFactorization, [575](#)
- numberFtranFullCounts_
 - CoinAbcTypeFactorization, [576](#)
- numberGoodColumns
 - CoinAbcAnyFactorization, [523](#)
- numberGoodL_
 - CoinAbcTypeFactorization, [569](#)
- numberGoodU_
 - CoinAbcAnyFactorization, [529](#)
- numberGubColumns
 - ClpDynamicMatrix, [216](#)
 - ClpGubDynamicMatrix, [240](#)
- numberGubColumns_
 - ClpDynamicMatrix, [220](#)
 - ClpGubDynamicMatrix, [242](#)
- numberGubEntries
 - ClpDynamicMatrix, [214](#)
- numberHash_
 - ClpHashValue, [258](#)
- numberInBlock_
 - blockStruct, [138](#)
 - blockStruct3, [139](#)
- numberInColumn
 - CoinAbcAnyFactorization, [524](#)
 - CoinAbcTypeFactorization, [555](#)
- numberInColumn_
 - CoinAbcTypeFactorization, [571](#)
- numberInColumnAddress_
 - CoinAbcTypeFactorization, [566](#)
- numberInColumnPlus_
 - CoinAbcTypeFactorization, [571](#)
- numberInColumnPlusAddress_
 - CoinAbcTypeFactorization, [566](#)
- numberInRow
 - CoinAbcAnyFactorization, [524](#)
 - CoinAbcTypeFactorization, [555](#)
- numberInRow_
 - CoinAbcTypeFactorization, [571](#)
- numberInRowAddress_
 - CoinAbcTypeFactorization, [566](#)
- numberInfeasibilities
 - AbcNonLinearCost, [50](#)
 - ClpNode, [357](#)
 - ClpNonLinearCost, [370](#)
- numberInfeasibilities_
 - ClpNode, [359](#)
 - ClpSimplexProgress, [510](#)
- numberIntegers
 - ampl_info, [135](#)
- numberIterations
 - ClpModel, [321](#)
- numberIterations_
 - ClpModel, [340](#)
 - ClpNodeStuff, [365](#)
- numberL
 - CoinAbcTypeFactorization, [553](#)
- numberL_
 - CoinAbcTypeFactorization, [569](#)
- numberLastSwapped
 - dualColumnResult, [580](#)
- numberNodesExplored_
 - ClpNodeStuff, [365](#)
- numberOrdinary
 - AbcSimplex, [78](#)
- numberOrdinary_
 - AbcSimplex, [96](#)
- numberPivots_
 - CoinAbcAnyFactorization, [529](#)
- numberPrice_
 - blockStruct, [138](#)
 - blockStruct3, [139](#)
- numberPrimalInfeasibilities
 - ClpSimplex, [462](#)
- numberPrimalInfeasibilities_
 - ClpDynamicMatrix, [219](#)
 - ClpGubMatrix, [255](#)
 - ClpSimplex, [485](#)
- numberR_
 - CoinAbcTypeFactorization, [569](#)
- numberReallyBadTimes_
 - ClpSimplexProgress, [510](#)
- numberRefinements
 - ClpSimplex, [467](#)
- numberRefinements_
 - AbcTolerancesEtc, [126](#)

- ClpSimplex, 485
- numberRemaining
 - dualColumnResult, 580
- numberRowBlocks
 - AbcMatrix, 39
- numberRowBlocks_
 - AbcMatrix, 40
- numberRows
 - AbcSimplexFactorization, 116
 - ampl_info, 135
 - ClpCholeskyBase, 152
 - ClpFactorization, 231
 - ClpModel, 320
 - CoinAbcAnyFactorization, 522
- numberRows_
 - AbcMatrix2, 43
 - ClpCholeskyBase, 154
 - ClpDummyMatrix, 200
 - ClpModel, 337
 - ClpNetworkMatrix, 353
 - ClpPackedMatrix2, 391
 - ClpPlusMinusOneMatrix, 409
 - CoinAbcAnyFactorization, 529
- numberRowsDropped
 - ClpCholeskyBase, 151
- numberRowsDropped_
 - ClpCholeskyBase, 155
- numberRowsExtra
 - CoinAbcTypeFactorization, 553
- numberRowsExtra_
 - CoinAbcTypeFactorization, 568
- numberRowsLeft_
 - CoinAbcTypeFactorization, 569
- numberRowsSmall_
 - CoinAbcTypeFactorization, 569
- numberSOS
 - OsiClpSolverInterface, 619
- numberSOS_
 - OsiClpSolverInterface, 630
- numberSets
 - ClpDynamicMatrix, 214
 - ClpGubMatrix, 253
- numberSets_
 - ClpDynamicMatrix, 218
 - ClpGubMatrix, 255
- numberSlacks
 - AbcSimplexFactorization, 116
 - CoinAbcAnyFactorization, 522
- numberSlacks_
 - CoinAbcAnyFactorization, 529
- numberSos
 - ampl_info, 135
- numberSprintColumns
 - AbcPrimalColumnPivot, 56
- ClpPrimalColumnPivot, 423
- ClpPrimalColumnSteepest, 429
- numberStaticRows
 - ClpDynamicMatrix, 216
- numberStaticRows_
 - ClpDynamicMatrix, 219
- numberSwapped
 - dualColumnResult, 580
- numberThreads
 - ClpModel, 332
- numberThreads_
 - ClpModel, 341
- numberTimes_
 - ClpSimplexProgress, 510
- numberTimesFlagged_
 - ClpSimplexProgress, 510
- numberTimesOptimal_
 - ClpSimplex, 486
- numberTotal
 - AbcSimplex, 77
- numberTotal_
 - AbcSimplex, 95
- numberTotalWithoutFixed
 - AbcSimplex, 77
- numberTotalWithoutFixed_
 - AbcSimplex, 95
- numberTrials_
 - ClpCholeskyBase, 154
 - CoinAbcTypeFactorization, 570
- numberU_
 - CoinAbcTypeFactorization, 570
- numberUp_
 - ClpNodeStuff, 363
- numberUpInfeasible_
 - ClpNodeStuff, 364
- numberValidRows_
 - AbcWarmStart, 131
- OBJECTIVE_SAME
 - ClpModel.hpp, 690
- objValue
 - ampl_info, 135
- objective
 - ampl_info, 135
 - ClpModel, 328
- objective_
 - ClpModel, 338
 - ClpSimplexProgress, 509
- objectiveAsObject
 - ClpModel, 334
- objectiveChange_
 - AbcSimplex, 94
- objectiveNorm_
 - ClpInterior, 274

- objectiveOffset
 - ClpDynamicMatrix, 215
 - ClpGubDynamicMatrix, 239
 - ClpModel, 321
- objectiveOffset_
 - AbcSimplex, 93
 - ClpDynamicMatrix, 218
 - ClpGubDynamicMatrix, 241
- objectiveScale
 - ClpModel, 328
- objectiveScale_
 - ClpDataSave, 182
 - ClpModel, 337
- objectiveValue
 - ClpLinearObjective, 281
 - ClpModel, 330
 - ClpNode, 356
 - ClpObjective, 374
 - ClpQuadraticObjective, 434
- objectiveValue_
 - ClpModel, 337
 - ClpNode, 358
- objectiveWork_
 - ClpSimplex, 483
- objval
 - IdiotResult, 586
- oddArraysExist
 - ClpNode, 357
- oddState
 - ClpSimplexProgress, 508
- oddState_
 - ClpSimplexProgress, 510
- offset
 - ampl_info, 135
 - ClpConstraint, 174
 - scatterStruct, 638
- offset_
 - AbcMatrix2, 43
 - AbcSimplex, 97
 - ClpConstraint, 174
 - ClpObjective, 375
 - ClpPackedMatrix2, 391
- offsetRhs_
 - AbcSimplex, 97
- onStopped
 - ClpModel, 337
- operator=
 - AbcDualRowDantzig, 18
 - AbcDualRowPivot, 21
 - AbcDualRowSteepest, 25
 - AbcMatrix, 39
 - AbcMatrix2, 42
 - AbcMatrix3, 45
 - AbcNonLinearCost, 48
 - AbcPrimalColumnDantzig, 52
 - AbcPrimalColumnPivot, 55
 - AbcPrimalColumnSteepest, 60
 - AbcSimplex, 75
 - AbcSimplexFactorization, 112
 - AbcTolerancesEtc, 125
 - AbcWarmStart, 130
 - AbcWarmStartOrganizer, 133
 - CbcOrClpParam, 143
 - ClpCholeskyBase, 153
 - ClpCholeskyDense, 160
 - ClpCholeskyTaucs, 164
 - ClpCholeskyWssmp, 168
 - ClpCholeskyWssmpKKT, 170
 - ClpConstraint, 173
 - ClpConstraintLinear, 177
 - ClpConstraintQuadratic, 180
 - ClpDataSave, 182
 - ClpDisasterHandler, 184
 - ClpDualRowDantzig, 186
 - ClpDualRowPivot, 190
 - ClpDualRowSteepest, 193
 - ClpDummyMatrix, 200
 - ClpDynamicExampleMatrix, 203
 - ClpDynamicMatrix, 214
 - ClpEventHandler, 224
 - ClpFactorization, 228
 - ClpGubDynamicMatrix, 239
 - ClpGubMatrix, 251
 - ClpHashValue, 258
 - ClpInterior, 266
 - ClpLinearObjective, 282
 - ClpLsq, 284
 - ClpMatrixBase, 300
 - ClpModel, 315
 - ClpNetworkBasis, 344
 - ClpNetworkMatrix, 352
 - ClpNode, 358
 - ClpNodeStuff, 362
 - ClpNonLinearCost, 368
 - ClpObjective, 374
 - ClpPackedMatrix, 388
 - ClpPackedMatrix2, 391
 - ClpPackedMatrix3, 394
 - ClpPdcoBase, 399
 - ClpPlusMinusOneMatrix, 408
 - ClpPrimalColumnDantzig, 420
 - ClpPrimalColumnPivot, 423
 - ClpPrimalColumnSteepest, 429
 - ClpPrimalQuadraticDantzig, 431
 - ClpQuadraticObjective, 434
 - ClpSimplex, 453
 - ClpSimplexProgress, 507
 - ClpSolve, 513

- CoinAbcAnyFactorization, [522](#)
- CoinAbcDenseFactorization, [533](#)
- CoinAbcTypeFactorization, [551](#)
- Idiot, [582](#)
- MyEventHandler, [588](#)
- MyMessageHandler, [590](#)
- OsiClpDisasterHandler, [595](#)
- OsiClpSolverInterface, [627](#)
- OptTol
 - Options, [592](#)
- optimizationDirection
 - ClpModel, [323](#)
- optimizationDirection_
 - ClpModel, [337](#)
- Options, [591](#)
 - delta, [592](#)
 - FeaTol, [592](#)
 - gamma, [592](#)
 - LSQRMaxIter, [592](#)
 - LSQRatol1, [592](#)
 - LSQRatol2, [593](#)
 - LSQRconlim, [593](#)
 - LSmethod, [592](#)
 - LSproblem, [592](#)
 - MaxIter, [592](#)
 - mu0, [592](#)
 - OptTol, [592](#)
 - StepTol, [592](#)
 - wait, [593](#)
 - x0min, [592](#)
 - z0min, [592](#)
- order
 - ClpCholeskyBase, [150](#)
 - ClpCholeskyDense, [158](#)
 - ClpCholeskyMumps, [162](#)
 - ClpCholeskyTaucs, [164](#)
 - ClpCholeskyUfl, [166](#)
 - ClpCholeskyWssmp, [168](#)
 - ClpCholeskyWssmpKKT, [170](#)
- ordinaryVariables
 - AbcSimplex, [78](#)
- ordinaryVariables_
 - AbcSimplex, [96](#)
- organizer_
 - AbcWarmStart, [131](#)
- originalBound
 - AbcSimplexDual, [107](#)
 - ClpSimplexDual, [493](#)
- originalColumns
 - ClpPresolve, [415](#)
- originalLower
 - AbcSimplex, [88](#)
 - ClpSimplex, [472](#)
- originalModel
 - AbcSimplex, [75](#)
 - ClpPresolve, [415](#)
 - ClpSimplex, [453](#)
- originalRows
 - ClpPresolve, [415](#)
- originalStatus
 - AbcNonLinearCost.hpp, [641](#)
 - ClpNonLinearCost.hpp, [693](#)
- originalUpper
 - AbcSimplex, [88](#)
 - ClpSimplex, [473](#)
- originalWanted
 - AbcMatrix, [38](#)
 - ClpMatrixBase, [300](#)
- originalWanted_
 - AbcMatrix, [40](#)
 - ClpMatrixBase, [301](#)
- OsiClpDisasterHandler, [593](#)
 - ~OsiClpDisasterHandler, [594](#)
 - check, [595](#)
 - clone, [595](#)
 - inTrouble, [596](#)
 - inTrouble_, [596](#)
 - intoSimplex, [595](#)
 - operator=, [595](#)
 - OsiClpDisasterHandler, [594](#)
 - osiModel, [595](#)
 - osiModel_, [596](#)
 - OsiClpDisasterHandler, [594](#)
 - phase, [596](#)
 - phase_, [596](#)
 - saveInfo, [595](#)
 - setOsiModel, [595](#)
 - setPhase, [595](#)
 - setWhereFrom, [595](#)
 - typeOfDisaster, [595](#)
 - whereFrom, [595](#)
 - whereFrom_, [596](#)
- OsiClpHasNDEBUG
 - OsiClpSolverInterface.hpp, [722](#)
- OsiClpInfinity
 - OsiClpSolverInterface.hpp, [722](#)
- OsiClpSolverInterface, [596](#)
 - ~OsiClpSolverInterface, [608](#)
 - addCol, [620](#)
 - addCols, [620](#)
 - addRow, [620](#)
 - addRows, [620](#), [621](#)
 - applyColCut, [627](#)
 - applyCuts, [621](#)
 - applyRowCut, [627](#)
 - applyRowCuts, [621](#)
 - assignProblem, [622](#)
 - baseModel_, [632](#)

