

Osi
0.107

Generated by Doxygen 1.8.9.1

Thu Oct 8 2015 22:38:27

Contents

1	Deprecated List	1
2	Namespace Index	3
2.1	Namespace List	3
3	Hierarchical Index	5
3.1	Class Hierarchy	5
4	Class Index	13
4.1	Class List	13
5	File Index	15
5.1	File List	15
6	Namespace Documentation	17
6.1	OsiUnitTest Namespace Reference	17
6.1.1	Detailed Description	18
6.1.2	Function Documentation	18
6.1.2.1	failureMessage	18
6.1.2.2	failureMessage	18
6.1.2.3	failureMessage	18
6.1.2.4	failureMessage	18
6.1.2.5	testingMessage	18
6.1.2.6	equivalentVectors	19
6.1.2.7	compareProblems	19
6.1.2.8	isEquivalent	19
6.1.2.9	processParameters	19
6.1.3	Variable Documentation	19
6.1.3.1	verbosity	19
6.1.3.2	haltonerror	19

6.1.3.3	outcomes	20
7	Class Documentation	21
7.1	OsiSolverInterface::ApplyCutsReturnCode Class Reference	21
7.1.1	Detailed Description	22
7.2	OsiCuts::const_iterator Class Reference	22
7.2.1	Detailed Description	22
7.3	glp_prob Struct Reference	22
7.3.1	Detailed Description	22
7.4	OsiCuts::iterator Class Reference	22
7.4.1	Detailed Description	23
7.5	OsiAuxInfo Class Reference	23
7.5.1	Detailed Description	23
7.6	OsiBabSolver Class Reference	23
7.6.1	Detailed Description	25
7.6.2	Member Function Documentation	26
7.6.2.1	setSolution	26
7.6.2.2	hasSolution	26
7.6.2.3	setSolverType	26
7.6.2.4	solverType	26
7.6.2.5	extraCharacteristics	26
7.6.2.6	setExtraCharacteristics	27
7.6.3	Member Data Documentation	27
7.6.3.1	solverType_	27
7.6.3.2	extraCharacteristics_	27
7.7	OsiBranchingInformation Class Reference	27
7.7.1	Detailed Description	29
7.7.2	Member Data Documentation	29
7.7.2.1	stateOfSearch_	29
7.8	OsiBranchingObject Class Reference	29
7.8.1	Detailed Description	31
7.8.2	Member Function Documentation	31
7.8.2.1	branch	31
7.8.2.2	branch	31
7.8.3	Member Data Documentation	31
7.8.3.1	branchIndex_	31
7.9	OsiChooseStrong Class Reference	32

7.9.1	Detailed Description	33
7.9.2	Member Function Documentation	33
7.9.2.1	setupList	33
7.9.2.2	doStrongBranching	33
7.10	OsiChooseVariable Class Reference	34
7.10.1	Detailed Description	37
7.10.2	Member Function Documentation	37
7.10.2.1	setupList	37
7.11	OsiColCut Class Reference	37
7.11.1	Detailed Description	38
7.11.2	Member Function Documentation	39
7.11.2.1	consistent	39
7.11.2.2	consistent	39
7.11.2.3	infeasible	39
7.11.2.4	violated	39
7.11.3	Friends And Related Function Documentation	39
7.11.3.1	OsiColCutUnitTest	39
7.12	OsiCpxSolverInterface Class Reference	40
7.12.1	Detailed Description	46
7.12.2	Member Enumeration Documentation	46
7.12.2.1	keepCachedFlag	46
7.12.3	Member Function Documentation	47
7.12.3.1	getEmptyWarmStart	47
7.12.3.2	setWarmStart	47
7.12.3.3	getRowSense	47
7.12.3.4	getRightHandSide	48
7.12.3.5	getRowRange	48
7.12.3.6	getIterationCount	48
7.12.3.7	getDualRays	48
7.12.3.8	getPrimalRays	48
7.12.3.9	setColLower	49
7.12.3.10	setColUpper	49
7.12.3.11	setColSetBounds	49
7.12.3.12	setRowLower	49
7.12.3.13	setRowUpper	49
7.12.3.14	setRowSetBounds	50
7.12.3.15	setRowSetTypes	51

7.12.3.16 setColSolution	51
7.12.3.17 setRowPrice	51
7.12.3.18 addCol	51
7.12.3.19 addCols	52
7.12.3.20 deleteCols	52
7.12.3.21 addRow	52
7.12.3.22 addRow	52
7.12.3.23 addRows	52
7.12.3.24 addRows	52
7.12.3.25 deleteRows	52
7.12.3.26 loadProblem	53
7.12.3.27 assignProblem	53
7.12.3.28 loadProblem	53
7.12.3.29 assignProblem	54
7.12.3.30 loadProblem	54
7.12.3.31 loadProblem	54
7.12.3.32 writeMps	54
7.12.3.33 applyColCut	54
7.12.4 Friends And Related Function Documentation	54
7.12.4.1 OsiCpxSolverInterfaceUnitTest	54
7.13 OsiCut Class Reference	55
7.13.1 Detailed Description	56
7.13.2 Member Function Documentation	56
7.13.2.1 consistent	56
7.13.2.2 consistent	56
7.13.2.3 infeasible	56
7.13.2.4 violated	57
7.14 OsiCuts Class Reference	57
7.14.1 Detailed Description	59
7.14.2 Member Function Documentation	59
7.14.2.1 insertIfNotDuplicate	59
7.14.2.2 insertIfNotDuplicate	59
7.14.2.3 insert	59
7.14.2.4 insert	60
7.14.2.5 dumpCuts	60
7.14.2.6 eraseAndDumpCuts	60
7.14.3 Friends And Related Function Documentation	60

7.14.3.1	OsiCutsUnitTest	60
7.15	OsiGlpkSolverInterface Class Reference	60
7.15.1	Detailed Description	66
7.15.2	Member Enumeration Documentation	66
7.15.2.1	keepCachedFlag	66
7.15.3	Member Function Documentation	67
7.15.3.1	setHintParam	67
7.15.3.2	getEmptyWarmStart	67
7.15.3.3	setWarmStart	67
7.15.3.4	getRowSense	67
7.15.3.5	getRightHandSide	68
7.15.3.6	getRowRange	68
7.15.3.7	getIterationCount	68
7.15.3.8	getDualRays	68
7.15.3.9	getPrimalRays	68
7.15.3.10	setColLower	69
7.15.3.11	setColUpper	69
7.15.3.12	setColSetBounds	69
7.15.3.13	setRowLower	69
7.15.3.14	setRowUpper	69
7.15.3.15	setRowSetBounds	70
7.15.3.16	setRowSetTypes	71
7.15.3.17	setColSolution	71
7.15.3.18	setRowPrice	71
7.15.3.19	addCol	71
7.15.3.20	addCols	72
7.15.3.21	deleteCols	72
7.15.3.22	addRow	72
7.15.3.23	addRow	72
7.15.3.24	addRows	72
7.15.3.25	addRows	72
7.15.3.26	deleteRows	72
7.15.3.27	loadProblem	73
7.15.3.28	assignProblem	73
7.15.3.29	loadProblem	73
7.15.3.30	assignProblem	74
7.15.3.31	loadProblem	74

7.15.3.32 loadProblem	74
7.15.3.33 writeMps	74
7.15.3.34 setRowName	74
7.15.3.35 setColName	74
7.15.3.36 incrementInstanceCounter	74
7.15.3.37 decrementInstanceCounter	75
7.15.3.38 applyColCut	75
7.15.4 Friends And Related Function Documentation	75
7.15.4.1 OsiGlpkSolverInterfaceUnitTest	75
7.16 OsiGrbSolverInterface Class Reference	75
7.16.1 Detailed Description	82
7.16.2 Member Enumeration Documentation	82
7.16.2.1 keepCachedFlag	82
7.16.3 Member Function Documentation	83
7.16.3.1 setHintParam	83
7.16.3.2 getHintParam	83
7.16.3.3 getHintParam	83
7.16.3.4 getHintParam	83
7.16.3.5 getEmptyWarmStart	83
7.16.3.6 setWarmStart	84
7.16.3.7 getRowSense	84
7.16.3.8 getRightHandSide	84
7.16.3.9 getRowRange	84
7.16.3.10 getIterationCount	85
7.16.3.11 getDualRays	85
7.16.3.12 getPrimalRays	85
7.16.3.13 setColLower	85
7.16.3.14 setColUpper	85
7.16.3.15 setColSetBounds	86
7.16.3.16 setRowLower	87
7.16.3.17 setRowUpper	87
7.16.3.18 setRowSetBounds	87
7.16.3.19 setRowSetTypes	87
7.16.3.20 setColSolution	88
7.16.3.21 setRowPrice	88
7.16.3.22 addCol	88
7.16.3.23 addCols	88

7.16.3.24 deleteCols	88
7.16.3.25 addRow	88
7.16.3.26 addRow	89
7.16.3.27 addRows	89
7.16.3.28 addRows	89
7.16.3.29 deleteRows	89
7.16.3.30 loadProblem	89
7.16.3.31 assignProblem	90
7.16.3.32 loadProblem	90
7.16.3.33 assignProblem	90
7.16.3.34 loadProblem	90
7.16.3.35 loadProblem	90
7.16.3.36 writeMps	91
7.16.3.37 incrementInstanceCounter	91
7.16.3.38 decrementInstanceCounter	91
7.16.3.39 switchToLP	91
7.16.3.40 applyCuts	91
7.16.3.41 applyColCut	92
7.16.4 Friends And Related Function Documentation	92
7.16.4.1 OsiGrbSolverInterfaceUnitTest	92
7.17 OsiHotInfo Class Reference	92
7.17.1 Detailed Description	94
7.17.2 Member Function Documentation	94
7.17.2.1 updateInformation	94
7.18 OsiIntegerBranchingObject Class Reference	94
7.18.1 Detailed Description	95
7.18.2 Constructor & Destructor Documentation	95
7.18.2.1 OsiIntegerBranchingObject	95
7.18.2.2 OsiIntegerBranchingObject	95
7.18.3 Member Function Documentation	95
7.18.3.1 branch	95
7.19 OsiLotsize Class Reference	95
7.19.1 Detailed Description	97
7.19.2 Member Function Documentation	97
7.19.2.1 feasibleRegion	97
7.19.2.2 createBranch	97
7.19.2.3 resetBounds	97

7.19.2.4	findRange	97
7.20	OsiLotsizeBranchingObject Class Reference	97
7.20.1	Detailed Description	98
7.20.2	Constructor & Destructor Documentation	98
7.20.2.1	OsiLotsizeBranchingObject	98
7.20.3	Member Function Documentation	99
7.20.3.1	branch	99
7.21	OsiMskSolverInterface Class Reference	99
7.21.1	Detailed Description	106
7.21.2	Member Enumeration Documentation	107
7.21.2.1	keepCachedFlag	107
7.21.3	Member Function Documentation	107
7.21.3.1	getEmptyWarmStart	107
7.21.3.2	setWarmStart	107
7.21.3.3	getRowSense	107
7.21.3.4	getRightHandSide	108
7.21.3.5	getRowRange	108
7.21.3.6	getIterationCount	108
7.21.3.7	getDualRays	108
7.21.3.8	getPrimalRays	109
7.21.3.9	setColLower	109
7.21.3.10	setColUpper	109
7.21.3.11	setColSetBounds	109
7.21.3.12	setRowLower	109
7.21.3.13	setRowUpper	109
7.21.3.14	setRowSetBounds	110
7.21.3.15	setRowSetTypes	110
7.21.3.16	setColSolution	110
7.21.3.17	setRowPrice	110
7.21.3.18	addCol	111
7.21.3.19	addCols	111
7.21.3.20	deleteCols	111
7.21.3.21	addRow	111
7.21.3.22	addRow	111
7.21.3.23	addRows	111
7.21.3.24	addRows	111
7.21.3.25	deleteRows	112

7.21.3.26	loadProblem	112
7.21.3.27	assignProblem	112
7.21.3.28	loadProblem	112
7.21.3.29	assignProblem	113
7.21.3.30	loadProblem	113
7.21.3.31	loadProblem	113
7.21.3.32	writeMps	113
7.21.3.33	incrementInstanceCounter	113
7.21.3.34	decrementInstanceCounter	114
7.21.3.35	applyColCut	114
7.22	OsiObject Class Reference	114
7.22.1	Detailed Description	116
7.22.2	Member Function Documentation	116
7.22.2.1	infeasibility	116
7.22.2.2	feasibleRegion	116
7.22.2.3	feasibleRegion	116
7.22.2.4	createBranch	117
7.22.2.5	setWhichWay	117
7.22.2.6	whichWay	117
7.22.2.7	resetBounds	117
7.22.3	Member Data Documentation	117
7.22.3.1	infeasibility_	117
7.23	OsiObject2 Class Reference	117
7.23.1	Detailed Description	118
7.24	OsiPresolve Class Reference	118
7.24.1	Detailed Description	119
7.24.2	Member Function Documentation	120
7.24.2.1	presolvedModel	120
7.24.2.2	postsolve	120
7.24.2.3	model	121
7.24.2.4	setNonLinearValue	121
7.24.2.5	setPresolveActions	121
7.24.2.6	presolve	121
7.24.2.7	postsolve	121
7.24.2.8	gutsOfDestroy	122
7.25	OsiPseudoCosts Class Reference	122
7.25.1	Detailed Description	123

7.26	OsiRowCut Class Reference	123
7.26.1	Detailed Description	125
7.26.2	Constructor & Destructor Documentation	125
7.26.2.1	OsiRowCut	125
7.26.3	Member Function Documentation	125
7.26.3.1	consistent	125
7.26.3.2	consistent	126
7.26.3.3	infeasible	126
7.26.3.4	violated	126
7.26.4	Friends And Related Function Documentation	126
7.26.4.1	OsiRowCutUnitTest	126
7.27	OsiRowCut2 Class Reference	126
7.27.1	Detailed Description	127
7.28	OsiRowCutDebugger Class Reference	127
7.28.1	Detailed Description	129
7.28.2	Constructor & Destructor Documentation	129
7.28.2.1	OsiRowCutDebugger	129
7.28.2.2	OsiRowCutDebugger	129
7.28.3	Member Function Documentation	129
7.28.3.1	validateCuts	129
7.28.3.2	invalidCut	129
7.28.3.3	onOptimalPath	129
7.28.3.4	activate	130
7.28.3.5	activate	130
7.28.3.6	redoSolution	130
7.28.4	Friends And Related Function Documentation	130
7.28.4.1	OsiRowCutDebuggerUnitTest	130
7.29	OsiSimpleInteger Class Reference	130
7.29.1	Detailed Description	132
7.29.2	Member Function Documentation	132
7.29.2.1	feasibleRegion	132
7.29.2.2	createBranch	132
7.29.2.3	resetBounds	132
7.30	OsiSolverBranch Class Reference	132
7.30.1	Detailed Description	133
7.31	OsiSolverInterface Class Reference	133
7.31.1	Detailed Description	146

7.31.2	Member Typedef Documentation	146
7.31.2.1	OsiNameVec	146
7.31.3	Member Function Documentation	146
7.31.3.1	resolve	146
7.31.3.2	setHintParam	146
7.31.3.3	getHintParam	146
7.31.3.4	getHintParam	147
7.31.3.5	getHintParam	147
7.31.3.6	copyParameters	147
7.31.3.7	getIntegerTolerance	147
7.31.3.8	getEmptyWarmStart	147
7.31.3.9	getWarmStart	148
7.31.3.10	getPointerToWarmStart	148
7.31.3.11	setWarmStart	148
7.31.3.12	getRowSense	148
7.31.3.13	getRightHandSide	148
7.31.3.14	getRowRange	149
7.31.3.15	getObjSense	149
7.31.3.16	isInteger	149
7.31.3.17	columnType	149
7.31.3.18	getColType	149
7.31.3.19	getMutableMatrixByRow	150
7.31.3.20	getMutableMatrixByCol	150
7.31.3.21	getRowActivity	150
7.31.3.22	getDualRays	150
7.31.3.23	getPrimalRays	150
7.31.3.24	getFractionalIndices	151
7.31.3.25	setObjective	151
7.31.3.26	setObjSense	151
7.31.3.27	setColLower	151
7.31.3.28	setColLower	151
7.31.3.29	setColUpper	152
7.31.3.30	setColUpper	152
7.31.3.31	setColBounds	152
7.31.3.32	setColSetBounds	152
7.31.3.33	setRowLower	152
7.31.3.34	setRowUpper	152

7.31.3.35 setRowBounds	153
7.31.3.36 setRowSetBounds	153
7.31.3.37 setRowSetTypes	153
7.31.3.38 setColSolution	153
7.31.3.39 setRowPrice	153
7.31.3.40 reducedCostFix	154
7.31.3.41 dfItRowColName	154
7.31.3.42 getRowName	154
7.31.3.43 getRowNames	154
7.31.3.44 setRowName	154
7.31.3.45 setRowNames	154
7.31.3.46 deleteRowNames	155
7.31.3.47 getColName	155
7.31.3.48 getColNames	155
7.31.3.49 setColName	155
7.31.3.50 setColNames	155
7.31.3.51 deleteColNames	155
7.31.3.52 setRowColNames	155
7.31.3.53 setRowColNames	156
7.31.3.54 setRowColNames	156
7.31.3.55 addCol	156
7.31.3.56 addCol	156
7.31.3.57 addCol	156
7.31.3.58 addCol	156
7.31.3.59 addCols	156
7.31.3.60 addCols	157
7.31.3.61 addCols	157
7.31.3.62 deleteCols	157
7.31.3.63 addRow	157
7.31.3.64 addRow	157
7.31.3.65 addRow	157
7.31.3.66 addRow	157
7.31.3.67 addRow	158
7.31.3.68 addRows	158
7.31.3.69 addRows	158
7.31.3.70 addRows	158
7.31.3.71 addRows	158

7.31.3.72 deleteRows	158
7.31.3.73 replaceMatrixOptional	159
7.31.3.74 replaceMatrix	159
7.31.3.75 saveBaseModel	159
7.31.3.76 restoreBaseModel	159
7.31.3.77 applyCuts	159
7.31.3.78 applyRowCuts	160
7.31.3.79 applyRowCuts	160
7.31.3.80 loadProblem	160
7.31.3.81 assignProblem	160
7.31.3.82 loadProblem	161
7.31.3.83 assignProblem	161
7.31.3.84 loadProblem	161
7.31.3.85 loadProblem	162
7.31.3.86 loadFromCoinModel	162
7.31.3.87 readMps	162
7.31.3.88 readMps	162
7.31.3.89 readGMPL	162
7.31.3.90 writeMps	162
7.31.3.91 writeMpsNative	163
7.31.3.92 writeLp	163
7.31.3.93 writeLp	163
7.31.3.94 writeLpNative	163
7.31.3.95 writeLpNative	164
7.31.3.96 readLp	164
7.31.3.97 readLp	164
7.31.3.98 differentModel	164
7.31.3.99 setApplicationData	164
7.31.3.100setAuxiliaryInfo	164
7.31.3.101passInMessageHandler	164
7.31.3.102findIntegers	165
7.31.3.103findIntegersAndSOS	165
7.31.3.104addObjects	165
7.31.3.105forceFeasible	165
7.31.3.106activateRowCutDebugger	165
7.31.3.107activateRowCutDebugger	165
7.31.3.108getRowCutDebugger	165

7.31.3.10	getRowCutDebuggerAlways	166
7.31.3.11	canDoSimplexInterface	166
7.31.3.11	enableFactorization	166
7.31.3.11	disableFactorization	166
7.31.3.11	basisIsAvailable	166
7.31.3.11	getBasisStatus	167
7.31.3.11	setBasisStatus	167
7.31.3.11	getReducedGradient	167
7.31.3.11	getBInvARow	168
7.31.3.11	getBasics	168
7.31.3.11	enableSimplexInterface	168
7.31.3.12	pivot	168
7.31.3.12	primalPivotResult	168
7.31.3.12	clone	168
7.31.3.12	reset	169
7.31.3.12	applyRowCut	169
7.31.3.12	applyColCut	169
7.31.3.12	setInitialData	169
7.31.4	Friends And Related Function Documentation	169
7.31.4.1	OsiSolverInterfaceCommonUnitTest	169
7.31.4.2	OsiSolverInterfaceMpsUnitTest	169
7.31.5	Member Data Documentation	170
7.31.5.1	rowCutDebugger_	170
7.31.5.2	defaultHandler_	170
7.32	OsiSolverResult Class Reference	170
7.32.1	Detailed Description	171
7.33	OsiSOS Class Reference	171
7.33.1	Detailed Description	172
7.33.2	Constructor & Destructor Documentation	173
7.33.2.1	OsiSOS	173
7.33.3	Member Function Documentation	173
7.33.3.1	feasibleRegion	173
7.33.3.2	createBranch	173
7.33.4	Member Data Documentation	173
7.33.4.1	members_	173
7.34	OsiSOSBranchingObject Class Reference	173
7.34.1	Detailed Description	174

7.35	OsiSpxSolverInterface Class Reference	174
7.35.1	Detailed Description	179
7.35.2	Member Enumeration Documentation	179
7.35.2.1	keepCachedFlag	179
7.35.3	Member Function Documentation	180
7.35.3.1	setWarmStart	180
7.35.3.2	getRowSense	180
7.35.3.3	getRightHandSide	180
7.35.3.4	getRowRange	180
7.35.3.5	getIterationCount	180
7.35.3.6	getDualRays	181
7.35.3.7	getPrimalRays	181
7.35.3.8	setColLower	181
7.35.3.9	setColUpper	181
7.35.3.10	setRowLower	181
7.35.3.11	setRowUpper	182
7.35.3.12	setColSolution	182
7.35.3.13	setRowPrice	182
7.35.3.14	addCol	182
7.35.3.15	deleteCols	182
7.35.3.16	addRow	182
7.35.3.17	addRow	183
7.35.3.18	deleteRows	183
7.35.3.19	loadProblem	183
7.35.3.20	assignProblem	183
7.35.3.21	loadProblem	183
7.35.3.22	assignProblem	184
7.35.3.23	loadProblem	184
7.35.3.24	loadProblem	184
7.35.3.25	writeMps	184
7.35.3.26	applyColCut	184
7.35.4	Friends And Related Function Documentation	185
7.35.4.1	OsiSpxSolverInterfaceUnitTest	185
7.36	OsiTwoWayBranchingObject Class Reference	185
7.36.1	Detailed Description	185
7.36.2	Constructor & Destructor Documentation	186
7.36.2.1	OsiTwoWayBranchingObject	186

7.36.3 Member Function Documentation	186
7.36.3.1 branch	186
7.37 OsiXprSolverInterface Class Reference	186
7.37.1 Detailed Description	192
7.37.2 Member Function Documentation	192
7.37.2.1 setWarmStart	192
7.37.2.2 getRowSense	192
7.37.2.3 getRightHandSide	193
7.37.2.4 getRowRange	193
7.37.2.5 getIterationCount	193
7.37.2.6 getDualRays	193
7.37.2.7 getPrimalRays	193
7.37.2.8 setColLower	194
7.37.2.9 setColUpper	194
7.37.2.10 setColSetBounds	194
7.37.2.11 setRowLower	194
7.37.2.12 setRowUpper	194
7.37.2.13 setRowSetBounds	195
7.37.2.14 setRowSetTypes	196
7.37.2.15 setColSolution	196
7.37.2.16 setRowPrice	196
7.37.2.17 addCol	196
7.37.2.18 addCols	197
7.37.2.19 deleteCols	197
7.37.2.20 addRow	197
7.37.2.21 addRow	197
7.37.2.22 addRows	197
7.37.2.23 addRows	197
7.37.2.24 deleteRows	197
7.37.2.25 loadProblem	198
7.37.2.26 assignProblem	198
7.37.2.27 loadProblem	198
7.37.2.28 assignProblem	199
7.37.2.29 loadProblem	199
7.37.2.30 loadProblem	199
7.37.2.31 writeMps	199
7.37.2.32 incrementInstanceCounter	199

7.37.2.33	decrementInstanceCounter	199
7.37.2.34	getNumInstances	200
7.37.2.35	setLogFileName	200
7.37.2.36	applyColCut	200
7.37.3	Friends And Related Function Documentation	200
7.37.3.1	OsiXprSolverInterfaceUnitTest	200
7.38	OsiUnitTest::TestOutcome Class Reference	200
7.38.1	Detailed Description	201
7.39	OsiUnitTest::TestOutcomes Class Reference	201
7.39.1	Detailed Description	202
7.39.2	Member Function Documentation	202
7.39.2.1	add	202
7.39.2.2	getCountBySeverity	202
8	File Documentation	203
8.1	OsiRowCutDebugger.hpp File Reference	203
8.2	OsiUnitTests.hpp File Reference	203
8.2.1	Detailed Description	205
8.2.2	Macro Definition Documentation	205
8.2.2.1	OSIUNITTEST_ASSERT_SEVERITY_EXPECTED	205
8.2.2.2	OSIUNITTEST_CATCH_SEVERITY_EXPECTED	206
8.2.3	Function Documentation	206
8.2.3.1	OsiSolverInterfaceMpsUnitTest	206
8.2.3.2	OsiSolverInterfaceCommonUnitTest	206
8.2.3.3	OsiColCutUnitTest	206
8.2.3.4	OsiRowCutUnitTest	206
8.2.3.5	OsiRowCutDebuggerUnitTest	206
8.2.3.6	OsiCutsUnitTest	206
Index		207

Chapter 1

Deprecated List

Member [OsiSolverInterface::columnType](#) (bool refresh=false) const

See [getColType](#)

Chapter 2

Namespace Index

2.1 Namespace List

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

OsiUnitTest	
A namespace so we can define a few 'global' variables to use during tests	17

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

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

```
_EKKfactinfo [external]
doubleton_action::action [external]
forcing_constraint_action::action [external]
tripleton_action::action [external]
remove_fixed_action::action [external]
std::allocator< T >
OsiSolverInterface::ApplyCutsReturnCode . . . . . 21
std::array< T >
std::auto_ptr< T >
std::basic_string< Char >
    std::string
    std::wstring
std::basic_string< char >
std::basic_string< wchar_t >
std::bitset< Bits >
BitVector128 [external]
CoinAbsFltEq [external]
CoinArrayWithLength [external]
    CoinArbitraryArrayWithLength [external]
    CoinBigIndexArrayWithLength [external]
    CoinDoubleArrayWithLength [external]
    CoinFactorizationDoubleArrayWithLength [external]
    CoinFactorizationLongDoubleArrayWithLength [external]
    CoinIntArrayWithLength [external]
    CoinUnsignedIntArrayWithLength [external]
    CoinVoidStarArrayWithLength [external]
CoinBaseModel [external]
    CoinModel [external]
    CoinStructuredModel [external]
CoinBuild [external]
CoinDenseVector< T > [external]
CoinError [external]
CoinExternalVectorFirstGreater_2< class, class, class > [external]
CoinExternalVectorFirstGreater_3< class, class, class, class > [external]
```

```

CoinExternalVectorFirstLess_2< class, class, class >[external]
CoinExternalVectorFirstLess_3< class, class, class, class >[external]
CoinFactorization[external]
CoinFileIOBase[external]
    CoinFileInput[external]
    CoinFileOutput[external]
CoinFirstAbsGreater_2< class, class >[external]
CoinFirstAbsGreater_3< class, class, class >[external]
CoinFirstAbsLess_2< class, class >[external]
CoinFirstAbsLess_3< class, class, class >[external]
CoinFirstGreater_2< class, class >[external]
CoinFirstGreater_3< class, class, class >[external]
CoinFirstLess_2< class, class >[external]
CoinFirstLess_3< class, class, class >[external]
CoinLpIO::CoinHashLink[external]
CoinMpsIO::CoinHashLink[external]
CoinIndexedVector[external]
    CoinPartitionedVector[external]
CoinLpIO[external]
CoinMessageHandler[external]
CoinMessages[external]
    CoinMessage[external]
CoinModelHash[external]
CoinModelHash2[external]
CoinModelHashLink[external]
CoinModelInfo2[external]
CoinModelLink[external]
CoinModelLinkedList[external]
CoinModelTriple[external]
CoinMpsCardReader[external]
CoinMpsIO[external]
CoinOneMessage[external]
CoinOtherFactorization[external]
    CoinDenseFactorization[external]
    CoinOslFactorization[external]
    CoinSimpFactorization[external]
CoinPackedMatrix[external]
CoinPackedVectorBase[external]
    CoinPackedVector[external]
    CoinShallowPackedVector[external]
CoinPair< S, T >[external]
CoinParam[external]
CoinPrePostsolveMatrix[external]
    CoinPostsolveMatrix[external]
    CoinPresolveMatrix[external]
CoinPresolveAction[external]
    do_tighten_action[external]
    doubleton_action[external]
    drop_empty_cols_action[external]
    drop_empty_rows_action[external]
    drop_zero_coefficients_action[external]
    dupcol_action[external]
    duprow3_action[external]
    duprow_action[external]

```

```

    forcing_constraint_action[external]
    gubrow_action[external]
    implied_free_action[external]
    isolated_constraint_action[external]
    make_fixed_action[external]
    remove_dual_action[external]
    remove_fixed_action[external]
    slack_doubleton_action[external]
    slack_singleton_action[external]
    subst_constraint_action[external]
    tripleton_action[external]
    twotwo_action[external]
    useless_constraint_action[external]
CoinPresolveMonitor[external]
CoinRational[external]
CoinRelFltEq[external]
CoinSearchTreeBase[external]
    CoinSearchTree< class >[external]
CoinSearchTreeCompareBest[external]
CoinSearchTreeCompareBreadth[external]
CoinSearchTreeCompareDepth[external]
CoinSearchTreeComparePreferred[external]
CoinSearchTreeManager[external]
CoinSet[external]
    CoinSosSet[external]
CoinSnapshot[external]
CoinThreadRandom[external]
CoinTimer[external]
CoinTreeNode[external]
CoinTreeSiblings[external]
CoinTriple< S, T, U >[external]
CoinWarmStart[external]
    CoinWarmStartBasis[external]
    CoinWarmStartDual[external]
    CoinWarmStartPrimalDual[external]
    CoinWarmStartVector< T >[external]
    CoinWarmStartVector< double >[external]
    CoinWarmStartVector< U >[external]
    CoinWarmStartVectorPair< T, U >[external]
CoinWarmStartDiff[external]
    CoinWarmStartBasisDiff[external]
    CoinWarmStartDualDiff[external]
    CoinWarmStartPrimalDualDiff[external]
    CoinWarmStartVectorDiff< T >[external]
    CoinWarmStartVectorDiff< double >[external]
    CoinWarmStartVectorDiff< U >[external]
    CoinWarmStartVectorPairDiff< T, U >[external]
CoinYacc[external]
std::complex

OsiCuts::const_iterator . . . . . 22
std::basic_string< Char >::const_iterator
std::string::const_iterator
std::wstring::const_iterator
std::deque< T >::const_iterator

```

```

std::list< T >::const_iterator
std::forward_list< T >::const_iterator
std::map< K, T >::const_iterator
std::unordered_map< K, T >::const_iterator
std::multimap< K, T >::const_iterator
std::unordered_multimap< K, T >::const_iterator
std::set< K >::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::basic_string< Char >::const_reverse_iterator
std::string::const_reverse_iterator
std::wstring::const_reverse_iterator
std::deque< T >::const_reverse_iterator
std::list< T >::const_reverse_iterator
std::forward_list< T >::const_reverse_iterator
std::map< K, T >::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::set< K >::const_reverse_iterator
std::unordered_set< K >::const_reverse_iterator
std::multiset< K >::const_reverse_iterator
std::unordered_multiset< K >::const_reverse_iterator
std::vector< T >::const_reverse_iterator
std::deque< T >
dropped_zero[external]
EKKHlink[external]
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
FactorPointers[external]
std::forward_list< T >

glp_prob . . . . . 22
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::set< K >::iterator
std::unordered_set< K >::iterator

OsiCuts::iterator . . . . . 22
std::unordered_multimap< K, T >::iterator
std::unordered_map< K, T >::iterator
std::basic_string< Char >::iterator

```

std::string::iterator	
std::forward_list< T >::iterator	
std::wstring::iterator	
std::map< K, T >::iterator	
std::deque< T >::iterator	
std::list< T >::iterator	
std::multimap< K, T >::iterator	
std::unordered_multiset< K >::iterator	
std::vector< T >::iterator	
std::multiset< K >::iterator	
std::list< T >	
OsiUnitTest::TestOutcomes	201
std::list< TestOutcome >	
std::map< K, T >	
std::multimap< K, T >	
std::multiset< K >	
OsiAuxInfo	23
OsiBabSolver	23
OsiBranchingInformation	27
OsiBranchingObject	29
OsiTwoWayBranchingObject	185
OsiIntegerBranchingObject	94
OsiLotsizeBranchingObject	97
OsiSOSBranchingObject	173
OsiChooseVariable	34
OsiChooseStrong	32
OsiCut	55
OsiColCut	37
OsiRowCut	123
OsiRowCut2	126
OsiCuts	57
OsiHotInfo	92
OsiObject	114
OsiObject2	117
OsiLotsize	95
OsiSimpleInteger	130
OsiSOS	171
OsiPresolve	118
OsiPseudoCosts	122
OsiRowCutDebugger	127
OsiSolverBranch	132
OsiSolverInterface	133
OsiCpxSolverInterface	40
OsiGlpkSolverInterface	60
OsiGrbSolverInterface	75
OsiMskSolverInterface	99
OsiSpxSolverInterface	174
OsiXprSolverInterface	186
OsiSolverResult	170
presolvehlink[external]	
std::priority_queue< T >	
std::queue< T >	

```

Coin::ReferencedObject[external]
std::set< K >::reverse_iterator
std::unordered_set< K >::reverse_iterator
std::basic_string< Char >::reverse_iterator
std::multimap< K, T >::reverse_iterator
std::unordered_multiset< K >::reverse_iterator
std::vector< T >::reverse_iterator
std::wstring::reverse_iterator
std::list< T >::reverse_iterator
std::map< K, T >::reverse_iterator
std::deque< T >::reverse_iterator
std::forward_list< T >::reverse_iterator
std::string::reverse_iterator
std::multiset< K >::reverse_iterator
std::unordered_map< K, T >::reverse_iterator
std::unordered_multimap< K, T >::reverse_iterator
std::set< K >
std::smart_ptr< T >
Coin::SmartPtr< T >[external]
std::stack< T >
symrec[external]
std::system_error
OsiUnitTest::TestOutcome . . . . . 200
std::thread
std::unique_ptr< T >
std::unordered_map< K, T >
std::unordered_multimap< K, T >
std::unordered_multiset< K >
std::unordered_set< K >
std::valarray< T >
std::vector< T >
std::vector< double >
std::vector< OsiColCut * >
std::vector< OsiRowCut * >
std::vector< std::string >
std::weak_ptr< T >
K
S
T
U

```


Chapter 4

Class Index

4.1 Class List

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

OsiSolverInterface::ApplyCutsReturnCode	Internal class for obtaining status from the applyCuts method	21
OsiCuts::const_iterator	Const Iterator	22
glp_prob	22
OsiCuts::iterator	Iterator	22
OsiAuxInfo	This class allows for a more structured use of algorithmic tweaking to an OsiSolverInterface	23
OsiBabSolver	This class allows for the use of more exotic solvers e.g	23
OsiBranchingInformation	27
OsiBranchingObject	Abstract branching object base class	29
OsiChooseStrong	This class chooses a variable to branch on	32
OsiChooseVariable	This class chooses a variable to branch on	34
OsiColCut	Column Cut Class	37
OsiCpxSolverInterface	CPLEX Solver Interface	40
OsiCut	55
OsiCuts	Collections of row cuts and column cuts	57
OsiGlpkSolverInterface	60
OsiGrbSolverInterface	Gurobi Solver Interface	75
OsiHotInfo	This class contains the result of strong branching on a variable When created it stores enough information for strong branching	92
OsiIntegerBranchingObject	Simple branching object for an integer variable	94

OsiLotsize	
Lotsize class	95
OsiLotsizeBranchingObject	
Lotsize branching object	97
OsiMskSolverInterface	99
OsiObject	
Abstract base class for 'objects'	114
OsiObject2	
Define a class to add a bit of complexity to OsiObject This assumes 2 way branching	117
OsiPresolve	
OSI interface to COIN problem simplification capabilities	118
OsiPseudoCosts	
This class is the placeholder for the pseudocosts used by OsiChooseStrong	122
OsiRowCut	
Row Cut Class	123
OsiRowCut2	
Row Cut Class which refers back to row which created it	126
OsiRowCutDebugger	
Validate cuts against a known solution	127
OsiSimpleInteger	
Define a single integer class	130
OsiSolverBranch	
Solver Branch Class	132
OsiSolverInterface	
Abstract Base Class for describing an interface to a solver	133
OsiSolverResult	
Solver Result Class	170
OsiSOS	
Define Special Ordered Sets of type 1 and 2	171
OsiSOSBranchingObject	
Branching object for Special ordered sets	173
OsiSpxSolverInterface	
SoPlex Solver Interface Instantiation of OsiSpxSolverInterface for SoPlex	174
OsiTwoWayBranchingObject	
This just adds two-wayness to a branching object	185
OsiXprSolverInterface	
XPRESS-MP Solver Interface	186
OsiUnitTest::TestOutcome	
A single test outcome record	200
OsiUnitTest::TestOutcomes	
Utility class to maintain a list of test outcomes	201

Chapter 5

File Index

5.1 File List

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

config_default.h	??
config_osi_default.h	??
OsiAuxInfo.hpp	??
OsiBranchingObject.hpp	??
OsiChooseVariable.hpp	??
OsiColCut.hpp	??
OsiCollections.hpp	??
OsiConfig.h	??
OsiCpxSolverInterface.hpp	??
OsiCut.hpp	??
OsiCuts.hpp	??
OsiGlpkSolverInterface.hpp	??
OsiGrbSolverInterface.hpp	??
OsiMskSolverInterface.hpp	??
OsiPresolve.hpp	??
OsiRowCut.hpp	??
OsiRowCutDebugger.hpp	
Provides a facility to validate cut constraints to ensure that they do not cut off a given solution	203
OsiSolverBranch.hpp	??
OsiSolverInterface.hpp	??
OsiSolverParameters.hpp	??
OsiSpxSolverInterface.hpp	??
OsiUnitTests.hpp	
Utility methods for OSI unit tests	203
OsiXprSolverInterface.hpp	??