basis_, 631
basisIsAvailable, 609
branchAndBound, 609
canDoSimplexInterface, 609
cleanupScaling, 625
cleanupScaling_, 632
clone, 627
ClpSimplex, 479
columnActivity_, 630
columnScale_, 633
computeLargestAway, 629
continuousModel_, 633
copyEnabledStuff, 611
copyEnabledSuff, 611
crossover, 609
crunch, 628
deleteCols, 620
deleteRows, 621
deleteScaleFactors, 628
disableFactorization, 609
disableSimplexInterface, 611
disasterHandler, 626
disasterHandler_, 633
downRange, 628
dualPivotResult, 611
enableFactorization, 609
enableSimplexInterface, 611
extractSenseRhsRange, 627
factorization_, 630
fakeMinInSimplex_, 632
fakeObjective, 626
fakeObjective_, 633
fillParamMaps, 627
findIntegersAndSOS, 619
freeCachedResults, 627
freeCachedResults0, 627
freeCachedResults1, 627
generateCpp, 624
getBlvACol, 610
getBlvARow, 610
getBlvCol, 610
getBlvRow, 610
getBasics, 610
getBasis, 627, 628
getBasisDiff, 628
getBasisStatus, 609
getColLower, 614
getColName, 614
getColSolution, 616
getColType, 615
getColUpper, 614
getConstPointerToWarmStart, 612
getDbiParam, 611
getDualRays, 616
getEmptyWarmStart, 612
getInfinity, 616
getIntParam, 611
getIterationCount, 616
getMatrixByCol, 616
getMatrixByRow, 615
getModelPtr, 624
getMutableMatrixByCol, 616
getNumCols, 613
getNumElements, 613
getNumRows, 613
getObjCoefficients, 615
getObjSense, 615
getObjValue, 616
getPointerToWarmStart, 612, 613
getPrimalRays, 617
getReducedCost, 616
getReducedGradient, 610
getRightHandSide, 614
getRowActivity, 616
getRowLower, 614
getRowName, 613
getRowPrice, 616
getRowRange, 614
getRowSense, 614
getRowUpper, 615
getSizeL, 626
getSizeU, 626
getStrParam, 611
getWarmStart, 612
gutsOfDestructor, 627
initialSolve, 609
integerInformation_, 631
isAbandoned, 612
isBinary, 615
isContinuous, 615
isDualObjectiveLimitReached, 612
isFreeBinary, 615
isInteger, 615
isIntegerNonBinary, 615
isIterationLimitReached, 612
isOptionalInteger, 615
isPrimalObjectiveLimitReached, 612
isProvenDualInfeasible, 612
isProvenOptimal, 612
isProvenPrimalInfeasible, 612
itlimOrig_, 631
largestAway, 629
largestAway_, 630
lastAlgorithm, 624
lastAlgorithm_, 631
lastNumberRows_, 633
lexSolve, 629
linearObjective_, 632

[loadFromCoinModel](#), 623
[loadProblem](#), 621, 622
[markHotStart](#), 613
[matrixByRow_](#), 631
[matrixByRowAtContinuous_](#), 631
[modelPtr_](#), 629
[modifyCoefficient](#), 621
[newLanguage](#), 624
[notOwned_](#), 631
[numberSOS](#), 619
[numberSOS_](#), 630
[operator=](#), 627
[OsiClpSolverInterface](#), 608
[OsiClpSolverInterfaceUnitTest](#), 629
[OsiClpSolverInterface](#), 608
[passInDisasterHandler](#), 626
[passInMessageHandler](#), 624
[passInRanges](#), 628
[pivot](#), 611
[primalPivotResult](#), 611
[readLp](#), 623
[readMps](#), 623
[redoScaleFactors](#), 628
[releaseClp](#), 627
[replaceMatrix](#), 624
[replaceMatrixOptional](#), 624
[reset](#), 627
[resolve](#), 609
[resolveGub](#), 609
[restoreBaseModel](#), 621
[rhs_](#), 629
[rowActivity_](#), 630
[rowScale_](#), 633
[rowrange_](#), 629
[rowsense_](#), 629
[saveBaseModel](#), 621
[saveData_](#), 632
[setBasis](#), 628
[setBasisStatus](#), 610
[setCleanupScaling](#), 625
[setColBounds](#), 617
[setColLower](#), 617, 618
[setColName](#), 618
[setColSetBounds](#), 617
[setColSolution](#), 619
[setColUpper](#), 617, 618
[setColumnStatus](#), 613
[setContinuous](#), 619
[setDbIParam](#), 611
[setFakeObjective](#), 626
[setHintParam](#), 612
[setInfo](#), 619
[setInfo_](#), 630
[setIntParam](#), 611
[setInteger](#), 619
[setLanguage](#), 624
[setLargestAway](#), 629
[setLastAlgorithm](#), 624
[setLogLevel](#), 624
[setObjCoeff](#), 617
[setObjSense](#), 619
[setObjective](#), 618
[setOptionalInteger](#), 615
[setRowBounds](#), 617
[setRowLower](#), 617
[setRowName](#), 618
[setRowPrice](#), 619
[setRowSetBounds](#), 618
[setRowSetTypes](#), 618
[setRowType](#), 618
[setRowUpper](#), 617
[setSOSData](#), 629
[setSmallestChangeInCut](#), 625
[setSmallestElementInCut](#), 625
[setSolveOptions](#), 625
[setSpecialOptions](#), 624
[setSpecialOptionsMutable](#), 627
[setStrParam](#), 611
[setStuff](#), 613
[setWarmStart](#), 612
[setupForRepeatedUse](#), 626
[smallModel_](#), 630
[smallestChangeInCut](#), 625
[smallestChangeInCut_](#), 630
[smallestElementInCut](#), 625
[smallestElementInCut_](#), 630
[solveFromHotStart](#), 613
[solveOptions_](#), 632
[spareArrays_](#), 631
[specialOptions](#), 624
[specialOptions_](#), 632
[startFastDual](#), 613
[stopFastDual](#), 613
[stuff_](#), 630
[swapModelPtr](#), 624
[synchronizeModel](#), 626
[tightenBounds](#), 625
[unmarkHotStart](#), 613
[upRange](#), 628
[whichRange_](#), 632
[writeLp](#), 623
[writeMps](#), 623
[writeMpsNative](#), 623
[ws_](#), 629
[OsiClpSolverInterface.hpp](#)
[OsiClpHasNDEBUG](#), 722
[OsiClpInfinity](#), 722
[OsiClpSolverInterfaceUnitTest](#), 722

- OsiClpSolverInterfaceUnitTest
 - OsiClpSolverInterface, [629](#)
 - OsiClpSolverInterface.hpp, [722](#)
- osiModel
 - OsiClpDisasterHandler, [595](#)
- osiModel_
 - OsiClpDisasterHandler, [596](#)
- OsiSolverInterface
 - Idiot.hpp, [720](#)
- out_
 - ClpSimplexProgress, [510](#)
- outDuplicateRows
 - ClpSimplex, [457](#)
- Outfo, [633](#)
 - atolnew, [634](#)
 - atolold, [634](#)
 - istop, [634](#)
 - itnrg, [634](#)
 - r3ratio, [634](#)
- PAN
 - AbcSimplex.hpp, [644](#)
- PESSIMISTIC
 - AbcSimplex.hpp, [645](#)
- pack
 - CoinAbcTypeFactorization, [559](#)
- packDown
 - ClpDynamicExampleMatrix, [203](#)
 - ClpDynamicMatrix, [213](#)
- parameterOption
 - CbcOrClpParam, [145](#)
- parametrics
 - ClpSimplexOther, [499](#)
- parametricsObj
 - ClpSimplexOther, [499](#)
- partialPricing
 - AbcMatrix, [37](#)
 - AbcPrimalColumnSteepest, [59](#)
 - ClpDynamicExampleMatrix, [203](#)
 - ClpDynamicMatrix, [212](#)
 - ClpGubDynamicMatrix, [238](#)
 - ClpGubMatrix, [249](#)
 - ClpMatrixBase, [295](#)
 - ClpNetworkMatrix, [351](#)
 - ClpPackedMatrix, [385](#)
 - ClpPlusMinusOneMatrix, [408](#)
 - ClpPrimalColumnSteepest, [427](#)
- passInCopy
 - ClpPlusMinusOneMatrix, [408](#)
- passInDisasterHandler
 - OsiClpSolverInterface, [626](#)
- passInEventHandler
 - ClpModel, [333](#)
 - ClpSimplex, [455](#)
- passInMessageHandler
 - ClpModel, [332](#)
 - OsiClpSolverInterface, [624](#)
- passInRanges
 - OsiClpSolverInterface, [628](#)
- pdco
 - ClpInterior, [266](#)
 - ClpPdco, [396](#)
- pdcoStuff_
 - ClpInterior, [274](#)
- permanentArrays
 - ClpModel, [336](#)
- permute
 - ClpFactorization, [229](#)
 - CoinAbcAnyFactorization, [525](#)
 - CoinAbcDenseFactorization, [536](#)
 - CoinAbcTypeFactorization, [551](#)
- permute_
 - ClpCholeskyBase, [155](#)
 - CoinAbcTypeFactorization, [571](#)
- permuteAddress_
 - CoinAbcTypeFactorization, [566](#)
- permuteBack
 - CoinAbcAnyFactorization, [524](#)
- permuteBasis
 - AbcSimplex, [79](#)
- permuteIn
 - AbcSimplex, [79](#)
- permuteInverse_
 - ClpCholeskyBase, [155](#)
- permuteOut
 - AbcSimplex, [79](#)
- Persistence
 - AbcDualRowSteepest, [24](#)
 - AbcPrimalColumnSteepest, [58](#)
 - ClpDualRowSteepest, [192](#)
 - ClpPrimalColumnSteepest, [426](#)
- persistence
 - AbcDualRowSteepest, [26](#)
 - AbcPrimalColumnSteepest, [61](#)
 - ClpDualRowSteepest, [194](#)
 - ClpPrimalColumnSteepest, [429](#)
- persistenceFlag
 - ClpFactorization, [232](#)
- perturb
 - AbcSimplexDual, [108](#)
 - AbcSimplexPrimal, [123](#)
 - ClpSimplexDual, [494](#)
 - ClpSimplexPrimal, [505](#)
- perturbB
 - AbcSimplexDual, [108](#)
- perturbation
 - ClpSimplex, [461](#)
- perturbation_

- AbcTolerancesEtc, [126](#)
 - ClpDataSave, [182](#)
 - ClpSimplex, [486](#)
- perturbationArray_
 - ClpSimplex, [488](#)
- perturbationBasic_
 - AbcSimplex, [98](#)
- perturbationFactor_
 - AbcSimplex, [93](#)
- perturbationSaved
 - AbcSimplex, [78](#)
- perturbationSaved_
 - AbcSimplex, [98](#)
- phase
 - OsiClpDisasterHandler, [596](#)
- phase_
 - OsiClpDisasterHandler, [596](#)
- pivot
 - ClpSimplex, [459](#)
 - CoinAbcTypeFactorization, [564](#), [565](#)
 - OsiClpSolverInterface, [611](#)
- pivotRow
 - ClpEventHandler, [223](#)
- pivotColumn
 - AbcPrimalColumnDantzig, [52](#)
 - AbcPrimalColumnPivot, [54](#)
 - AbcPrimalColumnSteepest, [59](#)
 - ClpFactorization, [229](#)
 - ClpPrimalColumnDantzig, [419](#)
 - ClpPrimalColumnPivot, [422](#)
 - ClpPrimalColumnSteepest, [427](#)
 - ClpPrimalQuadraticDantzig, [431](#)
 - ClpSimplexNonlinear, [496](#)
 - CoinAbcAnyFactorization, [525](#)
 - CoinAbcTypeFactorization, [552](#)
- pivotColumn_
 - CoinAbcTypeFactorization, [571](#)
- pivotColumnAddress_
 - CoinAbcTypeFactorization, [566](#)
- pivotColumnDantzig
 - AbcMatrix, [36](#)
- pivotColumnOldMethod
 - ClpPrimalColumnSteepest, [427](#)
- pivotColumnSingleton
 - CoinAbcTypeFactorization, [560](#)
- pivotLOrder
 - CoinAbcTypeFactorization, [552](#)
- pivotLOrderAddress_
 - CoinAbcTypeFactorization, [568](#)
- pivotLinkedBackwards
 - CoinAbcTypeFactorization, [552](#)
- pivotLinkedBackwardsAddress_
 - CoinAbcTypeFactorization, [568](#)
- pivotLinkedForwards
 - CoinAbcTypeFactorization, [552](#)
- pivotLinkedForwardsAddress_
 - CoinAbcTypeFactorization, [568](#)
- pivotNonlinearResult
 - ClpSimplexNonlinear, [496](#)
- pivotOneOtherRow
 - CoinAbcTypeFactorization, [560](#)
- pivotRegion
 - AbcSimplexFactorization, [116](#)
 - CoinAbcAnyFactorization, [523](#)
 - CoinAbcTypeFactorization, [552](#)
- pivotRegion_
 - CoinAbcTypeFactorization, [572](#)
- pivotRegionAddress_
 - CoinAbcTypeFactorization, [566](#)
- pivotResult
 - AbcSimplexPrimal, [121](#)
 - ClpSimplexPrimal, [504](#)
- pivotResult4
 - AbcSimplexPrimal, [121](#)
- pivotResultPart1
 - AbcSimplexDual, [108](#)
 - ClpSimplexDual, [494](#)
- pivotResultPart2
 - ClpSimplex, [460](#)
- pivotRow
 - AbcDualRowDantzig, [17](#)
 - AbcDualRowPivot, [20](#)
 - AbcDualRowSteepest, [24](#)
 - AbcPrimalColumnPivot, [55](#)
 - ClpDualRowDantzig, [186](#)
 - ClpDualRowPivot, [189](#)
 - ClpDualRowSteepest, [192](#)
 - ClpPrimalColumnPivot, [422](#)
 - ClpSimplex, [468](#)
 - CoinAbcAnyFactorization, [524](#)
- pivotRow_
 - AbcSimplexPrimal::pivotStruct, [638](#)
 - ClpSimplex, [484](#)
 - CoinAbcAnyFactorization, [529](#)
- pivotRowSingleton
 - CoinAbcTypeFactorization, [560](#)
- pivotTolerance
 - AbcSimplexFactorization, [116](#)
 - ClpFactorization, [231](#)
 - CoinAbcAnyFactorization, [523](#)
- pivotTolerance_
 - ClpDataSave, [182](#)
 - CoinAbcAnyFactorization, [528](#)
- pivotVariable
 - AbcSimplex, [82](#)
 - ClpSimplex, [467](#)
- pivotVariable_
 - ClpSimplex, [485](#)