Chapter 6

Namespace Documentation

6.1 OsiUnitTest Namespace Reference

A namespace so we can define a few 'global' variables to use during tests.

Classes

- class [TestOutcome](#)
A single test outcome record.
- class [TestOutcomes](#)
Utility class to maintain a list of test outcomes.

Functions

- void [failureMessage](#) (const std::string &solverName, const std::string &message)
Print an error message.
- void [failureMessage](#) (const [OsiSolverInterface](#) &si, const std::string &message)
- void [failureMessage](#) (const std::string &solverName, const std::string &testname, const std::string &testcond)
Print an error message, specifying the test name and condition.
- void [failureMessage](#) (const [OsiSolverInterface](#) &si, const std::string &testname, const std::string &testcond)
- void [testingMessage](#) (const char *const msg)
Print a message.
- bool [equivalentVectors](#) (const [OsiSolverInterface](#) *si1, const [OsiSolverInterface](#) *si2, double tol, const double *v1, const double *v2, int size)
Utility method to check equality.
- bool [compareProblems](#) ([OsiSolverInterface](#) *osi1, [OsiSolverInterface](#) *osi2)
Compare two problems for equality.
- bool [isEquivalent](#) (const **CoinPackedVectorBase** &pv, int n, const double *fv)
Compare a packed vector with an expanded vector.
- bool [processParameters](#) (int argc, const char **argv, std::map< std::string, std::string > &parms, const std::map< std::string, int > &ignorekeywords=std::map< std::string, int >())
Process command line parameters.

Variables

- unsigned int [verbosity](#)
Verbosity level of unit tests.
- unsigned int [haltonerror](#)
Behaviour on failing a test.
- [TestOutcomes](#) *outcomes*
Test outcomes.

6.1.1 Detailed Description

A namespace so we can define a few 'global' variables to use during tests.

6.1.2 Function Documentation

6.1.2.1 void OsiUnitTest::failureMessage (const std::string & *solverName*, const std::string & *message*)

Print an error message.

Formatted as "XxxSolverInterface testing issue: message" where Xxx is the string provided as *solverName*.

Flushes std::cout before printing to std::cerr.

6.1.2.2 void OsiUnitTest::failureMessage (const OsiSolverInterface & *si*, const std::string & *message*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

6.1.2.3 void OsiUnitTest::failureMessage (const std::string & *solverName*, const std::string & *testname*, const std::string & *testcond*)

Print an error message, specifying the test name and condition.

Formatted as "XxxSolverInterface testing issue: testname failed: testcond" where Xxx is the OsiStrParam::OsiSolver↵ Name parameter of the *si*. Flushes std::cout before printing to std::cerr.

6.1.2.4 void OsiUnitTest::failureMessage (const OsiSolverInterface & *si*, const std::string & *testname*, const std::string & *testcond*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

6.1.2.5 void OsiUnitTest::testingMessage (const char *const *msg*)

Print a message.

Prints the message as given. Flushes std::cout before printing to std::cerr.

6.1.2.6 `bool OsiUnitTest::equivalentVectors (const OsiSolverInterface * si1, const OsiSolverInterface * si2, double tol, const double * v1, const double * v2, int size)`

Utility method to check equality.

Tests for equality using **CoinRelFitEq** with tolerance `tol`. Understands the notion of solver infinity and obtains the value for infinity from the solver interfaces supplied as parameters.

6.1.2.7 `bool OsiUnitTest::compareProblems (OsiSolverInterface * osi1, OsiSolverInterface * osi2)`

Compare two problems for equality.

Compares the problems held in the two solvers: constraint matrix, row and column bounds, column type, and objective. Rows are checked using upper and lower bounds and using sense, bound, and range.

6.1.2.8 `bool OsiUnitTest::isEquivalent (const CoinPackedVectorBase & pv, int n, const double * fv)`

Compare a packed vector with an expanded vector.

Checks that all values present in the packed vector are present in the full vector and checks that there are no extra entries in the full vector. Uses **CoinRelFitEq** with the default tolerance.

6.1.2.9 `bool OsiUnitTest::processParameters (int argc, const char ** argv, std::map< std::string, std::string > & parms, const std::map< std::string, int > & ignorekeywords = std::map< std::string, int >())`

Process command line parameters.

An unrecognised keyword which is not in the `ignorekeywords` map will trigger the help message and a return value of false. For each keyword in `ignorekeywords`, you can specify the number of following parameters that should be ignored.

This should be replaced with the one of the standard CoinUtils parameter mechanisms.

6.1.3 Variable Documentation

6.1.3.1 `unsigned int OsiUnitTest::verbosity`

Verbosity level of unit tests.

0 (default) for minimal output; larger numbers produce more output

6.1.3.2 `unsigned int OsiUnitTest::haltonerror`

Behaviour on failing a test.

- 0 (= default) continue
- 1 press any key to continue
- 2 stop with abort()

6.1.3.3 `TestOutcomes` `OsiUnitTest::outcomes`

Test outcomes.

A global [TestOutcomes](#) object to store test outcomes during the run of the unit test for an OSI.

Chapter 7

Class Documentation

7.1 OsiSolverInterface::ApplyCutsReturnCode Class Reference

Internal class for obtaining status from the applyCuts method.

```
#include <OsiSolverInterface.hpp>
```

Public Member Functions

Constructors and destructors

- [ApplyCutsReturnCode](#) ()
Default constructor.
- [ApplyCutsReturnCode](#) (const [ApplyCutsReturnCode](#) &rhs)
Copy constructor.
- [ApplyCutsReturnCode](#) & operator= (const [ApplyCutsReturnCode](#) &rhs)
Assignment operator.
- [~ApplyCutsReturnCode](#) ()
Destructor.

Accessing return code attributes

- int [getNumInconsistent](#) () const
Number of logically inconsistent cuts.
- int [getNumInconsistentWrtIntegerModel](#) () const
Number of cuts inconsistent with the current model.
- int [getNumInfeasible](#) () const
Number of cuts that cause obvious infeasibility.
- int [getNumIneffective](#) () const
Number of redundant or ineffective cuts.
- int [getNumApplied](#) () const
Number of cuts applied.

Friends

- class **OsiSolverInterface**
- class **OsiGrbSolverInterface**

7.1.1 Detailed Description

Internal class for obtaining status from the applyCuts method.

Definition at line 74 of file OsiSolverInterface.hpp.

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

- OsiSolverInterface.hpp

7.2 OsiCuts::const_iterator Class Reference

Const Iterator.

```
#include <OsiCuts.hpp>
```

Friends

- class **OsiCuts**

7.2.1 Detailed Description

Const Iterator.

This is a class for iterating over the collection of cuts.

Definition at line 74 of file OsiCuts.hpp.

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

- OsiCuts.hpp

7.3 glp_prob Struct Reference

7.3.1 Detailed Description

Definition at line 30 of file OsiGlpkSolverInterface.hpp.

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

- OsiGlpkSolverInterface.hpp

7.4 OsiCuts::iterator Class Reference

Iterator.

```
#include <OsiCuts.hpp>
```

Friends

- class **OsiCuts**

7.4.1 Detailed Description

Iterator.

This is a class for iterating over the collection of cuts.

Definition at line 30 of file OsiCuts.hpp.

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

- OsiCuts.hpp

7.5 OsiAuxInfo Class Reference

This class allows for a more structured use of algorithmic tweaking to an [OsiSolverInterface](#).

```
#include <OsiAuxInfo.hpp>
```

Inheritance diagram for OsiAuxInfo:

Public Member Functions

- virtual [OsiAuxInfo](#) * [clone](#) () const
Clone.
- [OsiAuxInfo](#) & [operator=](#) (const [OsiAuxInfo](#) &rhs)
Assignment operator.
- void * [getApplicationData](#) () const
Get application data.

Protected Attributes

- void * [appData_](#)
Pointer to user-defined data structure.

7.5.1 Detailed Description

This class allows for a more structured use of algorithmic tweaking to an [OsiSolverInterface](#).

It is designed to replace the simple use of appData_ pointer.

This has been done to make it easier to use NonLinear solvers and other exotic beasts in a branch and bound mode. After this class definition there is one for a derived class for just such a purpose.

Definition at line 21 of file OsiAuxInfo.hpp.

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

- OsiAuxInfo.hpp

7.6 OsiBabSolver Class Reference

This class allows for the use of more exotic solvers e.g.

```
#include <OsiAuxInfo.hpp>
```

Inheritance diagram for OsiBabSolver:

Collaboration diagram for OsiBabSolver:

Public Member Functions

- virtual [OsiAuxInfo](#) * [clone](#) () const
Clone.
- [OsiBabSolver](#) & [operator=](#) (const [OsiBabSolver](#) &rhs)
Assignment operator.
- void [setSolver](#) (const [OsiSolverInterface](#) *solver)
Update solver.
- void [setSolver](#) (const [OsiSolverInterface](#) &solver)
Update solver.
- int [solution](#) (double &objectiveValue, double *newSolution, int numberColumns)
returns 0 if no heuristic solution, 1 if valid solution with better objective value than one passed in Sets solution values if good, sets objective value numberColumns is size of newSolution
- void [setSolution](#) (const double *solution, int numberColumns, double objectiveValue)
Set solution and objective value.
- bool [hasSolution](#) (double &solutionValue, double *solution)
returns true if the object stores a solution, false otherwise.
- void [setSolverType](#) (int value)
Sets solver type 0 - normal LP solver 1 - DW - may also return heuristic solutions 2 - NLP solver or similar - can't compute objective value just from solution check solver to see if feasible and what objective value is.
- int [solverType](#) () const
gets solver type 0 - normal LP solver 1 - DW - may also return heuristic solutions 2 - NLP solver or similar - can't compute objective value just from solution check this (rather than solver) to see if feasible and what objective value is
- bool [solutionAddsCuts](#) () const
Return true if getting solution may add cuts so hot start etc will be obsolete.
- bool [alwaysTryCutsAtRootNode](#) () const
Return true if we should try cuts at root even if looks satisfied.
- bool [solverAccurate](#) () const
Returns true if can use solver objective or feasible values, otherwise use mipBound etc.
- bool [reducedCostsAccurate](#) () const
Returns true if can use reduced costs for fixing.
- double [mipBound](#) () const
Get objective (well mip bound)
- bool [mipFeasible](#) () const
Returns true if node feasible.
- void [setMipBound](#) (double value)
Set mip bound (only used for some solvers)
- double [bestObjectiveValue](#) () const
Get objective value of saved solution.
- bool [tryCuts](#) () const
Says whether we want to try cuts at all.
- bool [warmStart](#) () const
Says whether we have a warm start (so can do strong branching)

- int [extraCharacteristics](#) () const
Get bit mask for odd actions of solvers 1 - solution or bound arrays may move in mysterious ways e.g.
- void [setExtraCharacteristics](#) (int value)
Set bit mask for odd actions of solvers 1 - solution or bound arrays may move in mysterious ways e.g.
- const double * [beforeLower](#) () const
Pointer to lower bounds before branch (only if extraCharacteristics set)
- void [setBeforeLower](#) (const double *array)
Set pointer to lower bounds before branch (only if extraCharacteristics set)
- const double * [beforeUpper](#) () const
Pointer to upper bounds before branch (only if extraCharacteristics set)
- void [setBeforeUpper](#) (const double *array)
Set pointer to upper bounds before branch (only if extraCharacteristics set)

Protected Attributes

- double [bestObjectiveValue_](#)
Objective value of best solution (if there is one) (minimization)
- double [mipBound_](#)
Current lower bound on solution (if > 1.0e50 infeasible)
- const [OsiSolverInterface](#) * [solver_](#)
Solver to use for getting/setting solutions etc.
- double * [bestSolution_](#)
Best integer feasible solution.
- const double * [beforeLower_](#)
Pointer to lower bounds before branch (only if extraCharacteristics set)
- const double * [beforeUpper_](#)
Pointer to upper bounds before branch (only if extraCharacteristics set)
- int [solverType_](#)
Solver type 0 - normal LP solver 1 - DW - may also return heuristic solutions 2 - NLP solver or similar - can't compute objective value just from solution check this (rather than solver) to see if feasible and what objective value is.
- int [sizeSolution_](#)
Size of solution.
- int [extraCharacteristics_](#)
Bit mask for odd actions of solvers 1 - solution or bound arrays may move in mysterious ways e.g.

7.6.1 Detailed Description

This class allows for the use of more exotic solvers e.g.

Non-Linear or Volume.

You can derive from this although at present I can't see the need.

Definition at line 49 of file OsiAuxInfo.hpp.

7.6.2 Member Function Documentation

7.6.2.1 void OsiBabSolver::setSolution (const double * *solution*, int *numberOfColumns*, double *objectiveValue*)

Set solution and objective value.

Number of columns and optimization direction taken from current solver. Size of solution is numberOfColumns (may be padded or truncated in function)

7.6.2.2 bool OsiBabSolver::hasSolution (double & *solutionValue*, double * *solution*)

returns true if the object stores a solution, false otherwise.

If there is a solution then solutionValue and solution will be filled out as well. In that case the user needs to allocate solution to be a big enough array.

7.6.2.3 void OsiBabSolver::setSolverType (int *value*) [inline]

Sets solver type 0 - normal LP solver 1 - DW - may also return heuristic solutions 2 - NLP solver or similar - can't compute objective value just from solution check solver to see if feasible and what objective value is.

- may also return heuristic solution 3 - NLP solver or similar - can't compute objective value just from solution check this (rather than solver) to see if feasible and what objective value is. Using Outer Approximation so called lp based
- may also return heuristic solution 4 - normal solver but cuts are needed for integral solution

Definition at line 102 of file OsiAuxInfo.hpp.

7.6.2.4 int OsiBabSolver::solverType () const [inline]

gets solver type 0 - normal LP solver 1 - DW - may also return heuristic solutions 2 - NLP solver or similar - can't compute objective value just from solution check this (rather than solver) to see if feasible and what objective value is

- may also return heuristic solution 3 - NLP solver or similar - can't compute objective value just from solution check this (rather than solver) to see if feasible and what objective value is. Using Outer Approximation so called lp based
- may also return heuristic solution 4 - normal solver but cuts are needed for integral solution

Definition at line 116 of file OsiAuxInfo.hpp.

7.6.2.5 int OsiBabSolver::extraCharacteristics () const [inline]

Get bit mask for odd actions of solvers 1 - solution or bound arrays may move in mysterious ways e.g.

cplex 2 - solver may want bounds before branch

Definition at line 152 of file OsiAuxInfo.hpp.

7.6.2.6 `void OsiBabSolver::setExtraCharacteristics (int value) [inline]`

Set bit mask for odd actions of solvers 1 - solution or bound arrays may move in mysterious ways e.g.

cplex 2 - solver may want bounds before branch

Definition at line 158 of file OsiAuxInfo.hpp.

7.6.3 Member Data Documentation

7.6.3.1 `int OsiBabSolver::solverType_ [protected]`

Solver type 0 - normal LP solver 1 - DW - may also return heuristic solutions 2 - NLP solver or similar - can't compute objective value just from solution check this (rather than solver) to see if feasible and what objective value is.

- may also return heuristic solution 3 - NLP solver or similar - can't compute objective value just from solution check this (rather than solver) to see if feasible and what objective value is. Using Outer Approximation so called lp based
- may also return heuristic solution

Definition at line 196 of file OsiAuxInfo.hpp.

7.6.3.2 `int OsiBabSolver::extraCharacteristics_ [protected]`

Bit mask for odd actions of solvers 1 - solution or bound arrays may move in mysterious ways e.g.

cplex 2 - solver may want bounds before branch

Definition at line 203 of file OsiAuxInfo.hpp.

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

- OsiAuxInfo.hpp

7.7 OsiBranchingInformation Class Reference

Collaboration diagram for OsiBranchingInformation:

Public Member Functions

- [OsiBranchingInformation \(\)](#)
Default Constructor.
- [OsiBranchingInformation \(const \[OsiSolverInterface\]\(#\) *solver, bool normalSolver, bool copySolution=false\)](#)
Useful Constructor (normalSolver true if has matrix etc etc) copySolution true if constructot should make a copy.
- [OsiBranchingInformation \(const \[OsiBranchingInformation\]\(#\) &\)](#)
Copy constructor.
- [OsiBranchingInformation & operator= \(const \[OsiBranchingInformation\]\(#\) &rhs\)](#)
Assignment operator.
- virtual [OsiBranchingInformation * clone \(\)](#) const
Clone.
- virtual [~OsiBranchingInformation \(\)](#)
Destructor.

Public Attributes

- int [stateOfSearch_](#)
data
- double [objectiveValue_](#)
Value of objective function (in minimization sense)
- double [cutoff_](#)
Value of objective cutoff (in minimization sense)
- double [direction_](#)
Direction 1.0 for minimization, -1.0 for maximization.
- double [integerTolerance_](#)
Integer tolerance.
- double [primalTolerance_](#)
Primal tolerance.
- double [timeRemaining_](#)
Maximum time remaining before stopping on time.
- double [defaultDual_](#)
Dual to use if row bound violated (if negative then pseudoShadowPrices off)
- const [OsiSolverInterface](#) * [solver_](#)
Pointer to solver.
- int [numberColumns_](#)
The number of columns.
- const double * [lower_](#)
Pointer to current lower bounds on columns.
- const double * [solution_](#)
Pointer to current solution.
- const double * [upper_](#)
Pointer to current upper bounds on columns.
- const double * [hotstartSolution_](#)
Highly optional target (hot start) solution.
- const double * [pi_](#)
Pointer to duals.
- const double * [rowActivity_](#)
Pointer to row activity.
- const double * [objective_](#)
Objective.
- const double * [rowLower_](#)
Pointer to current lower bounds on rows.
- const double * [rowUpper_](#)
Pointer to current upper bounds on rows.
- const double * [elementByColumn_](#)
Elements in column copy of matrix.
- const [CoinBigIndex](#) * [columnStart_](#)
Column starts.
- const int * [columnLength_](#)
Column lengths.
- const int * [row_](#)

Row indices.

- double * [usefulRegion_](#)
Useful region of length $\text{CoinMax}(\text{numberColumns}, 2 * \text{numberRows})$ This is allocated and deleted before [OsiObject](#)↔
↔[::infeasibility](#) It is zeroed on entry and should be so on exit It only exists if `defaultDual_ >= 0.0`.
- int * [indexRegion_](#)
Useful index region to go with `usefulRegion_`.
- int [numberSolutions_](#)
Number of solutions found.
- int [numberBranchingSolutions_](#)
Number of branching solutions found (i.e. exclude heuristics)
- int [depth_](#)
Depth in tree.
- bool [owningSolution_](#)
TEMP.

7.7.1 Detailed Description

Definition at line 367 of file `OsiBranchingObject.hpp`.

7.7.2 Member Data Documentation

7.7.2.1 int OsiBranchingInformation::stateOfSearch_

data

State of search 0 - no solution 1 - only heuristic solutions 2 - branched to a solution 3 - no solution but many nodes

Definition at line 402 of file `OsiBranchingObject.hpp`.

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

- `OsiBranchingObject.hpp`

7.8 OsiBranchingObject Class Reference

Abstract branching object base class.

```
#include <OsiBranchingObject.hpp>
```

Inheritance diagram for `OsiBranchingObject`:

Collaboration diagram for `OsiBranchingObject`:

Public Member Functions

- [OsiBranchingObject](#) ()
Default Constructor.
- [OsiBranchingObject](#) ([OsiSolverInterface](#) *solver, double value)
Constructor.
- [OsiBranchingObject](#) (const [OsiBranchingObject](#) &)

Copy constructor.

- `OsiBranchingObject & operator= (const OsiBranchingObject &rhs)`

Assignment operator.

- `virtual OsiBranchingObject * clone () const =0`

Clone.

- `virtual ~OsiBranchingObject ()`

Destructor.

- `int numberBranches () const`

The number of branch arms created for this branching object.

- `int numberBranchesLeft () const`

The number of branch arms left for this branching object.

- `void incrementNumberBranchesLeft ()`

Increment the number of branch arms left for this branching object.

- `void setNumberBranchesLeft (int)`

Set the number of branch arms left for this branching object Just for forcing.

- `void decrementNumberBranchesLeft ()`

Decrement the number of branch arms left for this branching object.

- `virtual double branch (OsiSolverInterface *solver)=0`

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

- `virtual double branch ()`

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

- `virtual bool boundBranch () const`

Return true if branch should fix variables.

- `int branchIndex () const`

Get the state of the branching object This is just the branch index.

- `void setBranchingIndex (int branchIndex)`

Set the state of the branching object.

- `double value () const`

Current value.

- `const OsiObject * originalObject () const`

Return pointer back to object which created.

- `void setOriginalObject (const OsiObject *object)`

Set pointer back to object which created.

- `virtual void checkIsCutoff (double)`

Double checks in case node can change its mind! Returns objective value Can change objective etc.

- `int columnNumber () const`

For debug.

- `virtual void print (const OsiSolverInterface *solver)=0`

Print something about branch - only if log level high.

Protected Attributes

- double `value_`
Current value - has some meaning about branch.
- const `OsiObject * originalObject_`
Pointer back to object which created.
- int `numberBranches_`
Number of branches.
- short `branchIndex_`
The state of the branching object.

7.8.1 Detailed Description

Abstract branching object base class.

In the abstract, an `OsiBranchingObject` contains instructions for how to branch. We want an abstract class so that we can describe how to branch on simple objects (e.g., integers) and more exotic objects (e.g., cliques or hyperplanes).

The `branch()` method is the crucial routine: it is expected to be able to step through a set of branch arms, executing the actions required to create each subproblem in turn. The base class is primarily virtual to allow for a wide range of problem modifications.

See `OsiObject` for an overview of the two classes (`OsiObject` and `OsiBranchingObject`) which make up Osi's branching model.

Definition at line 254 of file `OsiBranchingObject.hpp`.

7.8.2 Member Function Documentation

7.8.2.1 `virtual double OsiBranchingObject::branch (OsiSolverInterface * solver) [pure virtual]`

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

Returns change in guessed objective on next branch

Implemented in `OsiLotsizeBranchingObject`, `OsiSOSBranchingObject`, `OsiIntegerBranchingObject`, and `OsiTwoWayBranchingObject`.

7.8.2.2 `virtual double OsiBranchingObject::branch () [inline],[virtual]`

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

Returns change in guessed objective on next branch

Definition at line 309 of file `OsiBranchingObject.hpp`.

7.8.3 Member Data Documentation

7.8.3.1 `short OsiBranchingObject::branchIndex_ [protected]`

The state of the branching object.

i.e. branch index This starts at 0 when created

Definition at line 360 of file OsiBranchingObject.hpp.

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

- OsiBranchingObject.hpp

7.9 OsiChooseStrong Class Reference

This class chooses a variable to branch on.

```
#include <OsiChooseVariable.hpp>
```

Inheritance diagram for OsiChooseStrong:

Collaboration diagram for OsiChooseStrong:

Public Member Functions

- [OsiChooseStrong](#) ()
Default Constructor.
- [OsiChooseStrong](#) (const [OsiSolverInterface](#) *solver)
Constructor from solver (so we can set up arrays etc)
- [OsiChooseStrong](#) (const [OsiChooseStrong](#) &)
Copy constructor.
- [OsiChooseStrong](#) & [operator=](#) (const [OsiChooseStrong](#) &rhs)
Assignment operator.
- virtual [OsiChooseVariable](#) * [clone](#) () const
Clone.
- virtual [~OsiChooseStrong](#) ()
Destructor.
- virtual int [setUpList](#) ([OsiBranchingInformation](#) *info, bool initialize)
Sets up strong list and clears all if initialize is true.
- virtual int [chooseVariable](#) ([OsiSolverInterface](#) *solver, [OsiBranchingInformation](#) *info, bool fixVariables)
Choose a variable Returns - -1 Node is infeasible 0 Normal termination - we have a candidate 1 All looks satisfied - no candidate 2 We can change the bound on a variable - but we also have a strong branching candidate 3 We can change the bound on a variable - but we have a non-strong branching candidate 4 We can change the bound on a variable - no other candidates We can pick up branch from [bestObjectIndex\(\)](#) and [bestWhichWay\(\)](#) We can pick up a forced branch (can change bound) from [firstForcedObjectIndex\(\)](#) and [firstForcedWhichWay\(\)](#) If we have a solution then we can pick up from [goodObjectiveValue\(\)](#) and [goodSolution\(\)](#) If fixVariables is true then 2,3,4 are all really same as problem changed.
- int [shadowPriceMode](#) () const
Pseudo Shadow Price mode 0 - off 1 - use if no strong info 2 - use if strong not trusted 3 - use even if trusted.
- void [setShadowPriceMode](#) (int value)
Set Shadow price mode.
- const [OsiPseudoCosts](#) & [pseudoCosts](#) () const
Accessor method to pseudo cost object.
- [OsiPseudoCosts](#) & [pseudoCosts](#) ()
Accessor method to pseudo cost object.
- int [numberBeforeTrusted](#) () const
A feww pass-through methods to access members of pseudoCosts_ as if they were members of [OsiChooseStrong](#) object.

Protected Member Functions

- `int doStrongBranching (OsiSolverInterface *solver, OsiBranchingInformation *info, int numberToDo, int returnCriterion)`
This is a utility function which does strong branching on a list of objects and stores the results in OsiHotInfo.objects.
- `void resetResults (int num)`
Clear out the results array.

Protected Attributes

- `int shadowPriceMode_`
Pseudo Shadow Price mode 0 - off 1 - use and multiply by strong info 2 - use.
- `OsiPseudoCosts pseudoCosts_`
The pseudo costs for the chooser.
- `OsiHotInfo * results_`
The results of the strong branching done on the candidates where the pseudocosts were not sufficient.
- `int numResults_`
The number of OsiHotInfo objects that contain information.

7.9.1 Detailed Description

This class chooses a variable to branch on.

This chooses the variable and direction with reliability strong branching.

The flow is : a) initialize the process. This decides on strong branching list and stores indices of all infeasible objects b) do strong branching on list. If list is empty then just choose one candidate and return without strong branching. If not empty then go through list and return best. However we may find that the node is infeasible or that we can fix a variable. If so we return and it is up to user to call again (after fixing a variable).

Definition at line 318 of file OsiChooseVariable.hpp.

7.9.2 Member Function Documentation

7.9.2.1 `virtual int OsiChooseStrong::setupList (OsiBranchingInformation * info, bool initialize)` [virtual]

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

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

Reimplemented from [OsiChooseVariable](#).

7.9.2.2 `int OsiChooseStrong::doStrongBranching (OsiSolverInterface * solver, OsiBranchingInformation * info, int numberToDo, int returnCriterion)` [protected]

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

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

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

- [OsiChooseVariable.hpp](#)

7.10 OsiChooseVariable Class Reference

This class chooses a variable to branch on.

```
#include <OsiChooseVariable.hpp>
```

Inheritance diagram for OsiChooseVariable:

Collaboration diagram for OsiChooseVariable:

Public Member Functions

- [OsiChooseVariable](#) ()
Default Constructor.
- [OsiChooseVariable](#) (const [OsiSolverInterface](#) *solver)
Constructor from solver (so we can set up arrays etc)
- [OsiChooseVariable](#) (const [OsiChooseVariable](#) &)
Copy constructor.
- [OsiChooseVariable](#) & [operator=](#) (const [OsiChooseVariable](#) &rhs)
Assignment operator.
- virtual [OsiChooseVariable](#) * [clone](#) () const
Clone.
- virtual [~OsiChooseVariable](#) ()
Destructor.
- virtual int [setUpList](#) ([OsiBranchingInformation](#) *info, bool initialize)
Sets up strong list and clears all if initialize is true.
- virtual int [chooseVariable](#) ([OsiSolverInterface](#) *solver, [OsiBranchingInformation](#) *info, bool fixVariables)
Choose a variable Returns - -1 Node is infeasible 0 Normal termination - we have a candidate 1 All looks satisfied - no candidate 2 We can change the bound on a variable - but we also have a strong branching candidate 3 We can change the bound on a variable - but we have a non-strong branching candidate 4 We can change the bound on a variable - no other candidates We can pick up branch from [bestObjectIndex\(\)](#) and [bestWhichWay\(\)](#) We can pick up a forced branch (can change bound) from [firstForcedObjectIndex\(\)](#) and [firstForcedWhichWay\(\)](#) If we have a solution then we can pick up from [goodObjectiveValue\(\)](#) and [goodSolution\(\)](#) If fixVariables is true then 2,3,4 are all really same as problem changed.
- virtual bool [feasibleSolution](#) (const [OsiBranchingInformation](#) *info, const double *solution, int numberOfObjects, const [OsiObject](#) **objects)
Returns true if solution looks feasible against given objects.
- void [saveSolution](#) (const [OsiSolverInterface](#) *solver)
Saves a good solution.
- void [clearGoodSolution](#) ()
Clears out good solution after use.
- virtual void [updateInformation](#) (const [OsiBranchingInformation](#) *info, int branch, [OsiHotInfo](#) *hotInfo)
Given a candidate fill in useful information e.g. estimates.
- virtual void [updateInformation](#) (int whichObject, int branch, double changeInObjective, double changeInValue, int status)
Given a branch fill in useful information e.g. estimates.
- double [goodObjectiveValue](#) () const
Objective value for feasible solution.
- double [upChange](#) () const
Estimate of up change or change on chosen if n-way.
- double [downChange](#) () const

- Estimate of down change or max change on other possibilities if n-way.*

 - const double * [goodSolution](#) () const

Good solution - deleted by finalize.
- int [bestObjectIndex](#) () const

Index of chosen object.
- void [setBestObjectIndex](#) (int value)

Set index of chosen object.
- int [bestWhichWay](#) () const

Preferred way of chosen object.
- void [setBestWhichWay](#) (int value)

Set preferred way of chosen object.
- int [firstForcedObjectIndex](#) () const

Index of forced object.
- void [setFirstForcedObjectIndex](#) (int value)

Set index of forced object.
- int [firstForcedWhichWay](#) () const

Preferred way of forced object.
- void [setFirstForcedWhichWay](#) (int value)

Set preferred way of forced object.
- int [numberUnsatisfied](#) () const

Get the number of objects unsatisfied at this node - accurate on first pass.
- int [numberStrong](#) () const

Number of objects to choose for strong branching.
- void [setNumberStrong](#) (int value)

Set number of objects to choose for strong branching.
- int [numberOnList](#) () const

Number left on strong list.
- int [numberStrongDone](#) () const

Number of strong branches actually done.
- int [numberStrongIterations](#) () const

Number of strong iterations actually done.
- int [numberStrongFixed](#) () const

Number of strong branches which changed bounds.
- const int * [candidates](#) () const

List of candidates.
- bool [trustStrongForBound](#) () const

Trust results from strong branching for changing bounds.
- void [setTrustStrongForBound](#) (bool yesNo)

Set trust results from strong branching for changing bounds.
- bool [trustStrongForSolution](#) () const

Trust results from strong branching for valid solution.
- void [setTrustStrongForSolution](#) (bool yesNo)

Set trust results from strong branching for valid solution.
- void [setSolver](#) (const [OsiSolverInterface](#) *solver)

Set solver and redo arrays.
- int [status](#) () const

Return status - -1 Node is infeasible 0 Normal termination - we have a candidate 1 All looks satisfied - no candidate 2 We can change the bound on a variable - but we also have a strong branching candidate 3 We can change the bound on a variable - but we have a non-strong branching candidate 4 We can change the bound on a variable - no other candidates We can pick up branch from [bestObjectIndex\(\)](#) and [bestWhichWay\(\)](#) We can pick up a forced branch (can change bound) from [firstForcedObjectIndex\(\)](#) and [firstForcedWhichWay\(\)](#) If we have a solution then we can pick up from [goodObjectiveValue\(\)](#) and [goodSolution\(\)](#)

Protected Attributes

- double [goodObjectiveValue_](#)
Objective value for feasible solution.
- double [upChange_](#)
Estimate of up change or change on chosen if n-way.
- double [downChange_](#)
Estimate of down change or max change on other possibilities if n-way.
- double * [goodSolution_](#)
Good solution - deleted by finalize.
- int * [list_](#)
List of candidates.
- double * [useful_](#)
Useful array (for sorting etc)
- const [OsiSolverInterface](#) * [solver_](#)
Pointer to solver.
- int [bestObjectIndex_](#)
Index of chosen object.
- int [bestWhichWay_](#)
Preferred way of chosen object.
- int [firstForcedObjectIndex_](#)
Index of forced object.
- int [firstForcedWhichWay_](#)
Preferred way of forced object.
- int [numberUnsatisfied_](#)
The number of objects unsatisfied at this node.
- int [numberStrong_](#)
Number of objects to choose for strong branching.
- int [numberOnList_](#)
Number left on strong list.
- int [numberStrongDone_](#)
Number of strong branches actually done.
- int [numberStrongIterations_](#)
Number of strong iterations actually done.
- int [numberStrongFixed_](#)
Number of bound changes due to strong branching.
- bool [trustStrongForBound_](#)
List of unsatisfied objects - first numberOnList_ for strong branching Trust results from strong branching for changing bounds.
- bool [trustStrongForSolution_](#)
Trust results from strong branching for valid solution.

7.10.1 Detailed Description

This class chooses a variable to branch on.

The base class just chooses the variable and direction without strong branching but it has information which would normally be used by strong branching e.g. to re-enter having fixed a variable but using same candidates for strong branching.