- pivotVariables_
 - ClpNode, 359
- pivoted
 - AbcSimplex, 89
 - ClpSimplex, 474
- pivots
 - AbcSimplexFactorization, 114
 - ClpFactorization, 229
 - CoinAbcAnyFactorization, 522
- popMessageHandler
 - ClpModel, 332
- possiblePivotKey_
 - ClpGubMatrix, 255
- postProcess
 - CoinAbcAnyFactorization, 526
 - CoinAbcDenseFactorization, 533
 - CoinAbcTypeFactorization, 560
- postsolve
 - ClpPresolve, 418
- preOrder
 - ClpCholeskyBase, 154
- preProcess
 - CoinAbcAnyFactorization, 526
 - CoinAbcDenseFactorization, 533
 - CoinAbcTypeFactorization, 560
- preProcess3
 - CoinAbcTypeFactorization, 560
- preProcess4
 - CoinAbcTypeFactorization, 560
- presolve
 - ClpPresolve, 418
- presolveAfterFirstSolve
 - ClpEventHandler, 223
- presolveAfterSolve
 - ClpEventHandler, 223
- presolveBeforeSolve
 - ClpEventHandler, 223
- presolveEnd
 - ClpEventHandler, 223
- presolveInfeasible
 - ClpEventHandler, 223
- presolveNumber
 - ClpSolve, 513
- presolveNumberCost
 - ClpSolve, 513
- presolveOff
 - ClpSolve, 513
- presolveOn
 - ClpSolve, 513
- presolveSize
 - ClpEventHandler, 223
- presolveStart
 - ClpEventHandler, 223
- presolveActions
 - ClpPresolve, 417
 - ClpSolve, 516
- presolveStatus
 - ClpPresolve, 418
- presolveTolerance
 - ClpModel, 321
- PresolveType
 - ClpSolve, 513
- presolveType_
 - ClpNodeStuff, 365
- presolvedModel
 - ClpPresolve, 414
- presolvedModelToFile
 - ClpPresolve, 414
- previousBasis_
 - AbcWarmStart, 131
- primal
 - AbcSimplex, 76
 - AbcSimplexPrimal, 120
 - ClpSimplex, 455
 - ClpSimplexNonlinear, 495
 - ClpSimplexPrimal, 502
- primalColumn
 - AbcSimplexPrimal, 122
 - ClpSimplexPrimal, 504
- primalColumnDouble
 - AbcMatrix, 36
- primalColumnPivot
 - AbcSimplex, 79
 - ClpSimplex, 464
- primalColumnPivot_
 - ClpSimplex, 485
- primalColumnRow
 - AbcMatrix, 36
- primalColumnRowAndDjs
 - AbcMatrix, 36
- primalColumnSolution
 - ClpModel, 324
- primalColumnSparseDouble
 - AbcMatrix, 36
- primalColumnSubset
 - AbcMatrix, 37
- primalDual
 - ClpInterior, 267
- primalExpanded
 - ClpGubMatrix, 250
 - ClpMatrixBase, 295
- primalFeasible
 - ClpInterior, 267
 - ClpSimplex, 460
- primalObjective
 - ClpInterior, 267
- primalObjective_
 - ClpInterior, 274

- primalObjectiveLimit
 - ClpModel, [321](#)
- primalPivotResult
 - ClpSimplex, [460](#)
 - OsiClpSolverInterface, [611](#)
- primalR
 - ClpInterior, [269](#)
- primalR_
 - ClpInterior, [278](#)
- primalRanging
 - ClpSimplex, [457](#)
 - ClpSimplexOther, [498](#)
- primalRay
 - AbcSimplexPrimal, [123](#)
 - ClpSimplexPrimal, [505](#)
- primalRow
 - AbcSimplexPrimal, [122](#)
 - ClpSimplexPrimal, [504](#)
- primalRowSolution
 - ClpModel, [323](#)
- primalSLP
 - ClpSimplexNonlinear, [496](#)
- primalSolution
 - ampl_info, [136](#)
 - ClpNode, [356](#)
- primalSolution_
 - ClpNode, [358](#)
- primalTolerance
 - ClpModel, [320](#)
- primalTolerance_
 - AbcTolerancesEtc, [125](#)
 - ClpSimplex, [482](#)
- primalToleranceToGetOptimal_
 - AbcTolerancesEtc, [125](#)
 - ClpSimplex, [480](#)
- print
 - MyMessageHandler, [590](#)
- printLongHelp
 - CbcOrClpParam, [146](#)
- printOptions
 - CbcOrClpParam, [145](#)
- printRegion
 - CoinAbcTypeFactorization, [559](#)
- printString
 - CbcOrClpParam, [146](#)
- printStuff
 - AbcSimplex, [90](#)
- priorities
 - ampl_info, [136](#)
- priority_
 - ClpNodeStuff, [363](#)
- problemName
 - ClpModel, [321](#)
- problemStatus
 - ampl_info, [135](#)
 - ClpModel, [322](#)
- problemStatus_
 - ClpModel, [340](#)
- progress
 - ClpSimplex, [475](#)
- progress_
 - ClpSimplex, [488](#)
- progressFlag
 - ClpSimplex, [475](#)
- progressFlag_
 - ClpSimplex, [487](#)
- projectionTolerance
 - ClpInterior, [268](#)
- projectionTolerance_
 - ClpInterior, [275](#)
- pseudoDown
 - ampl_info, [136](#)
- pseudoUp
 - ampl_info, [136](#)
- pushMessageHandler
 - ClpModel, [332](#)
- putBackSolution
 - AbcSimplex, [75](#)
- putIntofUseful
 - AbcMatrix, [34](#)
- putStuffInBasis
 - AbcSimplex, [90](#)
- quadraticDjs
 - ClpInterior, [270](#)
- quadraticObjective
 - ClpQuadraticObjective, [435](#)
- r3norm
 - Info, [587](#)
- r3ratio
 - Outfo, [634](#)
- ROW_DUAL_OK
 - AbcSimplex.hpp, [645](#)
- ROW_LOWER_SAME
 - ClpModel.hpp, [690](#)
- ROW_PRIMAL_OK
 - AbcSimplex.hpp, [645](#)
- ROW_UPPER_SAME
 - ClpModel.hpp, [690](#)
- randomNumberGenerator
 - ClpModel, [333](#)
- randomNumberGenerator_
 - ClpModel, [341](#)
- rangeOfElements
 - ClpMatrixBase, [294](#)
 - ClpNetworkMatrix, [350](#)
 - ClpPackedMatrix, [383](#)
 - ClpPlusMinusOneMatrix, [405](#)

- rank
 - ClpCholeskyBase, [152](#)
- rawObjectiveValue
 - AbcSimplex, [90](#)
 - ClpInterior, [270](#)
 - ClpModel, [336](#)
 - ClpSimplex, [475](#)
- rawObjectiveValue_
 - AbcSimplex, [93](#)
- ray
 - ClpModel, [331](#)
- ray_
 - ClpModel, [339](#)
- rayExists
 - ClpModel, [331](#)
- readAmpl
 - Clp_ampl.h, [657](#)
- readBasis
 - ClpSimplex, [458](#)
 - ClpSimplexOther, [500](#)
- readGMPL
 - ClpModel, [316](#)
 - ClpSimplex, [454](#)
- readLp
 - ClpSimplex, [454](#)
 - OsiClpSolverInterface, [623](#)
- readMps
 - ClpInterior, [266](#)
 - ClpModel, [316](#)
 - ClpSimplex, [454](#)
 - OsiClpSolverInterface, [623](#)
- realInfeasibility_
 - ClpSimplexProgress, [509](#)
- reallyBadTimes
 - ClpSimplexProgress, [509](#)
- reallyScale
 - ClpConstraint, [173](#)
 - ClpConstraintLinear, [176](#)
 - ClpConstraintQuadratic, [179](#)
 - ClpLinearObjective, [282](#)
 - ClpMatrixBase, [294](#)
 - ClpObjective, [374](#)
 - ClpPackedMatrix, [385](#)
 - ClpQuadraticObjective, [434](#)
- rebalance
 - AbcMatrix, [36](#)
- recomputeInfeasibilities
 - AbcDualRowDantzig, [18](#)
 - AbcDualRowPivot, [21](#)
 - AbcDualRowSteepest, [25](#)
- redoScaleFactors
 - OsiClpSolverInterface, [628](#)
- redoSet
 - ClpGubMatrix, [251](#)
- reducedCost
 - AbcSimplex, [87](#)
 - ClpDynamicMatrix, [213](#)
 - ClpMatrixBase, [296](#)
 - ClpSimplex, [472](#)
- reducedCost_
 - ClpModel, [338](#)
- reducedCostAddress
 - AbcSimplex, [87](#)
 - ClpSimplex, [472](#)
- reducedCostWork_
 - ClpSimplex, [484](#)
- reducedGradient
 - ClpLinearObjective, [281](#)
 - ClpObjective, [373](#)
 - ClpQuadraticObjective, [434](#)
 - ClpSimplex, [456](#)
- reference
 - AbcPrimalColumnSteepest, [61](#)
 - ClpPrimalColumnSteepest, [429](#)
- refresh
 - AbcNonLinearCost, [49](#)
 - ClpDynamicMatrix, [213](#)
 - ClpMatrixBase, [294](#)
 - ClpNonLinearCost, [369](#)
 - ClpPackedMatrix, [385](#)
- refreshCosts
 - AbcNonLinearCost, [49](#)
 - AbcSimplex, [81](#)
 - ClpNonLinearCost, [369](#)
- refreshFrequency
 - ClpMatrixBase, [299](#)
- refreshFrequency_
 - ClpMatrixBase, [301](#)
- refreshFromPerturbed
 - AbcNonLinearCost, [49](#)
- refreshLower
 - AbcSimplex, [81](#)
- refreshUpper
 - AbcSimplex, [81](#)
- relaxAccuracyCheck
 - ClpFactorization, [232](#)
 - CoinAbcAnyFactorization, [523](#)
- relaxCheck_
 - CoinAbcAnyFactorization, [528](#)
- releaseClp
 - OsiClpSolverInterface, [627](#)
- releasePackedMatrix
 - ClpDummyMatrix, [199](#)
 - ClpMatrixBase, [295](#)
 - ClpNetworkMatrix, [351](#)
 - ClpPackedMatrix, [384](#)
 - ClpPlusMinusOneMatrix, [406](#)
- releaseSpecialColumnCopy

- ClpPackedMatrix, [387](#)
- removeSuperBasicSlacks
 - ClpSimplex, [457](#)
- reorderU
 - CoinAbcTypeFactorization, [561](#)
- replaceColumn
 - ClpFactorization, [228](#)
 - ClpNetworkBasis, [344](#)
 - CoinAbcDenseFactorization, [534](#)
- replaceColumnPFI
 - CoinAbcTypeFactorization, [564](#)
- replaceColumnPart3
 - AbcSimplexDual, [106](#)
 - AbcSimplexFactorization, [113](#)
 - CoinAbcAnyFactorization, [527](#)
 - CoinAbcDenseFactorization, [534](#)
 - CoinAbcTypeFactorization, [557](#)
- replaceColumnU
 - CoinAbcTypeFactorization, [557](#)
- replaceMatrix
 - ClpModel, [330](#)
 - OsiClpSolverInterface, [624](#)
- replaceMatrixOptional
 - OsiClpSolverInterface, [624](#)
- replaceVector
 - ClpPackedMatrix, [382](#)
- reserveSpace
 - ClpCholeskyDense, [159](#)
- reset
 - ClpSimplexProgress, [507](#)
 - OsiClpSolverInterface, [627](#)
- resetFakeBounds
 - AbcSimplexDual, [108](#)
 - ClpSimplexDual, [494](#)
- resetRowsDropped
 - ClpCholeskyBase, [151](#)
- resetStatistics
 - CoinAbcTypeFactorization, [559](#)
- resize
 - AbcSimplex, [92](#)
 - AbcWarmStart, [129](#)
 - ClpConstraint, [172](#)
 - ClpConstraintLinear, [176](#)
 - ClpConstraintQuadratic, [179](#)
 - ClpLinearObjective, [281](#)
 - ClpModel, [317](#)
 - ClpObjective, [374](#)
 - ClpQuadraticObjective, [434](#)
 - ClpSimplex, [479](#)
- resolve
 - OsiClpSolverInterface, [609](#)
- resolveGub
 - OsiClpSolverInterface, [609](#)
- restoreBaseModel
 - OsiClpSolverInterface, [621](#)
- restoreData
 - AbcSimplex, [79](#)
 - ClpSimplex, [464](#)
- restoreFromDual
 - ClpSimplexOther, [500](#)
- restoreGoodStatus
 - AbcSimplex, [81](#)
- restoreModel
 - ClpSimplex, [463](#)
- result
 - CoinAbcThreadInfo, [538](#)
- returnModel
 - ClpInterior, [266](#)
 - ClpModel, [319](#)
 - ClpSimplex, [464](#)
- reverseOrderedCopy
 - AbcMatrix, [33](#)
 - ClpDummyMatrix, [198](#)
 - ClpGubMatrix, [248](#)
 - ClpMatrixBase, [292](#)
 - ClpNetworkMatrix, [350](#)
 - ClpPackedMatrix, [383](#)
 - ClpPlusMinusOneMatrix, [405](#)
- reversePivotVariable_
 - AbcSimplex, [100](#)
- rhs_
 - ClpInterior, [273](#)
 - OsiClpSolverInterface, [629](#)
- rhsB_
 - ClpInterior, [278](#)
- rhsC_
 - ClpInterior, [278](#)
- rhsFixRegion_
 - ClpInterior, [276](#)
- rhsL_
 - ClpInterior, [278](#)
- rhsNorm_
 - ClpInterior, [274](#)
- rhsOffset
 - ClpDynamicMatrix, [212](#)
 - ClpGubDynamicMatrix, [238](#)
 - ClpGubMatrix, [251](#)
 - ClpMatrixBase, [299](#)
- rhsOffset_
 - ClpMatrixBase, [301](#)
- rhsScale
 - ClpModel, [328](#)
- rhsScale_
 - ClpModel, [337](#)
- rhsU_
 - ClpInterior, [278](#)
- rhsW_
 - ClpInterior, [278](#)

- rhsZ_
 - ClpInterior, 278
- row
 - ClpDynamicMatrix, 215
 - ClpGubDynamicMatrix, 239
- row_
 - AbcMatrix3, 46
 - ClpDynamicMatrix, 220
 - ClpGubDynamicMatrix, 241
 - ClpPackedMatrix3, 394
- rowActivity_
 - ClpModel, 338
 - OsiClpSolverInterface, 630
- rowActivityWork_
 - ClpSimplex, 485
- rowArray
 - ClpSimplex, 463
- rowArray_
 - ClpSimplex, 483
- rowColumns
 - AbcMatrix, 33
- rowCopy
 - ClpModel, 329
- rowCopy_
 - ClpCholeskyBase, 156
 - ClpModel, 339
 - ClpPackedMatrix, 389
- rowElements
 - AbcMatrix, 33
- rowEnd
 - AbcMatrix, 33
- rowGen
 - ClpDynamicExampleMatrix, 204
- rowGen_
 - ClpDynamicExampleMatrix, 205
- rowLower
 - ampl_info, 135
 - ClpModel, 324
- rowLower_
 - ClpModel, 338
- rowLowerWork_
 - ClpInterior, 273
 - ClpSimplex, 483
- rowName
 - ClpModel, 334
- rowNames
 - ClpModel, 334
- rowNames_
 - ClpModel, 341
- rowNamesAsChar
 - ClpModel, 337
- rowNumber
 - ClpConstraint, 173
- rowNumber_
 - ClpConstraint, 174
- rowObjective
 - ClpModel, 329
- rowObjective_
 - ClpModel, 338
- rowObjectiveWork_
 - ClpSimplex, 483
- rowPrimalSequence_
 - ClpSimplex, 480
- rowReducedCost_
 - ClpSimplex, 484
- rowScale
 - ClpModel, 327
- rowScale2
 - AbcSimplex, 82
- rowScale_
 - ClpModel, 339
 - OsiClpSolverInterface, 633
- rowStart
 - AbcMatrix, 33
- rowStart_
 - AbcMatrix, 39
 - AbcMatrix2, 43
 - ClpPackedMatrix2, 391
- rowStatus
 - ampl_info, 136
- rowUpper
 - ampl_info, 135
 - ClpModel, 324
- rowUpper_
 - ClpModel, 338
- rowUpperWork_
 - ClpInterior, 273
 - ClpSimplex, 483
- rowUseScale_
 - AbcSimplex.hpp, 644
- rowrange_
 - OsiClpSolverInterface, 629
- rows
 - ampl_info, 136
- rowsDropped
 - ClpCholeskyBase, 151
 - ClpCholeskyDenseC, 160
- rowsDropped_
 - ClpCholeskyBase, 155
- rowsense_
 - OsiClpSolverInterface, 629
- SCATTER_ATTRIBUTE
 - CoinAbcHelperFunctions.hpp, 715
- SLACK_VALUE
 - CoinAbcCommon.hpp, 703
- SWAP_FACTOR
 - CoinAbcCommonFactorization.hpp, 708

- saferTolerances
 - AbcSimplexFactorization, 115
 - ClpFactorization, 230
- sanityCheck
 - ClpInterior, 269
 - ClpSimplex, 469
- saveBaseModel
 - OsiClpSolverInterface, 621
- saveColumn_
 - CoinAbcTypeFactorization, 572
- saveColumnAddress_
 - CoinAbcTypeFactorization, 567
- saveCosts_
 - ClpNodeStuff, 364
- saveData
 - AbcSimplex, 79
 - ClpSimplex, 464
- saveData_
 - AbcSimplex, 101
 - OsiClpSolverInterface, 632
- saveDualIn_
 - AbcSimplexPrimal::pivotStruct, 637
- saveGoodStatus
 - AbcSimplex, 81
- saveInfo
 - ClpDisasterHandler, 184
 - OsiClpDisasterHandler, 595
- saveModel
 - ClpSimplex, 463
- saveNumber_
 - ClpGubMatrix, 255
- saveOptions_
 - ClpNodeStuff, 364
- saveSolution
 - CbcOrClpParam.hpp, 657
- saveStatus_
 - ClpGubMatrix, 254
 - ClpSimplex, 486
- saveWeights
 - AbcDualRowDantzig, 18
 - AbcDualRowPivot, 21
 - AbcDualRowSteepest, 25
 - AbcPrimalColumnDantzig, 52
 - AbcPrimalColumnPivot, 55
 - AbcPrimalColumnSteepest, 60
 - ClpDualRowPivot, 189
 - ClpDualRowSteepest, 193
 - ClpPrimalColumnDantzig, 419
 - ClpPrimalColumnPivot, 422
 - ClpPrimalColumnSteepest, 428
 - ClpPrimalQuadraticDantzig, 431
- savedBestDj
 - AbcMatrix, 38
 - ClpMatrixBase, 300
- savedBestDj_
 - AbcMatrix, 40
 - ClpMatrixBase, 301
- savedBestGubDual_
 - ClpDynamicMatrix, 217
- savedBestSequence
 - AbcMatrix, 38
 - ClpMatrixBase, 300
- savedBestSequence_
 - AbcMatrix, 40
 - ClpMatrixBase, 301
- savedBestSet_
 - ClpDynamicMatrix, 217
- savedColumnScale_
 - ClpModel, 342
- savedFirstAvailable_
 - ClpGubDynamicMatrix, 243
- savedKeyVariable_
 - ClpGubMatrix, 254
- savedRowScale_
 - ClpModel, 342
- savedSolution_
 - ClpSimplex, 486
- scale
 - AbcMatrix, 33
 - ClpMatrixBase, 293
 - ClpPackedMatrix, 383
- scaleFactor_
 - ClpInterior, 275
- scaleFromExternal
 - AbcSimplex, 82
- scaleFromExternal_
 - AbcSimplex, 97
- scaleObjective
 - ClpSimplex, 465
- scaleRealObjective
 - ClpSimplex, 458
- scaleRealRhs
 - ClpSimplex, 458
- scaleRowCopy
 - ClpMatrixBase, 293
 - ClpPackedMatrix, 383
- scaleToExternal
 - AbcSimplex, 82
- scaleToExternal_
 - AbcSimplex, 97
- scaledColumnCopy
 - ClpMatrixBase, 293
 - ClpPackedMatrix, 383
- scaledMatrix_
 - ClpModel, 339
- scaling
 - ClpModel, 328
- scalingFlag

- ClpModel, 328
- scalingFlag_
 - ClpDataSave, 182
 - ClpModel, 339
- scan
 - CoinAbcTypeFactorization, 556
- scatterStruct, 638
 - functionPointer, 638
 - number, 638
 - offset, 638
- scatterUpdate
 - CoinAbcHelperFunctions.hpp, 715
- secondaryStatus
 - ClpModel, 322
- secondaryStatus_
 - ClpModel, 340
- separateLinks
 - CoinAbcTypeFactorization, 562
- sequence
 - ClpNode, 357
 - dualColumnResult, 580
- sequence_
 - ClpNode, 359
- sequenceIn
 - AbcSimplex, 86
 - ClpSimplex, 470
- sequenceIn_
 - AbcSimplexPrimal::pivotStruct, 637
 - ClpSimplex, 484
- sequenceOut
 - AbcSimplex, 86
 - ClpSimplex, 470
- sequenceOut_
 - AbcSimplexPrimal::pivotStruct, 637
 - ClpSimplex, 484
- sequenceWithin
 - AbcSimplex, 86
 - ClpInterior, 270
 - ClpSimplex, 472
- setAbcState
 - ClpSimplex, 453
- setAbove
 - ClpGubMatrix, 252
- setActivated
 - ClpObjective, 375
- setActive
 - AbcSimplex, 90
 - ClpSimplex, 474
- setAlgorithm
 - ClpInterior, 267
 - ClpSimplex, 461
- setAlpha
 - ClpSimplex, 468
- setAlphaAccuracy
 - ClpSimplex, 466
- setAutomaticScaling
 - ClpSimplex, 467
- setAvailableArray
 - AbcSimplex, 89
- setAverageTheta
 - AbcNonLinearCost, 50
 - ClpNonLinearCost, 371
- setBasis
 - OsiClpSolverInterface, 628
- setBasisStatus
 - OsiClpSolverInterface, 610
- setBelow
 - ClpGubMatrix, 252
- setBiasLU
 - ClpFactorization, 232
- setCbcOrClpPrinting
 - CbcOrClpParam.hpp, 657
- setChangeInCost
 - AbcNonLinearCost, 50
 - ClpNonLinearCost, 371
- setCholesky
 - ClpInterior, 269
- setCleanupScaling
 - OsiClpSolverInterface, 625
- setClpScaledMatrix
 - ClpModel, 330
- setClpSimplexObjectiveValue
 - AbcSimplex, 77
- setColBounds
 - AbcSimplex, 92
 - ClpModel, 325
 - ClpSimplex, 477
 - OsiClpSolverInterface, 617
- setColLower
 - AbcSimplex, 91
 - ClpModel, 325
 - ClpSimplex, 477
 - OsiClpSolverInterface, 617, 618
- setColName
 - OsiClpSolverInterface, 618
- setColSetBounds
 - AbcSimplex, 92
 - ClpModel, 325
 - ClpSimplex, 477
 - OsiClpSolverInterface, 617
- setColSolution
 - ClpModel, 324
 - OsiClpSolverInterface, 619
- setColUpper
 - AbcSimplex, 91
 - ClpModel, 325
 - ClpSimplex, 477
 - OsiClpSolverInterface, 617, 618

- setColumnBounds
 - AbcSimplex, [91](#)
 - ClpModel, [325](#)
 - ClpSimplex, [477](#)
- setColumnLower
 - AbcSimplex, [91](#)
 - ClpModel, [325](#)
 - ClpSimplex, [477](#)
- setColumnName
 - ClpModel, [320](#)
- setColumnScale
 - ClpModel, [328](#)
- setColumnSetBounds
 - AbcSimplex, [91](#)
 - ClpModel, [325](#)
 - ClpSimplex, [477](#)
- setColumnStatus
 - ClpSimplex, [474](#)
 - OsiClpSolverInterface, [613](#)
- setColumnUpper
 - AbcSimplex, [91](#)
 - ClpModel, [325](#)
 - ClpSimplex, [477](#)
- setContinuous
 - ClpModel, [316](#)
 - OsiClpSolverInterface, [619](#)
- setCurrentDualTolerance
 - AbcSimplex, [78](#)
 - ClpSimplex, [467](#)
- setCurrentOption
 - CbcOrClpParam, [145](#)
- setCurrentOptionWithMessage
 - CbcOrClpParam, [145](#)
- setCurrentPrimalTolerance
 - ClpSimplex, [467](#)
- setCurrentStatus
 - AbcNonLinearCost.hpp, [642](#)
 - ClpNonLinearCost.hpp, [693](#)
- setCurrentWanted
 - AbcMatrix, [38](#)
 - ClpMatrixBase, [300](#)
- setDblParam
 - ClpModel, [335](#)
 - OsiClpSolverInterface, [611](#)
- setDefaultMessageHandler
 - ClpModel, [332](#)
- setDefaultValues
 - ClpFactorization, [232](#)
- setDelta
 - ClpInterior, [268](#)
- setDenseThreshold
 - AbcSimplexFactorization, [116](#)
 - ClpFactorization, [231](#)
 - CoinAbcTypeFactorization, [554](#)
- setDiagonalPerturbation
 - ClpInterior, [268](#)
- setDimensions
 - ClpMatrixBase, [294](#)
 - ClpPackedMatrix, [385](#)
 - ClpPlusMinusOneMatrix, [406](#)
- setDirectionIn
 - ClpSimplex, [471](#)
- setDirectionOut
 - ClpSimplex, [471](#)
- setDisasterHandler
 - ClpSimplex, [466](#)
- setDoDoubleton
 - ClpPresolve, [415](#)
 - ClpSolve, [515](#)
- setDoDual
 - ClpPresolve, [415](#)
 - ClpSolve, [514](#)
- setDoDupcol
 - ClpPresolve, [416](#)
 - ClpSolve, [516](#)
- setDoDuprow
 - ClpPresolve, [416](#)
 - ClpSolve, [516](#)
- setDoForcing
 - ClpPresolve, [416](#)
 - ClpSolve, [515](#)
- setDoGubrow
 - ClpPresolve, [417](#)
- setDoImpliedFree
 - ClpPresolve, [416](#)
 - ClpSolve, [515](#)
- setDoIntersection
 - ClpPresolve, [417](#)
- setDoKillSmall
 - ClpSolve, [516](#)
- setDoSingleton
 - ClpPresolve, [415](#)
 - ClpSolve, [515](#)
- setDoSingletonColumn
 - ClpPresolve, [417](#)
 - ClpSolve, [516](#)
- setDoTighten
 - ClpPresolve, [416](#)
 - ClpSolve, [515](#)
- setDoTripleton
 - ClpPresolve, [416](#)
 - ClpSolve, [515](#)
- setDoTwotwo
 - ClpPresolve, [417](#)
- setDoubleParameter
 - CbcOrClpParam, [143](#), [144](#)
 - ClpCholeskyBase, [153](#)
- setDoubleParameterWithMessage