The flow is : a) initialize the process. This decides on strong branching list and stores indices of all infeasible objects b) do strong branching on list. If list is empty then just choose one candidate and return without strong branching. If not empty then go through list and return best. However we may find that the node is infeasible or that we can fix a variable. If so we return and it is up to user to call again (after fixing a variable).

Definition at line 33 of file OsiChooseVariable.hpp.

7.10.2 Member Function Documentation

7.10.2.1 `virtual int OsiChooseVariable::setupList (OsiBranchingInformation * info, bool initialize) [virtual]`

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

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

Reimplemented in [OsiChooseStrong](#).

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

- OsiChooseVariable.hpp

7.11 OsiColCut Class Reference

Column Cut Class.

```
#include <OsiColCut.hpp>
```

Inheritance diagram for OsiColCut:

Collaboration diagram for OsiColCut:

Public Member Functions

Setting column bounds

- void [setLbs](#) (int nElements, const int *colIndices, const double *lbElements)
Set column lower bounds.
- void [setLbs](#) (const **CoinPackedVector** &lbs)
Set column lower bounds from a packed vector.
- void [setUbs](#) (int nElements, const int *colIndices, const double *ubElements)
Set column upper bounds.
- void [setUbs](#) (const **CoinPackedVector** &ubs)
Set column upper bounds from a packed vector.

Getting column bounds

- const **CoinPackedVector** & [lbs](#) () const

- *Get column lower bounds.*
const **CoinPackedVector** & **ubs** () const
Get column upper bounds.

Comparison operators

- virtual bool **operator==** (const **OsiColCut** &rhs) const
*equal - true if lower bounds, upper bounds, and **OsiCut** are equal.*
- virtual bool **operator!=** (const **OsiColCut** &rhs) const
not equal

Sanity checks on cut

- virtual bool **consistent** () const
Returns true if the cut is consistent with respect to itself.
- virtual bool **consistent** (const **OsiSolverInterface** &im) const
Returns true if cut is consistent with respect to the solver interface's model.
- virtual bool **infeasible** (const **OsiSolverInterface** &im) const
Returns true if the cut is infeasible with respect to its bounds and the column bounds in the solver interface's models.
- virtual double **violated** (const double *solution) const
Returns infeasibility of the cut with respect to solution passed in i.e.

Constructors and destructors

- **OsiColCut** & **operator=** (const **OsiColCut** &rhs)
Assignment operator.
- **OsiColCut** (const **OsiColCut** &)
Copy constructor.
- **OsiColCut** ()
Default Constructor.
- virtual **OsiColCut** * **clone** () const
Clone.
- virtual **~OsiColCut** ()
Destructor.

Debug stuff

- virtual void **print** () const
Print cuts in collection.

Friends

- void **OsiColCutUnitTest** (const **OsiSolverInterface** *baseSiP, const std::string &mpsDir)
*A function that tests the methods in the **OsiColCut** class.*

Additional Inherited Members

7.11.1 Detailed Description

Column Cut Class.

Column Cut Class has:

- a sparse vector of column lower bounds
- a sparse vector of column upper bounds

Definition at line 23 of file **OsiColCut.hpp**.

7.11.2 Member Function Documentation

7.11.2.1 `bool OsiColCut::consistent () const [inline], [virtual]`

Returns true if the cut is consistent with respect to itself.

This checks to ensure that:

- The bound vectors do not have duplicate indices,
- The bound vectors indices are ≥ 0

Implements [OsiCut](#).

Definition at line 226 of file OsiColCut.hpp.

7.11.2.2 `bool OsiColCut::consistent (const OsiSolverInterface & im) const [inline], [virtual]`

Returns true if cut is consistent with respect to the solver interface's model.

This checks to ensure that the lower & upperbound packed vectors:

- do not have an index \geq the number of column is the model.

Implements [OsiCut](#).

Definition at line 239 of file OsiColCut.hpp.

7.11.2.3 `bool OsiColCut::infeasible (const OsiSolverInterface & im) const [inline], [virtual]`

Returns true if the cut is infeasible with respect to its bounds and the column bounds in the solver interface's models.

This checks whether:

- the maximum of the new and existing lower bounds is strictly greater than the minimum of the new and existing upper bounds.

Implements [OsiCut](#).

Definition at line 290 of file OsiColCut.hpp.

7.11.2.4 `virtual double OsiColCut::violated (const double * solution) const [virtual]`

Returns infeasibility of the cut with respect to solution passed in i.e.

is positive if cuts off that solution. solution is `getNumCols()` long..

Implements [OsiCut](#).

7.11.3 Friends And Related Function Documentation

7.11.3.1 `void OsiColCutUnitTest (const OsiSolverInterface * baseSiP, const std::string & mpsDir) [friend]`

A function that tests the methods in the [OsiColCut](#) class.

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

- OsiColCut.hpp

7.12 OsiCpxSolverInterface Class Reference

CPLEX Solver Interface.

```
#include <OsiCpxSolverInterface.hpp>
```

Inheritance diagram for OsiCpxSolverInterface:

Collaboration diagram for OsiCpxSolverInterface:

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.
- const char * [getCtype](#) () const
return a vector of variable types (continous, binary, integer)

Solve methods

- virtual void [initialSolve](#) ()
Solve initial LP relaxation.
- virtual void [resolve](#) ()
Resolve an LP relaxation after problem modification.
- virtual void [branchAndBound](#) ()
Invoke solver's built-in enumeration algorithm.

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.

- bool [setIntParam](#) (OsiIntParam key, int value)
Set an integer parameter.
- bool [setDbParam](#) (OsiDbParam key, double value)
Set a double parameter.
- bool [setStrParam](#) (OsiStrParam key, const std::string &value)
Set a string parameter.
- bool [getIntParam](#) (OsiIntParam key, int &value) const
Get an integer parameter.
- bool [getDbParam](#) (OsiDbParam key, double &value) const

Get a double parameter.

- bool [getStrParam](#) (OsiStrParam key, std::string &value) const

Get a string parameter.

- void **setMipStart** (bool value)
- bool **getMipStart** () const

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

- **CoinWarmStart** * [getEmptyWarmStart](#) () const
Get an empty warm start object.
- virtual **CoinWarmStart** * [getWarmStart](#) () const
Get warmstarting information.
- virtual bool [setWarmStart](#) (const **CoinWarmStart** *warmstart)
Set warmstarting information.

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.

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 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 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 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 [setObjCoeffSet](#) (const int *indexFirst, const int *indexLast, const double *coeffList)

- Set a a set of objective function coefficients.*

 - virtual void [setColLower](#) (int elementIndex, double elementValue)
Set a single column lower bound
Use -COIN_DBL_MAX for -infinity.
 - virtual void [setColUpper](#) (int elementIndex, double elementValue)
Set a single column upper bound
Use COIN_DBL_MAX for infinity.
 - virtual void [setColBounds](#) (int elementIndex, double lower, double upper)
Set a single column lower and upper bound
The default implementation just invokes [setColLower\(\)](#) and [setColUpper\(\)](#)
 - 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 -COIN_DBL_MAX for -infinity.
 - virtual void [setRowUpper](#) (int elementIndex, double elementValue)
Set a single row upper bound
Use COIN_DBL_MAX for infinity.
 - virtual void [setRowBounds](#) (int elementIndex, double lower, double upper)
Set a single row lower and upper bound
The default implementation just invokes [setRowLower\(\)](#) and [setRowUpper\(\)](#)
 - 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\(\)](#) and over and over again.

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.

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)
Add a 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)
Add a set of columns (primal variables) to the problem.
- virtual void [deleteCols](#) (const int num, const int *colIndices)

- Remove a set of columns (primal variables) from the problem.*
- virtual void [addRow](#) (const **CoinPackedVectorBase** &vec, 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)
Add a row (constraint) to the problem.
- virtual void [addRows](#) (const int numRows, const **CoinPackedVectorBase** *const *rows, const double *rowlb, const double *rowub)
Add a set of rows (constraints) to the problem.
- virtual void [addRows](#) (const int numRows, const **CoinPackedVectorBase** *const *rows, const char *rowsen, const double *rowrhs, const double *rowrng)
Add a set of rows (constraints) to the problem.
- virtual void [deleteRows](#) (const int num, const int *rowIndices)
Delete a set of rows (constraints) from the problem.

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 int numcols, const int numRows, const int *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 int *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 [readMps](#) (const char *filename, const char *extension="mps")
Read an mps file from the given filename.
- 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.

Message handling

- void [passInMessageHandler](#) (**CoinMessageHandler** *handler)
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.

Constructors and destructor

- [OsiCpxSolverInterface](#) ()

Default Constructor.

- virtual [OsiSolverInterface](#) * [clone](#) (bool copyData=true) const

Clone.

- [OsiCpxSolverInterface](#) (const [OsiCpxSolverInterface](#) &)

Copy constructor.

- [OsiCpxSolverInterface](#) & [operator=](#) (const [OsiCpxSolverInterface](#) &rhs)

Assignment operator.

- virtual [~OsiCpxSolverInterface](#) ()

Destructor.

- virtual void [reset](#) ()

Resets as if default constructor.

OsiSimplexInterface methods

Cplex adds a slack with coeff +1 in "<=" and "=" constraints, with coeff -1 in ">=", slack being non negative.

We switch in order to get a "Clp tableau" where all the slacks have coefficient +1 in the original tableau.

If a slack for ">=" is non basic, invB is not changed; column of the slack in the optimal tableau is flipped.

If a slack for ">=" is basic, corresp. row of invB is flipped; whole row of the optimal tableau is flipped; then whole column for the slack in opt tableau is flipped.

Ranged rows are not supported. It might work, but no guarantee is given.

Code implemented only for Cplex9.0 and higher, lower version number of Cplex will abort the code.

- virtual int [canDoSimplexInterface](#) () const
Returns 1 if can just do getBlv etc 2 if has all OsiSimplex methods and 0 if it has none.
- virtual void [enableSimplexInterface](#) (int doingPrimal)
Useless function, defined only for compatibility with OsiSimplexInterface.
- virtual void [disableSimplexInterface](#) ()
Useless function, defined only for compatibility with OsiSimplexInterface.
- virtual void [enableFactorization](#) () const
Useless function, defined only for compatibility with OsiSimplexInterface.
- virtual void [disableFactorization](#) () const
Useless function, defined only for compatibility with OsiSimplexInterface.
- virtual bool [basisIsAvailable](#) () const
Returns true if a basis is available.
- virtual void [getBasisStatus](#) (int *cstat, int *rstat) const
Returns a basis status of the structural/artificial variables At present as warm start i.e 0: free, 1: basic, 2: upper, 3: lower.
- 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 [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 [getBlvCol](#) (int col, double *vec) const
Get a column of the basis inverse.
- virtual void [getBasics](#) (int *index) const
Get indices of the pivot variable in each row (order of indices corresponds to the order of elements in a vector returned by [getBlvACol\(\)](#) and [getBlvCol\(\)](#)).
- void [switchToLP](#) ()
switches CPLEX to prob type LP
- void [switchToMIP](#) ()
switches CPLEX to prob type MIP

Protected Member Functions

- CPXLPptr [getMutableLpPtr](#) () const
Get LP Pointer for const methods.
- CPXENVptr [getMutableEnvironmentPtr](#) () const
Get Environment Pointer for const methods.

Protected methods

- virtual void [applyRowCut](#) (const [OsiRowCut](#) &rc)
Apply a row cut. Return true if cut was applied.
- virtual void [applyColCut](#) (const [OsiColCut](#) &cc)
Apply a column cut (bound adjustment).

Friends

- void [OsiCpxSolverInterfaceUnitTest](#) (const std::string &mpsDir, const std::string &netlibDir)
A function that tests the methods in the [OsiCpxSolverInterface](#) class.

CPLEX specific public interfaces

- enum [keepCachedFlag](#) {
[KEEPCACHED_NONE](#) = 0, [KEEPCACHED_COLUMN](#) = 1, [KEEPCACHED_ROW](#) = 2, [KEEPCACHED_MATRIX](#) = 4,
[KEEPCACHED_RESULTS](#) = 8, [KEEPCACHED_PROBLEM](#) = [KEEPCACHED_COLUMN](#) | [KEEPCACHED_ROW](#) | [KEEPCACHED_MATRIX](#), [KEEPCACHED_ALL](#) = [KEEPCACHED_PROBLEM](#) | [KEEPCACHED_RESULTS](#),
[FREECACHED_COLUMN](#) = [KEEPCACHED_PROBLEM](#) & ~[KEEPCACHED_COLUMN](#),
[FREECACHED_ROW](#) = [KEEPCACHED_PROBLEM](#) & ~[KEEPCACHED_ROW](#), [FREECACHED_MATRIX](#) = [KEEPCACHED_PROBLEM](#) & ~[KEEPCACHED_MATRIX](#), [FREECACHED_RESULTS](#) = [KEEPCACHED_ALL](#) & ~[KEEPCACHED_RESULTS](#) }
Get pointer to CPLEX model and free all specified cached data entries (combined with logical or-operator '|'):
- CPXLPptr [getLpPtr](#) (int keepCached=[KEEPCACHED_NONE](#))
- CPXENVptr [getEnvironmentPtr](#) ()
Method to access CPLEX environment pointer.

Additional Inherited Members

7.12.1 Detailed Description

CPLEX Solver Interface.

Instantiation of [OsiCpxSolverInterface](#) for CPLEX

Definition at line 29 of file [OsiCpxSolverInterface.hpp](#).

7.12.2 Member Enumeration Documentation

7.12.2.1 enum [OsiCpxSolverInterface::keepCachedFlag](#)

Get pointer to CPLEX model and free all specified cached data entries (combined with logical or-operator '|'):

Enumerator

KEEPCACHED_NONE discard all cached data (default)

KEEPCACHED_COLUMN column information: objective values, lower and upper bounds, variable types

KEEPCACHED_ROW row information: right hand sides, ranges and senses, lower and upper bounds for row

KEEPCACHED_MATRIX problem matrix: matrix ordered by column and by row

KEEPCACHED_RESULTS LP solution: primal and dual solution, reduced costs, row activities.

KEEPCACHED_PROBLEM only discard cached LP solution

KEEPCACHED_ALL keep all cached data (similar to [getMutableLpPtr\(\)](#))

FREECACHED_COLUMN free only cached column and LP solution information

FREECACHED_ROW free only cached row and LP solution information

FREECACHED_MATRIX free only cached matrix and LP solution information

FREECACHED_RESULTS free only cached LP solution information

Definition at line 614 of file OsiCpxSolverInterface.hpp.

7.12.3 Member Function Documentation

7.12.3.1 `CoinWarmStart* OsiCpxSolverInterface::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.

Implements [OsiSolverInterface](#).

7.12.3.2 `virtual bool OsiCpxSolverInterface::setWarmStart (const CoinWarmStart * warmstart)` [virtual]

Set warmstarting information.

Return true/false depending on whether the warmstart information was accepted or not.

Implements [OsiSolverInterface](#).

7.12.3.3 `virtual const char* OsiCpxSolverInterface::getRowSense () const` [virtual]

Get pointer to array[[getNumRows\(\)](#)] of row constraint senses.

- 'L': \leq constraint
- 'E': $=$ constraint
- 'G': \geq constraint
- 'R': ranged constraint
- 'N': free constraint

Implements [OsiSolverInterface](#).

7.12.3.4 `virtual const double* OsiCpxSolverInterface::getRightHandSide () const [virtual]`

Get pointer to array[[getNumRows\(\)](#)] of rows right-hand sides.

- if `rowsense()[i] == 'L'` then `rhs()[i] == rowupper()[i]`
- if `rowsense()[i] == 'G'` then `rhs()[i] == rowlower()[i]`
- if `rowsense()[i] == 'R'` then `rhs()[i] == rowupper()[i]`
- if `rowsense()[i] == 'N'` then `rhs()[i] == 0.0`

Implements [OsiSolverInterface](#).

7.12.3.5 `virtual const double* OsiCpxSolverInterface::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 0.0

Implements [OsiSolverInterface](#).

7.12.3.6 `virtual int OsiCpxSolverInterface::getIterationCount () const [virtual]`

Get how many iterations it took to solve the problem (whatever "iteration" mean to the solver).

Implements [OsiSolverInterface](#).

7.12.3.7 `virtual std::vector<double*> OsiCpxSolverInterface::getDualRays (int maxNumRays, bool fullRay = false) const [virtual]`

Get as many dual rays as the solver can provide.

(In case of proven primal infeasibility there should be at least one.)

The first [getNumRows\(\)](#) ray components will always be associated with the row duals (as returned by [getRowPrice\(\)](#)). If `fullRay` is true, the final [getNumCols\(\)](#) entries will correspond to the ray components associated with the nonbasic variables. If the full ray is requested and the method cannot provide it, it will throw an exception.

NOTE for implementers of solver interfaces:

The double pointers in the vector should point to arrays of length [getNumRows\(\)](#) and they should be allocated via `new[]`.

NOTE for users of solver interfaces:

It is the user's responsibility to free the double pointers in the vector using `delete[]`.

Implements [OsiSolverInterface](#).

7.12.3.8 `virtual std::vector<double*> OsiCpxSolverInterface::getPrimalRays (int maxNumRays) const [virtual]`

Get as many primal rays as the solver can provide.

(In case of proven dual infeasibility there should be at least one.)

NOTE for implementers of solver interfaces:

The double pointers in the vector should point to arrays of length `getNumCols()` and they should be allocated via `new[]`.

NOTE for users of solver interfaces:

It is the user's responsibility to free the double pointers in the vector using `delete[]`.

Implements [OsiSolverInterface](#).

7.12.3.9 `virtual void OsiCpxSolverInterface::setColLower (int elementIndex, double elementValue)` [virtual]

Set a single column lower bound
Use `-COIN_DBL_MAX` for -infinity.

Implements [OsiSolverInterface](#).

7.12.3.10 `virtual void OsiCpxSolverInterface::setColUpper (int elementIndex, double elementValue)` [virtual]

Set a single column upper bound
Use `COIN_DBL_MAX` for infinity.

Implements [OsiSolverInterface](#).

7.12.3.11 `virtual void OsiCpxSolverInterface::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

<code><code>[<i>indexFirst</i>,<i>indexLast</i></code></code>	contains the indices of the constraints whose either bound changes
<code><i>boundList</i></code>	the new lower/upper bound pairs for the variables

Reimplemented from [OsiSolverInterface](#).

7.12.3.12 `virtual void OsiCpxSolverInterface::setRowLower (int elementIndex, double elementValue)` [virtual]

Set a single row lower bound
Use `-COIN_DBL_MAX` for -infinity.

Implements [OsiSolverInterface](#).

7.12.3.13 `virtual void OsiCpxSolverInterface::setRowUpper (int elementIndex, double elementValue)` [virtual]

Set a single row upper bound
Use `COIN_DBL_MAX` for infinity.

Implements [OsiSolverInterface](#).

7.12.3.14 `virtual void OsiCpxSolverInterface::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

<code><code>[indexfirst, index↵ Last]</code></code>	contains the indices of the constraints whose either bound changes
<code>boundList</code>	the new lower/upper bound pairs for the constraints

Reimplemented from [OsiSolverInterface](#).

7.12.3.15 `virtual void OsiCpxSolverInterface::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\(\)](#) and over and over again.

Parameters

<code><code>[indexfirst, index↵ Last]</code></code>	contains the indices of the constraints whose type changes
<code>senseList</code>	the new senses
<code>rhsList</code>	the new right hand sides
<code>rangeList</code>	the new ranges

Reimplemented from [OsiSolverInterface](#).

7.12.3.16 `virtual void OsiCpxSolverInterface::setColSolution (const double * colsol)` [virtual]

Set the primal solution column values.

`colsol[numcols()]` is an array of values of the problem column variables. These values are copied to memory owned by the solver object or the solver. They will be returned as the result of `colsol()` until changed by another call to `setColsol()` or by a call to any solver routine. Whether the solver makes use of the solution in any way is solver-dependent.

Implements [OsiSolverInterface](#).

7.12.3.17 `virtual void OsiCpxSolverInterface::setRowPrice (const double * rowprice)` [virtual]

Set dual solution vector.

`rowprice[numrows()]` is an array of values of the problem row dual variables. These values are copied to memory owned by the solver object or the solver. They will be returned as the result of `rowprice()` until changed by another call to `setRowprice()` or by a call to any solver routine. Whether the solver makes use of the solution in any way is solver-dependent.

Implements [OsiSolverInterface](#).

7.12.3.18 `virtual void OsiCpxSolverInterface::addCol (const CoinPackedVectorBase & vec, const double collb, const double colub, const double obj)` [virtual]

Add a column (primal variable) to the problem.

Implements [OsiSolverInterface](#).

7.12.3.19 `virtual void OsiCpxSolverInterface::addCols (const int numcols, const CoinPackedVectorBase *const * cols, const double * collb, const double * colub, const double * obj) [virtual]`

Add a set of columns (primal variables) to the problem.

The default implementation simply makes repeated calls to [addCol\(\)](#).

Reimplemented from [OsiSolverInterface](#).

7.12.3.20 `virtual void OsiCpxSolverInterface::deleteCols (const int num, const int * colIndices) [virtual]`

Remove a set of columns (primal variables) from the problem.

The solver interface for a basis-oriented solver will maintain valid warm start information if all deleted variables are nonbasic.

Implements [OsiSolverInterface](#).

7.12.3.21 `virtual void OsiCpxSolverInterface::addRow (const CoinPackedVectorBase & vec, const double rowlb, const double rowub) [virtual]`

Add a row (constraint) to the problem.

Implements [OsiSolverInterface](#).

7.12.3.22 `virtual void OsiCpxSolverInterface::addRow (const CoinPackedVectorBase & vec, const char rowSen, const double rowrhs, const double rowrng) [virtual]`

Add a row (constraint) to the problem.

Implements [OsiSolverInterface](#).

7.12.3.23 `virtual void OsiCpxSolverInterface::addRows (const int numrows, const CoinPackedVectorBase *const * rows, const double * rowlb, const double * rowub) [virtual]`

Add a set of rows (constraints) to the problem.

The default implementation simply makes repeated calls to [addRow\(\)](#).

Reimplemented from [OsiSolverInterface](#).

7.12.3.24 `virtual void OsiCpxSolverInterface::addRows (const int numrows, const CoinPackedVectorBase *const * rows, const char * rowSen, const double * rowrhs, const double * rowrng) [virtual]`

Add a set of rows (constraints) to the problem.

The default implementation simply makes repeated calls to [addRow\(\)](#).

Reimplemented from [OsiSolverInterface](#).

7.12.3.25 `virtual void OsiCpxSolverInterface::deleteRows (const int num, const int * rowIndices) [virtual]`

Delete a set of rows (constraints) from the problem.

The solver interface for a basis-oriented solver will maintain valid warm start information if all deleted rows are loose.

Implements [OsiSolverInterface](#).

7.12.3.26 `virtual void OsiCpxSolverInterface::loadProblem (const CoinPackedMatrix & matrix, const double * collb, const double * colub, const double * obj, const double * rowlb, const double * rowub) [virtual]`

Load in a problem by copying the arguments (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

Implements [OsiSolverInterface](#).

7.12.3.27 `virtual void OsiCpxSolverInterface::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.

Implements [OsiSolverInterface](#).

7.12.3.28 `virtual void OsiCpxSolverInterface::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 0 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

Implements [OsiSolverInterface](#).

7.12.3.29 `virtual void OsiCpxSolverInterface::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.

Implements [OsiSolverInterface](#).

7.12.3.30 `virtual void OsiCpxSolverInterface::loadProblem (const int numcols, const int numRows, const int * 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).

7.12.3.31 `virtual void OsiCpxSolverInterface::loadProblem (const int numcols, const int numRows, const int * 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).

7.12.3.32 `virtual void OsiCpxSolverInterface::writeMps (const char * filename, const char * extension = "mps", double objSense = 0.0) const [virtual]`

Write the problem into an mps file of the given filename.

If objSense is non zero then -1.0 forces the code to write a maximization objective and +1.0 to write a minimization one.

If 0.0 then solver can do what it wants

Implements [OsiSolverInterface](#).

7.12.3.33 `virtual void OsiCpxSolverInterface::applyColCut (const OsiColCut & cc) [protected],[virtual]`

Apply a column cut (bound adjustment).

Return true if cut was applied.

Implements [OsiSolverInterface](#).

7.12.4 Friends And Related Function Documentation

7.12.4.1 `void OsiCpxSolverInterfaceUnitTest (const std::string & mpsDir, const std::string & netlibDir) [friend]`

A function that tests the methods in the [OsiCpxSolverInterface](#) class.

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

- [OsiCpxSolverInterface.hpp](#)

7.13 OsiCut Class Reference

Inheritance diagram for OsiCut:

Public Member Functions

Effectiveness

- void [setEffectiveness](#) (double e)
Set effectiveness.
- double [effectiveness](#) () const
Get effectiveness.

GloballyValid

- void [setGloballyValid](#) (bool trueFalse)
Set globallyValid (nonzero true)
- void [setGloballyValid](#) ()
- void [setNotGloballyValid](#) ()
- bool [globallyValid](#) () const
Get globallyValid.
- void [setGloballyValidAsInteger](#) (int trueFalse)
Set globallyValid as integer (nonzero true)
- int [globallyValidAsInteger](#) () const
Get globallyValid.

Debug stuff

- virtual void [print](#) () const
Print cuts in collection.

Comparison operators

- virtual bool [operator==](#) (const [OsiCut](#) &rhs) const
equal. 2 cuts are equal if there effectiveness are equal
- virtual bool [operator!=](#) (const [OsiCut](#) &rhs) const
not equal
- virtual bool [operator<](#) (const [OsiCut](#) &rhs) const
less than. True if this.effectiveness < rhs.effectiveness
- virtual bool [operator>](#) (const [OsiCut](#) &rhs) const
less than. True if this.effectiveness > rhs.effectiveness

Sanity checks on cut

- virtual bool [consistent](#) () const =0
Returns true if the cut is consistent with respect to itself, without considering any data in the model.
- virtual bool [consistent](#) (const [OsiSolverInterface](#) &si) const =0
Returns true if cut is consistent when considering the solver interface's model.
- virtual bool [infeasible](#) (const [OsiSolverInterface](#) &si) const =0
Returns true if the cut is infeasible "with respect to itself" and cannot be satisfied.
- virtual double [violated](#) (const double *solution) const =0
Returns infeasibility of the cut with respect to solution passed in i.e.

Protected Member Functions

Constructors and destructors

- [OsiCut](#) ()
Default Constructor.
- [OsiCut](#) (const [OsiCut](#) &)
Copy constructor.
- [OsiCut](#) & [operator=](#) (const [OsiCut](#) &rhs)
Assignment operator.
- virtual [~OsiCut](#) ()
Destructor.

7.13.1 Detailed Description

Definition at line 36 of file [OsiCut.hpp](#).

7.13.2 Member Function Documentation

7.13.2.1 `virtual bool OsiCut::consistent () const [inline],[pure virtual]`

Returns true if the cut is consistent with respect to itself, without considering any data in the model.

For example, it might check to ensure that a column index is not negative.

Implemented in [OsiRowCut](#), and [OsiColCut](#).

7.13.2.2 `virtual bool OsiCut::consistent (const OsiSolverInterface & si) const [inline],[pure virtual]`

Returns true if cut is consistent when considering the solver interface's model.

For example, it might check to ensure that a column index is not greater than the number of columns in the model. Assumes [consistent\(\)](#) is true.

Implemented in [OsiRowCut](#), and [OsiColCut](#).

7.13.2.3 `virtual bool OsiCut::infeasible (const OsiSolverInterface & si) const [inline],[pure virtual]`

Returns true if the cut is infeasible "with respect to itself" and cannot be satisfied.

This method does NOT check whether adding the cut to the solver interface's model will make the -model- infeasible. A cut which returns [infeasible\(si\)](#) may very well make the model infeasible. (Of course, adding a cut with returns [infeasible\(si\)](#) will make the model infeasible.)

The "with respect to itself" is in quotes because in the case where the cut simply replaces existing bounds, it may make sense to test infeasibility with respect to the current bounds held in the solver interface's model. For example, if the cut has a single variable in it, it might check that the maximum of new and existing lower bounds is greater than the minimum of the new and existing upper bounds.

Assumes that [consistent\(si\)](#) is true.

Infeasible cuts can be a useful mechanism for a cut generator to inform the solver interface that its detected infeasibility of the problem.

Implemented in [OsiRowCut](#), and [OsiColCut](#).

7.13.2.4 `virtual double OsiCut::violated (const double * solution) const` `[pure virtual]`

Returns infeasibility of the cut with respect to solution passed in i.e.

is positive if cuts off that solution. solution is getNumCols() long..

Implemented in [OsiRowCut](#), and [OsiColCut](#).

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

- [OsiCut.hpp](#)

7.14 OsiCuts Class Reference

Collections of row cuts and column cuts.

```
#include <OsiCuts.hpp>
```

Classes

- class [const_iterator](#)
Const Iterator.
- class [iterator](#)
Iterator.

Public Member Functions

Inserting a cut into collection

- void [insert](#) (const [OsiRowCut](#) &rc)
Insert a row cut.
- void [insertIfNotDuplicate](#) ([OsiRowCut](#) &rc, **CoinAbsFitEq** treatAsSame=**CoinAbsFitEq**(1.0e-12))
Insert a row cut unless it is a duplicate - cut may get sorted.
- void [insertIfNotDuplicate](#) ([OsiRowCut](#) &rc, **CoinRelFitEq** treatAsSame)
Insert a row cut unless it is a duplicate - cut may get sorted.
- void [insert](#) (const [OsiColCut](#) &cc)
Insert a column cut.
- void [insert](#) ([OsiRowCut](#) *&rcPtr)
Insert a row cut.
- void [insert](#) ([OsiColCut](#) *&ccPtr)
Insert a column cut.
- void [insert](#) (const [OsiCuts](#) &cs)
Insert a set of cuts.

Number of cuts in collection

- int [sizeRowCuts](#) () const
Number of row cuts in collection.
- int [sizeColCuts](#) () const
Number of column cuts in collection.
- int [sizeCuts](#) () const
Number of cuts in collection.

Debug stuff

- void `printCuts ()` const
Print cuts in collection.

Get a cut from collection

- `OsiRowCut * rowCutPtr (int i)`
Get pointer to i'th row cut.
- const `OsiRowCut * rowCutPtr (int i) const`
Get const pointer to i'th row cut.
- `OsiColCut * colCutPtr (int i)`
Get pointer to i'th column cut.
- const `OsiColCut * colCutPtr (int i) const`
Get const pointer to i'th column cut.
- `OsiRowCut & rowCut (int i)`
Get reference to i'th row cut.
- const `OsiRowCut & rowCut (int i) const`
Get const reference to i'th row cut.
- `OsiColCut & colCut (int i)`
Get reference to i'th column cut.
- const `OsiColCut & colCut (int i) const`
Get const reference to i'th column cut.
- const `OsiCut * mostEffectiveCutPtr () const`
Get const pointer to the most effective cut.
- `OsiCut * mostEffectiveCutPtr ()`
Get pointer to the most effective cut.

Deleting cut from collection

- void `eraseRowCut (int i)`
Remove i'th row cut from collection.
- void `eraseColCut (int i)`
Remove i'th column cut from collection.
- `OsiRowCut * rowCutPtrAndZap (int i)`
Get pointer to i'th row cut and remove ptr from collection.
- void `dumpCuts ()`
Clear all row cuts without deleting them.
- void `eraseAndDumpCuts (const std::vector< int > to_erase)`
Selective delete and clear for row cuts.

Sorting collection

- void `sort ()`
Cuts with greatest effectiveness are first.

Iterators

Example of using an iterator to sum effectiveness of all cuts in the collection.

```
double sumEff=0.0;
for ( OsiCuts::iterator it=cuts.begin(); it!=cuts.end(); ++it )
    sumEff+= (*it)->effectiveness();
```

- [iterator begin \(\)](#)
Get iterator to beginning of collection.
- [const_iterator begin \(\)](#) const
Get const iterator to beginning of collection.
- [iterator end \(\)](#)
Get iterator to end of collection.
- [const_iterator end \(\)](#) const
Get const iterator to end of collection.

Constructors and destructors

- [OsiCuts \(\)](#)
Default constructor.
- [OsiCuts \(const OsiCuts &\)](#)
Copy constructor.
- [OsiCuts & operator= \(const OsiCuts &rhs\)](#)
Assignment operator.
- virtual [~OsiCuts \(\)](#)
Destructor.

Friends

- void [OsiCutsUnitTest \(\)](#)
A function that tests the methods in the [OsiCuts](#) class.

7.14.1 Detailed Description

Collections of row cuts and column cuts.

Definition at line 19 of file OsiCuts.hpp.

7.14.2 Member Function Documentation

7.14.2.1 void [OsiCuts::insertIfNotDuplicate \(OsiRowCut & rc, CoinAbsFltEq treatAsSame = CoinAbsFltEq \(1.0e-12\) \)](#)

Insert a row cut unless it is a duplicate - cut may get sorted.

Duplicate is defined as **CoinAbsFltEq** says same

7.14.2.2 void [OsiCuts::insertIfNotDuplicate \(OsiRowCut & rc, CoinRelFltEq treatAsSame \)](#)

Insert a row cut unless it is a duplicate - cut may get sorted.

Duplicate is defined as **CoinRelFltEq** says same

7.14.2.3 void [OsiCuts::insert \(OsiRowCut *& rcPtr \)](#) [inline]

Insert a row cut.

The [OsiCuts](#) object takes control of the cut object. On return, rcPtr is NULL.

Definition at line 319 of file OsiCuts.hpp.

7.14.2.4 `void OsiCuts::insert (OsiColCut *& ccPtr) [inline]`

Insert a column cut.

The [OsiCuts](#) object takes control of the cut object. On return `ccPtr` is NULL.

Definition at line 324 of file `OsiCuts.hpp`.

7.14.2.5 `void OsiCuts::dumpCuts () [inline]`

Clear all row cuts without deleting them.

Handy in case one wants to use CGL without managing cuts in one of the OSI containers. Client is ultimately responsible for deleting the data structures holding the row cuts.

Definition at line 461 of file `OsiCuts.hpp`.

7.14.2.6 `void OsiCuts::eraseAndDumpCuts (const std::vector< int > to_erase) [inline]`

Selective delete and clear for row cuts.

Deletes the cuts specified in `to_erase` then clears remaining cuts without deleting them. A hybrid of [eraseRowCut\(int\)](#) and [dumpCuts\(\)](#). Client is ultimately responsible for deleting the data structures for row cuts not specified in `to_erase`.

Definition at line 465 of file `OsiCuts.hpp`.

7.14.3 Friends And Related Function Documentation

7.14.3.1 `void OsiCutsUnitTest () [friend]`

A function that tests the methods in the [OsiCuts](#) class.

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

- `OsiCuts.hpp`

7.15 OsiGlpkSolverInterface Class Reference

Inheritance diagram for `OsiGlpkSolverInterface`:

Collaboration diagram for `OsiGlpkSolverInterface`:

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 `branchAndBound` ()
Invoke solver's built-in enumeration algorithm.

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.

- bool `setIntParam` (OsiIntParam key, int value)
Set an integer parameter.
- bool `setDblParam` (OsiDblParam key, double value)
Set a double parameter.
- bool `setStrParam` (OsiStrParam key, const std::string &value)
Set a string parameter.
- bool `setHintParam` (OsiHintParam key, bool sense=true, OsiHintStrength strength=OsiHintTry, void *info=0)
Set a hint parameter.
- bool `getIntParam` (OsiIntParam key, int &value) const
Get an integer parameter.
- bool `getDblParam` (OsiDblParam key, double &value) const
Get a double parameter.
- bool `getStrParam` (OsiStrParam key, std::string &value) const
Get a string parameter.

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?
- virtual bool `isTimeLimitReached` () const
Time limit reached?
- virtual bool `isFeasible` () const

(Integer) Feasible solution found?