- CbcOrClpParam, 143, 144
- setDoubleValue
 - CbcOrClpParam, 145
- setDropEnoughFeasibility
 - Idiot, 585
- setDropEnoughWeighted
 - Idiot, 585
- setDualBound
 - ClpSimplex, 461
- setDualIn
 - ClpSimplex, 468
- setDualObjectiveLimit
 - ClpModel, 321
- setDualOut
 - ClpSimplex, 471
- setDualRowPivotAlgorithm
 - AbcSimplex, 76
 - ClpSimplex, 459
- setDualTolerance
 - ClpModel, 321
- setDynamicStatus
 - ClpDynamicMatrix, 215
 - ClpGubDynamicMatrix, 239
- setDynamicStatusGen
 - ClpDynamicExampleMatrix, 204
- setElements
 - ClpHelperFunctions.hpp, 682
- setEmptyFactorization
 - ClpSimplex, 476
- setEndFraction
 - AbcMatrix, 38
 - ClpMatrixBase, 300
- setExitInfeasibility
 - Idiot, 583
- setFactorization
 - AbcSimplex, 76
 - AbcSimplexFactorization, 112
 - ClpFactorization, 233
 - ClpSimplex, 458
- setFactorizationFrequency
 - AbcSimplex, 77
 - ClpSimplex, 461
- setFakeBound
 - AbcSimplex, 89
 - ClpSimplex, 473
- setFakeKeyword
 - CbcOrClpParam, 146
- setFakeLower
 - ClpInterior, 271
- setFakeObjective
 - OsiClpSolverInterface, 626
- setFakeUpper
 - ClpInterior, 271
- setFeasibilityTolerance
 - Idiot, 583
- setFeasible
 - ClpGubMatrix, 252
- setFirstAvailable
 - ClpGubDynamicMatrix, 241
- setFixed
 - ClpInterior, 270
- setFixedOrFree
 - ClpInterior, 270
- setFlagged
 - AbcSimplex, 89
 - ClpDynamicMatrix, 215
 - ClpGubDynamicMatrix, 239
 - ClpGubMatrix, 252
 - ClpInterior, 270
 - ClpSimplex, 474
- setFlaggedGen
 - ClpDynamicExampleMatrix, 205
- setFlaggedSlack
 - ClpDynamicMatrix, 214
- setForrestTomlin
 - ClpFactorization, 232
- setGamma
 - ClpInterior, 268
- setGoDense
 - ClpCholeskyBase, 151
- setGoDenseThreshold
 - AbcSimplexFactorization, 117
 - ClpFactorization, 233
- setGoLongThreshold
 - AbcSimplexFactorization, 117
- setGoOslThreshold
 - ClpFactorization, 232
- setGoSmallThreshold
 - AbcSimplexFactorization, 117
 - ClpFactorization, 233
- setGubBasis
 - ClpSimplexOther, 500
- setHintParam
 - OsiClpSolverInterface, 612
- setInfeasibility
 - ClpSimplexProgress, 508
- setInfeasibilityCost
 - ClpSimplex, 461
- setInfeasible
 - ClpNonLinearCost, 371
- setInfeasibleReturn
 - ClpSolve, 514
- setInfo
 - OsiClpSolverInterface, 619
- setInfo_
 - OsiClpSolverInterface, 630
- setInitialDenseFactorization
 - AbcSimplex, 86

- ClpSimplex, 470
- setInitialStatus
 - AbcNonLinearCost.hpp, 642
 - ClpNonLinearCost.hpp, 693
- setIntParam
 - ClpModel, 335
 - OsiClpSolverInterface, 611
- setIntParameter
 - CbcOrClpParam, 143, 144
- setIntParameterWithMessage
 - CbcOrClpParam, 143, 144
- setIntValue
 - CbcOrClpParam, 145
- setInteger
 - ClpModel, 317
 - OsiClpSolverInterface, 619
- setIntegerParameter
 - ClpCholeskyBase, 152
- setInternalColumnStatus
 - AbcSimplex, 86
- setInternalStatus
 - AbcSimplex, 86
- setKKT
 - ClpCholeskyBase, 152
- setLanguage
 - ClpModel, 332
 - OsiClpSolverInterface, 624
- setLargeValue
 - ClpSimplex, 466
- setLargestAway
 - OsiClpSolverInterface, 629
- setLargestDualError
 - ClpSimplex, 467
- setLargestPrimalError
 - ClpSimplex, 466
- setLastAlgorithm
 - OsiClpSolverInterface, 624
- setLastBadIteration
 - ClpSimplex, 475
- setLengthNames
 - ClpModel, 334
- setLightweight
 - Idiot, 584
- setLinearPerturbation
 - ClpInterior, 268
- setLogLevel
 - ClpModel, 333
 - Idiot, 584
 - OsiClpSolverInterface, 624
- setLonghelp
 - CbcOrClpParam, 146
- setLooksOptimal
 - AbcPrimalColumnPivot, 55
 - ClpPrimalColumnPivot, 423
- setLowerBound
 - ClpInterior, 271
- setLowerOut
 - ClpSimplex, 471
- setMajorIterations
 - Idiot, 584
- setMatrixNull
 - ClpPackedMatrix, 387
- setMaximumBarrierIterations
 - ClpInterior, 269
- setMaximumIterations
 - ClpModel, 322
- setMaximumSeconds
 - ClpModel, 322
- setMethod
 - ClpNonLinearCost, 371
- setMinimumGoodReducedCosts
 - AbcMatrix, 37
 - ClpMatrixBase, 299
- setMinimumObjectsScan
 - AbcMatrix, 37
 - ClpMatrixBase, 299
- setMinorIterations
 - Idiot, 584
- setMinorIterations0
 - Idiot, 584
- setModel
 - AbcDualRowPivot, 21
 - AbcMatrix, 32
 - AbcPrimalColumnPivot, 56
 - AbcWarmStart, 129
 - ClpCholeskyBase, 153
 - ClpDualRowPivot, 190
 - ClpPrimalColumnPivot, 423
 - MyMessageHandler, 590
- setMoreSpecialOptions
 - ClpSimplex, 473
- setMultipleSequenceIn
 - AbcSimplex, 80
- setNewRowCopy
 - ClpModel, 330
- setNoGotLCopy
 - CoinAbcTypeFactorization, 565
- setNoGotRCopy
 - CoinAbcTypeFactorization, 565
- setNoGotSparse
 - CoinAbcTypeFactorization, 565
- setNoGotUCopy
 - CoinAbcTypeFactorization, 565
- setNonLinearValue
 - ClpPresolve, 415
- setNumberActiveColumns
 - ClpPackedMatrix, 388
- setNumberDualInfeasibilities

- ClpSimplex, 462
- setNumberElementsU
 - CoinAbcTypeFactorization, 554
- setNumberIterations
 - ClpModel, 321
- setNumberOrdinary
 - AbcSimplex, 78
- setNumberPrimalInfeasibilities
 - ClpSimplex, 463
- setNumberRefinements
 - ClpSimplex, 468
- setNumberRows
 - CoinAbcAnyFactorization, 522
- setNumberSlacks
 - CoinAbcAnyFactorization, 522
- setNumberThreads
 - ClpModel, 332
- setObjCoeff
 - AbcSimplex, 91
 - ClpModel, 325
 - ClpSimplex, 477
 - OsiClpSolverInterface, 617
- setObjSense
 - OsiClpSolverInterface, 619
- setObjective
 - ClpModel, 334
 - OsiClpSolverInterface, 618
- setObjectiveCoefficient
 - AbcSimplex, 91
 - ClpModel, 324
 - ClpSimplex, 476
- setObjectiveOffset
 - ClpModel, 321
- setObjectivePointer
 - ClpModel, 334
- setObjectiveScale
 - ClpModel, 328
- setObjectiveValue
 - ClpModel, 330
 - ClpNode, 356
- setOne
 - AbcNonLinearCost, 49
 - ClpNonLinearCost, 369
- setOneBasic
 - AbcNonLinearCost, 49
- setOneOutgoing
 - AbcNonLinearCost, 49
 - ClpNonLinearCost, 369
- setOptimizationDirection
 - ClpModel, 323
- setOptionalInteger
 - OsiClpSolverInterface, 615
- setOriginalModel
 - ClpPresolve, 415
- setOriginalStatus
 - AbcNonLinearCost.hpp, 642
 - ClpNonLinearCost.hpp, 693
- setOriginalWanted
 - AbcMatrix, 38
 - ClpMatrixBase, 300
- setOsiModel
 - OsiClpDisasterHandler, 595
- setParam
 - ClpLsq, 285
- setPersistence
 - AbcDualRowSteepest, 26
 - AbcPrimalColumnSteepest, 61
 - ClpDualRowSteepest, 194
 - ClpPrimalColumnSteepest, 429
- setPersistenceFlag
 - ClpFactorization, 232
 - ClpSimplex, 453
- setPerturbation
 - ClpSimplex, 461
- setPhase
 - OsiClpDisasterHandler, 595
- setPivotRow
 - ClpSimplex, 468
- setPivoted
 - AbcSimplex, 89
 - ClpSimplex, 474
- setPivots
 - AbcSimplexFactorization, 115
 - CoinAbcAnyFactorization, 522
- setPresolveActions
 - ClpPresolve, 417
 - ClpSolve, 516
- setPresolveType
 - ClpSolve, 514
- setPrimalColumnPivotAlgorithm
 - AbcSimplex, 76
 - ClpSimplex, 459
- setPrimalObjectiveLimit
 - ClpModel, 321
- setPrimalTolerance
 - ClpModel, 320
- setProblemStatus
 - ClpModel, 322
- setProjectionTolerance
 - ClpInterior, 268
- setRandomSeed
 - ClpModel, 334
- setReasonablyFeasible
 - Idiot, 583
- setReduceIterations
 - Idiot, 584
- setReference
 - AbcPrimalColumnSteepest, 61

- ClpPrimalColumnSteepest, 429
- setRefreshFrequency
 - ClpMatrixBase, 299
- setRhsScale
 - ClpModel, 328
- setRowBounds
 - AbcSimplex, 92
 - ClpModel, 327
 - ClpSimplex, 479
 - OsiClpSolverInterface, 617
- setRowLower
 - AbcSimplex, 92
 - ClpModel, 327
 - ClpSimplex, 479
 - OsiClpSolverInterface, 617
- setRowName
 - ClpModel, 319
 - OsiClpSolverInterface, 618
- setRowObjective
 - ClpModel, 316
- setRowPrice
 - OsiClpSolverInterface, 619
- setRowScale
 - ClpModel, 328
- setRowSetBounds
 - AbcSimplex, 92
 - ClpModel, 327
 - ClpSimplex, 479
 - OsiClpSolverInterface, 618
- setRowSetTypes
 - OsiClpSolverInterface, 618
- setRowStatus
 - ClpSimplex, 474
- setRowType
 - OsiClpSolverInterface, 618
- setRowUpper
 - AbcSimplex, 92
 - ClpModel, 327
 - ClpSimplex, 479
 - OsiClpSolverInterface, 617
- setSOSData
 - OsiClpSolverInterface, 629
- setSameStatus
 - AbcNonLinearCost.hpp, 642
 - ClpNonLinearCost.hpp, 693
- setSavedBestDj
 - AbcMatrix, 38
 - ClpMatrixBase, 300
- setSavedBestSequence
 - AbcMatrix, 38
 - ClpMatrixBase, 300
- setSecondaryStatus
 - ClpModel, 322
- setSequenceIn

- AbcSimplex, 86
- ClpSimplex, 470
- setSequenceOut
 - AbcSimplex, 86
 - ClpSimplex, 470
- setSimplex
 - ClpDisasterHandler, 184
 - ClpEventHandler, 224
- setSize
 - AbcWarmStart, 129
- setSkipDualCheck
 - ClpMatrixBase, 299
- setSmallElementValue
 - ClpModel, 329
- setSmallestChangeInCut
 - OsiClpSolverInterface, 625
- setSmallestElementInCut
 - OsiClpSolverInterface, 625
- setSolveMode
 - CoinAbcAnyFactorization, 525
- setSolveOptions
 - OsiClpSolverInterface, 625
- setSolveType
 - ClpModel, 322
 - ClpSolve, 514
- setSparseFactorization
 - ClpSimplex, 461
- setSpecialOption
 - ClpSolve, 514
- setSpecialOptions
 - ClpModel, 336
 - OsiClpSolverInterface, 624
- setSpecialOptionsMutable
 - OsiClpSolverInterface, 627
- setStartFraction
 - AbcMatrix, 37
 - ClpMatrixBase, 300
- setStartingWeight
 - Idiot, 583
- setStateOfProblem
 - AbcSimplex, 82
- setStatistics
 - CoinAbcCommonFactorization.hpp, 707
- setStatus
 - AbcSimplexFactorization, 115
 - ClpDynamicMatrix, 214
 - ClpFactorization, 230
 - ClpGubMatrix, 251
 - ClpSimplex, 470
 - CoinAbcAnyFactorization, 522
- setStrParam
 - ClpModel, 335
 - OsiClpSolverInterface, 611
- setStrategy

- Idiot, [585](#)
- setStringValue
 - CbcOrClpParam, [145](#)
- setStuff
 - OsiClpSolverInterface, [613](#)
- setSubstitution
 - ClpPresolve, [417](#)
 - ClpSolve, [516](#)
- setSumDualInfeasibilities
 - ClpSimplex, [462](#)
- setSumOfRelaxedDualInfeasibilities
 - ClpSimplex, [462](#)
- setSumOfRelaxedPrimalInfeasibilities
 - ClpSimplex, [462](#)
- setSumPrimalInfeasibilities
 - ClpSimplex, [462](#)
- setTheta
 - ClpSimplex, [471](#)
- setToBaseModel
 - AbcSimplex, [75](#)
 - ClpSimplex, [453](#)
- setTrustedUserPointer
 - ClpModel, [332](#)
- setType
 - ClpCholeskyBase, [153](#)
 - ClpMatrixBase, [298](#)
 - ClpObjective, [375](#)
 - ClpPdcoBase, [398](#)
- setUpperBound
 - ClpInterior, [271](#)
- setUpperOut
 - ClpSimplex, [471](#)
- setUsedArray
 - AbcSimplex, [89](#)
- setUsefulInformation
 - CoinAbcAnyFactorization, [525](#)
- setUserPointer
 - ClpModel, [331](#)
- setValueOut
 - ClpSimplex, [471](#)
- setValuesPassAction
 - AbcSimplex, [81](#)
 - ClpSimplex, [465](#)
- setWarmStart
 - OsiClpSolverInterface, [612](#)
- setWeightFactor
 - Idiot, [583](#)
- setWhatsChanged
 - ClpModel, [332](#)
- setWhereFrom
 - OsiClpDisasterHandler, [595](#)
- setYesGotLCopy
 - CoinAbcTypeFactorization, [565](#)
- setYesGotRCopy
 - CoinAbcTypeFactorization, [565](#)
- setYesGotSparse
 - CoinAbcTypeFactorization, [565](#)
- setYesGotUCopy
 - CoinAbcTypeFactorization, [565](#)
- setZeroTolerance
 - ClpSimplex, [467](#)
- setupDualValuesPass
 - AbcSimplex, [77](#)
- setupForRepeatedUse
 - OsiClpSolverInterface, [626](#)
- setupForSolve
 - ClpPredictorCorrector, [411](#)
- setupForStrongBranching
 - AbcSimplexDual, [105](#)
 - ClpSimplexDual, [492](#)
- setupPointers
 - AbcSimplex, [81](#)
- shortHelp
 - CbcOrClpParam, [143](#)
- show_self
 - CoinAbcTypeFactorization, [551](#)
- simplex
 - ClpDisasterHandler, [184](#)
 - ClpEventHandler, [224](#)
- size
 - ClpCholeskyBase, [152](#)
- sizeBases_
 - AbcWarmStartOrganizer, [133](#)
- sizeD1
 - ClpPdcoBase, [398](#)
- sizeD2
 - ClpPdcoBase, [398](#)
- sizeFactor_
 - ClpCholeskyBase, [156](#)
- sizeIndex_
 - ClpCholeskyBase, [156](#)
- sizeSparseArray_
 - CoinAbcTypeFactorization, [577](#)
- skipDualCheck
 - ClpMatrixBase, [299](#)
- skipDualCheck_
 - ClpMatrixBase, [302](#)
- slackValue2_
 - CoinAbcDenseFactorization.hpp, [709](#)
- slightlyInfeasible
 - ClpEventHandler, [223](#)
- smallChange_
 - ClpNodeStuff, [363](#)
- smallElement_
 - ClpModel, [337](#)
- smallModel_
 - OsiClpSolverInterface, [630](#)
- smallestChangeInCut