WarmStart related methods

- **CoinWarmStart** * [getEmptyWarmStart](#) () const
Get an empty warm start object.
- virtual **CoinWarmStart** * [getWarmStart](#) () const
Get warmstarting information.
- virtual bool [setWarmStart](#) (const **CoinWarmStart** *warmstart)
Set warmstarting information.

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.

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 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 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 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 -COIN_DBL_MAX for -infinity.
- virtual void [setColUpper](#) (int elementIndex, double elementValue)
Set a single column upper bound
Use COIN_DBL_MAX for infinity.
- virtual void [setColBounds](#) (int elementIndex, double lower, double upper)
Set a single column lower and upper bound
The default implementation just invokes [setColLower\(\)](#) and [setColUpper\(\)](#)
- 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 -COIN_DBL_MAX for -infinity.
- virtual void [setRowUpper](#) (int elementIndex, double elementValue)
Set a single row upper bound
Use COIN_DBL_MAX for infinity.
- virtual void [setRowBounds](#) (int elementIndex, double lower, double upper)
Set a single row lower and upper bound
The default implementation just invokes [setRowLower\(\)](#) and [setRowUpper\(\)](#)
- 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.

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.

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)
Add a 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)
Add a set of columns (primal variables) to the problem.
- virtual void [deleteCols](#) (const int num, const int *colIndices)
Remove a set of columns (primal variables) from the problem.
- virtual void [addRow](#) (const **CoinPackedVectorBase** &vec, 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)
Add a row (constraint) to the problem.
- virtual void [addRows](#) (const int numRows, const **CoinPackedVectorBase** *const *rows, const double *rowlb, const double *rowub)
Add a set of rows (constraints) to the problem.
- virtual void [addRows](#) (const int numRows, const **CoinPackedVectorBase** *const *rows, const char *rowsen, const double *rowrhs, const double *rowrng)
Add a set of rows (constraints) to the problem.
- virtual void [deleteRows](#) (const int num, const int *rowIndices)
Delete a set of rows (constraints) from the problem.

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 int numcols, const int numRows, const int *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 int *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 [readMps](#) (const char *filename, const char *extension="mps")
Read an mps file from the given filename.
- 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.

Methods for row and column names.

Only the set methods need to be overridden to ensure consistent names between OsiGlpk and the OSI base class.

- void [setObjName](#) (std::string name)
Set the objective function name.
- void [setRowName](#) (int ndx, std::string name)
Set a row name.
- void [setColName](#) (int ndx, std::string name)
Set a column name.

Constructors and destructor

- [OsiGlpkSolverInterface](#) ()
Default Constructor.
- virtual [OsiSolverInterface](#) * [clone](#) (bool copyData=true) const
Clone.
- [OsiGlpkSolverInterface](#) (const [OsiGlpkSolverInterface](#) &)
Copy constructor.
- [OsiGlpkSolverInterface](#) & [operator=](#) (const [OsiGlpkSolverInterface](#) &rhs)
Assignment operator.
- virtual [~OsiGlpkSolverInterface](#) ()
Destructor.
- virtual void [reset](#) ()
Resets as if default constructor.

Static Public Member Functions

Static instance counter methods

- static void [incrementInstanceCounter](#) ()
GLPK has a context which must be freed after all GLPK LPs (or MIPs) are freed.
- static void [decrementInstanceCounter](#) ()
GLPK has a context which must be freed after all GLPK LPs (or MIPs) are freed.
- static unsigned int [getNumInstances](#) ()
Return the number of LP/MIP instances of instantiated objects using the GLPK environment.

Protected Member Functions

Protected methods

- virtual void `applyRowCut` (const `OsiRowCut` &rc)
Apply a row cut. Return true if cut was applied.
- virtual void `applyColCut` (const `OsiColCut` &cc)
Apply a column cut (bound adjustment).
- `LPX * getMutableModelPtr` () const
Pointer to the model.

Friends

- void `OsiGlpkSolverInterfaceUnitTest` (const std::string &mpsDir, const std::string &netlibDir)
A function that tests the methods in the `OsiGlpkSolverInterface` class.

GLPK specific public interfaces

- enum `keepCachedFlag` {
`KEEPCACHED_NONE` = 0, `KEEPCACHED_COLUMN` = 1, `KEEPCACHED_ROW` = 2, `KEEPCACHED_MATRIX` = 4,
`KEEPCACHED_RESULTS` = 8, `KEEPCACHED_PROBLEM` = `KEEPCACHED_COLUMN` | `KEEPCACHED_ROW` | `KEEPCACHED_MATRIX`, `KEEPCACHED_ALL` = `KEEPCACHED_PROBLEM` | `KEEPCACHED_RESULTS`,
`FREECACHED_COLUMN` = `KEEPCACHED_PROBLEM` & ~`KEEPCACHED_COLUMN`,
`FREECACHED_ROW` = `KEEPCACHED_PROBLEM` & ~`KEEPCACHED_ROW`, `FREECACHED_MATRIX` = `KEEPCACHED_PROBLEM` & ~`KEEPCACHED_MATRIX`, `FREECACHED_RESULTS` = `KEEPCACHED_ALL` & ~`KEEPCACHED_RESULTS` }
- `LPX * getModelPtr` ()
Get pointer to GLPK model.

Additional Inherited Members

7.15.1 Detailed Description

Definition at line 34 of file `OsiGlpkSolverInterface.hpp`.

7.15.2 Member Enumeration Documentation

7.15.2.1 enum `OsiGlpkSolverInterface::keepCachedFlag`

Enumerator

`KEEPCACHED_NONE` discard all cached data (default)
`KEEPCACHED_COLUMN` column information: objective values, lower and upper bounds, variable types
`KEEPCACHED_ROW` row information: right hand sides, ranges and senses, lower and upper bounds for row
`KEEPCACHED_MATRIX` problem matrix: matrix ordered by column and by row
`KEEPCACHED_RESULTS` LP solution: primal and dual solution, reduced costs, row activities.
`KEEPCACHED_PROBLEM` only discard cached LP solution
`KEEPCACHED_ALL` keep all cached data (similar to `getMutableLpPtr()`)

FREECACHED_COLUMN free only cached column and LP solution information

FREECACHED_ROW free only cached row and LP solution information

FREECACHED_MATRIX free only cached matrix and LP solution information

FREECACHED_RESULTS free only cached LP solution information

Definition at line 635 of file OsiGlpkSolverInterface.hpp.

7.15.3 Member Function Documentation

7.15.3.1 `bool OsiGlpkSolverInterface::setHintParam (OsiHintParam key, bool yesNo = true, OsiHintStrength strength = OsiHintTry, void * = 0) [virtual]`

Set a hint parameter.

The `otherInformation` parameter can be used to pass in an arbitrary block of information which is interpreted by the OSI and the underlying solver. Users are cautioned that this hook is solver-specific.

Implementors: The default implementation completely ignores `otherInformation` and always throws an exception for `OsiForceDo`. This is almost certainly not the behaviour you want; you really should override this method.

Reimplemented from [OsiSolverInterface](#).

7.15.3.2 `CoinWarmStart* OsiGlpkSolverInterface::getEmptyWarmStart () const [inline],[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.

Implements [OsiSolverInterface](#).

Definition at line 117 of file OsiGlpkSolverInterface.hpp.

7.15.3.3 `virtual bool OsiGlpkSolverInterface::setWarmStart (const CoinWarmStart * warmstart) [virtual]`

Set warmstarting information.

Return true/false depending on whether the warmstart information was accepted or not.

Implements [OsiSolverInterface](#).

7.15.3.4 `virtual const char* OsiGlpkSolverInterface::getRowSense () const [virtual]`

Get pointer to array[[getNumRows\(\)](#)] of row constraint senses.

- 'L': \leq constraint
- 'E': $=$ constraint
- 'G': \geq constraint
- 'R': ranged constraint
- 'N': free constraint

Implements [OsiSolverInterface](#).

7.15.3.5 `virtual const double* OsiGlpkSolverInterface::getRightHandSide () const [virtual]`

Get pointer to array[[getNumRows\(\)](#)] of rows right-hand sides.

- if `rowsense()[i] == 'L'` then `rhs()[i] == rowupper()[i]`
- if `rowsense()[i] == 'G'` then `rhs()[i] == rowlower()[i]`
- if `rowsense()[i] == 'R'` then `rhs()[i] == rowupper()[i]`
- if `rowsense()[i] == 'N'` then `rhs()[i] == 0.0`

Implements [OsiSolverInterface](#).

7.15.3.6 `virtual const double* OsiGlpkSolverInterface::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 0.0

Implements [OsiSolverInterface](#).

7.15.3.7 `virtual int OsiGlpkSolverInterface::getIterationCount () const [virtual]`

Get how many iterations it took to solve the problem (whatever "iteration" mean to the solver).

Implements [OsiSolverInterface](#).

7.15.3.8 `virtual std::vector<double*> OsiGlpkSolverInterface::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.)

NOTE for implementers of solver interfaces:

The double pointers in the vector should point to arrays of length [getNumRows\(\)](#) and they should be allocated via `new[]`.

NOTE for users of solver interfaces:

It is the user's responsibility to free the double pointers in the vector using `delete[]`.

Implements [OsiSolverInterface](#).

7.15.3.9 `virtual std::vector<double*> OsiGlpkSolverInterface::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.)

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 [getNumCols\(\)](#) and they should be allocated via `new[]`.

NOTE for users of solver interfaces:

It is the user's responsibility to free the double pointers in the vector using `delete[]`.

Implements [OsiSolverInterface](#).

7.15.3.10 `virtual void OsiGlpkSolverInterface::setColLower (int elementIndex, double elementValue) [virtual]`

Set a single column lower bound
Use `-COIN_DBL_MAX` for -infinity.

Implements [OsiSolverInterface](#).

7.15.3.11 `virtual void OsiGlpkSolverInterface::setColUpper (int elementIndex, double elementValue) [virtual]`

Set a single column upper bound
Use `COIN_DBL_MAX` for infinity.

Implements [OsiSolverInterface](#).

7.15.3.12 `virtual void OsiGlpkSolverInterface::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, indexLast</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

Reimplemented from [OsiSolverInterface](#).

7.15.3.13 `virtual void OsiGlpkSolverInterface::setRowLower (int elementIndex, double elementValue) [virtual]`

Set a single row lower bound
Use `-COIN_DBL_MAX` for -infinity.

Implements [OsiSolverInterface](#).

7.15.3.14 `virtual void OsiGlpkSolverInterface::setRowUpper (int elementIndex, double elementValue) [virtual]`

Set a single row upper bound
Use `COIN_DBL_MAX` for infinity.

Implements [OsiSolverInterface](#).

7.15.3.15 `virtual void OsiGlpkSolverInterface::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>index↔ First,indexLast</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

Reimplemented from [OsiSolverInterface](#).

7.15.3.16 `virtual void OsiGlpkSolverInterface::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>index↔ First,indexLast</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

Reimplemented from [OsiSolverInterface](#).

7.15.3.17 `virtual void OsiGlpkSolverInterface::setColSolution (const double * colsol)` [virtual]

Set the primal solution column values.

`colsol[numcols()]` is an array of values of the problem column variables. These values are copied to memory owned by the solver object or the solver. They will be returned as the result of `colsol()` until changed by another call to `setColsol()` or by a call to any solver routine. Whether the solver makes use of the solution in any way is solver-dependent.

Implements [OsiSolverInterface](#).

7.15.3.18 `virtual void OsiGlpkSolverInterface::setRowPrice (const double * rowprice)` [virtual]

Set dual solution vector.

`rowprice[numrows()]` is an array of values of the problem row dual variables. These values are copied to memory owned by the solver object or the solver. They will be returned as the result of `rowprice()` until changed by another call to `setRowprice()` or by a call to any solver routine. Whether the solver makes use of the solution in any way is solver-dependent.

Implements [OsiSolverInterface](#).

7.15.3.19 `virtual void OsiGlpkSolverInterface::addCol (const CoinPackedVectorBase & vec, const double collb, const double colub, const double obj)` [virtual]

Add a column (primal variable) to the problem.

Implements [OsiSolverInterface](#).

7.15.3.20 `virtual void OsiGlpkSolverInterface::addCols (const int numcols, const CoinPackedVectorBase *const * cols, const double * collb, const double * colub, const double * obj) [virtual]`

Add a set of columns (primal variables) to the problem.

The default implementation simply makes repeated calls to [addCol\(\)](#).

Reimplemented from [OsiSolverInterface](#).

7.15.3.21 `virtual void OsiGlpkSolverInterface::deleteCols (const int num, const int * colIndices) [virtual]`

Remove a set of columns (primal variables) from the problem.

The solver interface for a basis-oriented solver will maintain valid warm start information if all deleted variables are nonbasic.

Implements [OsiSolverInterface](#).

7.15.3.22 `virtual void OsiGlpkSolverInterface::addRow (const CoinPackedVectorBase & vec, const double rowlb, const double rowub) [virtual]`

Add a row (constraint) to the problem.

Implements [OsiSolverInterface](#).

7.15.3.23 `virtual void OsiGlpkSolverInterface::addRow (const CoinPackedVectorBase & vec, const char rowSen, const double rowrhs, const double rowrng) [virtual]`

Add a row (constraint) to the problem.

Implements [OsiSolverInterface](#).

7.15.3.24 `virtual void OsiGlpkSolverInterface::addRows (const int numrows, const CoinPackedVectorBase *const * rows, const double * rowlb, const double * rowub) [virtual]`

Add a set of rows (constraints) to the problem.

The default implementation simply makes repeated calls to [addRow\(\)](#).

Reimplemented from [OsiSolverInterface](#).

7.15.3.25 `virtual void OsiGlpkSolverInterface::addRows (const int numrows, const CoinPackedVectorBase *const * rows, const char * rowSen, const double * rowrhs, const double * rowrng) [virtual]`

Add a set of rows (constraints) to the problem.

The default implementation simply makes repeated calls to [addRow\(\)](#).

Reimplemented from [OsiSolverInterface](#).

7.15.3.26 `virtual void OsiGlpkSolverInterface::deleteRows (const int num, const int * rowIndices) [virtual]`

Delete a set of rows (constraints) from the problem.

The solver interface for a basis-oriented solver will maintain valid warm start information if all deleted rows are loose.

Implements [OsiSolverInterface](#).

7.15.3.27 `virtual void OsiGlpkSolverInterface::loadProblem (const CoinPackedMatrix & matrix, const double * collb, const double * colub, const double * obj, const double * rowlb, const double * rowub) [virtual]`

Load in a problem by copying the arguments (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

Implements [OsiSolverInterface](#).

7.15.3.28 `virtual void OsiGlpkSolverInterface::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.

Implements [OsiSolverInterface](#).

7.15.3.29 `virtual void OsiGlpkSolverInterface::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 0 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

Implements [OsiSolverInterface](#).

7.15.3.30 `virtual void OsiGlpkSolverInterface::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.

Implements [OsiSolverInterface](#).

7.15.3.31 `virtual void OsiGlpkSolverInterface::loadProblem (const int numcols, const int numRows, const int * 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).

7.15.3.32 `virtual void OsiGlpkSolverInterface::loadProblem (const int numcols, const int numRows, const int * 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).

7.15.3.33 `virtual void OsiGlpkSolverInterface::writeMps (const char * filename, const char * extension = "mps", double objSense = 0.0) const [virtual]`

Write the problem into an mps file of the given filename.

If objSense is non zero then -1.0 forces the code to write a maximization objective and +1.0 to write a minimization one.

If 0.0 then solver can do what it wants

Implements [OsiSolverInterface](#).

7.15.3.34 `void OsiGlpkSolverInterface::setRowName (int ndx, std::string name) [virtual]`

Set a row name.

Quietly does nothing if the name discipline (#OsiNameDiscipline) is auto. Quietly fails if the row index is invalid.

Reimplemented from [OsiSolverInterface](#).

7.15.3.35 `void OsiGlpkSolverInterface::setColName (int ndx, std::string name) [virtual]`

Set a column name.

Quietly does nothing if the name discipline (#OsiNameDiscipline) is auto. Quietly fails if the column index is invalid.

Reimplemented from [OsiSolverInterface](#).

7.15.3.36 `static void OsiGlpkSolverInterface::incrementInstanceCounter () [inline], [static]`

GLPK has a context which must be freed after all GLPK LPs (or MIPs) are freed.

It is automatically created when the first LP is created. This method:

- Increments by 1 the number of uses of the GLPK environment.

Definition at line 674 of file OsiGlpkSolverInterface.hpp.

7.15.3.37 `static void OsiGlpkSolverInterface::decrementInstanceCounter () [static]`

GLPK has a context which must be freed after all GLPK LPs (or MIPs) are freed.

This method:

- Decrements by 1 the number of uses of the GLPK environment.
- Deletes the GLPK environment when the number of uses is change to 0 from 1.

7.15.3.38 `virtual void OsiGlpkSolverInterface::applyColCut (const OsiColCut & cc) [protected],[virtual]`

Apply a column cut (bound adjustment).

Return true if cut was applied.

Implements [OsiSolverInterface](#).

7.15.4 Friends And Related Function Documentation

7.15.4.1 `void OsiGlpkSolverInterfaceUnitTest (const std::string & mpsDir, const std::string & netlibDir) [friend]`

A function that tests the methods in the [OsiGlpkSolverInterface](#) class.

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

- OsiGlpkSolverInterface.hpp

7.16 OsiGrbSolverInterface Class Reference

Gurobi Solver Interface.

```
#include <OsiGrbSolverInterface.hpp>
```

Inheritance diagram for OsiGrbSolverInterface:

Collaboration diagram for OsiGrbSolverInterface:

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.

- const char * [getCType](#) () const
return a vector of variable types (continous, binary, integer)
- virtual [OsiSolverInterface::ApplyCutsReturnCode](#) [applyCuts](#) (const [OsiCuts](#) &cs, double effectivenessLb=0.0)
Apply a collection of cuts.

Solve methods

- virtual void [initialSolve](#) ()
Solve initial LP relaxation.
- virtual void [resolve](#) ()
Resolve an LP relaxation after problem modification.
- virtual void [branchAndBound](#) ()
Invoke solver's built-in enumeration algorithm.

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.

- bool [setIntParam](#) ([OsiIntParam](#) key, int value)
Set an integer parameter.
- bool [setDbIParam](#) ([OsiDbIParam](#) key, double value)
Set a double parameter.
- bool [setStrParam](#) ([OsiStrParam](#) key, const std::string &value)
Set a string parameter.
- bool [setHintParam](#) ([OsiHintParam](#) key, bool yesNo=true, [OsiHintStrength](#) strength=[OsiHintTry](#), void *=NULL)
Set a hint parameter.
- bool [getIntParam](#) ([OsiIntParam](#) key, int &value) const
Get an integer parameter.
- bool [getDbIParam](#) ([OsiDbIParam](#) key, double &value) const
Get a double parameter.
- bool [getStrParam](#) ([OsiStrParam](#) key, std::string &value) const
Get a string parameter.
- bool [getHintParam](#) ([OsiHintParam](#) key, bool &yesNo, [OsiHintStrength](#) &strength, void *&otherInformation) const
Get a hint parameter (all information)
- bool [getHintParam](#) ([OsiHintParam](#) key, bool &yesNo, [OsiHintStrength](#) &strength) const
Get a hint parameter (sense and strength only)
- bool [getHintParam](#) ([OsiHintParam](#) key, bool &yesNo) const
Get a hint parameter (sense only)
- void [setMipStart](#) (bool value)
- bool [getMipStart](#) () const

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

- **CoinWarmStart** * `getEmptyWarmStart` () const
Get an empty warm start object.
- virtual **CoinWarmStart** * `getWarmStart` () const
Get warmstarting information.
- virtual bool `setWarmStart` (const **CoinWarmStart** *warmstart)
Set warmstarting information.

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.

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 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 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 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 [setObjCoeffSet](#) (const int *indexFirst, const int *indexLast, const double *coeffList)
Set a a set of objective function coefficients.
- virtual void [setColLower](#) (int elementIndex, double elementValue)
Set a single column lower bound
Use -COIN_DBL_MAX for -infinity.
- virtual void [setColUpper](#) (int elementIndex, double elementValue)
Set a single column upper bound
Use COIN_DBL_MAX for infinity.
- virtual void [setColBounds](#) (int elementIndex, double lower, double upper)
Set a single column lower and upper bound
The default implementation just invokes [setColLower](#) () and [setColUpper](#) ()
- 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 `-COIN_DBL_MAX` for -infinity.*
- virtual void `setRowUpper` (int elementIndex, double elementValue)
 - Set a single row upper bound*
 - Use `COIN_DBL_MAX` for infinity.*
- virtual void `setRowBounds` (int elementIndex, double lower, double upper)
 - Set a single row lower and upper bound*
 - The default implementation just invokes `setRowLower()` and `setRowUpper()`*
- 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()` and over and over again.*

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.*

Naming methods

- virtual void `setRowName` (int ndx, std::string name)
 - Set a row name.*
- virtual void `setColName` (int ndx, std::string name)
 - Set a column name.*

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)
 - Add a 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)
 - Add a set of columns (primal variables) to the problem.*
- virtual void `deleteCols` (const int num, const int *colIndices)
 - Remove a set of columns (primal variables) from the problem.*
- virtual void `addRow` (const **CoinPackedVectorBase** &vec, 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)
Add a row (constraint) to the problem.
- virtual void [addRows](#) (const int numRows, const **CoinPackedVectorBase** *const *rows, const double *rowlb, const double *rowub)
Add a set of rows (constraints) to the problem.
- virtual void [addRows](#) (const int numRows, const **CoinPackedVectorBase** *const *rows, const char *rowsen, const double *rowrhs, const double *rowrng)
Add a set of rows (constraints) to the problem.
- virtual void [deleteRows](#) (const int num, const int *rowIndices)
Delete a set of rows (constraints) from the problem.

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 int numcols, const int numRows, const int *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 int *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 [readMps](#) (const char *filename, const char *extension="mps")
Read an mps file from the given filename.
- 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.

Constructors and destructor

- [OsiGrbSolverInterface](#) (bool use_local_env=false)
Default Constructor.
- [OsiGrbSolverInterface](#) (GRBEnv *localgrbenv)
Constructor that takes a gurobi environment and assumes membership.
- virtual [OsiSolverInterface](#) * [clone](#) (bool copyData=true) const
Clone.
- [OsiGrbSolverInterface](#) (const [OsiGrbSolverInterface](#) &)
Copy constructor.
- [OsiGrbSolverInterface](#) & [operator=](#) (const [OsiGrbSolverInterface](#) &rhs)
Assignment operator.

- virtual `~OsiGrbSolverInterface ()`
Destructor.
- virtual void `reset ()`
Resets as if default constructor.

OsiSimplexInterface methods

Gurobi adds a slack with coeff +1 in "<=" and "=" constraints, with coeff -1 in ">=", slack being non negative.

We switch in order to get a "Clp tableau" where all the slacks have coefficient +1 in the original tableau.

If a slack for ">=" is non basic, invB is not changed; column of the slack in the optimal tableau is flipped.

If a slack for ">=" is basic, corresp. row of invB is flipped; whole row of the optimal tableau is flipped; then whole column for the slack in opt tableau is flipped.

Ranged rows are not supported. It might work, but no guarantee is given.

- virtual int `canDoSimplexInterface ()` const
Returns 1 if can just do getBlmv etc 2 if has all OsiSimplex methods and 0 if it has none.
- virtual void `enableSimplexInterface (int doingPrimal)`
Useless function, defined only for compatibility with OsiSimplexInterface.
- virtual void `disableSimplexInterface ()`
Useless function, defined only for compatibility with OsiSimplexInterface.
- virtual void `enableFactorization ()` const
Useless function, defined only for compatibility with OsiSimplexInterface.
- virtual void `disableFactorization ()` const
Useless function, defined only for compatibility with OsiSimplexInterface.
- virtual bool `basisIsAvailable ()` const
Returns true if a basis is available.
- virtual void `getBasisStatus (int *cstat, int *rstat)` const
Returns a basis status of the structural/artificial variables At present as warm start i.e 0: free, 1: basic, 2: upper, 3: lower.
- void `switchToLP ()`
Get indices of the pivot variable in each row (order of indices corresponds to the order of elements in a vector returned by `getBlmvACol()` and `getBlmvCol()`).
- void `switchToMIP ()`
switches Gurobi to prob type MIP

Static Public Member Functions

Static instance counter methods

- static void `incrementInstanceCounter ()`
Gurobi has a context which must be created prior to all other Gurobi calls.
- static void `decrementInstanceCounter ()`
Gurobi has a context which should be deleted after Gurobi calls.
- static void `setEnvironment (GRBEnv *globalenv)`
sets the global gurobi environment to a user given one
- static unsigned int `getNumInstances ()`
Return the number of instances of instantiated objects using Gurobi services.

Protected Member Functions

Protected methods

- virtual void `applyRowCut (const OsiRowCut &rc)`
Apply a row cut. Return true if cut was applied.
- virtual void `applyColCut (const OsiColCut &cc)`
Apply a column cut (bound adjustment).

Friends

- void [OsiGrbSolverInterfaceUnitTest](#) (const std::string &mpsDir, const std::string &netlibDir)

A function that tests the methods in the [OsiGrbSolverInterface](#) class.

Gurobi specific public interfaces

- enum [keepCachedFlag](#) {
[KEEPCACHED_NONE](#) = 0, [KEEPCACHED_COLUMN](#) = 1, [KEEPCACHED_ROW](#) = 2, [KEEPCACHED_MATRIX](#) = 4,
[KEEPCACHED_RESULTS](#) = 8, [KEEPCACHED_PROBLEM](#) = [KEEPCACHED_COLUMN](#) | [KEEPCACHED_ROW](#) | [KEEPCACHED_MATRIX](#), [KEEPCACHED_ALL](#) = [KEEPCACHED_PROBLEM](#) | [KEEPCACHED_RESULTS](#),
[FREECACHED_COLUMN](#) = [KEEPCACHED_PROBLEM](#) & ~[KEEPCACHED_COLUMN](#),
[FREECACHED_ROW](#) = [KEEPCACHED_PROBLEM](#) & ~[KEEPCACHED_ROW](#), [FREECACHED_MATRIX](#) = [KEEPCACHED_PROBLEM](#) & ~[KEEPCACHED_MATRIX](#), [FREECACHED_RESULTS](#) = [KEEPCACHED_ALL](#) & ~[KEEPCACHED_RESULTS](#) }

Get pointer to Gurobi model and free all specified cached data entries (combined with logical or-operator '|'):

- GRBmodel * [getLpPtr](#) (int keepCached=[KEEPCACHED_NONE](#))
- GRBenv * [getEnvironmentPtr](#) () const
- bool [isDemoLicense](#) () const

Method to access Gurobi environment pointer.

Return whether the current Gurobi environment runs in demo mode.

Additional Inherited Members

7.16.1 Detailed Description

Gurobi Solver Interface.

Instantiation of [OsiGrbSolverInterface](#) for Gurobi

Definition at line 29 of file [OsiGrbSolverInterface.hpp](#).

7.16.2 Member Enumeration Documentation

7.16.2.1 enum [OsiGrbSolverInterface::keepCachedFlag](#)

Get pointer to Gurobi model and free all specified cached data entries (combined with logical or-operator '|'):

Enumerator

[KEEPCACHED_NONE](#) discard all cached data (default)

[KEEPCACHED_COLUMN](#) column information: objective values, lower and upper bounds, variable types

[KEEPCACHED_ROW](#) row information: right hand sides, ranges and senses, lower and upper bounds for row

[KEEPCACHED_MATRIX](#) problem matrix: matrix ordered by column and by row

[KEEPCACHED_RESULTS](#) LP solution: primal and dual solution, reduced costs, row activities.

[KEEPCACHED_PROBLEM](#) only discard cached LP solution

[KEEPCACHED_ALL](#) keep all cached data (similar to [getMutableLpPtr\(\)](#))

[FREECACHED_COLUMN](#) free only cached column and LP solution information

FREECACHED_ROW free only cached row and LP solution information

FREECACHED_MATRIX free only cached matrix and LP solution information

FREECACHED_RESULTS free only cached LP solution information

Definition at line 546 of file OsiGrbSolverInterface.hpp.

7.16.3 Member Function Documentation

7.16.3.1 `bool OsiGrbSolverInterface::setHintParam (OsiHintParam key, bool yesNo = true, OsiHintStrength strength = OsiHintTry, void * = NULL) [virtual]`

Set a hint parameter.

The `otherInformation` parameter can be used to pass in an arbitrary block of information which is interpreted by the OSI and the underlying solver. Users are cautioned that this hook is solver-specific.

Implementors: The default implementation completely ignores `otherInformation` and always throws an exception for `OsiForceDo`. This is almost certainly not the behaviour you want; you really should override this method.

Reimplemented from [OsiSolverInterface](#).

7.16.3.2 `bool OsiGrbSolverInterface::getHintParam (OsiHintParam key, bool & yesNo, OsiHintStrength & strength, void *& otherInformation) const [virtual]`

Get a hint parameter (all information)

Return all available information for the hint: sense, strength, and any extra information associated with the hint.

Implementors: The default implementation will always set `otherInformation` to `NULL`. This is almost certainly not the behaviour you want; you really should override this method.

Reimplemented from [OsiSolverInterface](#).

7.16.3.3 `bool OsiGrbSolverInterface::getHintParam (OsiHintParam key, bool & yesNo, OsiHintStrength & strength) const [virtual]`

Get a hint parameter (sense and strength only)

Return only the sense and strength of the hint.

Reimplemented from [OsiSolverInterface](#).

7.16.3.4 `bool OsiGrbSolverInterface::getHintParam (OsiHintParam key, bool & yesNo) const [virtual]`

Get a hint parameter (sense only)

Return only the sense (true/false) of the hint.

Reimplemented from [OsiSolverInterface](#).

7.16.3.5 `CoinWarmStart* OsiGrbSolverInterface::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.

Implements [OsiSolverInterface](#).

7.16.3.6 `virtual bool OsiGrbSolverInterface::setWarmStart (const CoinWarmStart * warmstart) [virtual]`

Set warmstarting information.

Return true/false depending on whether the warmstart information was accepted or not.

Implements [OsiSolverInterface](#).

7.16.3.7 `virtual const char* OsiGrbSolverInterface::getRowSense () const [virtual]`

Get pointer to array[[getNumRows\(\)](#)] of row constraint senses.

- 'L': \leq constraint
- 'E': = constraint
- 'G': \geq constraint
- 'R': ranged constraint
- 'N': free constraint

Implements [OsiSolverInterface](#).

7.16.3.8 `virtual const double* OsiGrbSolverInterface::getRightHandSide () const [virtual]`

Get pointer to array[[getNumRows\(\)](#)] of rows right-hand sides.

- if `rowsense()[i] == 'L'` then `rhs()[i] == rowupper()[i]`
- if `rowsense()[i] == 'G'` then `rhs()[i] == rowlower()[i]`
- if `rowsense()[i] == 'R'` then `rhs()[i] == rowupper()[i]`
- if `rowsense()[i] == 'N'` then `rhs()[i] == 0.0`

Implements [OsiSolverInterface](#).

7.16.3.9 `virtual const double* OsiGrbSolverInterface::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 0.0

Implements [OsiSolverInterface](#).

7.16.3.10 `virtual int OsiGrbSolverInterface::getIterationCount () const [virtual]`

Get how many iterations it took to solve the problem (whatever "iteration" mean to the solver.

Implements [OsiSolverInterface](#).

7.16.3.11 `virtual std::vector<double*> OsiGrbSolverInterface::getDualRays (int maxNumRays, bool fullRay = false) const [virtual]`

Get as many dual rays as the solver can provide.

(In case of proven primal infeasibility there should be at least one.)

The first [getNumRows\(\)](#) ray components will always be associated with the row duals (as returned by [getRowPrice\(\)](#)). If `fullRay` is true, the final [getNumCols\(\)](#) entries will correspond to the ray components associated with the nonbasic variables. If the full ray is requested and the method cannot provide it, it will throw an exception.

NOTE for implementers of solver interfaces:

The double pointers in the vector should point to arrays of length [getNumRows\(\)](#) and they should be allocated via `new[]`.

NOTE for users of solver interfaces:

It is the user's responsibility to free the double pointers in the vector using `delete[]`.

Implements [OsiSolverInterface](#).

7.16.3.12 `virtual std::vector<double*> OsiGrbSolverInterface::getPrimalRays (int maxNumRays) const [virtual]`

Get as many primal rays as the solver can provide.

(In case of proven dual infeasibility there should be at least one.)

NOTE for implementers of solver interfaces:

The double pointers in the vector should point to arrays of length [getNumCols\(\)](#) and they should be allocated via `new[]`.

NOTE for users of solver interfaces:

It is the user's responsibility to free the double pointers in the vector using `delete[]`.

Implements [OsiSolverInterface](#).

7.16.3.13 `virtual void OsiGrbSolverInterface::setColLower (int elementIndex, double elementValue) [virtual]`

Set a single column lower bound
Use `-COIN_DBL_MAX` for -infinity.

Implements [OsiSolverInterface](#).

7.16.3.14 `virtual void OsiGrbSolverInterface::setColUpper (int elementIndex, double elementValue) [virtual]`

Set a single column upper bound
Use `COIN_DBL_MAX` for infinity.

Implements [OsiSolverInterface](#).

7.16.3.15 `virtual void OsiGrbSolverInterface::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

<code><code>[indexfirst, index↵ Last]</code></code>	contains the indices of the constraints whose either bound changes
<code>boundList</code>	the new lower/upper bound pairs for the variables

Reimplemented from [OsiSolverInterface](#).

7.16.3.16 `virtual void OsiGrbSolverInterface::setRowLower (int elementIndex, double elementValue)` [virtual]

Set a single row lower bound

Use -COIN_DBL_MAX for -infinity.

Implements [OsiSolverInterface](#).

7.16.3.17 `virtual void OsiGrbSolverInterface::setRowUpper (int elementIndex, double elementValue)` [virtual]

Set a single row upper bound

Use COIN_DBL_MAX for infinity.

Implements [OsiSolverInterface](#).

7.16.3.18 `virtual void OsiGrbSolverInterface::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

<code><code>[indexfirst, index↵ Last]</code></code>	contains the indices of the constraints whose either bound changes
<code>boundList</code>	the new lower/upper bound pairs for the constraints

Reimplemented from [OsiSolverInterface](#).

7.16.3.19 `virtual void OsiGrbSolverInterface::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\(\)](#) and over and over again.

Parameters

<code><code>[indexfirst, index↵ Last]</code></code>	contains the indices of the constraints whose type changes
<code>senseList</code>	the new senses
<code>rhsList</code>	the new right hand sides

<i>rangeList</i>	the new ranges
------------------	----------------

Reimplemented from [OsiSolverInterface](#).

7.16.3.20 `virtual void OsiGrbSolverInterface::setColSolution (const double * colsol) [virtual]`

Set the primal solution column values.

`colsol[numcols()]` is an array of values of the problem column variables. These values are copied to memory owned by the solver object or the solver. They will be returned as the result of `colsol()` until changed by another call to `setColsol()` or by a call to any solver routine. Whether the solver makes use of the solution in any way is solver-dependent.

Implements [OsiSolverInterface](#).

7.16.3.21 `virtual void OsiGrbSolverInterface::setRowPrice (const double * rowprice) [virtual]`

Set dual solution vector.

`rowprice[numrows()]` is an array of values of the problem row dual variables. These values are copied to memory owned by the solver object or the solver. They will be returned as the result of `rowprice()` until changed by another call to `setRowprice()` or by a call to any solver routine. Whether the solver makes use of the solution in any way is solver-dependent.

Implements [OsiSolverInterface](#).

7.16.3.22 `virtual void OsiGrbSolverInterface::addCol (const CoinPackedVectorBase & vec, const double collb, const double colub, const double obj) [virtual]`

Add a column (primal variable) to the problem.

Implements [OsiSolverInterface](#).

7.16.3.23 `virtual void OsiGrbSolverInterface::addCols (const int numcols, const CoinPackedVectorBase *const * cols, const double * collb, const double * colub, const double * obj) [virtual]`

Add a set of columns (primal variables) to the problem.

The default implementation simply makes repeated calls to [addCol\(\)](#).

Reimplemented from [OsiSolverInterface](#).

7.16.3.24 `virtual void OsiGrbSolverInterface::deleteCols (const int num, const int * colIndices) [virtual]`

Remove a set of columns (primal variables) from the problem.

The solver interface for a basis-oriented solver will maintain valid warm start information if all deleted variables are nonbasic.

Implements [OsiSolverInterface](#).

7.16.3.25 `virtual void OsiGrbSolverInterface::addRow (const CoinPackedVectorBase & vec, const double rowlb, const double rowub) [virtual]`

Add a row (constraint) to the problem.

Implements [OsiSolverInterface](#).

7.16.3.26 `virtual void OsiGrbSolverInterface::addRow (const CoinPackedVectorBase & vec, const char rowsen, const double rowrhs, const double rowrng) [virtual]`

Add a row (constraint) to the problem.

Implements [OsiSolverInterface](#).

7.16.3.27 `virtual void OsiGrbSolverInterface::addRows (const int numrows, const CoinPackedVectorBase *const * rows, const double * rowlb, const double * rowub) [virtual]`

Add a set of rows (constraints) to the problem.

The default implementation simply makes repeated calls to [addRow\(\)](#).

Reimplemented from [OsiSolverInterface](#).

7.16.3.28 `virtual void OsiGrbSolverInterface::addRows (const int numrows, const CoinPackedVectorBase *const * rows, const char * rowsen, const double * rowrhs, const double * rowrng) [virtual]`

Add a set of rows (constraints) to the problem.

The default implementation simply makes repeated calls to [addRow\(\)](#).

Reimplemented from [OsiSolverInterface](#).

7.16.3.29 `virtual void OsiGrbSolverInterface::deleteRows (const int num, const int * rowIndices) [virtual]`

Delete a set of rows (constraints) from the problem.

The solver interface for a basis-oriented solver will maintain valid warm start information if all deleted rows are loose.

Implements [OsiSolverInterface](#).

7.16.3.30 `virtual void OsiGrbSolverInterface::loadProblem (const CoinPackedMatrix & matrix, const double * colub, const double * colub, const double * obj, const double * rowlb, const double * rowub) [virtual]`

Load in a problem by copying the arguments (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
- `colub`: all columns have lower bound 0
- `rowub`: all rows have upper bound infinity
- `rowlb`: all rows have lower bound -infinity
- `obj`: all variables have 0 objective coefficient

Implements [OsiSolverInterface](#).

7.16.3.31 `virtual void OsiGrbSolverInterface::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.

Implements [OsiSolverInterface](#).

7.16.3.32 `virtual void OsiGrbSolverInterface::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 0 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

Implements [OsiSolverInterface](#).

7.16.3.33 `virtual void OsiGrbSolverInterface::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.

Implements [OsiSolverInterface](#).

7.16.3.34 `virtual void OsiGrbSolverInterface::loadProblem (const int numcols, const int numRows, const int * 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).

7.16.3.35 `virtual void OsiGrbSolverInterface::loadProblem (const int numcols, const int numRows, const int * 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).

7.16.3.36 `virtual void OsiGrbSolverInterface::writeMps (const char * filename, const char * extension = "mps", double objSense = 0.0) const` [virtual]

Write the problem into an mps file of the given filename.

If objSense is non zero then -1.0 forces the code to write a maximization objective and +1.0 to write a minimization one. If 0.0 then solver can do what it wants

Implements [OsiSolverInterface](#).

7.16.3.37 `static void OsiGrbSolverInterface::incrementInstanceCounter ()` [static]

Gurobi has a context which must be created prior to all other Gurobi calls.

This method:

- Increments by 1 the number of uses of the Gurobi environment.
- Creates the Gurobi context when the number of uses is change to 1 from 0.

7.16.3.38 `static void OsiGrbSolverInterface::decrementInstanceCounter ()` [static]

Gurobi has a context which should be deleted after Gurobi calls.

This method:

- Decrements by 1 the number of uses of the Gurobi environment.
- Deletes the Gurobi context when the number of uses is change to 0 from 1.

7.16.3.39 `void OsiGrbSolverInterface::switchToLP ()`

Get indices of the pivot variable in each row (order of indices corresponds to the order of elements in a vector returned by [getBlvACol\(\)](#) and [getBlvCol\(\)](#)).

switches Gurobi to prob type LP

7.16.3.40 `virtual OsiSolverInterface::ApplyCutsReturnCode OsiGrbSolverInterface::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()`

Reimplemented from [OsiSolverInterface](#).

7.16.3.41 `virtual void OsiGrbSolverInterface::applyColCut (const OsiColCut & cc)` [protected],[virtual]

Apply a column cut (bound adjustment).

Return true if cut was applied.

Implements [OsiSolverInterface](#).

7.16.4 Friends And Related Function Documentation

7.16.4.1 `void OsiGrbSolverInterfaceUnitTest (const std::string & mpsDir, const std::string & netlibDir)` [friend]

A function that tests the methods in the [OsiGrbSolverInterface](#) class.

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

- `OsiGrbSolverInterface.hpp`

7.17 OsiHotInfo Class Reference

This class contains the result of strong branching on a variable When created it stores enough information for strong branching.

```
#include <OsiChooseVariable.hpp>
```

Collaboration diagram for OsiHotInfo:

Public Member Functions

- [OsiHotInfo](#) ()
Default Constructor.
- [OsiHotInfo](#) ([OsiSolverInterface](#) *solver, const [OsiBranchingInformation](#) *info, const [OsiObject](#) *const *objects, int whichObject)
Constructor from useful information.
- [OsiHotInfo](#) (const [OsiHotInfo](#) &)
Copy constructor.
- [OsiHotInfo](#) & `operator=` (const [OsiHotInfo](#) &rhs)
Assignment operator.
- virtual [OsiHotInfo](#) * `clone` () const
Clone.
- virtual `~OsiHotInfo` ()
Destructor.
- int `updateInformation` (const [OsiSolverInterface](#) *solver, const [OsiBranchingInformation](#) *info, [OsiChooseVariable](#) *choose)
Fill in useful information after strong branch.

- double [originalObjectiveValue](#) () const
Original objective value.
- double [upChange](#) () const
Up change - invalid if n-way.
- double [downChange](#) () const
Down change - invalid if n-way.
- void [setUpChange](#) (double value)
Set up change - invalid if n-way.
- void [setDownChange](#) (double value)
Set down change - invalid if n-way.
- double [change](#) (int k) const
Change on way k.
- int [upIterationCount](#) () const
Up iteration count - invalid if n-way.
- int [downIterationCount](#) () const
Down iteration count - invalid if n-way.
- int [iterationCount](#) (int k) const
Iteration count on way k.
- int [upStatus](#) () const
Up status - invalid if n-way.
- int [downStatus](#) () const
Down status - invalid if n-way.
- void [setUpStatus](#) (int value)
Set up status - invalid if n-way.
- void [setDownStatus](#) (int value)
Set down status - invalid if n-way.
- int [status](#) (int k) const
Status on way k.
- [OsiBranchingObject](#) * [branchingObject](#) () const
Branching object.

Protected Attributes

- double [originalObjectiveValue_](#)
Original objective value.
- double * [changes_](#)
Objective changes.
- int * [iterationCounts_](#)
Iteration counts.
- int * [statuses_](#)
Status - 1 - not done 0 - feasible and finished 1 - infeasible 2 - not finished.
- [OsiBranchingObject](#) * [branchingObject_](#)
Branching object.
- int [whichObject_](#)
Which object on list.

7.17.1 Detailed Description

This class contains the result of strong branching on a variable. When created it stores enough information for strong branching.

Definition at line 432 of file `OsiChooseVariable.hpp`.

7.17.2 Member Function Documentation

7.17.2.1 `int OsiHotInfo::updateInformation (const OsiSolverInterface * solver, const OsiBranchingInformation * info, OsiChooseVariable * choose)`

Fill in useful information after strong branch.

Return status

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

- `OsiChooseVariable.hpp`

7.18 OsiIntegerBranchingObject Class Reference

Simple branching object for an integer variable.

```
#include <OsiBranchingObject.hpp>
```

Inheritance diagram for `OsiIntegerBranchingObject`:

Collaboration diagram for `OsiIntegerBranchingObject`:

Public Member Functions

- `OsiIntegerBranchingObject ()`
Default constructor.
- `OsiIntegerBranchingObject (OsiSolverInterface *solver, const OsiSimpleInteger *originalObject, int way, double value)`
Create a standard floor/ceiling branch object.
- `OsiIntegerBranchingObject (OsiSolverInterface *solver, const OsiSimpleInteger *originalObject, int way, double value, double downUpperBound, double upLowerBound)`
Create a standard floor/ceiling branch object.
- `OsiIntegerBranchingObject (const OsiIntegerBranchingObject &)`
Copy constructor.
- `OsiIntegerBranchingObject & operator= (const OsiIntegerBranchingObject &rhs)`
Assignment operator.
- `virtual OsiBranchingObject * clone () const`
Clone.
- `virtual ~OsiIntegerBranchingObject ()`
Destructor.
- `virtual double branch (OsiSolverInterface *solver)`
Sets the bounds for the variable according to the current arm of the branch and advances the object state to the next arm.
- `virtual void print (const OsiSolverInterface *solver=NULL)`
Print something about branch - only if log level high.

Protected Attributes

- double `down_` [2]
Lower [0] and upper [1] bounds for the down arm (`way_ = -1`)
- double `up_` [2]
Lower [0] and upper [1] bounds for the up arm (`way_ = 1`)

7.18.1 Detailed Description

Simple branching object for an integer variable.

This object can specify a two-way branch on an integer variable. For each arm of the branch, the upper and lower bounds on the variable can be independently specified. 0 -> down, 1-> up.

Definition at line 607 of file `OsiBranchingObject.hpp`.

7.18.2 Constructor & Destructor Documentation

7.18.2.1 `OsiIntegerBranchingObject::OsiIntegerBranchingObject (OsiSolverInterface * solver, const OsiSimpleInteger * originalObject, int way, double value)`

Create a standard floor/ceiling branch object.

Specifies a simple two-way branch. Let `value = x*`. One arm of the branch will be $lb \leq x \leq \text{floor}(x*)$, the other $\text{ceil}(x*) \leq x \leq ub$. Specify `way = -1` to set the object state to perform the down arm first, `way = 1` for the up arm.

7.18.2.2 `OsiIntegerBranchingObject::OsiIntegerBranchingObject (OsiSolverInterface * solver, const OsiSimpleInteger * originalObject, int way, double value, double downUpperBound, double upLowerBound)`

Create a standard floor/ceiling branch object.

Specifies a simple two-way branch in a more flexible way. One arm of the branch will be $lb \leq x \leq \text{downUpperBound}$, the other $\text{upLowerBound} \leq x \leq ub$. Specify `way = -1` to set the object state to perform the down arm first, `way = 1` for the up arm.

7.18.3 Member Function Documentation

7.18.3.1 `virtual double OsiIntegerBranchingObject::branch (OsiSolverInterface * solver)` [virtual]

Sets the bounds for the variable according to the current arm of the branch and advances the object state to the next arm.

state. Returns change in guessed objective on next branch

Implements [OsiTwoWayBranchingObject](#).

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

- `OsiBranchingObject.hpp`

7.19 OsiLotsize Class Reference

Lotsize class.

```
#include <OsiBranchingObject.hpp>
```

Inheritance diagram for OsiLotsize:

Collaboration diagram for OsiLotsize:

Public Member Functions

- virtual [OsiObject](#) * [clone](#) () const
Clone.
- virtual double [infeasibility](#) (const [OsiBranchingInformation](#) *info, int &[whichWay](#)) const
Infeasibility - large is 0.5.
- virtual double [feasibleRegion](#) ([OsiSolverInterface](#) *solver, const [OsiBranchingInformation](#) *info) const
Set bounds to contain the current solution.
- virtual [OsiBranchingObject](#) * [createBranch](#) ([OsiSolverInterface](#) *solver, const [OsiBranchingInformation](#) *info, int way) const
Creates a branching object.
- void [setColumnNumber](#) (int value)
Set solver column number.
- virtual int [columnNumber](#) () const
Column number if single column object -1 otherwise, so returns ≥ 0 Used by heuristics.
- virtual void [resetBounds](#) (const [OsiSolverInterface](#) *solver)
Reset original upper and lower bound values from the solver.
- bool [findRange](#) (double value, double integerTolerance) const
Finds range of interest so value is feasible in range range_ or infeasible between hi[range_] and lo[range_+1].
- virtual void [floorCeiling](#) (double &floorLotsize, double &ceilingLotsize, double value, double tolerance) const
Returns floor and ceiling.
- double [originalLowerBound](#) () const
Original bounds.
- int [rangeType](#) () const
Type - 1 points, 2 ranges.
- int [numberRanges](#) () const
Number of points.
- double * [bound](#) () const
Ranges.
- virtual void [resetSequenceEtc](#) (int numberColumns, const int *originalColumns)
Change column numbers after preprocessing.
- virtual double [upEstimate](#) () const
Return "up" estimate (default 1.0e-5)
- virtual double [downEstimate](#) () const
Return "down" estimate (default 1.0e-5)
- virtual bool [canHandleShadowPrices](#) () const
Return true if knows how to deal with Pseudo Shadow Prices.
- virtual bool [canDoHeuristics](#) () const
Return true if object can take part in normal heuristics.

Additional Inherited Members

7.19.1 Detailed Description

Lotsize class.

Definition at line 827 of file OsiBranchingObject.hpp.

7.19.2 Member Function Documentation

7.19.2.1 `virtual double OsiLotsize::feasibleRegion (OsiSolverInterface * solver, const OsiBranchingInformation * info) const [virtual]`

Set bounds to contain the current solution.

More precisely, for the variable associated with this object, take the value given in the current solution, force it within the current bounds if required, then set the bounds to fix the variable at the integer nearest the solution value. Returns amount it had to move variable.

Implements [OsiObject](#).

7.19.2.2 `virtual OsiBranchingObject* OsiLotsize::createBranch (OsiSolverInterface * solver, const OsiBranchingInformation * info, int way) const [virtual]`

Creates a branching object.

The preferred direction is set by `way`, 0 for down, 1 for up.

Reimplemented from [OsiObject](#).

7.19.2.3 `virtual void OsiLotsize::resetBounds (const OsiSolverInterface * solver) [virtual]`

Reset original upper and lower bound values from the solver.

Handy for updating bounds held in this object after bounds held in the solver have been tightened.

Reimplemented from [OsiObject](#).

7.19.2.4 `bool OsiLotsize::findRange (double value, double integerTolerance) const`

Finds range of interest so value is feasible in range `range_` or infeasible between `hi[range_]` and `lo[range_+1]`.

Returns true if feasible.

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

- OsiBranchingObject.hpp

7.20 OsiLotsizeBranchingObject Class Reference

Lotsize branching object.

```
#include <OsiBranchingObject.hpp>
```

Inheritance diagram for OsiLotsizeBranchingObject:

Collaboration diagram for OsiLotsizeBranchingObject:

Public Member Functions

- [OsiLotsizeBranchingObject](#) ()
Default constructor.
- [OsiLotsizeBranchingObject](#) ([OsiSolverInterface](#) *solver, const [OsiLotsize](#) *originalObject, int way, double value)
Create a lotsize floor/ceiling branch object.
- [OsiLotsizeBranchingObject](#) (const [OsiLotsizeBranchingObject](#) &)
Copy constructor.
- [OsiLotsizeBranchingObject](#) & operator= (const [OsiLotsizeBranchingObject](#) &rhs)
Assignment operator.
- virtual [OsiBranchingObject](#) * clone () const
Clone.
- virtual ~[OsiLotsizeBranchingObject](#) ()
Destructor.
- virtual double branch ([OsiSolverInterface](#) *solver)
Sets the bounds for the variable according to the current arm of the branch and advances the object state to the next arm.
- virtual void print (const [OsiSolverInterface](#) *solver=NULL)
Print something about branch - only if log level high.

Protected Attributes

- double down_ [2]
Lower [0] and upper [1] bounds for the down arm (way_ = -1)
- double up_ [2]
Lower [0] and upper [1] bounds for the up arm (way_ = 1)

7.20.1 Detailed Description

Lotsize branching object.

This object can specify a two-way branch on an integer variable. For each arm of the branch, the upper and lower bounds on the variable can be independently specified.

Variable_ holds the index of the integer variable in the integerVariable_ array of the model.

Definition at line 957 of file OsiBranchingObject.hpp.

7.20.2 Constructor & Destructor Documentation

7.20.2.1 [OsiLotsizeBranchingObject::OsiLotsizeBranchingObject](#) ([OsiSolverInterface](#) * solver, const [OsiLotsize](#) * originalObject, int way, double value)

Create a lotsize floor/ceiling branch object.

Specifies a simple two-way branch. Let value = x*. One arm of the branch will be is lb <= x <= valid range below(x*), the other valid range above(x*) <= x <= ub. Specify way = -1 to set the object state to perform the down arm first, way = 1 for the up arm.

7.20.3 Member Function Documentation

7.20.3.1 virtual double OsiLotsizeBranchingObject::branch (OsiSolverInterface * solver) [virtual]

Sets the bounds for the variable according to the current arm of the branch and advances the object state to the next arm.

state. Returns change in guessed objective on next branch

Implements [OsiTwoWayBranchingObject](#).

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

- OsiBranchingObject.hpp

7.21 OsiMskSolverInterface Class Reference

Inheritance diagram for OsiMskSolverInterface:

Collaboration diagram for OsiMskSolverInterface:

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.
- const char * [getCtype](#) () const
return a vector of variable types (continous, binary, integer)

Solve methods

- virtual void [initialSolve](#) ()
Solve initial LP relaxation.
- virtual void [resolve](#) ()
Resolve an LP relaxation after problem modification.
- virtual void [branchAndBound](#) ()
Invoke solver's built-in enumeration algorithm.

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.

- bool [setIntParam](#) (OsiIntParam key, int value)

- *Set an integer parameter.*
- bool [setDbiParam](#) (OsiDbiParam key, double value)
Set a double parameter.
- bool [setStrParam](#) (OsiStrParam key, const std::string &value)
Set a string parameter.
- bool [getIntParam](#) (OsiIntParam key, int &value) const
Get an integer parameter.
- bool [getDbiParam](#) (OsiDbiParam key, double &value) const
Get a double parameter.
- bool [getStrParam](#) (OsiStrParam key, std::string &value) const
Get a string parameter.

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?
- virtual bool [isLicenseError](#) () const
Has there been a license problem?
- int [getRescode](#) () const
Get rescode return of last Mosek optimizer call.

WarmStart related methods

- **CoinWarmStart** * [getEmptyWarmStart](#) () const
Get an empty warm start object.
- virtual **CoinWarmStart** * [getWarmStart](#) () const
Get warmstarting information.
- virtual bool [setWarmStart](#) (const **CoinWarmStart** *warmstart)
Set warmstarting information.

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.

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 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 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 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 [setObjCoeffSet](#) (const int *indexFirst, const int *indexLast, const double *coeffList)
Set a a set of objective function coefficients.
- virtual void [setColLower](#) (int elementIndex, double elementValue)
Set a single column lower bound
Use -COIN_DBL_MAX for -infinity.
- virtual void [setColUpper](#) (int elementIndex, double elementValue)
Set a single column upper bound
Use COIN_DBL_MAX for infinity.
- virtual void [setColBounds](#) (int elementIndex, double lower, double upper)
Set a single column lower and upper bound
The default implementation just invokes [setColLower\(\)](#) and [setColUpper\(\)](#)
- 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 -COIN_DBL_MAX for -infinity.
- virtual void [setRowUpper](#) (int elementIndex, double elementValue)
Set a single row upper bound
Use COIN_DBL_MAX for infinity.
- virtual void [setRowBounds](#) (int elementIndex, double lower, double upper)
Set a single row lower and upper bound
The default implementation just invokes [setRowLower\(\)](#) and [setRowUpper\(\)](#)
- 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\(\)](#) and over and over again.

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.

**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)
Add a 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)
Add a set of columns (primal variables) to the problem.
- virtual void [deleteCols](#) (const int num, const int *colIndices)
Remove a set of columns (primal variables) from the problem.
- virtual void [addRow](#) (const **CoinPackedVectorBase** &vec, 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)
Add a row (constraint) to the problem.
- virtual void [addRows](#) (const int numrows, const **CoinPackedVectorBase** *const *rows, const double *rowlb, const double *rowub)
Add a set of rows (constraints) to the problem.
- virtual void [addRows](#) (const int numrows, const **CoinPackedVectorBase** *const *rows, const char *rowsen, const double *rowrhs, const double *rowrng)
Add a set of rows (constraints) to the problem.
- virtual void [deleteRows](#) (const int num, const int *rowIndices)
Delete a set of rows (constraints) from the problem.

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 int numcols, const int numrows, const int *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 int *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 [readMps](#) (const char *filename, const char *extension="mps")
Read an mps file from the given filename.
- 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.

Message handling

- void [passInMessageHandler](#) (**CoinMessageHandler** *handler)
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.

Constructors and destructor

- [OsiMskSolverInterface](#) (MSKenv_t mskenv=NULL)
Default Constructor optional argument mskenv can be used to reach in an initialized user environment OsiMsk assumes membership of mskenv, so it will be freed when the last instantiation of OsiMsk is deleted.
- virtual [OsiSolverInterface](#) * [clone](#) (bool copyData=true) const
Clone.
- [OsiMskSolverInterface](#) (const [OsiMskSolverInterface](#) &)
Copy constructor.
- [OsiMskSolverInterface](#) & [operator=](#) (const [OsiMskSolverInterface](#) &rhs)
Assignment operator.
- virtual [~OsiMskSolverInterface](#) ()
Destructor.

Static Public Member Functions

Static instance counter methods

- static void [incrementInstanceCounter](#) ()
MOSEK has a context which must be created prior to all other MOSEK calls.
- static void [decrementInstanceCounter](#) ()
MOSEK has a context which should be deleted after MOSEK calls.
- static unsigned int [getNumInstances](#) ()
Return the number of instances of instantiated objects using MOSEK services.

Public Attributes

Private member data

- MSKtask_t [task_](#)
MOSEK model represented by this class instance.
- int * [hotStartCStat_](#)
Hotstart information.
- int [hotStartCStatSize_](#)
- int * [hotStartRStat_](#)
- int [hotStartRStatSize_](#)
- int [hotStartMaxIteration_](#)

Cached information derived from the MOSEK model

- double * [obj_](#)
Pointer to objective vector.
- double * [collower_](#)
Pointer to dense vector of variable lower bounds.
- double * [colupper_](#)
Pointer to dense vector of variable lower bounds.

- 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)
- double * [rowlower_](#)
Pointer to dense vector of row lower bounds.
- double * [rowupper_](#)
Pointer to dense vector of row upper bounds.
- double * [colsol_](#)
Pointer to primal solution vector.
- double * [rowsol_](#)
Pointer to dual solution vector.
- double * [redcost_](#)
Pointer to reduced cost vector.
- double * [rowact_](#)
Pointer to row activity (slack) vector.
- **CoinPackedMatrix** * [matrixByRow_](#)
Pointer to row-wise copy of problem matrix coefficients.
- **CoinPackedMatrix** * [matrixByCol_](#)
Pointer to row-wise copy of problem matrix coefficients.

Additional information needed for storing MIP problems

- char * [coltype_](#)
Pointer to dense vector of variable types (continuous, binary, integer)
- int [coltypesize_](#)
Size of allocated memory for coltype_.
- bool [probtymip_](#)
Stores whether MOSEK' prob type is currently set to MIP.

Protected Member Functions

Protected methods

- virtual void [applyRowCut](#) (const [OsiRowCut](#) &rc)
Apply a row cut. Return true if cut was applied.
- virtual void [applyColCut](#) (const [OsiColCut](#) &cc)
Apply a column cut (bound adjustment).

Friends

- void [OsiMskSolverInterfaceUnitTest](#) (const std::string &mpsDir, const std::string &netlibDir)
A function that tests the methods in the [OsiMskSolverInterface](#) class.

MOSEK specific public interfaces

- enum `keepCachedFlag` {
`KEEPCACHED_NONE` = 0, `KEEPCACHED_COLUMN` = 1, `KEEPCACHED_ROW` = 2, `KEEPCACHED_MATRIX` = 4,
`KEEPCACHED_RESULTS` = 8, `KEEPCACHED_PROBLEM` = `KEEPCACHED_COLUMN` | `KEEPCACHED_ROW` | `KEEPCACHED_MATRIX`, `KEEPCACHED_ALL` = `KEEPCACHED_PROBLEM` | `KEEPCACHED_RESULTS`,
`FREECACHED_COLUMN` = `KEEPCACHED_PROBLEM` & ~`KEEPCACHED_COLUMN`,
`FREECACHED_ROW` = `KEEPCACHED_PROBLEM` & ~`KEEPCACHED_ROW`, `FREECACHED_MATRIX` = `KEEPCACHED_PROBLEM` & ~`KEEPCACHED_MATRIX`, `FREECACHED_RESULTS` = `KEEPCACHED_ALL` & ~`KEEPCACHED_RESULTS` }

Get pointer to MOSEK model and free all specified cached data entries (combined with logical or-operator '|'):

- `MSKtask_t getLpPtr` (int keepCached=`KEEPCACHED_NONE`)
- `MSKenv_t getEnvironmentPtr` ()

Method to access MOSEK environment pointer.

Private methods

- `MSKtask_t getMutableLpPtr` () const
Get task Pointer for const methods.
- void `gutsOfCopy` (const `OsiMskSolverInterface` &source)
The real work of a copy constructor (used by copy and assignment)
- void `gutsOfConstructor` ()
The real work of the constructor.
- void `gutsOfDestructor` ()
The real work of the destructor.
- void `freeCachedColRim` ()
free cached column rim vectors
- void `freeCachedRowRim` ()
free cached row rim vectors
- void `freeCachedResults` ()
free cached result vectors
- void `freeCachedMatrix` ()
free cached matrices
- void `freeCachedData` (int keepCached=`KEEPCACHED_NONE`)
free all cached data (except specified entries, see getLpPtr())
- void `freeAllMemory` ()
free all allocated memory

Additional Inherited Members

7.21.1 Detailed Description

Definition at line 23 of file `OsiMskSolverInterface.hpp`.

7.21.2 Member Enumeration Documentation

7.21.2.1 enum OsiMskSolverInterface::keepCachedFlag

Get pointer to MOSEK model and free all specified cached data entries (combined with logical or-operator '|'):

Enumerator

KEEPCACHED_NONE discard all cached data (default)
KEEPCACHED_COLUMN column information: objective values, lower and upper bounds, variable types
KEEPCACHED_ROW row information: right hand sides, ranges and senses, lower and upper bounds for row
KEEPCACHED_MATRIX problem matrix: matrix ordered by column and by row
KEEPCACHED_RESULTS LP solution: primal and dual solution, reduced costs, row activities.
KEEPCACHED_PROBLEM only discard cached LP solution
KEEPCACHED_ALL keep all cached data (similar to [getMutableLpPtr\(\)](#))
FREECACHED_COLUMN free only cached column and LP solution information
FREECACHED_ROW free only cached row and LP solution information
FREECACHED_MATRIX free only cached matrix and LP solution information
FREECACHED_RESULTS free only cached LP solution information

Definition at line 598 of file OsiMskSolverInterface.hpp.

7.21.3 Member Function Documentation

7.21.3.1 CoinWarmStart* OsiMskSolverInterface::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.

Implements [OsiSolverInterface](#).

7.21.3.2 virtual bool OsiMskSolverInterface::setWarmStart (const CoinWarmStart * warmstart) [virtual]

Set warmstarting information.

Return true/false depending on whether the warmstart information was accepted or not.

Implements [OsiSolverInterface](#).

7.21.3.3 virtual const char* OsiMskSolverInterface::getRowSense () const [virtual]

Get pointer to array[[getNumRows\(\)](#)] of row constraint senses.

- 'L': <= constraint
- 'E': = constraint
- 'G': >= constraint
- 'R': ranged constraint

- 'N': free constraint

Implements [OsiSolverInterface](#).

7.21.3.4 `virtual const double* OsiMskSolverInterface::getRightHandSide () const` [virtual]

Get pointer to array[[getNumRows\(\)](#)] of rows right-hand sides.

- if `rowsense()[i] == 'L'` then `rhs()[i] == rowupper()[i]`
- if `rowsense()[i] == 'G'` then `rhs()[i] == rowlower()[i]`
- if `rowsense()[i] == 'R'` then `rhs()[i] == rowupper()[i]`
- if `rowsense()[i] == 'N'` then `rhs()[i] == 0.0`

Implements [OsiSolverInterface](#).

7.21.3.5 `virtual const double* OsiMskSolverInterface::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 0.0

Implements [OsiSolverInterface](#).

7.21.3.6 `virtual int OsiMskSolverInterface::getIterationCount () const` [virtual]

Get how many iterations it took to solve the problem (whatever "iteration" mean to the solver).

Implements [OsiSolverInterface](#).

7.21.3.7 `virtual std::vector<double*> OsiMskSolverInterface::getDualRays (int maxNumRays, bool fullRay = false) const` [virtual]

Get as many dual rays as the solver can provide.

(In case of proven primal infeasibility there should be at least one.)

The first [getNumRows\(\)](#) ray components will always be associated with the row duals (as returned by [getRowPrice\(\)](#)). If `fullRay` is true, the final [getNumCols\(\)](#) entries will correspond to the ray components associated with the nonbasic variables. If the full ray is requested and the method cannot provide it, it will throw an exception.

NOTE for implementers of solver interfaces:

The double pointers in the vector should point to arrays of length [getNumRows\(\)](#) and they should be allocated via `new[]`.

NOTE for users of solver interfaces:

It is the user's responsibility to free the double pointers in the vector using `delete[]`.

Implements [OsiSolverInterface](#).

7.21.3.8 `virtual std::vector<double*> OsiMskSolverInterface::getPrimalRays (int maxNumRays) const` [virtual]

Get as many primal rays as the solver can provide.

(In case of proven dual infeasibility there should be at least one.)

NOTE for implementers of solver interfaces:

The double pointers in the vector should point to arrays of length `getNumCols()` and they should be allocated via `new[]`.

NOTE for users of solver interfaces:

It is the user's responsibility to free the double pointers in the vector using `delete[]`.

Implements [OsiSolverInterface](#).

7.21.3.9 `virtual void OsiMskSolverInterface::setColLower (int elementIndex, double elementValue)` [virtual]

Set a single column lower bound

Use `-COIN_DBL_MAX` for -infinity.

Implements [OsiSolverInterface](#).

7.21.3.10 `virtual void OsiMskSolverInterface::setColUpper (int elementIndex, double elementValue)` [virtual]

Set a single column upper bound

Use `COIN_DBL_MAX` for infinity.

Implements [OsiSolverInterface](#).

7.21.3.11 `virtual void OsiMskSolverInterface::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

<code><code>[<i>indexFirst</i>,<i>indexLast</i></code></code>	contains the indices of the constraints whose either bound changes
<code><i>boundList</i></code>	the new lower/upper bound pairs for the variables

Reimplemented from [OsiSolverInterface](#).

7.21.3.12 `virtual void OsiMskSolverInterface::setRowLower (int elementIndex, double elementValue)` [virtual]

Set a single row lower bound

Use `-COIN_DBL_MAX` for -infinity.

Implements [OsiSolverInterface](#).

7.21.3.13 `virtual void OsiMskSolverInterface::setRowUpper (int elementIndex, double elementValue)` [virtual]

Set a single row upper bound

Use `COIN_DBL_MAX` for infinity.

Implements [OsiSolverInterface](#).

7.21.3.14 `virtual void OsiMskSolverInterface::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

<code><code>[<i>indexfirst</i>,<i>index</i>↵ <i>Last</i>]</code>	contains the indices of the constraints whose either bound changes
<code><i>boundList</i></code>	the new lower/upper bound pairs for the constraints

Reimplemented from [OsiSolverInterface](#).

7.21.3.15 `virtual void OsiMskSolverInterface::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\(\)](#) and over and over again.

Parameters

<code><code>[<i>indexfirst</i>,<i>index</i>↵ <i>Last</i>]</code>	contains the indices of the constraints whose type changes
<code><i>senseList</i></code>	the new senses
<code><i>rhsList</i></code>	the new right hand sides
<code><i>rangeList</i></code>	the new ranges

Reimplemented from [OsiSolverInterface](#).

7.21.3.16 `virtual void OsiMskSolverInterface::setColSolution (const double * colsol)` [virtual]

Set the primal solution column values.

`colsol[numcols()]` is an array of values of the problem column variables. These values are copied to memory owned by the solver object or the solver. They will be returned as the result of `colsol()` until changed by another call to `setColsol()` or by a call to any solver routine. Whether the solver makes use of the solution in any way is solver-dependent.

Implements [OsiSolverInterface](#).

7.21.3.17 `virtual void OsiMskSolverInterface::setRowPrice (const double * rowprice)` [virtual]

Set dual solution vector.

`rowprice[numrows()]` is an array of values of the problem row dual variables. These values are copied to memory owned by the solver object or the solver. They will be returned as the result of `rowprice()` until changed by another call to `setRowprice()` or by a call to any solver routine. Whether the solver makes use of the solution in any way is solver-dependent.

Implements [OsiSolverInterface](#).

7.21.3.18 `virtual void OsiMskSolverInterface::addCol (const CoinPackedVectorBase & vec, const double collb, const double colub, const double obj) [virtual]`

Add a column (primal variable) to the problem.

Implements [OsiSolverInterface](#).

7.21.3.19 `virtual void OsiMskSolverInterface::addCols (const int numcols, const CoinPackedVectorBase *const * cols, const double * collb, const double * colub, const double * obj) [virtual]`

Add a set of columns (primal variables) to the problem.

The default implementation simply makes repeated calls to [addCol\(\)](#).

Reimplemented from [OsiSolverInterface](#).

7.21.3.20 `virtual void OsiMskSolverInterface::deleteCols (const int num, const int * colIndices) [virtual]`

Remove a set of columns (primal variables) from the problem.

The solver interface for a basis-oriented solver will maintain valid warm start information if all deleted variables are nonbasic.

Implements [OsiSolverInterface](#).

7.21.3.21 `virtual void OsiMskSolverInterface::addRow (const CoinPackedVectorBase & vec, const double rowlb, const double rowub) [virtual]`

Add a row (constraint) to the problem.

Implements [OsiSolverInterface](#).

7.21.3.22 `virtual void OsiMskSolverInterface::addRow (const CoinPackedVectorBase & vec, const char rowSen, const double rowrhs, const double rowrng) [virtual]`

Add a row (constraint) to the problem.

Implements [OsiSolverInterface](#).

7.21.3.23 `virtual void OsiMskSolverInterface::addRows (const int numrows, const CoinPackedVectorBase *const * rows, const double * rowlb, const double * rowub) [virtual]`

Add a set of rows (constraints) to the problem.

The default implementation simply makes repeated calls to [addRow\(\)](#).

Reimplemented from [OsiSolverInterface](#).

7.21.3.24 `virtual void OsiMskSolverInterface::addRows (const int numrows, const CoinPackedVectorBase *const * rows, const char * rowSen, const double * rowrhs, const double * rowrng) [virtual]`

Add a set of rows (constraints) to the problem.

The default implementation simply makes repeated calls to [addRow\(\)](#).

Reimplemented from [OsiSolverInterface](#).

7.21.3.25 `virtual void OsiMskSolverInterface::deleteRows (const int num, const int * rowIndices) [virtual]`

Delete a set of rows (constraints) from the problem.

The solver interface for a basis-oriented solver will maintain valid warm start information if all deleted rows are loose.

Implements [OsiSolverInterface](#).

7.21.3.26 `virtual void OsiMskSolverInterface::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 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

Implements [OsiSolverInterface](#).