- OsiClpSolverInterface, 625
- smallestChangeInCut_
 - OsiClpSolverInterface, 630
- smallestElementInCut
 - OsiClpSolverInterface, 625
- smallestElementInCut_
 - OsiClpSolverInterface, 630
- smallestInfeasibility_
 - ClpInterior, 276
- soloKey
 - ClpDynamicMatrix, 211
- solution
 - AbcSimplex, 87
 - ClpEventHandler, 223
 - ClpSimplex, 472
- solution_
 - ClpInterior, 277
 - ClpSimplex, 485
- solutionAddress
 - AbcSimplex, 87
 - ClpSimplex, 472
- solutionBasic
 - AbcSimplex, 85
- solutionBasic_
 - AbcSimplex, 99
- solutionNorm_
 - ClpInterior, 274
- solutionRegion
 - AbcSimplex, 84
 - ClpSimplex, 469
- solutionSaved_
 - AbcSimplex, 99
- solve
 - ClpCholeskyBase, 151, 153
 - ClpCholeskyDense, 159
 - ClpCholeskyMumps, 162
 - ClpCholeskyTaucs, 164
 - ClpCholeskyUfl, 166
 - ClpCholeskyWssmp, 168
 - ClpCholeskyWssmpKKT, 170
 - ClpPredictorCorrector, 411
 - ClpSimplex, 456
 - Idiot, 582
- solve2
 - Idiot, 585
- solveB1
 - ClpCholeskyDense, 159
- solveB2
 - ClpCholeskyDense, 159
- solveBenders
 - ClpSimplex, 465
- solveDW
 - ClpSimplex, 465
- solveF1
 - ClpCholeskyDense, 159
- solveF2
 - ClpCholeskyDense, 159
- solveFromHotStart
 - OsiClpSolverInterface, 613
- solveKKT
 - ClpCholeskyBase, 151
 - ClpCholeskyWssmpKKT, 170
- solveMode
 - CoinAbcAnyFactorization, 525
- solveMode_
 - CoinAbcAnyFactorization, 530
- solveOptions_
 - OsiClpSolverInterface, 632
- solveSystem
 - ClpPredictorCorrector, 411
- SolveType
 - ClpSolve, 513
- solveType
 - ClpModel, 321
- solveType_
 - ClpModel, 340
- solverOptions_
 - ClpNodeStuff, 365
- sort
 - CoinAbcTypeFactorization, 551
- sortBlocks
 - AbcMatrix3, 45
 - ClpPackedMatrix3, 394
- sortUseful
 - AbcMatrix, 34
- sosIndices
 - ampl_info, 137
- sosPriority
 - ampl_info, 136
- sosReference
 - ampl_info, 137
- sosStart
 - ampl_info, 137
- sosType
 - ampl_info, 136
- space
 - ClpCholeskyDense, 159
- spaceForForrestTomlin
 - CoinAbcTypeFactorization, 554
- spare
 - ClpNode::branchState, 139
- spareArrays_
 - OsiClpSolverInterface, 631
- spareDoubleArray_
 - ClpSimplex, 488
- spareIntArray_
 - ClpSimplex, 488
- sparse_

- CoinAbcTypeFactorization, 574
- sparseAddress_
 - CoinAbcTypeFactorization, 568
- sparseFactor
 - ClpCholeskyBase, 152
- sparseFactor_
 - ClpCholeskyBase, 155
- sparseFactorization
 - ClpSimplex, 461
- sparseThreshold
 - ClpFactorization, 230
 - CoinAbcTypeFactorization, 559
- sparseThreshold_
 - ClpDataSave, 182
 - CoinAbcTypeFactorization, 569
- special
 - ampl_info, 137
- specialColumnCopy
 - ClpPackedMatrix, 388
- specialOptions
 - ClpModel, 335
 - OsiClpSolverInterface, 624
- specialOptions_
 - ClpDataSave, 182
 - ClpModel, 341
 - OsiClpSolverInterface, 632
- specialRowCopy
 - ClpPackedMatrix, 388
- src/AbcCommon.hpp, 639
- src/AbcDualRowDantzig.hpp, 639
- src/AbcDualRowPivot.hpp, 639
- src/AbcDualRowSteepest.hpp, 639
- src/AbcMatrix.hpp, 640
- src/AbcNonLinearCost.hpp, 640
- src/AbcPrimalColumnDantzig.hpp, 642
- src/AbcPrimalColumnPivot.hpp, 642
- src/AbcPrimalColumnSteepest.hpp, 643
- src/AbcSimplex.hpp, 643
- src/AbcSimplexDual.hpp, 646
- src/AbcSimplexFactorization.hpp, 647
- src/AbcSimplexPrimal.hpp, 647
- src/AbcWarmStart.hpp, 647
- src/CbcOrClpParam.hpp, 648
- src/Clp_C_Interface.h, 657
- src/Clp_ampl.h, 657
- src/ClpCholeskyBase.hpp, 674
- src/ClpCholeskyDense.hpp, 675
- src/ClpCholeskyMumps.hpp, 677
- src/ClpCholeskyTaucs.hpp, 677
- src/ClpCholeskyUfl.hpp, 677
- src/ClpCholeskyWssmp.hpp, 678
- src/ClpCholeskyWssmpKKT.hpp, 678
- src/ClpConfig.h, 678
- src/ClpConstraint.hpp, 678
- src/ClpConstraintLinear.hpp, 679
- src/ClpConstraintQuadratic.hpp, 679
- src/ClpDualRowDantzig.hpp, 679
- src/ClpDualRowPivot.hpp, 679
- src/ClpDualRowSteepest.hpp, 679
- src/ClpDummyMatrix.hpp, 680
- src/ClpDynamicExampleMatrix.hpp, 680
- src/ClpDynamicMatrix.hpp, 680
- src/ClpEventHandler.hpp, 680
- src/ClpFactorization.hpp, 681
- src/ClpGubDynamicMatrix.hpp, 681
- src/ClpGubMatrix.hpp, 681
- src/ClpHelperFunctions.hpp, 682
- src/ClpInterior.hpp, 683
- src/ClpLinearObjective.hpp, 684
- src/ClpLsqr.hpp, 684
- src/ClpMatrixBase.hpp, 684
- src/ClpMessage.hpp, 685
- src/ClpModel.hpp, 688
- src/ClpNetworkBasis.hpp, 690
- src/ClpNetworkMatrix.hpp, 691
- src/ClpNode.hpp, 691
- src/ClpNonLinearCost.hpp, 691
- src/ClpObjective.hpp, 693
- src/ClpPackedMatrix.hpp, 693
- src/ClpParameters.hpp, 694
- src/ClpPdco.hpp, 696
- src/ClpPdcoBase.hpp, 696
- src/ClpPlusMinusOneMatrix.hpp, 696
- src/ClpPredictorCorrector.hpp, 696
- src/ClpPresolve.hpp, 697
- src/ClpPrimalColumnDantzig.hpp, 697
- src/ClpPrimalColumnPivot.hpp, 697
- src/ClpPrimalColumnSteepest.hpp, 697
- src/ClpPrimalQuadraticDantzig.hpp, 698
- src/ClpQuadraticObjective.hpp, 698
- src/ClpSimplex.hpp, 698
- src/ClpSimplexDual.hpp, 699
- src/ClpSimplexNonlinear.hpp, 699
- src/ClpSimplexOther.hpp, 700
- src/ClpSimplexPrimal.hpp, 700
- src/ClpSolve.hpp, 700
- src/CoinAbcBaseFactorization.hpp, 701
- src/CoinAbcCommon.hpp, 702
- src/CoinAbcCommonFactorization.hpp, 706
- src/CoinAbcDenseFactorization.hpp, 709
- src/CoinAbcFactorization.hpp, 709
- src/CoinAbcHelperFunctions.hpp, 710
- src/Idiot.hpp, 720
- src/MyEventHandler.hpp, 720
- src/MyMessageHandler.hpp, 721
- src/OsiClp/OsiClpSolverInterface.hpp, 721
- src/config_clp_default.h, 719
- src/config_default.h, 719

- stack
 - CoinAbcStack, [537](#)
- stamp_
 - AbcWarmStart, [131](#)
- start
 - ClpConstraintQuadratic, [180](#)
 - ClpGubMatrix, [252](#)
 - CoinAbcStack, [537](#)
- startOfIterationInDual
 - ClpEventHandler, [223](#)
- startOfStatusOfProblemInDual
 - ClpEventHandler, [223](#)
- startOfStatusOfProblemInPrimal
 - ClpEventHandler, [223](#)
- start_
 - AbcMatrix3, [46](#)
 - ClpGubMatrix, [253](#)
 - ClpPackedMatrix3, [394](#)
- startAtLowerNoOther_
 - AbcSimplex.hpp, [645](#)
- startAtLowerOther_
 - AbcSimplex, [96](#)
- startAtUpperNoOther_
 - AbcSimplex, [96](#)
- startAtUpperOther_
 - AbcSimplex, [96](#)
- startCheck
 - ClpSimplexProgress, [508](#)
- startColumn
 - ClpDynamicMatrix, [215](#)
 - ClpGubDynamicMatrix, [239](#)
- startColumn_
 - ClpDynamicMatrix, [220](#)
 - ClpGubDynamicMatrix, [241](#)
- startColumnBlock
 - AbcMatrix, [38](#)
- startColumnBlock_
 - AbcMatrix, [40](#)
- startColumnGen
 - ClpDynamicExampleMatrix, [204](#)
- startColumnGen_
 - ClpDynamicExampleMatrix, [205](#)
- startColumnL
 - CoinAbcTypeFactorization, [552](#)
- startColumnL_
 - CoinAbcTypeFactorization, [573](#)
- startColumnLAddress_
 - CoinAbcTypeFactorization, [567](#)
- startColumnR
 - CoinAbcTypeFactorization, [555](#)
- startColumnRAddress_
 - CoinAbcTypeFactorization, [568](#)
- startColumnU
 - CoinAbcTypeFactorization, [556](#)
- startColumnU_
 - CoinAbcTypeFactorization, [572](#)
- startColumnUAddress_
 - CoinAbcTypeFactorization, [566](#)
- startElements_
 - blockStruct, [138](#)
 - blockStruct3, [138](#)
- startFastDual
 - OsiClpSolverInterface, [613](#)
- startFastDual2
 - ClpSimplex, [459](#)
- startFixed_
 - AbcSimplex, [96](#)
- startFraction
 - AbcMatrix, [37](#)
 - ClpMatrixBase, [299](#)
- startFraction_
 - AbcMatrix, [40](#)
 - ClpMatrixBase, [301](#)
- startIndices_
 - blockStruct, [138](#)
 - blockStruct3, [138](#)
- startNegative
 - ClpPlusMinusOneMatrix, [408](#)
- startNegative_
 - ClpPlusMinusOneMatrix, [409](#)
- startOther_
 - AbcSimplex, [96](#)
- startPermanentArrays
 - ClpModel, [336](#)
 - ClpSimplex, [470](#)
- startPositive
 - ClpPlusMinusOneMatrix, [408](#)
- startPositive_
 - ClpPlusMinusOneMatrix, [409](#)
- startRowL
 - CoinAbcTypeFactorization, [552](#)
- startRowL_
 - CoinAbcTypeFactorization, [573](#)
- startRowLAddress_
 - CoinAbcTypeFactorization, [567](#)
- startRowU_
 - CoinAbcTypeFactorization, [571](#)
- startRowUAddress_
 - CoinAbcTypeFactorization, [566](#)
- startSet_
 - ClpDynamicMatrix, [220](#)
- startSets
 - ClpDynamicMatrix, [214](#)
- startingDepth_
 - ClpNodeStuff, [365](#)
- startingTheta
 - ClpSimplexOther::parametricsData, [635](#)
- starts

- ampl_info, 136
- CoinAbcAnyFactorization, 524
- CoinAbcTypeFactorization, 555
- startup
 - AbcSimplex, 90
 - ClpSimplex, 460
- startupSolve
 - AbcSimplexDual, 108
 - ClpSimplexDual, 494
- state_
 - CoinAbcTypeFactorization, 577
- stateDualColumn_
 - AbcSimplex, 95
- stateOfIteration_
 - AbcSimplex, 94
- stateOfProblem
 - AbcSimplex, 82
- stateOfProblem_
 - AbcSimplex, 96
- stateOfSearch_
 - ClpNodeStuff, 365
- statistics
 - ClpPresolve, 417
- Status
 - AbcSimplex, 74
 - ClpSimplex, 452
- status
 - AbcSimplexFactorization, 115
 - ClpCholeskyBase, 151
 - ClpFactorization, 230
 - ClpModel, 322
 - CoinAbcAnyFactorization, 522
 - CoinAbcThreadInfo, 539
- status_
 - ClpCholeskyBase, 154
 - ClpDynamicMatrix, 218
 - ClpGubMatrix, 254
 - ClpModel, 340
 - ClpNode, 358
 - CoinAbcAnyFactorization, 529
- statusArray
 - AbcNonLinearCost, 50
 - ClpModel, 331
 - ClpNode, 357
 - ClpNonLinearCost, 371
- statusCopy
 - ClpModel, 331
- statusExists
 - ClpModel, 331
- statusOfProblem
 - ClpSimplex, 460
- statusOfProblemInDual
 - AbcSimplexDual, 107
 - ClpSimplexDual, 493
- statusOfProblemInPrimal
 - AbcSimplexPrimal, 122
 - ClpSimplexNonlinear, 496
 - ClpSimplexPrimal, 504
- StdVectorDouble
 - MyMessageHandler.hpp, 721
- stepLength
 - ClpLinearObjective, 281
 - ClpObjective, 373
 - ClpQuadraticObjective, 434
- stepLength_
 - ClpInterior, 274
- StepTol
 - Options, 592
- stopFastDual
 - OsiClpSolverInterface, 613
- stopFastDual2
 - ClpSimplex, 459
- stopPermanentArrays
 - ClpModel, 336
- storeFT
 - CoinAbcTypeFactorization, 562
- strParam_
 - ClpModel, 342
- stringValue
 - CbcOrClpParam, 145
- strongBranching
 - AbcSimplexDual, 105
 - ClpSimplex, 459
 - ClpSimplexDual, 492
- stuff
 - CoinAbcThreadInfo, 539
- stuff_
 - OsiClpSolverInterface, 630
- subsetClone
 - ClpGubMatrix, 251
 - ClpLinearObjective, 282
 - ClpMatrixBase, 298
 - ClpNetworkMatrix, 352
 - ClpObjective, 374
 - ClpPackedMatrix, 388
 - ClpPlusMinusOneMatrix, 408
 - ClpQuadraticObjective, 435
- subsetTimes2
 - ClpMatrixBase, 298
 - ClpPackedMatrix, 387
 - ClpPlusMinusOneMatrix, 407
- subsetTransposeTimes
 - AbcMatrix, 37
 - ClpDummyMatrix, 199
 - ClpGubMatrix, 249
 - ClpMatrixBase, 297
 - ClpNetworkMatrix, 352
 - ClpPackedMatrix, 386

- ClpPlusMinusOneMatrix, [407](#)
- substitution
 - ClpSolve, [516](#)
- sumDualInfeasibilities
 - ClpInterior, [267](#)
 - ClpSimplex, [462](#)
- sumDualInfeasibilities_
 - ClpDynamicMatrix, [217](#)
 - ClpGubMatrix, [253](#)
 - ClpInterior, [272](#)
 - ClpSimplex, [482](#)
- sumFakeInfeasibilities_
 - AbcSimplex, [93](#)
- sumInfeasibilities
 - AbcNonLinearCost, [50](#)
 - ClpNode, [357](#)
 - ClpNonLinearCost, [370](#)
- sumInfeasibilities_
 - ClpNode, [358](#)
- sumNonBasicCosts_
 - AbcSimplex, [93](#)
- sumOfRelaxedDualInfeasibilities
 - ClpSimplex, [462](#)
- sumOfRelaxedDualInfeasibilities_
 - ClpDynamicMatrix, [217](#)
 - ClpGubMatrix, [253](#)
 - ClpSimplex, [482](#)
- sumOfRelaxedPrimalInfeasibilities
 - ClpSimplex, [462](#)
- sumOfRelaxedPrimalInfeasibilities_
 - ClpDynamicMatrix, [217](#)
 - ClpGubMatrix, [253](#)
 - ClpSimplex, [482](#)
- sumPrimalInfeasibilities
 - ClpInterior, [267](#)
 - ClpSimplex, [462](#)
- sumPrimalInfeasibilities_
 - ClpDynamicMatrix, [217](#)
 - ClpGubMatrix, [253](#)
 - ClpInterior, [272](#)
 - ClpSimplex, [482](#)
- sumSquared
 - IdiotResult, [586](#)
- superBasic
 - AbcSimplex, [74](#)
 - ClpSimplex, [452](#)
- swap
 - AbcSimplex, [89](#)
- swapDualStuff
 - AbcSimplex, [89](#)
- swapFactorization
 - AbcSimplex, [76](#)
 - ClpSimplex, [458](#)
- swapModelPtr
 - OsiClpSolverInterface, [624](#)
- swapOne
 - AbcMatrix3, [45](#)
 - ClpPackedMatrix3, [394](#)
- swapPrimalStuff
 - AbcSimplex, [89](#)
- swapRowScale
 - ClpModel, [328](#)
- swapScaledMatrix
 - ClpModel, [330](#)
- swappedAlgorithm_
 - AbcSimplex, [97](#)
- switchOffCheck
 - ClpDynamicMatrix, [217](#)
 - ClpGubMatrix, [253](#)
- switchOffSprint
 - AbcPrimalColumnPivot, [56](#)
 - ClpPrimalColumnPivot, [423](#)
 - ClpPrimalColumnSteepest, [429](#)
- symbolic
 - ClpCholeskyBase, [150](#)
 - ClpCholeskyDense, [158](#)
 - ClpCholeskyMumps, [162](#)
 - ClpCholeskyTaucs, [164](#)
 - ClpCholeskyUfl, [166](#)
 - ClpCholeskyWssmp, [168](#)
 - ClpCholeskyWssmpKKT, [170](#)
- symbolic1
 - ClpCholeskyBase, [153](#)
- symbolic2
 - ClpCholeskyBase, [153](#)
- synchronize
 - AbcSimplexFactorization, [117](#)
 - ClpGubDynamicMatrix, [238](#)
 - ClpGubMatrix, [251](#)
- synchronizeModel
 - OsiClpSolverInterface, [626](#)
- TEST_INT_NONZERO
 - CoinAbcCommon.hpp, [704](#)
- TRY_ABC_GUS
 - AbcSimplex.hpp, [644](#)
- takeOutOfUseful
 - AbcMatrix, [33](#)
- targetGap_
 - ClpInterior, [275](#)
- tempArray_
 - AbcSimplex, [97](#)
- tentativeTheta
 - dualColumnResult, [579](#)
- theta
 - ClpEventHandler, [223](#)
 - ClpSimplex, [473](#)
 - dualColumnResult, [579](#)

- theta_
 - AbcSimplexPrimal::pivotStruct, [637](#)
 - ClpSimplex, [481](#)
- thisPivotValue
 - dualColumnResult, [579](#)
- thruThis
 - dualColumnResult, [579](#)
- tightenBounds
 - OsiClpSolverInterface, [625](#)
- tightenIntegerBounds
 - ClpSimplexOther, [500](#)
- tightenPrimalBounds
 - AbcSimplex, [76](#)
 - ClpSimplex, [458](#)
- timeToRefactorize
 - AbcSimplexFactorization, [115](#)
 - ClpFactorization, [231](#)
- times
 - ClpDummyMatrix, [199](#)
 - ClpDynamicMatrix, [212](#)
 - ClpGubDynamicMatrix, [238](#)
 - ClpMatrixBase, [296](#), [297](#)
 - ClpModel, [335](#)
 - ClpNetworkMatrix, [351](#)
 - ClpPackedMatrix, [385](#)
 - ClpPlusMinusOneMatrix, [406](#)
- timesFlagged
 - ClpSimplexProgress, [509](#)
- timesIncludingSlacks
 - AbcMatrix, [34](#)
- timesModifyExcludingSlacks
 - AbcMatrix, [34](#)
- timesModifyIncludingSlacks
 - AbcMatrix, [34](#)
- toIndex_
 - ClpDynamicMatrix, [218](#)
 - ClpGubMatrix, [255](#)
- toLongArray
 - CoinAbcTypeFactorization, [556](#)
- totalElements_
 - CoinAbcTypeFactorization, [569](#)
- totalThru
 - dualColumnResult, [579](#)
- translate
 - AbcSimplex, [84](#)
- transposeTimes
 - AbcMatrix, [37](#)
 - AbcMatrix2, [42](#)
 - AbcMatrix3, [45](#)
 - ClpDummyMatrix, [199](#)
 - ClpGubMatrix, [249](#)
 - ClpMatrixBase, [297](#)
 - ClpModel, [335](#)
 - ClpNetworkMatrix, [351](#), [352](#)
 - ClpPackedMatrix, [385](#), [386](#)
 - ClpPackedMatrix2, [391](#)
 - ClpPackedMatrix3, [393](#)
 - ClpPlusMinusOneMatrix, [406](#), [407](#)
- transposeTimes2
 - AbcMatrix3, [45](#)
 - ClpMatrixBase, [298](#)
 - ClpPackedMatrix, [387](#)
 - ClpPackedMatrix3, [393](#)
 - ClpPlusMinusOneMatrix, [407](#)
 - ClpPrimalColumnSteepest, [428](#)
- transposeTimesAll
 - AbcMatrix, [35](#)
- transposeTimesBasic
 - AbcMatrix, [35](#)
- transposeTimesByColumn
 - ClpPackedMatrix, [386](#)
- transposeTimesByRow
 - ClpGubMatrix, [249](#)
 - ClpPackedMatrix, [386](#)
 - ClpPlusMinusOneMatrix, [407](#)
- transposeTimesNonBasic
 - AbcMatrix, [34](#), [35](#)
- transposeTimesSubset
 - ClpPackedMatrix, [386](#)
- treeStatus
 - ClpEventHandler, [223](#)
- trueNetwork
 - ClpNetworkMatrix, [352](#)
- trueNetwork_
 - ClpNetworkMatrix, [353](#)
- trueSequenceIn_
 - ClpMatrixBase, [302](#)
- trueSequenceOut_
 - ClpMatrixBase, [302](#)
- trustedUserPointer_
 - ClpModel, [340](#)
- twiddleBtranFactor1
 - CoinAbcCommonFactorization.hpp, [707](#)
- twiddleBtranFactor2
 - CoinAbcCommonFactorization.hpp, [708](#)
- twiddleBtranFullFactor1
 - CoinAbcCommonFactorization.hpp, [708](#)
- twiddleFactor1S
 - CoinAbcCommonFactorization.hpp, [707](#)
- twiddleFactor2S
 - CoinAbcCommonFactorization.hpp, [707](#)
- twiddleFtranFTFactor1
 - CoinAbcCommonFactorization.hpp, [707](#)
- twiddleFtranFTFactor2
 - CoinAbcCommonFactorization.hpp, [708](#)
- twiddleFtranFactor1
 - CoinAbcCommonFactorization.hpp, [707](#)
- twiddleFtranFactor2

- CoinAbcCommonFactorization.hpp, 707
- type
 - AbcDualRowPivot, 21
 - AbcPrimalColumnPivot, 56
 - CbcOrClpParam, 146
 - ClpCholeskyBase, 153
 - ClpConstraint, 173
 - ClpDualRowPivot, 190
 - ClpMatrixBase, 298
 - ClpObjective, 374
 - ClpPdcoBase, 398
 - ClpPrimalColumnPivot, 423
- type_
 - AbcDualRowPivot, 22
 - AbcPrimalColumnPivot, 56
 - ClpCholeskyBase, 154
 - ClpConstraint, 174
 - ClpDualRowPivot, 190
 - ClpMatrixBase, 301
 - ClpObjective, 375
 - ClpPdcoBase, 399
 - ClpPrimalColumnPivot, 424
- typeCall
 - ClpTrustedData, 517
- typeExtraInformation_
 - AbcWarmStart, 130
- typeOfDisaster
 - ClpDisasterHandler, 184
 - OsiClpDisasterHandler, 595
- typeOfFactorization
 - AbcSimplexFactorization, 117
- typeStruct
 - ClpTrustedData, 517
- UNROLL_GATHER
 - CoinAbcHelperFunctions.hpp, 715
- UNROLL_SCATTER
 - CoinAbcHelperFunctions.hpp, 714
- USE_TEST_INT_ZERO
 - CoinAbcCommon.hpp, 704
- unPerturb
 - AbcSimplexPrimal, 123
 - ClpSimplexPrimal, 505
- unboundedRay
 - ClpModel, 331
- unflag
 - AbcSimplexPrimal, 123
 - ClpSimplexPrimal, 505
- unmarkHotStart
 - OsiClpSolverInterface, 613
- unpack
 - AbcMatrix, 34
 - AbcSimplex, 80
 - ClpDummyMatrix, 198
 - ClpGubMatrix, 248
 - ClpMatrixBase, 294
 - ClpNetworkMatrix, 350
 - ClpPackedMatrix, 384
 - ClpPlusMinusOneMatrix, 405
 - ClpSimplex, 465
 - CoinAbcTypeFactorization, 559
- unpackPacked
 - ClpDummyMatrix, 198
 - ClpGubMatrix, 248
 - ClpMatrixBase, 294
 - ClpNetworkMatrix, 350
 - ClpPackedMatrix, 384
 - ClpPlusMinusOneMatrix, 405
 - ClpSimplex, 465
- unrollWeights
 - AbcPrimalColumnSteepest, 60
 - ClpDualRowPivot, 189
 - ClpDualRowSteepest, 193
 - ClpPrimalColumnSteepest, 428
- unscale
 - ClpModel, 328
- unscaledChangesOffset
 - ClpSimplexOther::parametricsData, 636
- unsetFlagged
 - ClpDynamicExampleMatrix, 205
 - ClpDynamicMatrix, 215
 - ClpGubDynamicMatrix, 239
- unsetFlaggedSlack
 - ClpDynamicMatrix, 214
- upPseudo_
 - ClpNodeStuff, 363
- upRange
 - OsiClpSolverInterface, 628
- update
 - ClpNodeStuff, 363
- updateDualsInDual
 - ClpEventHandler, 223
- updateColumn
 - AbcSimplexFactorization, 113
 - ClpFactorization, 228
 - ClpNetworkBasis, 344
 - CoinAbcAnyFactorization, 527
 - CoinAbcDenseFactorization, 535
 - CoinAbcTypeFactorization, 558
- updateColumnCpu
 - AbcSimplexFactorization, 114
 - CoinAbcAnyFactorization, 528
 - CoinAbcTypeFactorization, 558
- updateColumnFT
 - AbcSimplexFactorization, 113
 - ClpFactorization, 228
 - CoinAbcAnyFactorization, 527
 - CoinAbcDenseFactorization, 534, 535