7.21.3.27 `virtual void OsiMskSolverInterface::assignProblem (CoinPackedMatrix *& matrix, double *& collb, double *& colub, double *& obj, double *& rowlb, double *& rowub) [virtual]`

Load in an problem by assuming ownership of the arguments (the constraints on the rows are given by lower and upper bounds).

For default values see the previous method.

WARNING: The arguments passed to this method will be freed using the C++ `delete` and `delete[]` functions.

Implements [OsiSolverInterface](#).

7.21.3.28 `virtual void OsiMskSolverInterface::loadProblem (const CoinPackedMatrix & matrix, const double * collb, const double * colub, const double * obj, const char * rowsen, const double * rowrhs, const double * rowrng) [virtual]`

Load in an problem by copying the arguments (the constraints on the rows are given by sense/rhs/range triplets).

If a pointer is 0 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

Implements [OsiSolverInterface](#).

7.21.3.29 `virtual void OsiMskSolverInterface::assignProblem (CoinPackedMatrix *& matrix, double *& collb, double *& colub, double *& obj, char *& rowsen, double *& rowrhs, double *& rowrng) [virtual]`

Load in an problem by assuming ownership of the arguments (the constraints on the rows are given by sense/rhs/range triplets).

For default values see the previous method.

WARNING: The arguments passed to this method will be freed using the C++ `delete` and `delete[]` functions.

Implements [OsiSolverInterface](#).

7.21.3.30 `virtual void OsiMskSolverInterface::loadProblem (const int numcols, const int numRows, const int * 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).

7.21.3.31 `virtual void OsiMskSolverInterface::loadProblem (const int numcols, const int numRows, const int * 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).

7.21.3.32 `virtual void OsiMskSolverInterface::writeMps (const char * filename, const char * extension = "mps", double objSense = 0.0) const [virtual]`

Write the problem into an mps file of the given filename.

If `objSense` is non zero then -1.0 forces the code to write a maximization objective and +1.0 to write a minimization one.

If 0.0 then solver can do what it wants

Implements [OsiSolverInterface](#).

7.21.3.33 `static void OsiMskSolverInterface::incrementInstanceCounter () [static]`

MOSEK has a context which must be created prior to all other MOSEK calls.

This method:

- Increments by 1 the number of uses of the MOSEK environment.
- Creates the MOSEK context when the number of uses is change to 1 from 0.

7.21.3.34 `static void OsiMskSolverInterface::decrementInstanceCounter () [static]`

MOSEK has a context which should be deleted after MOSEK calls.

This method:

- Decrements by 1 the number of uses of the MOSEK environment.
- Deletes the MOSEK context when the number of uses is change to 0 from 1.

7.21.3.35 `virtual void OsiMskSolverInterface::applyColCut (const OsiColCut & cc) [protected],[virtual]`

Apply a column cut (bound adjustment).

Return true if cut was applied.

Implements [OsiSolverInterface](#).

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

- `OsiMskSolverInterface.hpp`

7.22 OsiObject Class Reference

Abstract base class for 'objects'.

```
#include <OsiBranchingObject.hpp>
```

Inheritance diagram for OsiObject:

Public Member Functions

- [OsiObject](#) ()
Default Constructor.
- [OsiObject](#) (const [OsiObject](#) &)
Copy constructor.
- [OsiObject](#) & `operator=` (const [OsiObject](#) &rhs)
Assignment operator.
- virtual [OsiObject](#) * `clone` () const =0
Clone.
- virtual `~OsiObject` ()
Destructor.
- double `infeasibility` (const [OsiSolverInterface](#) *solver, int &`whichWay`) const
Infeasibility of the object.
- virtual double `feasibleRegion` ([OsiSolverInterface](#) *solver) const
For the variable(s) referenced by the object, look at the current solution and set bounds to match the solution.
- virtual double `feasibleRegion` ([OsiSolverInterface](#) *solver, const [OsiBranchingInformation](#) *info) const =0
For the variable(s) referenced by the object, look at the current solution and set bounds to match the solution.
- virtual [OsiBranchingObject](#) * `createBranch` ([OsiSolverInterface](#) *, const [OsiBranchingInformation](#) *, int) const
Create a branching object and indicate which way to branch first.
- virtual bool `canDoHeuristics` () const

- Return true if object can take part in normal heuristics.*

 - virtual bool `canMoveToNearest` () const
- Return true if object can take part in move to nearest heuristic.*

 - virtual int `columnNumber` () const
- Column number if single column object -1 otherwise, Used by heuristics.*

 - int `priority` () const
- Return Priority - note 1 is highest priority.*

 - void `setPriority` (int `priority`)
- Set priority.*

 - virtual bool `boundBranch` () const
- Return true if branch should only bound variables.*

 - virtual bool `canHandleShadowPrices` () const
- Return true if knows how to deal with Pseudo Shadow Prices.*

 - int `numberWays` () const
- Return maximum number of ways branch may have.*

 - void `setNumberWays` (int `numberWays`)
- Set maximum number of ways branch may have.*

 - void `setWhichWay` (int `way`)
- Return preferred way to branch.*

 - int `whichWay` () const
- Return current preferred way to branch.*

 - virtual int `preferredWay` () const
- Get pre-emptive preferred way of branching - -1 off, 0 down, 1 up (for 2-way)*

 - double `infeasibility` () const
- Return infeasibility.*

 - virtual double `upEstimate` () const
- Return "up" estimate (default 1.0e-5)*

 - virtual double `downEstimate` () const
- Return "down" estimate (default 1.0e-5)*

 - virtual void `resetBounds` (const `OsiSolverInterface` *)
- Reset variable bounds to their original values.*

 - virtual void `resetSequenceEtc` (int, const int *)
- Change column numbers after preprocessing.*

 - virtual void `updateBefore` (const `OsiObject` *)
- Updates stuff like pseudocosts before threads.*

 - virtual void `updateAfter` (const `OsiObject` *, const `OsiObject` *)
- Updates stuff like pseudocosts after threads finished.*

Protected Attributes

- double `infeasibility_`
data
- short `whichWay_`
Computed preferred way to branch.
- short `numberWays_`
Maximum number of ways on branch.
- int `priority_`
Priority.

7.22.1 Detailed Description

Abstract base class for 'objects'.

The branching model used in Osi is based on the idea of an *object*. In the abstract, an object is something that has a feasible region, can be evaluated for infeasibility, can be branched on (*i.e.*, there's some constructive action to be taken to move toward feasibility), and allows comparison of the effect of branching.

This class ([OsiObject](#)) is the base class for an object. To round out the branching model, the class [OsiBranchingObject](#) describes how to perform a branch, and the class [OsiBranchDecision](#) describes how to compare two [OsiBranchingObject](#)s.

To create a new type of object you need to provide three methods: [infeasibility\(\)](#), [feasibleRegion\(\)](#), and [createBranch\(\)](#), described below.

This base class is primarily virtual to allow for any form of structure. Any form of discontinuity is allowed.

As there is an overhead in getting information from solvers and because other useful information is available there is also an [OsiBranchingInformation](#) class which can contain pointers to information. If used it must at minimum contain pointers to current value of objective, maximum allowed objective and pointers to arrays for bounds and solution and direction of optimization. Also integer and primal tolerance.

Classes which inherit might have other information such as depth, number of solutions, pseudo-shadow prices etc etc. May be easier just to throw in here - as I keep doing

Definition at line 56 of file [OsiBranchingObject.hpp](#).

7.22.2 Member Function Documentation

7.22.2.1 `double OsiObject::infeasibility (const OsiSolverInterface * solver, int & whichWay) const`

Infeasibility of the object.

This is some measure of the infeasibility of the object. 0.0 indicates that the object is satisfied.

The preferred branching direction is returned in `whichWay`, where for normal two-way branching 0 is down, 1 is up

This is used to prepare for strong branching but should also think of case when no strong branching

The object may also compute an estimate of cost of going "up" or "down". This will probably be based on pseudo-cost ideas

This should also set mutable `infeasibility_` and `whichWay_` This is for instant re-use for speed

Default for this just calls `infeasibility` with [OsiBranchingInformation](#) NOTE - Convention says that an infeasibility of `COIN_DBL_MAX` means object has worked out it can't be satisfied!

7.22.2.2 `virtual double OsiObject::feasibleRegion (OsiSolverInterface * solver) const [virtual]`

For the variable(s) referenced by the object, look at the current solution and set bounds to match the solution.

Returns measure of how much it had to move solution to make feasible

7.22.2.3 `virtual double OsiObject::feasibleRegion (OsiSolverInterface * solver, const OsiBranchingInformation * info) const [pure virtual]`

For the variable(s) referenced by the object, look at the current solution and set bounds to match the solution.

Returns measure of how much it had to move solution to make feasible Faster version

Implemented in [OsiLotsize](#), [OsiSOS](#), and [OsiSimpleInteger](#).

7.22.2.4 `virtual OsiBranchingObject* OsiObject::createBranch (OsiSolverInterface *, const OsiBranchingInformation *, int) const [inline],[virtual]`

Create a branching object and indicate which way to branch first.

The branching object has to know how to create branches (fix variables, etc.)

Reimplemented in [OsiLotsize](#), [OsiSOS](#), and [OsiSimpleInteger](#).

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

7.22.2.5 `void OsiObject::setWhichWay (int way) [inline]`

Return preferred way to branch.

If two then way=0 means down and 1 means up, otherwise way points to preferred branch

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

7.22.2.6 `int OsiObject::whichWay () const [inline]`

Return current preferred way to branch.

If two then way=0 means down and 1 means up, otherwise way points to preferred branch

Definition at line 164 of file `OsiBranchingObject.hpp`.

7.22.2.7 `virtual void OsiObject::resetBounds (const OsiSolverInterface *) [inline],[virtual]`

Reset variable bounds to their original values.

Bounds may be tightened, so it may be good to be able to reset them to their original values.

Reimplemented in [OsiLotsize](#), and [OsiSimpleInteger](#).

Definition at line 180 of file `OsiBranchingObject.hpp`.

7.22.3 Member Data Documentation

7.22.3.1 `double OsiObject::infeasibility_ [mutable],[protected]`

data

Computed infeasibility

Definition at line 193 of file `OsiBranchingObject.hpp`.

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

- `OsiBranchingObject.hpp`

7.23 OsiObject2 Class Reference

Define a class to add a bit of complexity to [OsiObject](#) This assumes 2 way branching.

```
#include <OsiBranchingObject.hpp>
```

Inheritance diagram for OsiObject2:

Collaboration diagram for OsiObject2:

Public Member Functions

- [OsiObject2](#) ()
Default Constructor.
- [OsiObject2](#) (const [OsiObject2](#) &)
Copy constructor.
- [OsiObject2](#) & [operator=](#) (const [OsiObject2](#) &rhs)
Assignment operator.
- virtual [~OsiObject2](#) ()
Destructor.
- void [setPreferredWay](#) (int value)
Set preferred way of branching - -1 off, 0 down, 1 up (for 2-way)
- virtual int [preferredWay](#) () const
Get preferred way of branching - -1 off, 0 down, 1 up (for 2-way)

Protected Attributes

- int [preferredWay_](#)
Preferred way of branching - -1 off, 0 down, 1 up (for 2-way)
- double [otherInfeasibility_](#)
"Infeasibility" on other way

7.23.1 Detailed Description

Define a class to add a bit of complexity to [OsiObject](#) This assumes 2 way branching.

Definition at line 206 of file OsiBranchingObject.hpp.

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

- OsiBranchingObject.hpp

7.24 OsiPresolve Class Reference

OSI interface to COIN problem simplification capabilities.

```
#include <OsiPresolve.hpp>
```

Public Member Functions

- [OsiPresolve](#) ()
Default constructor (empty object)
- virtual [~OsiPresolve](#) ()

Virtual destructor.

- virtual [OsiSolverInterface](#) * [presolvedModel](#) ([OsiSolverInterface](#) &origModel, double feasibilityTolerance=0.0, bool keepIntegers=true, int numberPasses=5, const char *prohibited=NULL, bool doStatus=true, const char *row↵Prohibited=NULL)

Create a new [OsiSolverInterface](#) loaded with the presolved problem.

- virtual void [postsolve](#) (bool updateStatus=true)

Restate the solution to the presolved problem in terms of the original problem and load it into the original model.

- [OsiSolverInterface](#) * [model](#) () const

Return a pointer to the presolved model.

- [OsiSolverInterface](#) * [originalModel](#) () const

Return a pointer to the original model.

- void [setOriginalModel](#) ([OsiSolverInterface](#) *model)

Set the pointer to the original model.

- const int * [originalColumns](#) () const

Return a pointer to the original columns.

- const int * [originalRows](#) () const

Return a pointer to the original rows.

- int [getNumRows](#) () const

Return number of rows in original model.

- int [getNumCols](#) () const

Return number of columns in original model.

- void [setNonLinearValue](#) (double value)

"Magic" number.

- void [setPresolveActions](#) (int action)

Fine control over presolve actions.

Protected Member Functions

- virtual const **CoinPresolveAction** * [presolve](#) (**CoinPresolveMatrix** *prob)

Apply presolve transformations to the problem.

- virtual void [postsolve](#) (**CoinPostsolveMatrix** &prob)

Reverse presolve transformations to recover the solution to the original problem.

- void [gutsOfDestroy](#) ()

Destroys queued postsolve actions.

7.24.1 Detailed Description

OSI interface to COIN problem simplification capabilities.

COIN provides a number of classes which implement problem simplification algorithms (**CoinPresolveAction**, **Coin↵PrePostsolveMatrix**, and derived classes). The model of operation is as follows:

- Create a copy of the original problem.
- Subject the copy to a series of transformations (the *presolve* methods) to produce a presolved model. Each transformation is also expected to provide a method to reverse the transformation (the *postsolve* method). The postsolve methods are collected in a linked list; the postsolve method for the final presolve transformation is at the head of the list.

- Hand the presolved problem to the solver for optimization.
- Apply the collected postsolve methods to the presolved problem and solution, restating the solution in terms of the original problem.

The COIN presolve algorithms are unaware of OSI. The [OsiPresolve](#) class takes care of the interface. Given an [OsiSolverInterface](#) `origModel`, it will take care of creating a clone properly loaded with the presolved problem and ready for optimization. After optimization, it will apply postsolve transformations and load the result back into `origModel`.

Assuming a problem has been loaded into an [OsiSolverInterface](#) `origModel`, a bare-bones application looks like this:

```
OsiPresolve pinfo ;
OsiSolverInterface *presolvedModel ;
// Return an OsiSolverInterface loaded with the presolved problem.
presolvedModel = pinfo.presolvedModel(*origModel, 1.0e-8, false, numberPasses) ;
presolvedModel->initialSolve() ;
// Restate the solution and load it back into origModel.
pinfo.postsolve(true) ;
delete presolvedModel ;
```

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

7.24.2 Member Function Documentation

7.24.2.1 `virtual OsiSolverInterface* OsiPresolve::presolvedModel (OsiSolverInterface & origModel, double feasibilityTolerance = 0.0, bool keepIntegers = true, int numberPasses = 5, const char * prohibited = NULL, bool doStatus = true, const char * rowProhibited = NULL) [virtual]`

Create a new [OsiSolverInterface](#) loaded with the presolved problem.

This method implements the first two steps described in the class documentation. It clones `origModel` and applies presolve transformations, storing the resulting list of postsolve transformations. It returns a pointer to a new [OsiSolverInterface](#) loaded with the presolved problem, or NULL if the problem is infeasible or unbounded. If `keepIntegers` is true then bounds may be tightened in the original. Bounds will be moved by up to `feasibilityTolerance` to try and stay feasible. When `doStatus` is true, the current solution will be transformed to match the presolved model.

This should be paired with [postsolve\(\)](#). It is up to the client to destroy the returned [OsiSolverInterface](#), *after* calling [postsolve\(\)](#).

This method is virtual. Override this method if you need to customize the steps of creating a model to apply presolve transformations.

In some sense, a wrapper for [presolve\(CoinPresolveMatrix*\)](#).

7.24.2.2 `virtual void OsiPresolve::postsolve (bool updateStatus = true) [virtual]`

Restate the solution to the presolved problem in terms of the original problem and load it into the original model.

[postsolve\(\)](#) restates the solution in terms of the original problem and updates the original [OsiSolverInterface](#) supplied to [presolvedModel\(\)](#). 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 a basic solution, then set `updateStatus`

The advantage of going back to the original problem is that it will be exactly as it was, *i.e.*, 0.0 will not become 1.0e-19.

Note that if you modified the original problem after presolving, then you must “undo” these modifications before calling [postsolve\(\)](#).

In some sense, a wrapper for [postsolve\(CoinPostsolveMatrix&\)](#).

7.24.2.3 OsiSolverInterface* OsiPresolve::model () const

Return a pointer to the presolved model.

7.24.2.4 void OsiPresolve::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 144 of file OsiPresolve.hpp.

7.24.2.5 void OsiPresolve::setPresolveActions (int *action*) [inline]

Fine control over presolve actions.

Set/clear the following bits to allow or suppress actions:

- 0x01 allow duplicate column processing on integer columns and dual stuff on integers
- 0x02 switch off actions which can change +1 to something else (doubleton, tripleton, implied free)
- 0x04 allow transfer of costs from singletons and between integer variables (when advantageous)
- 0x08 do not allow $x+y+z=1$ transform
- 0x10 allow actions that don't easily unroll
- 0x20 allow dubious gub element reduction

GUB element reduction is only partially implemented in CoinPresolve (see **gubrow_action**) and will cause an abort at postsolve. It's not clear what's meant by 'dual stuff on integers'. – lh, 110605 –

Definition at line 166 of file OsiPresolve.hpp.

7.24.2.6 virtual const CoinPresolveAction* OsiPresolve::presolve (CoinPresolveMatrix * *prob*) [protected], [virtual]

Apply presolve transformations to the problem.

Handles the core activity of applying presolve transformations.

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

7.24.2.7 virtual void OsiPresolve::postsolve (CoinPostsolveMatrix & *prob*) [protected], [virtual]

Reverse presolve transformations to recover the solution to the original problem.

Handles the core activity of applying postsolve transformations.

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.

7.24.2.8 void OsiPresolve::gutsOfDestroy () [protected]

Destroys queued postsolve actions.

E.g., when [presolve\(\)](#) determines the problem is infeasible, so that it will not be necessary to actually solve the presolved problem and convert the result back to the original problem.

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

- OsiPresolve.hpp

7.25 OsiPseudoCosts Class Reference

This class is the placeholder for the pseudocosts used by [OsiChooseStrong](#).

```
#include <OsiChooseVariable.hpp>
```

Public Member Functions

- int [numberBeforeTrusted](#) () const
Number of times before trusted.
- void [setNumberBeforeTrusted](#) (int value)
Set number of times before trusted.
- void [initialize](#) (int n)
Initialize the pseudocosts with n entries.
- int [numberObjects](#) () const
Give the number of objects for which pseudo costs are stored.
- virtual void [updateInformation](#) (const [OsiBranchingInformation](#) *info, int branch, [OsiHotInfo](#) *hotInfo)
Given a candidate fill in useful information e.g. estimates.
- virtual void [updateInformation](#) (int whichObject, int branch, double changeInObjective, double changeInValue, int status)
Given a branch fill in useful information e.g. estimates.

Accessor methods to pseudo costs data

- double * [upTotalChange](#) ()
- const double * [upTotalChange](#) () const
- double * [downTotalChange](#) ()
- const double * [downTotalChange](#) () const
- int * [upNumber](#) ()
- const int * [upNumber](#) () const
- int * [downNumber](#) ()
- const int * [downNumber](#) () const

Protected Attributes

- double * [upTotalChange_](#)
Total of all changes up.
- double * [downTotalChange_](#)
Total of all changes down.

- int * [upNumber_](#)
Number of times up.
- int * [downNumber_](#)
Number of times down.
- int [numberObjects_](#)
Number of objects (could be found from solver)
- int [numberBeforeTrusted_](#)
Number before we trust.

7.25.1 Detailed Description

This class is the placeholder for the pseudocosts used by [OsiChooseStrong](#).

It can also be used by any other pseudocost based strong branching algorithm.

Definition at line 240 of file OsiChooseVariable.hpp.

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

- OsiChooseVariable.hpp

7.26 OsiRowCut Class Reference

Row Cut Class.

```
#include <OsiRowCut.hpp>
```

Inheritance diagram for OsiRowCut:

Collaboration diagram for OsiRowCut:

Public Member Functions

- void [sortIncrIndex](#) ()
Allow access row sorting function.

Row bounds

- OsiRowCut_ inline double [lb](#) () const
Get lower bound.
- OsiRowCut_ inline void [setLb](#) (double [lb](#))
Set lower bound.
- OsiRowCut_ inline double [ub](#) () const
Get upper bound.
- OsiRowCut_ inline void [setUb](#) (double [ub](#))
Set upper bound.

Row rhs, sense, range

- char [sense](#) () const
Get sense ('E', 'G', 'L', 'N', 'R')
- double [rhs](#) () const

Get right-hand side.

- double [range](#) () const
Get range (ub - lb for 'R' rows, 0 otherwise)

Row elements

- `OsiRowCut`_inline void [setRow](#) (int size, const int *colIndices, const double *elements, bool testForDuplicate↔ Index=COIN_DEFAULT_VALUE_FOR_DUPLICATE)
Set row elements.
- `OsiRowCut`_inline void [setRow](#) (const **CoinPackedVector** &v)
Set row elements from a packed vector.
- `OsiRowCut`_inline const **CoinPackedVector** & [row](#) () const
Get row elements.
- `OsiRowCut`_inline **CoinPackedVector** & [mutableRow](#) ()
Get row elements for changing.

Comparison operators

- `OsiRowCut`_inline bool [operator==](#) (const [OsiRowCut](#) &rhs) const
equal - true if lower bound, upper bound, row elements, and [OsiCut](#) are equal.
- `OsiRowCut`_inline bool [operator!=](#) (const [OsiRowCut](#) &rhs) const
not equal

Sanity checks on cut

- `OsiRowCut`_inline bool [consistent](#) () const
Returns true if the cut is consistent.
- `OsiRowCut`_inline bool [consistent](#) (const [OsiSolverInterface](#) &im) const
Returns true if cut is consistent with respect to the solver interface's model.
- `OsiRowCut`_inline bool [infeasible](#) (const [OsiSolverInterface](#) &im) const
Returns true if the row cut itself is infeasible and cannot be satisfied.
- virtual double [violated](#) (const double *solution) const
Returns infeasibility of the cut with respect to solution passed in i.e.

Arithmetic operators. Apply **CoinPackedVector** methods to the vector

- void [operator+=](#) (double value)
add value to every vector entry
- void [operator-=](#) (double value)
subtract value from every vector entry
- void [operator*=](#) (double value)
multiply every vector entry by value
- void [operator/=](#) (double value)
divide every vector entry by value

Constructors and destructors

- [OsiRowCut](#) & [operator=](#) (const [OsiRowCut](#) &rhs)
Assignment operator.
- [OsiRowCut](#) (const [OsiRowCut](#) &)
Copy constructor.
- virtual [OsiRowCut](#) * [clone](#) () const
Clone.

- [OsiRowCut](#) ()
Default Constructor.
- [OsiRowCut](#) (double *cutlb*, double *cutub*, int *capacity*, int *size*, int **colIndices*, double **elements*)
Ownership Constructor.
- virtual [~OsiRowCut](#) ()
Destructor.

Debug stuff

- virtual void [print](#) () const
Print cuts in collection.

Friends

- void [OsiRowCutUnitTest](#) (const [OsiSolverInterface](#) **baseSiP*, const std::string &*mpsDir*)
A function that tests the methods in the [OsiRowCut](#) class.

Additional Inherited Members

7.26.1 Detailed Description

Row Cut Class.

A row cut has:

- a lower bound
- an upper bound
- a vector of row elements

Definition at line 29 of file `OsiRowCut.hpp`.

7.26.2 Constructor & Destructor Documentation

7.26.2.1 `OsiRowCut::OsiRowCut (double cutlb, double cutub, int capacity, int size, int *colIndices, double *elements)`

Ownership Constructor.

This constructor assumes ownership of the vectors passed as parameters for indices and elements. `colIndices` and `elements` will be NULL on return.

7.26.3 Member Function Documentation

7.26.3.1 `OsiRowCut_inline bool OsiRowCut::consistent () const` `[virtual]`

Returns true if the cut is consistent.

This checks to ensure that:

- The row element vector does not have duplicate indices
- The row element vector indices are ≥ 0

Implements [OsiCut](#).

7.26.3.2 `OsiRowCut_inline bool OsiRowCut::consistent (const OsiSolverInterface & im) const` `[virtual]`

Returns true if cut is consistent with respect to the solver interface's model.

This checks to ensure that

- The row element vector indices are $<$ the number of columns in the model

Implements [OsiCut](#).

7.26.3.3 `OsiRowCut_inline bool OsiRowCut::infeasible (const OsiSolverInterface & im) const` `[virtual]`

Returns true if the row cut itself is infeasible and cannot be satisfied.

This checks whether

- the lower bound is strictly greater than the upper bound.

Implements [OsiCut](#).

7.26.3.4 `virtual double OsiRowCut::violated (const double * solution) const` `[virtual]`

Returns infeasibility of the cut with respect to solution passed in i.e.

is positive if cuts off that solution. solution is `getNumCols()` long..

Implements [OsiCut](#).

7.26.4 Friends And Related Function Documentation

7.26.4.1 `void OsiRowCutUnitTest (const OsiSolverInterface * baseSiP, const std::string & mpsDir)` `[friend]`

A function that tests the methods in the [OsiRowCut](#) class.

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

- `OsiRowCut.hpp`

7.27 OsiRowCut2 Class Reference

Row Cut Class which refers back to row which created it.

```
#include <OsiRowCut.hpp>
```

Inheritance diagram for `OsiRowCut2`:

Collaboration diagram for `OsiRowCut2`:

Public Member Functions

Which row

- int [whichRow](#) () const
Get row.
- void [setWhichRow](#) (int [row](#))
Set row.

Constructors and destructors

- [OsiRowCut2](#) & [operator=](#) (const [OsiRowCut2](#) &[rhs](#))
Assignment operator.
- [OsiRowCut2](#) (const [OsiRowCut2](#) &)
Copy constructor.
- virtual [OsiRowCut](#) * [clone](#) () const
Clone.
- [OsiRowCut2](#) (int [row](#)==1)
Default Constructor.
- virtual [~OsiRowCut2](#) ()
Destructor.

Additional Inherited Members

7.27.1 Detailed Description

Row Cut Class which refers back to row which created it.

It may be useful to strengthen a row rather than add a cut. To do this we need to know which row is strengthened. This trivial extension to [OsiRowCut](#) does that.

Definition at line 290 of file [OsiRowCut.hpp](#).

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

- [OsiRowCut.hpp](#)

7.28 OsiRowCutDebugger Class Reference

Validate cuts against a known solution.

```
#include <OsiRowCutDebugger.hpp>
```

Public Member Functions

Validate Row Cuts

Check that the specified cuts do not cut off the known solution.

- virtual int [validateCuts](#) (const [OsiCuts](#) &[cs](#), int first, int last) const
Check that the set of cuts does not cut off the solution known to the debugger.
- virtual bool [invalidCut](#) (const [OsiRowCut](#) &[rowcut](#)) const
Check that the cut does not cut off the solution known to the debugger.

- bool `onOptimalPath` (const `OsiSolverInterface` &si) const
Returns true if the solution held in the solver is compatible with the known solution.

Activate the Debugger

The debugger is considered to be active when it holds a known solution.

- bool `activate` (const `OsiSolverInterface` &si, const char *model)
Activate a debugger using the name of a problem.
- bool `activate` (const `OsiSolverInterface` &si, const double *solution, bool keepContinuous=false)
Activate a debugger using a full solution array.
- bool `active` () const
Returns true if the debugger is active.

Query or Manipulate the Known Solution

- const double * `optimalSolution` () const
Return the known solution.
- int `numberColumns` () const
Return the number of columns in the known solution.
- double `optimalValue` () const
Return the value of the objective for the known solution.
- void `redoSolution` (int `numberColumns`, const int *originalColumns)
Edit the known solution to reflect column changes.
- int `printOptimalSolution` (const `OsiSolverInterface` &si) const
Print optimal solution (returns -1 bad debug, 0 on optimal, 1 not)

Constructors and Destructors

- `OsiRowCutDebugger` ()
Default constructor - no checking.
- `OsiRowCutDebugger` (const `OsiSolverInterface` &si, const char *model)
Constructor with name of model.
- `OsiRowCutDebugger` (const `OsiSolverInterface` &si, const double *solution, bool enforceOptimality=false)
Constructor with full solution.
- `OsiRowCutDebugger` (const `OsiRowCutDebugger` &)
Copy constructor.
- `OsiRowCutDebugger` & `operator=` (const `OsiRowCutDebugger` &rhs)
Assignment operator.
- virtual `~OsiRowCutDebugger` ()
Destructor.

Friends

- void `OsiRowCutDebuggerUnitTest` (const `OsiSolverInterface` *siP, const std::string &mpsDir)
A function that tests the methods in the `OsiRowCutDebugger` class.

7.28.1 Detailed Description

Validate cuts against a known solution.

[OsiRowCutDebugger](#) provides a facility for validating cuts against a known solution for a problem. The debugger knows an optimal solution for many of the miplib3 problems. Check the source for [activate\(const OsiSolverInterface&,const char*\)](#) in `OsiRowCutDebugger.cpp` for the full set of known problems.

A full solution vector can be supplied as a parameter with ([activate\(const OsiSolverInterface&,const double*,bool\)](#)). Only the integer values need to be valid. The default behaviour is to solve an lp relaxation with the integer variables fixed to the specified values and use the optimal solution to fill in the continuous variables in the solution. The debugger can be instructed to preserve the continuous variables (useful when debugging solvers where the linear relaxation doesn't capture all the constraints).

Note that the solution must match the problem held in the solver interface. If you want to use the row cut debugger on a problem after applying presolve transformations, your solution must match the presolved problem. (But see [redo←Solution\(\)](#).)

Definition at line 42 of file `OsiRowCutDebugger.hpp`.

7.28.2 Constructor & Destructor Documentation

7.28.2.1 `OsiRowCutDebugger::OsiRowCutDebugger (const OsiSolverInterface & si, const char * model)`

Constructor with name of model.

See [activate\(const OsiSolverInterface&,const char*\)](#).

7.28.2.2 `OsiRowCutDebugger::OsiRowCutDebugger (const OsiSolverInterface & si, const double * solution, bool enforceOptimality = false)`

Constructor with full solution.

See [activate\(const OsiSolverInterface&,const double*,bool\)](#).

7.28.3 Member Function Documentation

7.28.3.1 `virtual int OsiRowCutDebugger::validateCuts (const OsiCuts & cs, int first, int last) const` [virtual]

Check that the set of cuts does not cut off the solution known to the debugger.

Check if any generated cuts cut off the solution known to the debugger! If so then print offending cuts. Return the number of invalid cuts.

7.28.3.2 `virtual bool OsiRowCutDebugger::invalidCut (const OsiRowCut & rowcut) const` [virtual]

Check that the cut does not cut off the solution known to the debugger.

Return true if cut is invalid

7.28.3.3 `bool OsiRowCutDebugger::onOptimalPath (const OsiSolverInterface & si) const`

Returns true if the solution held in the solver is compatible with the known solution.

More specifically, returns true if the known solution satisfies the column bounds held in the solver.

7.28.3.4 `bool OsiRowCutDebugger::activate (const OsiSolverInterface & si, const char * model)`

Activate a debugger using the name of a problem.

The debugger knows an optimal solution for most of miplib3. Check the source code for the full list. Returns true if the debugger is successfully activated.

7.28.3.5 `bool OsiRowCutDebugger::activate (const OsiSolverInterface & si, const double * solution, bool keepContinuous = false)`

Activate a debugger using a full solution array.

The solution must have one entry for every variable, but only the entries for integer values are used. By default the debugger will solve an lp relaxation with the integer variables fixed and fill in values for the continuous variables from this solution. If the debugger should preserve the given values for the continuous variables, set `keepContinuous` to true.

Returns true if debugger activates successfully.

7.28.3.6 `void OsiRowCutDebugger::redoSolution (int numberColumns, const int * originalColumns)`

Edit the known solution to reflect column changes.

Given a translation array `originalColumns[numberColumns]` which can translate current column indices to original column indices, this method will edit the solution held in the debugger so that it matches the current set of columns.

Useful when the original problem is preprocessed prior to cut generation. The debugger does keep a record of the changes.

7.28.4 Friends And Related Function Documentation

7.28.4.1 `void OsiRowCutDebuggerUnitTest (const OsiSolverInterface * siP, const std::string & mpsDir)` `[friend]`

A function that tests the methods in the [OsiRowCutDebugger](#) class.

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

- [OsiRowCutDebugger.hpp](#)

7.29 OsiSimpleInteger Class Reference

Define a single integer class.

```
#include <OsiBranchingObject.hpp>
```

Inheritance diagram for OsiSimpleInteger:

Collaboration diagram for OsiSimpleInteger:

Public Member Functions

- [OsiSimpleInteger \(\)](#)
Default Constructor.

- [OsiSimpleInteger](#) (const [OsiSolverInterface](#) *solver, int iColumn)
Useful constructor - passed solver index.
- [OsiSimpleInteger](#) (int iColumn, double lower, double upper)
Useful constructor - passed solver index and original bounds.
- [OsiSimpleInteger](#) (const [OsiSimpleInteger](#) &)
Copy constructor.
- virtual [OsiObject](#) * [clone](#) () const
Clone.
- [OsiSimpleInteger](#) & [operator=](#) (const [OsiSimpleInteger](#) &rhs)
Assignment operator.
- virtual [~OsiSimpleInteger](#) ()
Destructor.
- virtual double [infeasibility](#) (const [OsiBranchingInformation](#) *info, int &[whichWay](#)) const
Infeasibility - large is 0.5.
- virtual double [feasibleRegion](#) ([OsiSolverInterface](#) *solver, const [OsiBranchingInformation](#) *info) const
Set bounds to fix the variable at the current (integer) value.
- virtual [OsiBranchingObject](#) * [createBranch](#) ([OsiSolverInterface](#) *solver, const [OsiBranchingInformation](#) *info, int way) const
Creates a branching object.
- void [setColumnNumber](#) (int value)
Set solver column number.
- virtual int [columnNumber](#) () const
Column number if single column object -1 otherwise, so returns ≥ 0 Used by heuristics.
- double [originalLowerBound](#) () const
Original bounds.
- virtual void [resetBounds](#) (const [OsiSolverInterface](#) *solver)
Reset variable bounds to their original values.
- virtual void [resetSequenceEtc](#) (int numberColumns, const int *originalColumns)
Change column numbers after preprocessing.
- virtual double [upEstimate](#) () const
Return "up" estimate (default 1.0e-5)
- virtual double [downEstimate](#) () const
Return "down" estimate (default 1.0e-5)
- virtual bool [canHandleShadowPrices](#) () const
Return true if knows how to deal with Pseudo Shadow Prices.

Protected Attributes

- double [originalLower_](#)
data Original lower bound
- double [originalUpper_](#)
Original upper bound.
- int [columnNumber_](#)
Column number in solver.

7.29.1 Detailed Description

Define a single integer class.

Definition at line 511 of file OsiBranchingObject.hpp.

7.29.2 Member Function Documentation

7.29.2.1 `virtual double OsiSimpleInteger::feasibleRegion (OsiSolverInterface * solver, const OsiBranchingInformation * info) const` `[virtual]`

Set bounds to fix the variable at the current (integer) value.

Given an integer value, set the lower and upper bounds to fix the variable. Returns amount it had to move variable.

Implements [OsiObject](#).

7.29.2.2 `virtual OsiBranchingObject* OsiSimpleInteger::createBranch (OsiSolverInterface * solver, const OsiBranchingInformation * info, int way) const` `[virtual]`

Creates a branching object.

The preferred direction is set by `way`, 0 for down, 1 for up.