- CoinAbcTypeFactorization, [557](#), [558](#)
- updateColumnFTPPart1
 - AbcSimplexFactorization, [113](#)
 - CoinAbcAnyFactorization, [527](#)
 - CoinAbcDenseFactorization, [534](#)
 - CoinAbcTypeFactorization, [557](#)
- updateColumnFTPPart2
 - AbcSimplexFactorization, [113](#)
 - CoinAbcAnyFactorization, [527](#)
 - CoinAbcDenseFactorization, [534](#)
 - CoinAbcTypeFactorization, [557](#)
- updateColumnForDebug
 - ClpFactorization, [229](#)
- updateColumnL
 - CoinAbcTypeFactorization, [562](#)
- updateColumnLDense
 - CoinAbcTypeFactorization, [562](#)
- updateColumnLDensish
 - CoinAbcTypeFactorization, [562](#)
- updateColumnLSparse
 - CoinAbcTypeFactorization, [562](#)
- updateColumnPFI
 - CoinAbcTypeFactorization, [563](#)
- updateColumnR
 - CoinAbcTypeFactorization, [562](#)
- updateColumnTranspose
 - AbcSimplexFactorization, [113](#)
 - ClpFactorization, [229](#)
 - ClpNetworkBasis, [344](#), [345](#)
 - CoinAbcAnyFactorization, [527](#)
 - CoinAbcDenseFactorization, [535](#)
 - CoinAbcTypeFactorization, [558](#)
- updateColumnTransposeCpu
 - AbcSimplexFactorization, [114](#)
 - CoinAbcAnyFactorization, [528](#)
 - CoinAbcTypeFactorization, [558](#)
- updateColumnTransposeL
 - CoinAbcTypeFactorization, [564](#)
- updateColumnTransposeLByRow
 - CoinAbcTypeFactorization, [564](#)
- updateColumnTransposeLDensish
 - CoinAbcTypeFactorization, [564](#)
- updateColumnTransposeLSparse
 - CoinAbcTypeFactorization, [564](#)
- updateColumnTransposePFI
 - CoinAbcTypeFactorization, [563](#)
- updateColumnTransposeR
 - CoinAbcTypeFactorization, [564](#)
- updateColumnTransposeRDensish
 - CoinAbcTypeFactorization, [564](#)
- updateColumnTransposeRSparse
 - CoinAbcTypeFactorization, [564](#)
- updateColumnTransposeU
 - CoinAbcTypeFactorization, [563](#)
- updateColumnTransposeUByColumn
 - CoinAbcTypeFactorization, [563](#)
- updateColumnTransposeUDensish
 - CoinAbcTypeFactorization, [563](#)
- updateColumnTransposeUSparse
 - CoinAbcTypeFactorization, [563](#)
- updateColumnU
 - CoinAbcTypeFactorization, [562](#)
- updateColumnUDense
 - CoinAbcTypeFactorization, [563](#)
- updateColumnUDensish
 - CoinAbcTypeFactorization, [563](#)
- updateColumnUSparse
 - CoinAbcTypeFactorization, [563](#)
- updateDense
 - ClpCholeskyBase, [154](#)
- updateDualsInDual
 - AbcSimplexDual, [106](#)
 - ClpSimplexDual, [492](#)
- updateDualsInValuesPass
 - ClpSimplexDual, [492](#)
- updateFullColumn
 - AbcSimplexFactorization, [114](#)
 - CoinAbcAnyFactorization, [527](#)
 - CoinAbcDenseFactorization, [535](#)
 - CoinAbcTypeFactorization, [558](#)
- updateFullColumnTranspose
 - AbcSimplexFactorization, [114](#)
 - CoinAbcAnyFactorization, [528](#)
 - CoinAbcDenseFactorization, [535](#)
 - CoinAbcTypeFactorization, [558](#)
- updateMinorCandidate
 - AbcSimplexPrimal, [122](#)
- updatePartialUpdate
 - AbcSimplexPrimal, [122](#)
 - CoinAbcTypeFactorization, [557](#)
- updatePivot
 - ClpDynamicMatrix, [212](#)
 - ClpGubDynamicMatrix, [238](#)
 - ClpGubMatrix, [250](#)
 - ClpMatrixBase, [296](#)
- updatePrimalSolution
 - AbcDualRowDantzig, [17](#)
 - AbcDualRowPivot, [20](#)
 - AbcDualRowSteepest, [25](#)
 - AbcSimplexDual, [106](#)
 - ClpDualRowDantzig, [186](#)
 - ClpDualRowPivot, [189](#)
 - ClpDualRowSteepest, [193](#)
- updatePrimalSolutionAndWeights
 - AbcDualRowPivot, [20](#)
 - AbcDualRowSteepest, [25](#)
- updatePrimalsInPrimal
 - AbcSimplexPrimal, [121](#), [122](#)

- ClpSimplexPrimal, 504
- updateSolution
 - ClpPredictorCorrector, 411
- updateTwoColumnsFT
 - AbcSimplexFactorization, 113
 - ClpFactorization, 229
 - CoinAbcAnyFactorization, 527
 - CoinAbcDenseFactorization, 535
 - CoinAbcTypeFactorization, 558
- updateTwoColumnsUDensish
 - CoinAbcTypeFactorization, 563
- updateWeights
 - AbcDualRowDantzig, 17
 - AbcDualRowPivot, 20
 - AbcDualRowSteepest, 24
 - AbcPrimalColumnPivot, 55
 - AbcPrimalColumnSteepest, 60
 - AbcSimplexFactorization, 114
 - ClpDualRowDantzig, 186
 - ClpDualRowPivot, 189
 - ClpDualRowSteepest, 192
 - ClpPrimalColumnPivot, 422
 - ClpPrimalColumnSteepest, 428
 - CoinAbcAnyFactorization, 528
 - CoinAbcDenseFactorization, 535
 - CoinAbcTypeFactorization, 558
- updateWeights1
 - AbcDualRowDantzig, 17
 - AbcDualRowPivot, 20
 - AbcDualRowSteepest, 24
- updateWeights2
 - AbcDualRowDantzig, 17
 - AbcDualRowPivot, 20
 - AbcDualRowSteepest, 25
- updateWeightsOnly
 - AbcDualRowDantzig, 17
 - AbcDualRowPivot, 20
 - AbcDualRowSteepest, 24
- upper
 - AbcSimplex, 88
 - ClpGubMatrix, 252
 - ClpNonLinearCost, 370
 - ClpSimplex, 472
- upperFake
 - AbcSimplex, 74
 - ClpSimplex, 452
- upper_
 - ClpGubMatrix, 254
 - ClpInterior, 273
 - ClpNode, 359
 - ClpSimplex, 483
- upperActive
 - ClpSimplexOther::parametricsData, 636
- upperAddress
 - AbcSimplex, 88
 - ClpSimplex, 472
- upperBasic
 - AbcSimplex, 85
- upperBasic_
 - AbcSimplex, 99
- upperBound
 - ClpInterior, 271
- upperChange
 - ClpSimplexOther::parametricsData, 635
- upperCoefficient
 - ClpSimplexOther::parametricsData, 636
- upperColumn
 - ClpGubDynamicMatrix, 240
- upperColumn_
 - ClpGubDynamicMatrix, 242
- upperGap
 - ClpSimplexOther::parametricsData, 636
- upperIn_
 - AbcSimplexPrimal::pivotStruct, 637
 - ClpSimplex, 481
- upperList
 - ClpSimplexOther::parametricsData, 635
- upperOut_
 - AbcSimplexPrimal::pivotStruct, 637
 - ClpSimplex, 482
- upperRegion
 - AbcSimplex, 84, 85
 - ClpSimplex, 469, 470
- upperSaved_
 - AbcSimplex, 99
- upperSet
 - ClpDynamicMatrix, 216
 - ClpGubDynamicMatrix, 240
- upperSet_
 - ClpDynamicMatrix, 218
 - ClpGubDynamicMatrix, 242
- upperSlack_
 - ClpInterior, 276
- upperTheta
 - AbcSimplex, 83
- upperTheta_
 - AbcSimplex, 101
- useBarrier
 - ClpSolve, 513
- useBarrierNoCross
 - ClpSolve, 513
- useDual
 - ClpSolve, 513
- usePrimal
 - ClpSolve, 513
- usePrimalorSprint
 - ClpSolve, 513
- useEffectiveRhs

- ClpGubDynamicMatrix, [238](#)
- ClpGubMatrix, [250](#)
- ClpMatrixBase, [298](#)
- ClpPackedMatrix, [387](#)
- useThru
 - dualColumnResult, [579](#)
- usefulArray
 - AbcSimplex, [77](#)
- usefulArray_
 - AbcSimplex, [100](#)
- usefullInfo
 - AbcMatrix2, [42](#)
 - ClpPackedMatrix2, [391](#)
- userPointer_
 - ClpModel, [340](#)
- usingFT
 - AbcSimplexFactorization, [114](#)
- VALUES_PASS
 - AbcSimplex.hpp, [646](#)
- VALUES_PASS2
 - AbcSimplex.hpp, [646](#)
- validate
 - AbcNonLinearCost, [51](#)
 - ClpNonLinearCost, [371](#)
- value
 - ClpHashValue::CoinHashLink, [578](#)
- valueIn_
 - AbcSimplexPrimal::pivotStruct, [637](#)
 - ClpSimplex, [481](#)
- valueIncomingDual
 - AbcSimplex, [83](#)
 - ClpSimplex, [468](#)
- valueOut
 - ClpSimplex, [471](#)
- valueOut_
 - AbcSimplexPrimal::pivotStruct, [637](#)
 - ClpSimplex, [481](#)
- valuesPass_
 - AbcSimplexPrimal::pivotStruct, [638](#)
- wVec_
 - ClpInterior, [278](#)
- wait
 - Options, [593](#)
- wantToGoDense
 - CoinAbcTypeFactorization, [561](#)
- wantsSpecialColumnCopy
 - ClpPackedMatrix, [387](#)
- wantsTableauColumn
 - CoinAbcAnyFactorization, [525](#)
 - CoinAbcTypeFactorization, [557](#)
- way
 - ClpNode, [357](#)
- way_
 - ClpSimplexProgress, [510](#)
- weight
 - ClpGubMatrix, [252](#)
- weighted
 - IdiotResult, [586](#)
- weights
 - AbcDualRowSteepest, [26](#)
- weights_
 - ClpNode, [358](#)
- whatNext
 - AbcSimplexDual, [107](#)
- whatsChanged
 - ClpModel, [332](#)
- whatsChanged_
 - ClpModel, [340](#)
- whereFrom
 - OsiClpDisasterHandler, [595](#)
- whereFrom_
 - OsiClpDisasterHandler, [596](#)
- whereUsed
 - CbcOrClpParam, [146](#)
- whichColumn_
 - ClpNodeStuff, [364](#)
- whichDense_
 - ClpCholeskyBase, [156](#)
- whichParam
 - CbcOrClpParam.hpp, [657](#)
- whichRange_
 - OsiClpSolverInterface, [632](#)
- whichRow_
 - ClpNodeStuff, [364](#)
- whichSet
 - ClpDynamicMatrix, [217](#)
 - ClpGubDynamicMatrix, [241](#)
- whileIterating
 - AbcSimplexPrimal, [121](#)
 - ClpSimplexDual, [492](#)
 - ClpSimplexNonlinear, [496](#)
 - ClpSimplexPrimal, [504](#)
- whileIterating2
 - AbcSimplexDual, [105](#)
- whileIterating3
 - AbcSimplexDual, [106](#)
- whileIteratingParallel
 - AbcSimplexDual, [105](#)
- whileIteratingSerial
 - AbcSimplexDual, [105](#)
- work
 - ClpCholeskyDenseC, [160](#)
- work_
 - AbcMatrix2, [43](#)
 - ClpPackedMatrix2, [392](#)
- workArea
 - CoinAbcAnyFactorization, [524](#)

- workArea2_
 - CoinAbcTypeFactorization, [573](#)
- workArea2Address_
 - CoinAbcTypeFactorization, [568](#)
- workArea_
 - CoinAbcAnyFactorization, [530](#)
 - CoinAbcTypeFactorization, [573](#)
- workAreaAddress_
 - CoinAbcTypeFactorization, [568](#)
- workArray_
 - ClpInterior, [277](#)
- workDouble
 - ClpCholeskyBase, [152](#)
- workDouble_
 - ClpCholeskyBase, [155](#)
- workInteger_
 - ClpCholeskyBase, [156](#)
- worstComplementarity_
 - ClpInterior, [272](#)
- worstDirectionAccuracy_
 - ClpInterior, [276](#)
- writeAmpl
 - Clp_ampl.h, [657](#)
- writeBasis
 - ClpSimplex, [458](#)
 - ClpSimplexOther, [499](#)
- writeLp
 - OsiClpSolverInterface, [623](#)
- writeMps
 - ClpDynamicMatrix, [213](#)
 - ClpModel, [320](#)
 - OsiClpSolverInterface, [623](#)
- writeMpsNative
 - OsiClpSolverInterface, [623](#)
- ws_
 - OsiClpSolverInterface, [629](#)
- x0min
 - Options, [592](#)
- x_
 - ClpInterior, [273](#)
- xsize_
 - ClpInterior, [272](#)
- y_
 - ClpInterior, [273](#)
- z0min
 - Options, [592](#)
- zVec_
 - ClpInterior, [278](#)
- zap
 - ClpNodeStuff, [362](#)
- zapCosts
 - AbcNonLinearCost, [49](#)
 - ClpNonLinearCost, [369](#)
- zeroFactorizationTolerance_
 - ClpDataSave, [182](#)
- zeroSimplexTolerance_
 - ClpDataSave, [182](#)
- zeroTolerance
 - AbcSimplexFactorization, [115](#)
 - ClpFactorization, [230](#)
 - ClpSimplex, [467](#)
 - CoinAbcAnyFactorization, [524](#)
- zeroTolerance_
 - AbcTolerancesEtc, [125](#)
 - ClpSimplex, [480](#)
 - CoinAbcAnyFactorization, [528](#)
- zeros
 - ClpPackedMatrix, [387](#)
- zsize_
 - ClpInterior, [272](#)