Reimplemented from [OsiObject](#).

7.29.2.3 `virtual void OsiSimpleInteger::resetBounds (const OsiSolverInterface * solver)` `[virtual]`

Reset variable bounds to their original values.

Bounds may be tightened, so it may be good to be able to reset them to their original values.

Reimplemented from [OsiObject](#).

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

- [OsiBranchingObject.hpp](#)

7.30 OsiSolverBranch Class Reference

Solver Branch Class.

```
#include <OsiSolverBranch.hpp>
```

Public Member Functions

Add and Get methods

- void [addBranch](#) (int iColumn, double value)
Add a simple branch (i.e. first sets ub of floor(value), second lb of ceil(value))
- void [addBranch](#) (int way, int numberTighterLower, const int *whichLower, const double *newLower, int numberTighterUpper, const int *whichUpper, const double *newUpper)
Add bounds - way = -1 is first , +1 is second.

- void [addBranch](#) (int way, int numberColumns, const double *oldLower, const double *newLower, const double *oldUpper, const double *newUpper)
Add bounds - way == -1 is first , +1 is second.
- void [applyBounds](#) ([OsiSolverInterface](#) &solver, int way) const
Apply bounds.
- bool [feasibleOneWay](#) (const [OsiSolverInterface](#) &solver) const
Returns true if current solution satisfies one side of branch.
- const int * [starts](#) () const
Starts.
- const int * [which](#) () const
Which variables.
- const double * [bounds](#) () const
Bounds.

Constructors and destructors

- [OsiSolverBranch](#) ()
Default Constructor.
- [OsiSolverBranch](#) (const [OsiSolverBranch](#) &rhs)
Copy constructor.
- [OsiSolverBranch](#) & [operator=](#) (const [OsiSolverBranch](#) &rhs)
Assignment operator.
- [~OsiSolverBranch](#) ()
Destructor.

7.30.1 Detailed Description

Solver Branch Class.

This provides information on a branch as a set of tighter bounds on both ways

Definition at line 18 of file [OsiSolverBranch.hpp](#).

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

- [OsiSolverBranch.hpp](#)

7.31 OsiSolverInterface Class Reference

Abstract Base Class for describing an interface to a solver.

```
#include <OsiSolverInterface.hpp>
```

Inheritance diagram for [OsiSolverInterface](#):

Collaboration diagram for [OsiSolverInterface](#):

Classes

- class [ApplyCutsReturnCode](#)
Internal class for obtaining status from the [applyCuts](#) method.

Public Types

- typedef std::vector< std::string > [OsiNameVec](#)

Data type for name vectors.

Public Member Functions

Solve methods

- virtual void [initialSolve](#) ()=0
Solve initial LP relaxation.
- virtual void [resolve](#) ()=0
Resolve an LP relaxation after problem modification.
- virtual void [branchAndBound](#) ()=0
Invoke solver's built-in enumeration algorithm.

Parameter set/get methods

The set methods return true if the parameter was set to the given value, false otherwise.

When a set method returns false, the original value (if any) should be unchanged. 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.

Note

There is a default implementation of the set/get methods, namely to store/retrieve the given value using an array in the base class. A specific solver implementation can use this feature, for example, to store parameters that should be used later on. Implementors of a solver interface should overload these functions to provide the proper interface to and accurately reflect the capabilities of a specific solver.

The format for hints is slightly different in that a boolean specifies the sense of the hint and an enum specifies the strength of the hint. Hints should be initialised when a solver is instantiated. (See [OsiSolverParameters.hpp](#) for defined hint parameters and strength.) When specifying the sense of the hint, a value of true means to work with the hint, false to work against it. For example,

- [setHintParam](#)([OsiDoScale](#), true, [OsiHintTry](#))
is a mild suggestion to the solver to scale the constraint system.
- [setHintParam](#)([OsiDoScale](#), false, [OsiForceDo](#))
tells the solver to disable scaling, or throw an exception if it cannot comply.

As another example, a solver interface could use the value and strength of the [OsiDoReducePrint](#) hint to adjust the amount of information printed by the interface and/or solver. The extent to which a solver obeys hints is left to the solver. The value and strength returned by [getHintParam](#) will match the most recent call to [setHintParam](#), and will not necessarily reflect the solver's ability to comply with the hint. If the hint strength is [OsiForceDo](#), the solver is required to throw an exception if it cannot perform the specified action.

Note

As with the other set/get methods, there is a default implementation which maintains arrays in the base class for hint sense and strength. The default implementation does not store the [otherInformation](#) pointer, and always throws an exception for strength [OsiForceDo](#). Implementors of a solver interface should override these functions to provide the proper interface to and accurately reflect the capabilities of a specific solver.

- virtual bool [setIntParam](#) (OsiIntParam key, int value)
Set an integer parameter.
- virtual bool [setDbiParam](#) (OsiDbiParam key, double value)
Set a double parameter.
- virtual bool [setStrParam](#) (OsiStrParam key, const std::string &value)
Set a string parameter.
- virtual bool [setHintParam](#) (OsiHintParam key, bool yesNo=true, OsiHintStrength strength=OsiHintTry, void *=NULL)
Set a hint parameter.
- virtual bool [getIntParam](#) (OsiIntParam key, int &value) const
Get an integer parameter.
- virtual bool [getDbiParam](#) (OsiDbiParam key, double &value) const
Get a double parameter.
- virtual bool [getStrParam](#) (OsiStrParam key, std::string &value) const
Get a string parameter.
- virtual bool [getHintParam](#) (OsiHintParam key, bool &yesNo, OsiHintStrength &strength, void *&other← Information) const
Get a hint parameter (all information)
- virtual bool [getHintParam](#) (OsiHintParam key, bool &yesNo, OsiHintStrength &strength) const
Get a hint parameter (sense and strength only)
- virtual bool [getHintParam](#) (OsiHintParam key, bool &yesNo) const
Get a hint parameter (sense only)
- void [copyParameters](#) (OsiSolverInterface &rhs)
Copy all parameters in this section from one solver to another.
- double [getIntegerTolerance](#) () const
Return the integrality tolerance of the underlying solver.

Methods returning info on how the solution process terminated

- virtual bool [isAbandoned](#) () const =0
Are there numerical difficulties?
- virtual bool [isProvenOptimal](#) () const =0
Is optimality proven?
- virtual bool [isProvenPrimalInfeasible](#) () const =0
Is primal infeasibility proven?
- virtual bool [isProvenDualInfeasible](#) () const =0
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 =0
Iteration limit reached?

Warm start methods

Note that the warm start methods return a generic **CoinWarmStart** object.

The precise characteristics of this object are solver-dependent. Clients who wish to maintain a maximum degree of solver independence should take care to avoid unnecessary assumptions about the properties of a warm start object.

- virtual **CoinWarmStart** * [getEmptyWarmStart](#) () const =0
Get an empty warm start object.

- virtual **CoinWarmStart** * [getWarmStart](#) () const =0
Get warm start information.
- virtual **CoinWarmStart** * [getPointerToWarmStart](#) (bool &mustDelete)
Get warm start information.
- virtual bool [setWarmStart](#) (const **CoinWarmStart** *warmstart)=0
Set warm start information.

Hot start methods

Primarily used in strong branching.

The user can create a hot start object — a snapshot of the optimization process — then reoptimize over and over again, starting from the same point.

Note

- *Between hot started optimizations only bound changes are allowed.*
- *The copy constructor and assignment operator should NOT copy any hot start information.*
- *The default implementation simply extracts a warm start object in `markHotStart`, resets to the warm start object in `solveFromHotStart`, and deletes the warm start object in `unmarkHotStart`. Actual solver implementations are encouraged to do better.*
- virtual void [markHotStart](#) ()
Create a hot start snapshot of the optimization process.
- virtual void [solveFromHotStart](#) ()
Optimize starting from the hot start snapshot.
- virtual void [unmarkHotStart](#) ()
Delete the hot start snapshot.

Problem query methods

Querying a problem that has no data associated with it will result in zeros for the number of rows and columns, and NULL pointers from the methods that return vectors.

Const pointers returned from any data-query method are valid as long as the data is unchanged and the solver is not called.

- virtual int [getNumCols](#) () const =0
Get the number of columns.
- virtual int [getNumRows](#) () const =0
Get the number of rows.
- virtual int [getNumElements](#) () const =0
Get the number of nonzero elements.
- virtual int [getNumIntegers](#) () const
Get the number of integer variables.
- virtual const double * [getColLower](#) () const =0
Get a pointer to an array[getNumCols()] of column lower bounds.
- virtual const double * [getColUpper](#) () const =0
Get a pointer to an array[getNumCols()] of column upper bounds.
- virtual const char * [getRowSense](#) () const =0
Get a pointer to an array[getNumRows()] of row constraint senses.
- virtual const double * [getRightHandSide](#) () const =0
Get a pointer to an array[getNumRows()] of row right-hand sides.
- virtual const double * [getRowRange](#) () const =0
Get a pointer to an array[getNumRows()] of row ranges.
- virtual const double * [getRowLower](#) () const =0
Get a pointer to an array[getNumRows()] of row lower bounds.

- virtual const double * [getRowUpper](#) () const =0
Get a pointer to an array[getNumRows()] of row upper bounds.
- virtual const double * [getObjCoefficients](#) () const =0
Get a pointer to an array[getNumCols()] of objective function coefficients.
- virtual double [getObjSense](#) () const =0
Get the objective function sense.
- virtual bool [isContinuous](#) (int colIndex) const =0
Return true if the variable is continuous.
- virtual bool [isBinary](#) (int colIndex) const
Return true if the variable is binary.
- virtual bool [isInteger](#) (int colIndex) const
Return true if the variable is integer.
- virtual bool [isIntegerNonBinary](#) (int colIndex) const
Return true if the variable is general integer.
- virtual bool [isFreeBinary](#) (int colIndex) const
Return true if the variable is binary and not fixed.
- const char * [columnType](#) (bool refresh=false) const
Return an array[getNumCols()] of column types.
- virtual const char * [getColType](#) (bool refresh=false) const
Return an array[getNumCols()] of column types.
- virtual const **CoinPackedMatrix** * [getMatrixByRow](#) () const =0
Get a pointer to a row-wise copy of the matrix.
- virtual const **CoinPackedMatrix** * [getMatrixByCol](#) () const =0
Get a pointer to a column-wise copy of the matrix.
- virtual **CoinPackedMatrix** * [getMutableMatrixByRow](#) () const
Get a pointer to a mutable row-wise copy of the matrix.
- virtual **CoinPackedMatrix** * [getMutableMatrixByCol](#) () const
Get a pointer to a mutable column-wise copy of the matrix.
- virtual double [getInfinity](#) () const =0
Get the solver's value for infinity.

Solution query methods

- virtual const double * [getColSolution](#) () const =0
Get a pointer to an array[getNumCols()] of primal variable values.
- virtual const double * [getStrictColSolution](#) ()
Get a pointer to an array[getNumCols()] of primal variable values guaranteed to be between the column lower and upper bounds.
- virtual const double * [getRowPrice](#) () const =0
Get pointer to array[getNumRows()] of dual variable values.
- virtual const double * [getReducedCost](#) () const =0
Get a pointer to an array[getNumCols()] of reduced costs.
- virtual const double * [getRowActivity](#) () const =0
Get a pointer to array[getNumRows()] of row activity levels.
- virtual double [getObjValue](#) () const =0
Get the objective function value.
- virtual int [getIterationCount](#) () const =0
Get the number of iterations it took to solve the problem (whatever 'iteration' means to the solver).
- virtual std::vector< double * > [getDualRays](#) (int maxNumRays, bool fullRay=false) const =0
Get as many dual rays as the solver can provide.
- virtual std::vector< double * > [getPrimalRays](#) (int maxNumRays) const =0
Get as many primal rays as the solver can provide.
- virtual OsiVectorInt [getFractionalIndices](#) (const double etol=1.e-05) const

Get vector of indices of primal variables which are integer variables but have fractional values in the current solution.

Methods to modify the objective, bounds, and solution

For functions which take a set of indices as parameters (`setObjCoeffSet()`, `setColSetBounds()`, `setRowSetBounds()`, `setRowSetTypes()`), the parameters follow the C++ STL iterator convention: `indexFirst` points to the first index in the set, and `indexLast` points to a position one past the last index in the set.

- virtual void `setObjCoeff` (int elementIndex, double elementValue)=0
Set an objective function coefficient.
- virtual void `setObjCoeffSet` (const int *indexFirst, const int *indexLast, const double *coeffList)
Set a set of objective function coefficients.
- virtual void `setObjective` (const double *array)
Set the objective coefficients for all columns.
- virtual void `setObjSense` (double s)=0
Set the objective function sense.
- virtual void `setColLower` (int elementIndex, double elementValue)=0
Set a single column lower bound.
- virtual void `setColLower` (const double *array)
Set the lower bounds for all columns.
- virtual void `setColUpper` (int elementIndex, double elementValue)=0
Set a single column upper bound.
- virtual void `setColUpper` (const double *array)
Set the upper bounds for all columns.
- 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 upper and lower bounds of a set of columns.
- virtual void `setRowLower` (int elementIndex, double elementValue)=0
Set a single row lower bound.
- virtual void `setRowUpper` (int elementIndex, double elementValue)=0
Set a single row upper bound.
- virtual void `setRowBounds` (int elementIndex, double lower, double upper)
Set a single row lower and upper bound.
- virtual void `setRowSetBounds` (const int *indexFirst, const int *indexLast, const double *boundList)
Set the bounds on a set of rows.
- virtual void `setRowType` (int index, char sense, double rightHandSide, double range)=0
Set the type of a single row.
- virtual void `setRowSetTypes` (const int *indexFirst, const int *indexLast, const char *senseList, const double *rhsList, const double *rangeList)
Set the type of a set of rows.
- virtual void `setColSolution` (const double *colsol)=0
Set the primal solution variable values.
- virtual void `setRowPrice` (const double *rowprice)=0
Set dual solution variable values.
- virtual int `reducedCostFix` (double gap, bool justInteger=true)
Fix variables at bound based on reduced cost.

Methods to set variable type

- virtual void `setContinuous` (int index)=0
Set the index-th variable to be a continuous variable.
- virtual void `setInteger` (int index)=0
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.

Methods for row and column names

Osi defines three name management disciplines: 'auto names' (0), 'lazy names' (1), and 'full names' (2).

See the description of `#OsiNameDiscipline` for details. Changing the name discipline (via [setIntParam\(\)](#)) will not automatically add or remove name information, but setting the discipline to auto will make existing information inaccessible until the discipline is reset to lazy or full.

By definition, a row index of [getNumRows\(\)](#) (i.e., one larger than the largest valid row index) refers to the objective function.

OSI users and implementors: While the OSI base class can define an interface and provide rudimentary support, use of names really depends on support by the `OsiXXX` class to ensure that names are managed correctly. If an `OsiXXX` class does not support names, it should return false for calls to [getIntParam\(\)](#) or [setIntParam\(\)](#) that reference `OsiNameDiscipline`.

- virtual std::string [dfltRowColName](#) (char rc, int ndx, unsigned digits=7) const
Generate a standard name of the form Rnnnnnnn or Cnnnnnnn.
- virtual std::string [getObjName](#) (unsigned maxlen=static_cast< unsigned >(std::string::npos)) const
Return the name of the objective function.
- virtual void [setObjName](#) (std::string name)
Set the name of the objective function.
- virtual std::string [getRowName](#) (int rowIndex, unsigned maxlen=static_cast< unsigned >(std::string::npos)) const
Return the name of the row.
- virtual const [OsiNameVec](#) & [getRowNames](#) ()
Return a pointer to a vector of row names.
- virtual void [setRowName](#) (int ndx, std::string name)
Set a row name.
- virtual void [setRowNames](#) ([OsiNameVec](#) &srcNames, int srcStart, int len, int tgtStart)
Set multiple row names.
- virtual void [deleteRowNames](#) (int tgtStart, int len)
Delete len row names starting at index tgtStart.
- virtual std::string [getColName](#) (int colIndex, unsigned maxlen=static_cast< unsigned >(std::string::npos)) const
Return the name of the column.
- virtual const [OsiNameVec](#) & [getColNames](#) ()
Return a pointer to a vector of column names.
- virtual void [setColName](#) (int ndx, std::string name)
Set a column name.
- virtual void [setColNames](#) ([OsiNameVec](#) &srcNames, int srcStart, int len, int tgtStart)
Set multiple column names.
- virtual void [deleteColNames](#) (int tgtStart, int len)
Delete len column names starting at index tgtStart.
- void [setRowColNames](#) (const [CoinMpsIO](#) &mps)
Set row and column names from a [CoinMpsIO](#) object.
- void [setRowColNames](#) ([CoinModel](#) &mod)
Set row and column names from a [CoinModel](#) object.
- void [setRowColNames](#) ([CoinLpIO](#) &mod)
Set row and column names from a [CoinLpIO](#) object.

Methods to modify the constraint system.

Note that new columns are added as continuous variables.

- virtual void [addCol](#) (const **CoinPackedVectorBase** &vec, const double collb, const double colub, const double obj)=0
Add a column (primal variable) to the problem.
- 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)
Add a set of columns (primal variables) to the problem.
- virtual void [addCols](#) (const int numcols, const int *columnStarts, const int *rows, const double *elements, const double *collb, const double *colub, const double *obj)
Add a set of columns (primal variables) to the problem.
- void [addCols](#) (const **CoinBuild** &buildObject)
*Add columns using a **CoinBuild** object.*
- int [addCols](#) (**CoinModel** &modelObject)
Add columns from a model object.
- virtual void [deleteCols](#) (const int num, const int *colIndices)=0
Remove a set of columns (primal variables) from the problem.
- virtual void [addRow](#) (const **CoinPackedVectorBase** &vec, const double rowlb, const double rowub)=0
Add a row (constraint) to the problem.
- 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)=0
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 [addRow](#) (int numberElements, const int *columns, const double *element, const double rowlb, const double rowub)
Add a row (constraint) to the problem.
- virtual void [addRows](#) (const int numRows, const **CoinPackedVectorBase** *const *rows, const double *rowlb, const double *rowub)
Add a set of rows (constraints) to the problem.
- virtual void [addRows](#) (const int numRows, const **CoinPackedVectorBase** *const *rows, const char *rowsen, const double *rowrhs, const double *rowrng)
Add a set of rows (constraints) to the problem.
- virtual void [addRows](#) (const int numRows, const int *rowStarts, const int *columns, const double *element, const double *rowlb, const double *rowub)
Add a set of rows (constraints) to the problem.
- void [addRows](#) (const **CoinBuild** &buildObject)
*Add rows using a **CoinBuild** object.*
- int [addRows](#) (**CoinModel** &modelObject)
*Add rows from a **CoinModel** object.*

- virtual void [deleteRows](#) (const int num, const int *rowIndices)=0
Delete a set of rows (constraints) from the problem.
- virtual void [replaceMatrixOptional](#) (const **CoinPackedMatrix** &)
Replace the constraint matrix.
- virtual void [replaceMatrix](#) (const **CoinPackedMatrix** &)
Replace the constraint matrix.
- virtual void [saveBaseModel](#) ()
Save a copy of the base model.
- virtual void [restoreBaseModel](#) (int numberOfRows)
Reduce the constraint system to the specified number of constraints.
- virtual [ApplyCutsReturnCode](#) [applyCuts](#) (const **OsiCuts** &cs, double effectivenessLb=0.0)
Apply a collection of cuts.
- 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.
- void [deleteBranchingInfo](#) (int numberDeleted, const int *which)
Deletes branching information before columns deleted.

Methods for problem input and output

- virtual void [loadProblem](#) (const **CoinPackedMatrix** &matrix, const double *collb, const double *colub, const double *obj, const double *rowlb, const double *rowub)=0
Load in a problem by copying the arguments.
- virtual void [assignProblem](#) (**CoinPackedMatrix** *&matrix, double *&collb, double *&colub, double *&obj, double *&rowlb, double *&rowub)=0
Load in a problem by assuming ownership of the arguments.
- virtual void [loadProblem](#) (const **CoinPackedMatrix** &matrix, const double *collb, const double *colub, const double *obj, const char *rowsen, const double *rowrhs, const double *rowrng)=0
Load in a problem by copying the arguments.
- virtual void [assignProblem](#) (**CoinPackedMatrix** *&matrix, double *&collb, double *&colub, double *&obj, char *&rowsen, double *&rowrhs, double *&rowrng)=0
Load in a problem by assuming ownership of the arguments.
- 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)=0
Load in a problem by copying the arguments.
- 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)=0
Load in a problem by copying the arguments.
- virtual int [loadFromCoinModel](#) (**CoinModel** &modelObject, bool keepSolution=false)
*Load a model from a **CoinModel** object.*
- virtual int [readMps](#) (const char *filename, const char *extension="mps")
Read a problem in MPS format from the given filename.
- virtual int [readMps](#) (const char *filename, const char *extension, int &numberSets, **CoinSet** **&sets)
Read a problem in MPS format from the given full filename.
- virtual int [readGMPL](#) (const char *filename, const char *dataname=NULL)
Read a problem in GMPL format from the given filenames.
- virtual void [writeMps](#) (const char *filename, const char *extension="mps", double objSense=0.0) const =0
Write the problem in MPS format to the specified file.
- int [writeMpsNative](#) (const char *filename, const char **rowNames, const char **columnNames, int format←Type=0, int numberAcross=2, double objSense=0.0, int numberSOS=0, const **CoinSet** *setInfo=NULL) const
Write the problem in MPS format to the specified file with more control over the output.

- 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 with the specified extension.
- 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.
- int [writeLpNative](#) (const char *filename, char const *const *const rowNames, char const *const *const columnNames, const double epsilon=1.0e-5, const int numberAcross=10, const int decimals=5, const double objSense=0.0, const bool useRowNames=true) const
Write the problem into an Lp file.
- int [writeLpNative](#) (FILE *fp, char const *const *const rowNames, char const *const *const columnNames, const double epsilon=1.0e-5, const int numberAcross=10, const int decimals=5, const double objSense=0.0, const bool useRowNames=true) const
Write the problem into the file pointed to by the parameter fp.
- virtual int [readLp](#) (const char *filename, const double epsilon=1e-5)
Read file in LP format from file with name filename.
- int [readLp](#) (FILE *fp, const double epsilon=1e-5)
Read file in LP format from the file pointed to by fp.

Miscellaneous

- int [differentModel](#) ([OsiSolverInterface](#) &other, bool ignoreNames=true)
Check two models against each other.

Setting/Accessing application data

- void [setApplicationData](#) (void *appData)
Set application data.
- void [setAuxiliaryInfo](#) ([OsiAuxInfo](#) *auxiliaryInfo)
Create a clone of an Auxiliary Information object.
- void * [getApplicationData](#) () const
Get application data.
- [OsiAuxInfo](#) * [getAuxiliaryInfo](#) () const
Get pointer to auxiliary info object.

Message handling

See the COIN library documentation for additional information about COIN message facilities.

- virtual void [passInMessageHandler](#) ([CoinMessageHandler](#) *handler)
Pass in a message handler.
- void [newLanguage](#) ([CoinMessages::Language](#) language)
Set language.
- void [setLanguage](#) ([CoinMessages::Language](#) language)
- [CoinMessageHandler](#) * [messageHandler](#) () const
Return a pointer to the current message handler.
- [CoinMessages](#) [messages](#) ()
Return the current set of messages.
- [CoinMessages](#) * [messagesPointer](#) ()
Return a pointer to the current set of messages.
- bool [defaultHandler](#) () const
Return true if default handler.

Methods for dealing with discontinuities other than integers.

Osi should be able to know about SOS and other types.

This is an optional section where such information can be stored.

- void [findIntegers](#) (bool justCount)
Identify integer variables and create corresponding objects.
- virtual int [findIntegersAndSOS](#) (bool justCount)
Identify integer variables and SOS and create corresponding objects.
- int [numberObjects](#) () const
Get the number of objects.
- void [setNumberObjects](#) (int number)
Set the number of objects.
- [OsiObject](#) ** [objects](#) () const
Get the array of objects.
- const [OsiObject](#) * [object](#) (int which) const
Get the specified object.
- [OsiObject](#) * [modifiableObject](#) (int which) const
Get the specified object.
- void [deleteObjects](#) ()
Delete all object information.
- void [addObjects](#) (int [numberObjects](#), [OsiObject](#) **[objects](#))
Add in object information.
- double [forceFeasible](#) ()
Use current solution to set bounds so current integer feasible solution will stay feasible.

Methods related to testing generated cuts

See the documentation for [OsiRowCutDebugger](#) for additional details.

- virtual void [activateRowCutDebugger](#) (const char *modelName)
Activate the row cut debugger.
- virtual void [activateRowCutDebugger](#) (const double *solution, bool enforceOptimality=true)
Activate the row cut debugger using a full solution array.
- const [OsiRowCutDebugger](#) * [getRowCutDebugger](#) () const
Get the row cut debugger provided the solution known to the debugger is within the feasible region held in the solver.
- [OsiRowCutDebugger](#) * [getRowCutDebuggerAlways](#) () const
Get the row cut debugger object.

OsiSimplexInterface

Simplex Interface

Methods for an advanced interface to a simplex solver. The interface comprises two groups of methods. Group 1 contains methods for tableau access. Group 2 contains methods for dictating individual simplex pivots.

- virtual int [canDoSimplexInterface](#) () const
Return the simplex implementation level.

OsiSimplex Group 1

Tableau access methods.

This group of methods provides access to rows and columns of the basis inverse and to rows and columns of the tableau.

- virtual void [enableFactorization](#) () const
Prepare the solver for the use of tableau access methods.

- virtual void [disableFactorization](#) () const
Undo the effects of [enableFactorization](#).
- virtual bool [basisIsAvailable](#) () const
Check if an optimal basis is available.
- bool [optimalBasisIsAvailable](#) () const
Synonym for [basisIsAvailable](#).
- virtual void [getBasisStatus](#) (int *cstat, int *rstat) const
Retrieve status information for column and row variables.
- virtual int [setBasisStatus](#) (const int *cstat, const int *rstat)
Set the status of column and row variables and update the basis factorization and solution.
- virtual void [getReducedGradient](#) (double *columnReducedCosts, double *duals, const double *c) const
Calculate duals and reduced costs for the given objective coefficients.
- virtual void [getBlnvARow](#) (int row, double *z, double *slack=NULL) const
Get a row of the tableau.
- virtual void [getBlnvRow](#) (int row, double *z) const
Get a row of the basis inverse.
- virtual void [getBlnvACol](#) (int col, double *vec) const
Get a column of the tableau.
- virtual void [getBlnvCol](#) (int col, double *vec) const
Get a column of the basis inverse.
- virtual void [getBasics](#) (int *index) const
Get indices of basic variables.

OsiSimplex Group 2

Pivoting methods

This group of methods provides for control of individual pivots by a simplex solver.

- virtual void [enableSimplexInterface](#) (bool doingPrimal)
Enables normal operation of subsequent functions.
- virtual void [disableSimplexInterface](#) ()
Undo whatever setting changes the above method had to make.
- virtual int [pivot](#) (int colIn, int colOut, int outStatus)
Perform a pivot by substituting a colIn for colOut in the basis.
- virtual int [primalPivotResult](#) (int colIn, int sign, int &colOut, int &outStatus, double &t, **CoinPackedVector** *dx)
*Obtain a result of the primal pivot Outputs: colOut – leaving column, outStatus – its status, t – step size, and, if dx!=NULL, *dx – primal ray direction.*
- virtual int [dualPivotResult](#) (int &colIn, int &sign, int colOut, int outStatus, double &t, **CoinPackedVector** *dx)
*Obtain a result of the dual pivot (similar to the previous method) Differences: entering variable and a sign of its change are now the outputs, the leaving variable and its status – the inputs If dx!=NULL, then *dx contains dual ray Return code: same.*

Constructors and destructors

- [OsiSolverInterface](#) ()
Default Constructor.
- virtual [OsiSolverInterface](#) * [clone](#) (bool copyData=true) const =0
Clone.
- [OsiSolverInterface](#) (const [OsiSolverInterface](#) &)
Copy constructor.
- [OsiSolverInterface](#) & [operator=](#) (const [OsiSolverInterface](#) &rhs)
Assignment operator.
- virtual [~OsiSolverInterface](#) ()
Destructor.
- virtual void [reset](#) ()
Reset the solver interface.

Protected Member Functions

Protected methods

- virtual void [applyRowCut](#) (const [OsiRowCut](#) &rc)=0
Apply a row cut (append to the constraint matrix).
- virtual void [applyColCut](#) (const [OsiColCut](#) &cc)=0
Apply a column cut (adjust the bounds of one or more variables).
- void [convertBoundToSense](#) (const double lower, const double upper, char &sense, double &right, double &range) const
A quick inlined function to convert from the lb/ub style of constraint definition to the sense/rhs/range style.
- void [convertSenseToBound](#) (const char sense, const double right, const double range, double &lower, double &upper) const
A quick inlined function to convert from the sense/rhs/range style of constraint definition to the lb/ub style.
- template<class T >
T [forceIntoRange](#) (const T value, const T lower, const T upper) const
A quick inlined function to force a value to be between a minimum and a maximum value.
- void [setInitialData](#) ()
Set [OsiSolverInterface](#) object state for default constructor.

Protected Attributes

Protected member data

- [OsiRowCutDebugger](#) * [rowCutDebugger_](#)
Pointer to row cut debugger object.
- [CoinMessageHandler](#) * [handler_](#)
Message handler.
- bool [defaultHandler_](#)
Flag to say if the current handler is the default handler.
- [CoinMessages](#) [messages_](#)
Messages.
- int [numberIntegers_](#)
Number of integers.
- int [numberObjects_](#)
Total number of objects.
- [OsiObject](#) ** [object_](#)
Integer and ... information (integer info normally at beginning)
- char * [columnType_](#)
Column type 0 - continuous 1 - binary (may get fixed later) 2 - general integer (may get fixed later)

Friends

- void [OsiSolverInterfaceCommonUnitTest](#) (const [OsiSolverInterface](#) *emptySi, const std::string &mpsDir, const std::string &netlibDir)
A function that tests the methods in the [OsiSolverInterface](#) class.
- void [OsiSolverInterfaceMpsUnitTest](#) (const std::vector< [OsiSolverInterface](#) * > &vecSiP, const std::string &mpsDir)
A function that tests that a lot of problems given in MPS files (mostly the NETLIB problems) solve properly with all the specified solvers.

7.31.1 Detailed Description

Abstract Base Class for describing an interface to a solver.

Many [OsiSolverInterface](#) query methods return a const pointer to the requested read-only data. If the model data is changed or the solver is called, these pointers may no longer be valid and should be refreshed by invoking the member function to obtain an updated copy of the pointer. For example:

```
OsiSolverInterface solverInterfacePtr ;
const double * ruBnds = solverInterfacePtr->getRowUpper();
solverInterfacePtr->applyCuts(someSetOfCuts);
// ruBnds is no longer a valid pointer and must be refreshed
ruBnds = solverInterfacePtr->getRowUpper();
```

Querying a problem that has no data associated with it will result in zeros for the number of rows and columns, and NULL pointers from the methods that return vectors.

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

7.31.2 Member Typedef Documentation

7.31.2.1 `typedef std::vector<std::string> OsiSolverInterface::OsiNameVec`

Data type for name vectors.

Definition at line 888 of file `OsiSolverInterface.hpp`.

7.31.3 Member Function Documentation

7.31.3.1 `virtual void OsiSolverInterface::resolve () [pure virtual]`

Resolve an LP relaxation after problem modification.

Note the 're-' in 'resolve'. [initialSolve\(\)](#) should be used to solve the problem for the first time.

Implemented in [OsiSpxSolverInterface](#), [OsiGlpkSolverInterface](#), [OsiCpxSolverInterface](#), [OsiGrbSolverInterface](#), [Osi↔MskSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.2 `virtual bool OsiSolverInterface::setHintParam (OsiHintParam key, bool yesNo = true, OsiHintStrength strength = OsiHintTry, void * = NULL) [inline], [virtual]`

Set a hint parameter.

The `otherInformation` parameter can be used to pass in an arbitrary block of information which is interpreted by the OSI and the underlying solver. Users are cautioned that this hook is solver-specific.

Implementors: The default implementation completely ignores `otherInformation` and always throws an exception for `OsiForceDo`. This is almost certainly not the behaviour you want; you really should override this method.

Reimplemented in [OsiGlpkSolverInterface](#), and [OsiGrbSolverInterface](#).

Definition at line 294 of file `OsiSolverInterface.hpp`.

7.31.3.3 `virtual bool OsiSolverInterface::getHintParam (OsiHintParam key, bool & yesNo, OsiHintStrength & strength, void *& otherInformation) const [inline], [virtual]`

Get a hint parameter (all information)

Return all available information for the hint: sense, strength, and any extra information associated with the hint.

Implementors: The default implementation will always set `otherInformation` to `NULL`. This is almost certainly not the behaviour you want; you really should override this method.

Reimplemented in [OsiGrbSolverInterface](#).

Definition at line 333 of file `OsiSolverInterface.hpp`.

```
7.31.3.4  virtual bool OsiSolverInterface::getHintParam ( OsiHintParam key, bool & yesNo, OsiHintStrength & strength ) const
          [inline],[virtual]
```

Get a hint parameter (sense and strength only)

Return only the sense and strength of the hint.

Reimplemented in [OsiGrbSolverInterface](#).

Definition at line 347 of file `OsiSolverInterface.hpp`.

```
7.31.3.5  virtual bool OsiSolverInterface::getHintParam ( OsiHintParam key, bool & yesNo ) const  [inline],[virtual]
```

Get a hint parameter (sense only)

Return only the sense (true/false) of the hint.

Reimplemented in [OsiGrbSolverInterface](#).

Definition at line 359 of file `OsiSolverInterface.hpp`.

```
7.31.3.6  void OsiSolverInterface::copyParameters ( OsiSolverInterface & rhs )
```

Copy all parameters in this section from one solver to another.

Note that the current implementation also copies the `appData` block, message handler, and `rowCutDebugger`. Arguably these should have independent copy methods.

```
7.31.3.7  double OsiSolverInterface::getIntegerTolerance ( ) const  [inline]
```

Return the integrality tolerance of the underlying solver.

We should be able to get an integrality tolerance, but until that time just use the primal tolerance

Definition at line 386 of file `OsiSolverInterface.hpp`.

```
7.31.3.8  virtual CoinWarmStart* OsiSolverInterface::getEmptyWarmStart ( ) const  [pure virtual]
```

Get an empty warm start object.

This routine returns an empty warm start object. Its purpose is to provide a way for a client to acquire a warm start object of the appropriate type for the solver, which can then be resized and modified as desired.

Implemented in [OsiGrbSolverInterface](#), [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.9 `virtual CoinWarmStart* OsiSolverInterface::getWarmStart () const` [pure 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.

Implemented in [OsiGrbSolverInterface](#), [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.10 `virtual CoinWarmStart* OsiSolverInterface::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.

7.31.3.11 `virtual bool OsiSolverInterface::setWarmStart (const CoinWarmStart * warmstart)` [pure virtual]

Set warm start information.

Return true or false depending on whether the warm start information was accepted or not. By definition, a call to `setWarmStart` with a null parameter should cause the solver interface to refresh its warm start information from the underlying solver.

Implemented in [OsiGrbSolverInterface](#), [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.12 `virtual const char* OsiSolverInterface::getRowSense () const` [pure virtual]

Get a pointer to an array[[getNumRows\(\)](#)] of row constraint senses.

- 'L': <= constraint
- 'E': = constraint
- 'G': >= constraint
- 'R': ranged constraint
- 'N': free constraint

Implemented in [OsiGrbSolverInterface](#), [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.13 `virtual const double* OsiSolverInterface::getRightHandSide () const` [pure virtual]

Get a pointer to an array[[getNumRows\(\)](#)] of row right-hand sides.

- if [getRowSense\(\)\[i\]](#) == 'L' then [getRightHandSide\(\)\[i\]](#) == [getRowUpper\(\)\[i\]](#)
- if [getRowSense\(\)\[i\]](#) == 'G' then [getRightHandSide\(\)\[i\]](#) == [getRowLower\(\)\[i\]](#)
- if [getRowSense\(\)\[i\]](#) == 'R' then [getRightHandSide\(\)\[i\]](#) == [getRowUpper\(\)\[i\]](#)

- if `getRowSense()[i] == 'N'` then `getRightHandSide()[i] == 0.0`

Implemented in [OsiGrbSolverInterface](#), [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.14 `virtual const double* OsiSolverInterface::getRowRange () const` `[pure virtual]`

Get a pointer to an array[`getNumRows()`] of row ranges.

- if `getRowSense()[i] == 'R'` then `getRowRange()[i] == getRowUpper()[i] - getRowLower()[i]`
- if `getRowSense()[i] != 'R'` then `getRowRange()[i]` is 0.0

Implemented in [OsiGrbSolverInterface](#), [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.15 `virtual double OsiSolverInterface::getObjSense () const` `[pure virtual]`

Get the objective function sense.

- 1 for minimisation (default)
- -1 for maximisation

Implemented in [OsiGrbSolverInterface](#), [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.16 `virtual bool OsiSolverInterface::isInteger (int colIndex) const` `[virtual]`

Return true if the variable is integer.

This method returns true if the variable is binary or general integer.

7.31.3.17 `const char* OsiSolverInterface::columnType (bool refresh = false) const` `[inline]`

Return an array[`getNumCols()`] of column types.

Deprecated See [getColType](#)

Definition at line 593 of file `OsiSolverInterface.hpp`.

7.31.3.18 `virtual const char* OsiSolverInterface::getColType (bool refresh = false) const` `[virtual]`

Return an array[`getNumCols()`] of column types.

- 0 - continuous
- 1 - binary
- 2 - general integer

If `refresh` is true, the classification of integer variables as binary or general integer will be reevaluated. If the current bounds are [0,1], or if the variable is fixed at 0 or 1, it will be classified as binary, otherwise it will be classified as general integer.

7.31.3.19 `virtual CoinPackedMatrix* OsiSolverInterface::getMutableMatrixByRow () const [inline],[virtual]`

Get a pointer to a mutable row-wise copy of the matrix.

Returns NULL if the request is not meaningful (i.e., the OSI will not recognise any modifications to the matrix).

Definition at line 620 of file `OsiSolverInterface.hpp`.

7.31.3.20 `virtual CoinPackedMatrix* OsiSolverInterface::getMutableMatrixByCol () const [inline],[virtual]`

Get a pointer to a mutable column-wise copy of the matrix.

Returns NULL if the request is not meaningful (i.e., the OSI will not recognise any modifications to the matrix).

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

7.31.3.21 `virtual const double* OsiSolverInterface::getRowActivity () const [pure virtual]`

Get a pointer to array[[getNumRows\(\)](#)] of row activity levels.

The row activity for a row is the left-hand side evaluated at the current solution.

Implemented in [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.22 `virtual std::vector<double*> OsiSolverInterface::getDualRays (int maxNumRays, bool fullRay = false) const [pure virtual]`

Get as many dual rays as the solver can provide.

In case of proven primal infeasibility there should (with high probability) 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

Implementors of solver interfaces note that the double pointers in the vector should point to arrays of length [getNumRows\(\)](#) (`fullRay = false`) or ([getNumRows\(\)](#)+[getNumCols\(\)](#)) (`fullRay = true`) and they should be allocated with `new[]`.

Clients of solver interfaces note that it is the client's responsibility to free the double pointers in the vector using `delete[]`. Clients are reminded that a problem can be dual and primal infeasible.

Implemented in [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.23 `virtual std::vector<double*> OsiSolverInterface::getPrimalRays (int maxNumRays) const [pure virtual]`

Get as many primal rays as the solver can provide.

In case of proven dual infeasibility there should (with high probability) be at least one.

Note

Implementors of solver interfaces note that the double pointers in the vector should point to arrays of length `getNumCols()` and they should be allocated with `new[]`.

Clients of solver interfaces note that it is the client's responsibility to free the double pointers in the vector using `delete[]`. Clients are reminded that a problem can be dual and primal infeasible.

Implemented in [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.24 `virtual OsiVectorInt OsiSolverInterface::getFractionalIndices (const double etol = 1.e-05) const` [virtual]

Get vector of indices of primal variables which are integer variables but have fractional values in the current solution.

7.31.3.25 `virtual void OsiSolverInterface::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.

7.31.3.26 `virtual void OsiSolverInterface::setObjSense (double s)` [pure virtual]

Set the objective function sense.

Use 1 for minimisation (default), -1 for maximisation.

Note

Implementors note that objective function sense is a parameter of the OSI, not a property of the problem. Objective sense can be set prior to problem load and should not be affected by loading a new problem.

Implemented in [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.27 `virtual void OsiSolverInterface::setColLower (int elementIndex, double elementValue)` [pure virtual]

Set a single column lower bound.

Use `-getInfinity()` for -infinity.

Implemented in [OsiGlpkSolverInterface](#), [OsiCpxSolverInterface](#), [OsiSpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.28 `virtual void OsiSolverInterface::setColLower (const double * array)` [virtual]

Set the lower bounds for all columns.

array `[getNumCols()]` is an array of values for the lower bounds. This defaults to a series of set operations and is here for speed.

7.31.3.29 `virtual void OsiSolverInterface::setColUpper (int elementIndex, double elementValue)` [pure virtual]

Set a single column upper bound.

Use `getInfinity()` for infinity.

Implemented in [OsiGlpkSolverInterface](#), [OsiCpxSolverInterface](#), [OsiSpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.30 `virtual void OsiSolverInterface::setColUpper (const double * array)` [virtual]

Set the upper bounds for all columns.

array [`getNumCols()`] is an array of values for the upper bounds. This defaults to a series of set operations and is here for speed.

7.31.3.31 `virtual void OsiSolverInterface::setColBounds (int elementIndex, double lower, double upper)` [inline],
[virtual]

Set a single column lower and upper bound.

The default implementation just invokes `setColLower()` and `setColUpper()`

Reimplemented in [OsiGlpkSolverInterface](#), [OsiCpxSolverInterface](#), [OsiSpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

Definition at line 778 of file `OsiSolverInterface.hpp`.

7.31.3.32 `virtual void OsiSolverInterface::setColSetBounds (const int * indexFirst, const int * indexLast, const double * boundList)` [virtual]

Set the upper and lower bounds of a set of columns.

The default implementation just invokes `setColBounds()` over and over again. For each column, `boundList` must contain both a lower and upper bound, in that order.

Reimplemented in [OsiGlpkSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.33 `virtual void OsiSolverInterface::setRowLower (int elementIndex, double elementValue)` [pure virtual]

Set a single row lower bound.

Use `-getInfinity()` for -infinity.

Implemented in [OsiGlpkSolverInterface](#), [OsiCpxSolverInterface](#), [OsiSpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.34 `virtual void OsiSolverInterface::setRowUpper (int elementIndex, double elementValue)` [pure virtual]

Set a single row upper bound.

Use `getInfinity()` for infinity.

Implemented in [OsiGlpkSolverInterface](#), [OsiCpxSolverInterface](#), [OsiSpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.35 `virtual void OsiSolverInterface::setRowBounds (int elementIndex, double lower, double upper) [inline],
[virtual]`

Set a single row lower and upper bound.

The default implementation just invokes [setRowLower\(\)](#) and [setRowUpper\(\)](#)

Reimplemented in [OsiGlpkSolverInterface](#), [OsiCpxSolverInterface](#), [OsiSpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

Definition at line 805 of file `OsiSolverInterface.hpp`.

7.31.3.36 `virtual void OsiSolverInterface::setRowSetBounds (const int * indexFirst, const int * indexLast, const double *
boundList) [virtual]`

Set the bounds on a set of rows.

The default implementation just invokes [setRowBounds\(\)](#) over and over again. For each row, `boundList` must contain both a lower and upper bound, in that order.

Reimplemented in [OsiGlpkSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.37 `virtual void OsiSolverInterface::setRowSetTypes (const int * indexFirst, const int * indexLast, const char * senseList,
const double * rhsList, const double * rangeList) [virtual]`

Set the type of a set of rows.

The default implementation just invokes [setRowType\(\)](#) over and over again.

Reimplemented in [OsiGlpkSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.38 `virtual void OsiSolverInterface::setColSolution (const double * colsol) [pure virtual]`

Set the primal solution variable values.

`colsol[getNumCols()]` is an array of values for the primal variables. These values are copied to memory owned by the solver interface object or the solver. They will be returned as the result of [getColSolution\(\)](#) until changed by another call to [setColSolution\(\)](#) or by a call to any solver routine. Whether the solver makes use of the solution in any way is solver-dependent.

Implemented in [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.39 `virtual void OsiSolverInterface::setRowPrice (const double * rowprice) [pure virtual]`

Set dual solution variable values.

`rowprice[getNumRows()]` is an array of values for the dual variables. These values are copied to memory owned by the solver interface object or the solver. They will be returned as the result of [getRowPrice\(\)](#) 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.

Implemented in [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.40 `virtual int OsiSolverInterface::reducedCostFix (double gap, bool justInteger = true) [virtual]`

Fix variables at bound based on reduced cost.

For variables currently at bound, fix the variable at bound if the reduced cost exceeds the gap. Return the number of variables fixed.

If *justInteger* is set to false, the routine will also fix continuous variables, but the test still assumes a delta of 1.0.

7.31.3.41 `virtual std::string OsiSolverInterface::dfltRowColName (char rc, int ndx, unsigned digits = 7) const [virtual]`

Generate a standard name of the form Rnnnnnnn or Cnnnnnnn.

Set *rc* to 'r' for a row name, 'c' for a column name. The 'nnnnnnn' part is generated from *ndx* and will contain 7 digits by default, padded with zeros if necessary. As a special case, *ndx* = [getNumRows\(\)](#) is interpreted as a request for the name of the objective function. OBJECTIVE is returned, truncated to *digits*+1 characters to match the row and column names.

7.31.3.42 `virtual std::string OsiSolverInterface::getRowName (int rowIndex, unsigned maxLen = static_cast< unsigned >(std::string::npos)) const [virtual]`

Return the name of the row.

The routine will *always* return some name, regardless of the name discipline or the level of support by an OsiXXX derived class. Use *maxLen* to limit the length.

7.31.3.43 `virtual const OsiNameVec& OsiSolverInterface::getRowNames () [virtual]`

Return a pointer to a vector of row names.

If the name discipline (#OsiNameDiscipline) is auto, the return value will be a vector of length zero. If the name discipline is lazy, the vector will contain only names supplied by the client and will be no larger than needed to hold those names; entries not supplied will be null strings. In particular, the objective name is *not* included in the vector for lazy names. If the name discipline is full, the vector will have [getNumRows\(\)](#) names, either supplied or generated, plus one additional entry for the objective name.

7.31.3.44 `virtual void OsiSolverInterface::setRowName (int ndx, std::string name) [virtual]`

Set a row name.

Quietly does nothing if the name discipline (#OsiNameDiscipline) is auto. Quietly fails if the row index is invalid.

Reimplemented in [OsiGlpkSolverInterface](#), and [OsiGrbSolverInterface](#).

7.31.3.45 `virtual void OsiSolverInterface::setRowNames (OsiNameVec & srcNames, int srcStart, int len, int tgtStart) [virtual]`

Set multiple row names.

The run of *len* entries starting at *srcNames*[*srcStart*] are installed as row names starting at row index *tgtStart*. The base class implementation makes repeated calls to [setRowName](#).

7.31.3.46 `virtual void OsiSolverInterface::deleteRowNames (int tgtStart, int len) [virtual]`

Delete *len* row names starting at index *tgtStart*.

The specified row names are removed and the remaining row names are copied down to close the gap.

7.31.3.47 `virtual std::string OsiSolverInterface::getColName (int colIndex, unsigned maxLen = static_cast< unsigned >(std::string::npos)) const [virtual]`

Return the name of the column.

The routine will *always* return some name, regardless of the name discipline or the level of support by an OsiXXX derived class. Use *maxLen* to limit the length.

7.31.3.48 `virtual const OsiNameVec& OsiSolverInterface::getColNames () [virtual]`

Return a pointer to a vector of column names.

If the name discipline (#OsiNameDiscipline) is auto, the return value will be a vector of length zero. If the name discipline is lazy, the vector will contain only names supplied by the client and will be no larger than needed to hold those names; entries not supplied will be null strings. If the name discipline is full, the vector will have [getNumCols\(\)](#) names, either supplied or generated.

7.31.3.49 `virtual void OsiSolverInterface::setColName (int ndx, std::string name) [virtual]`

Set a column name.

Quietly does nothing if the name discipline (#OsiNameDiscipline) is auto. Quietly fails if the column index is invalid.

Reimplemented in [OsiGlpkSolverInterface](#), and [OsiGrbSolverInterface](#).

7.31.3.50 `virtual void OsiSolverInterface::setColNames (OsiNameVec & srcNames, int srcStart, int len, int tgtStart) [virtual]`

Set multiple column names.

The run of *len* entries starting at *srcNames*[*srcStart*] are installed as column names starting at column index *tgtStart*. The base class implementation makes repeated calls to `setColName`.

7.31.3.51 `virtual void OsiSolverInterface::deleteColNames (int tgtStart, int len) [virtual]`

Delete *len* column names starting at index *tgtStart*.

The specified column names are removed and the remaining column names are copied down to close the gap.

7.31.3.52 `void OsiSolverInterface::setRowColNames (const CoinMpsIO & mps)`

Set row and column names from a **CoinMpsIO** object.

Also sets the name of the objective function. If the name discipline is auto, you get what you asked for. This routine does not use `setRowName` or `setColName`.

7.31.3.53 void OsiSolverInterface::setRowColNames (CoinModel & mod)

Set row and column names from a **CoinModel** object.

If the name discipline is auto, you get what you asked for. This routine does not use setRowName or setColName.

7.31.3.54 void OsiSolverInterface::setRowColNames (CoinLpIO & mod)

Set row and column names from a **CoinLpIO** object.

Also sets the name of the objective function. If the name discipline is auto, you get what you asked for. This routine does not use setRowName or setColName.

7.31.3.55 virtual void OsiSolverInterface::addCol (const CoinPackedVectorBase & vec, const double collb, const double colub, const double obj) [pure virtual]

Add a column (primal variable) to the problem.

Implemented in [OsiGlpkSolverInterface](#), [OsiCpxSolverInterface](#), [OsiSpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.56 virtual void OsiSolverInterface::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.

The default implementation adds the column, then changes the name. This can surely be made more efficient within an OsiXXX class.

7.31.3.57 virtual void OsiSolverInterface::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.

7.31.3.58 virtual void OsiSolverInterface::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.

The default implementation adds the column, then changes the name. This can surely be made more efficient within an OsiXXX class.

7.31.3.59 virtual void OsiSolverInterface::addCols (const int numcols, const CoinPackedVectorBase *const * cols, const double * collb, const double * colub, const double * obj) [virtual]

Add a set of columns (primal variables) to the problem.

The default implementation simply makes repeated calls to [addCol\(\)](#).

Reimplemented in [OsiGlpkSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.60 `virtual void OsiSolverInterface::addCols (const int numcols, const int * columnStarts, const int * rows, const double * elements, const double * collb, const double * colub, const double * obj) [virtual]`

Add a set of columns (primal variables) to the problem.

The default implementation simply makes repeated calls to [addCol\(\)](#).

7.31.3.61 `int OsiSolverInterface::addCols (CoinModel & modelObject)`

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

7.31.3.62 `virtual void OsiSolverInterface::deleteCols (const int num, const int * colIndices) [pure virtual]`

Remove a set of columns (primal variables) from the problem.

The solver interface for a basis-oriented solver will maintain valid warm start information if all deleted variables are nonbasic.

Implemented in [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.63 `virtual void OsiSolverInterface::addRow (const CoinPackedVectorBase & vec, const double rowlb, const double rowub) [pure virtual]`

Add a row (constraint) to the problem.

Implemented in [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.64 `virtual void OsiSolverInterface::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.

7.31.3.65 `virtual void OsiSolverInterface::addRow (const CoinPackedVectorBase & vec, const char rowSen, const double rowrhs, const double rowrng) [pure virtual]`

Add a row (constraint) to the problem.

Implemented in [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.66 `virtual void OsiSolverInterface::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.

The default implementation adds the row, then changes the name. This can surely be made more efficient within an OsiXXX class.

```
7.31.3.67 virtual void OsiSolverInterface::addRow ( int numberElements, const int * columns, const double * element, const
double * rowlb, const double * rowub ) [virtual]
```

Add a row (constraint) to the problem.

Converts to addRow(CoinPackedVectorBase&,const double,const double).

```
7.31.3.68 virtual void OsiSolverInterface::addRows ( const int numrows, const CoinPackedVectorBase *const * rows, const
double * rowlb, const double * rowub ) [virtual]
```

Add a set of rows (constraints) to the problem.

The default implementation simply makes repeated calls to [addRow\(\)](#).

Reimplemented in [OsiGlpkSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

```
7.31.3.69 virtual void OsiSolverInterface::addRows ( const int numrows, const CoinPackedVectorBase *const * rows, const
char * rowSEN, const double * rowRHS, const double * rowRNG ) [virtual]
```

Add a set of rows (constraints) to the problem.

The default implementation simply makes repeated calls to [addRow\(\)](#).

Reimplemented in [OsiGlpkSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

```
7.31.3.70 virtual void OsiSolverInterface::addRows ( const int numrows, const int * rowStarts, const int * columns, const double *
element, const double * rowlb, const double * rowub ) [virtual]
```

Add a set of rows (constraints) to the problem.

The default implementation simply makes repeated calls to [addRow\(\)](#).

```
7.31.3.71 int OsiSolverInterface::addRows ( CoinModel & modelObject )
```

Add rows from a **CoinModel** object.

Returns -1 if the object is in the wrong state (*i.e.*, has column-major information), otherwise the number of errors.

The modelObject is not const as it can be regularized as part of the build.

```
7.31.3.72 virtual void OsiSolverInterface::deleteRows ( const int num, const int * rowIndices ) [pure virtual]
```

Delete a set of rows (constraints) from the problem.

The solver interface for a basis-oriented solver will maintain valid warm start information if all deleted rows are loose.

Implemented in [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.73 `virtual void OsiSolverInterface::replaceMatrixOptional (const CoinPackedMatrix &) [inline],[virtual]`

Replace the constraint matrix.

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.

Definition at line 1228 of file OsiSolverInterface.hpp.

7.31.3.74 `virtual void OsiSolverInterface::replaceMatrix (const CoinPackedMatrix &) [inline],[virtual]`

Replace the constraint matrix.

And if it does matter (not used at present)

Definition at line 1234 of file OsiSolverInterface.hpp.

7.31.3.75 `virtual void OsiSolverInterface::saveBaseModel () [inline],[virtual]`

Save a copy of the base model.

If solver wants it can save a copy of "base" (continuous) model here.

Definition at line 1240 of file OsiSolverInterface.hpp.

7.31.3.76 `virtual void OsiSolverInterface::restoreBaseModel (int numberOfRows) [virtual]`

Reduce the constraint system to the specified number of constraints.

If solver wants it can restore a copy of "base" (continuous) model here.

Note

The name is somewhat misleading. Implementors should consider the opportunity to optimise behaviour in the common case where `numberOfRows` is exactly the number of original constraints. Do not, however, neglect the possibility that `numberOfRows` does not equal the number of original constraints.

7.31.3.77 `virtual ApplyCutsReturnCode OsiSolverInterface::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()`

Reimplemented in [OsiGrbSolverInterface](#).

7.31.3.78 `virtual void OsiSolverInterface::applyRowCuts (int numberCuts, const OsiRowCut * cuts) [virtual]`

Apply a collection of row cuts which are all effective.

`applyCuts` seems to do one at a time which seems inefficient. Would be even more efficient to pass an array of pointers.

7.31.3.79 `virtual void OsiSolverInterface::applyRowCuts (int numberCuts, const OsiRowCut ** cuts) [virtual]`

Apply a collection of row cuts which are all effective.

This is passed in as an array of pointers.

7.31.3.80 `virtual void OsiSolverInterface::loadProblem (const CoinPackedMatrix & matrix, const double * collb, const double * colub, const double * obj, const double * rowlb, const double * rowub) [pure virtual]`

Load in a problem by copying the arguments.

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

Note that the default values for `rowub` and `rowlb` produce the constraint $-inf \leq ax \leq inf$. This is probably not what you want.

Implemented in [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiXprSolverInterface](#), and [OsiGrbSolverInterface](#).

7.31.3.81 `virtual void OsiSolverInterface::assignProblem (CoinPackedMatrix *& matrix, double *& collb, double *& colub, double *& obj, double *& rowlb, double *& rowub) [pure 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 argument values see the matching `loadProblem` method.

Warning

The arguments passed to this method will be freed using the C++ `delete` and `delete[]` functions.

Implemented in [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiXprSolverInterface](#), and [OsiGrbSolverInterface](#).

7.31.3.82 `virtual void OsiSolverInterface::loadProblem (const CoinPackedMatrix & matrix, const double * collb, const double * colub, const double * obj, const char * rowsen, const double * rowrhs, const double * rowrng) [pure 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 0 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

Note that the default values for `rowsen`, `rowrhs`, and `rowrng` produce the constraint $ax \geq 0$.

Implemented in [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiXprSolverInterface](#), and [OsiGrbSolverInterface](#).

7.31.3.83 `virtual void OsiSolverInterface::assignProblem (CoinPackedMatrix *& matrix, double *& collb, double *& colub, double *& obj, char *& rowsen, double *& rowrhs, double *& rowrng) [pure 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 argument values see the matching `loadProblem` method.

Warning

The arguments passed to this method will be freed using the C++ `delete` and `delete[]` functions.

Implemented in [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiXprSolverInterface](#), and [OsiGrbSolverInterface](#).

7.31.3.84 `virtual void OsiSolverInterface::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) [pure virtual]`

Load in a problem by copying the arguments.

The constraint matrix is specified with standard column-major column starts / row indices / coefficients vectors. The constraints on the rows are given by lower and upper bounds.

The matrix vectors must be gap-free. Note that `start` must have `numcols+1` entries so that the length of the last column can be calculated as `start[numcols]-start[numcols-1]`.

See the previous `loadProblem` method using `rowlb` and `rowub` for default argument values.

```
7.31.3.85 virtual void OsiSolverInterface::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 ) [pure virtual]
```

Load in a problem by copying the arguments.

The constraint matrix is specified with standard column-major column starts / row indices / coefficients vectors. The constraints on the rows are given by sense/rhs/range triplets.

The matrix vectors must be gap-free. Note that *start* must have *numcols*+1 entries so that the length of the last column can be calculated as *start*[*numcols*]-*start*[*numcols*-1].

See the previous *loadProblem* method using sense/rhs/range for default argument values.

```
7.31.3.86 virtual int OsiSolverInterface::loadFromCoinModel ( CoinModel & modelObject, bool keepSolution = false )
[virtual]
```

Load a model from a **CoinModel** object.

Return the number of errors encountered.

The *modelObject* parameter cannot be const as it may be changed as part of process. If *keepSolution* is true will try and keep *warmStart*.

```
7.31.3.87 virtual int OsiSolverInterface::readMps ( const char * filename, const char * extension = "mps" ) [virtual]
```

Read a problem in MPS format from the given filename.

The default implementation uses **CoinMpsIO::readMps()** to read the MPS file and returns the number of errors encountered.

Reimplemented in [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiXprSolverInterface](#), and [OsiGrbSolverInterface](#).

```
7.31.3.88 virtual int OsiSolverInterface::readMps ( const char * filename, const char * extension, int & numberSets, CoinSet **&
sets ) [virtual]
```

Read a problem in MPS format from the given full filename.

This uses **CoinMpsIO::readMps()** to read the MPS file and returns the number of errors encountered. It also may return an array of set information

```
7.31.3.89 virtual int OsiSolverInterface::readGMPL ( const char * filename, const char * dataname = NULL ) [virtual]
```

Read a problem in GMPL format from the given filenames.

The default implementation uses **CoinMpsIO::readGMPL()**. This capability is available only if the third-party package Glpk is installed.

```
7.31.3.90 virtual void OsiSolverInterface::writeMps ( const char * filename, const char * extension = "mps", double objSense =
0.0 ) const [pure virtual]
```

Write the problem in MPS format to the specified file.

If *objSense* is non-zero, a value of -1.0 causes the problem to be written with a maximization objective; +1.0 forces a minimization objective. If *objSense* is zero, the choice is left to the implementation.

Implemented in [OsiGlpkSolverInterface](#), [OsiSpxSolverInterface](#), [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), [OsiXprSolverInterface](#), and [OsiGrbSolverInterface](#).

```
7.31.3.91 int OsiSolverInterface::writeMpsNative ( const char * filename, const char ** rowNames, const char ** columnNames,
int formatType = 0, int numberAcross = 2, double objSense = 0.0, int numberSOS = 0, const CoinSet * setInfo =
NULL ) const
```

Write the problem in MPS format to the specified file with more control over the output.

Row and column names may be null. formatType is

- 0 - normal
- 1 - extra accuracy
- 2 - IEEE hex

Returns non-zero on I/O error

```
7.31.3.92 virtual void OsiSolverInterface::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 with the specified extension.

Coefficients with value less than epsilon away from an integer value are written as integers. Write at most numberAcross monomials on a line. Write non integer numbers with decimals digits after the decimal point.

The written problem is always a minimization problem. If the current problem is a maximization problem, the intended objective function for the written problem is the current objective function multiplied by -1. If the current problem is a minimization problem, the intended objective function for the written problem is the current objective function. If objSense < 0, the intended objective function is multiplied by -1 before writing the problem. It is left unchanged otherwise.

Write objective function name and constraint names if useRowNames is true. This version calls [writeLpNative\(\)](#).

```
7.31.3.93 virtual void OsiSolverInterface::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.

```
7.31.3.94 int OsiSolverInterface::writeLpNative ( const char * filename, char const *const *const rowNames, char const *const
*const columnNames, const double epsilon = 1.0e-5, const int numberAcross = 10, const int decimals = 5, const
double objSense = 0.0, const bool useRowNames = true ) const
```

Write the problem into an Lp file.

Parameters are similar to those of [writeLp\(\)](#), but in addition row names and column names may be given.

Parameter rowNames may be NULL, in which case default row names are used. If rowNames is not NULL, it must have exactly one entry per row in the problem and one additional entry (rowNames[getNumRows()]) with the objective function name. These getNumRows()+1 entries must be distinct. If this is not the case, default row names are used. In addition, format restrictions are imposed on names (see [CoinLpIO::is_invalid_name\(\)](#) for details).

Similar remarks can be made for the parameter `columnNames` which must either be NULL or have exactly `getNumCols()` distinct entries.

Write objective function name and constraint names if `useRowNames` is true.

```
7.31.3.95  int OsiSolverInterface::writeLpNative ( FILE * fp, char const *const *const rowNames, char const *const *const
          columnNames, const double epsilon = 1.0e-5, const int numberAcross = 10, const int decimals = 5, const double
          objSense = 0.0, const bool useRowNames = true ) const
```

Write the problem into the file pointed to by the parameter `fp`.

Other parameters are similar to those of `writeLpNative()` with first parameter filename.

```
7.31.3.96  virtual int OsiSolverInterface::readLp ( const char * filename, const double epsilon = 1e-5 ) [virtual]
```

Read file in LP format from file with name `filename`.

See class **CoinLpIO** for description of this format.

```
7.31.3.97  int OsiSolverInterface::readLp ( FILE * fp, const double epsilon = 1e-5 )
```

Read file in LP format from the file pointed to by `fp`.

See class **CoinLpIO** for description of this format.

```
7.31.3.98  int OsiSolverInterface::differentModel ( OsiSolverInterface & other, bool ignoreNames = true )
```

Check two models against each other.

Return nonzero if different. Ignore names if that set. (Note initial version does not check names) May modify both models by cleaning up

```
7.31.3.99  void OsiSolverInterface::setApplicationData ( void * appData )
```

Set application data.

This is a pointer that the application can store into and retrieve from the solver interface. This field is available for the application to optionally define and use.

```
7.31.3.100 void OsiSolverInterface::setAuxiliaryInfo ( OsiAuxInfo * auxiliaryInfo )
```

Create a clone of an Auxiliary Information object.

The base class just stores an application data pointer but can be more general. Application data pointer is designed for one user while this can be extended to cope with more general extensions.

```
7.31.3.101 virtual void OsiSolverInterface::passInMessageHandler ( CoinMessageHandler * handler ) [virtual]
```

Pass in a message handler.

It is the client's responsibility to destroy a message handler installed by this routine; it will not be destroyed when the solver interface is destroyed.

Reimplemented in [OsiCpxSolverInterface](#), [OsiMskSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.102 void OsiSolverInterface::findIntegers (bool *justCount*)

Identify integer variables and create corresponding objects.

Record integer variables and create an [OsiSimpleInteger](#) object for each one. All existing [OsiSimpleInteger](#) objects will be destroyed. If *justCount* then no objects created and we just store `numberIntegers_`

7.31.3.103 virtual int OsiSolverInterface::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

7.31.3.104 void OsiSolverInterface::addObjects (int *numberOfObjects*, OsiObject ** *objects*)

Add in object information.

Objects are cloned; the owner can delete the originals.

7.31.3.105 double OsiSolverInterface::forceFeasible ()

Use current solution to set bounds so current integer feasible solution will stay feasible.

Only feasible bounds will be used, even if current solution outside bounds. The amount of such violation will be returned (and if small can be ignored)

7.31.3.106 virtual void OsiSolverInterface::activateRowCutDebugger (const char * *modelName*) [virtual]

Activate the row cut debugger.

If *modelName* is in the set of known models then all cuts are checked to see that they do NOT cut off the optimal solution known to the debugger.

7.31.3.107 virtual void OsiSolverInterface::activateRowCutDebugger (const double * *solution*, bool *enforceOptimality* = true) [virtual]

Activate the row cut debugger using a full solution array.

Activate the debugger for a model not included in the debugger's internal database. Cuts will be checked to see that they do NOT cut off the given solution.

solution must be a full solution vector, but only the integer variables need to be correct. The debugger will fill in the continuous variables by solving an lp relaxation with the integer variables fixed as specified. If the given values for the continuous variables should be preserved, set `keepContinuous` to true.

7.31.3.108 const OsiRowCutDebugger* OsiSolverInterface::getRowCutDebugger () const

Get the row cut debugger provided the solution known to the debugger is within the feasible region held in the solver.

If there is a row cut debugger object associated with model AND if the solution known to the debugger is within the solver's current feasible region (i.e., the column bounds held in the solver are compatible with the known solution) then a pointer to the debugger is returned which may be used to test validity of cuts.

Otherwise NULL is returned

7.31.3.109 `OsiRowCutDebugger*` `OsiSolverInterface::getRowCutDebuggerAlways () const`

Get the row cut debugger object.

Return the row cut debugger object if it exists. One common usage of this method is to obtain a debugger object in order to execute `OsiRowCutDebugger::redoSolution` (so that the stored solution is again compatible with the problem held in the solver).

7.31.3.110 `virtual int` `OsiSolverInterface::canDoSimplexInterface () const` `[virtual]`

Return the simplex implementation level.

The return codes are:

- 0: the simplex interface is not implemented.
- 1: the Group 1 (tableau access) methods are implemented.
- 2: the Group 2 (pivoting) methods are implemented

The codes are cumulative - a solver which implements Group 2 also implements Group 1.

Reimplemented in `OsiCpxSolverInterface`, and `OsiGrbSolverInterface`.

7.31.3.111 `virtual void` `OsiSolverInterface::enableFactorization () const` `[virtual]`

Prepare the solver for the use of tableau access methods.

Prepares the solver for the use of the tableau access methods, if any such preparation is required.

The `const` attribute is required due to the places this method may be called (e.g., within `CglCutGenerator::generateCuts()`).

Reimplemented in `OsiCpxSolverInterface`, and `OsiGrbSolverInterface`.

7.31.3.112 `virtual void` `OsiSolverInterface::disableFactorization () const` `[virtual]`

Undo the effects of `enableFactorization`.

Reimplemented in `OsiCpxSolverInterface`, and `OsiGrbSolverInterface`.

7.31.3.113 `virtual bool` `OsiSolverInterface::basisIsAvailable () const` `[virtual]`

Check if an optimal basis is available.

Returns true if the problem has been solved to optimality and a basis is available. This should be used to see if the tableau access operations are possible and meaningful.

Note

Implementors please note that this method may be called before [enableFactorization](#).

Reimplemented in [OsiCpxSolverInterface](#), and [OsiGrbSolverInterface](#).

7.31.3.114 `virtual void OsiSolverInterface::getBasisStatus (int * cstat, int * rstat) const` `[virtual]`

Retrieve status information for column and row variables.

This method returns status as integer codes:

- 0: free
- 1: basic
- 2: nonbasic at upper bound
- 3: nonbasic at lower bound

The [getWarmStart](#) method provides essentially the same functionality for a simplex-oriented solver, but the implementation details are very different.

Note

Logical variables associated with rows are all assumed to have +1 coefficients, so for a \leq constraint the logical will be at lower bound if the constraint is tight.

Implementors may choose to implement this method as a wrapper which converts a **CoinWarmStartBasis** to the requested representation.

Reimplemented in [OsiCpxSolverInterface](#), and [OsiGrbSolverInterface](#).

7.31.3.115 `virtual int OsiSolverInterface::setBasisStatus (const int * cstat, const int * rstat)` `[virtual]`

Set the status of column and row variables and update the basis factorization and solution.

Status information should be coded as documented for [getBasisStatus](#). Returns 0 if all goes well, 1 if something goes wrong.

This method differs from [setWarmStart](#) in the format of the input and in its immediate effect. Think of it as [setWarmStart](#) immediately followed by [resolve](#), but no pivots are allowed.

Note

Implementors may choose to implement this method as a wrapper that calls [setWarmStart](#) and [resolve](#) if the no pivot requirement can be satisfied.

7.31.3.116 `virtual void OsiSolverInterface::getReducedGradient (double * columnReducedCosts, double * duals, const double * c) const` `[virtual]`

Calculate duals and reduced costs for the given objective coefficients.

The solver's objective coefficient vector is not changed.

7.31.3.117 `virtual void OsiSolverInterface::getBlvARow (int row, double * z, double * slack = NULL) const [virtual]`

Get a row of the tableau.

If `slack` is not null, it will be loaded with the coefficients for the artificial (logical) variables (i.e., the row of the basis inverse).

Reimplemented in [OsiCpxSolverInterface](#).

7.31.3.118 `virtual void OsiSolverInterface::getBasics (int * index) const [virtual]`

Get indices of basic variables.

If the logical (artificial) for row `i` is basic, the index should be coded as (`getNumCols` + `i`). The order of indices must match the order of elements in the vectors returned by `getBlvACol` and `getBlvCol`.

Reimplemented in [OsiCpxSolverInterface](#).

7.31.3.119 `virtual void OsiSolverInterface::enableSimplexInterface (bool doingPrimal) [virtual]`

Enables normal operation of subsequent functions.

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. says whether will be doing primal or dual

7.31.3.120 `virtual int OsiSolverInterface::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 `outStatus`. Where 1 is to upper bound, -1 to lower bound Return code was undefined - now for OsiClp is 0 for okay, 1 if inaccuracy forced re-factorization (should be okay) and -1 for singular factorization

7.31.3.121 `virtual int OsiSolverInterface::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

7.31.3.122 `virtual OsiSolverInterface* OsiSolverInterface::clone (bool copyData =true) const [pure virtual]`

Clone.

The result of calling `clone(false)` is defined to be equivalent to calling the default constructor `OsiSolverInterface()`.

Implemented in [OsiGlpkSolverInterface](#), [OsiMskSolverInterface](#), [OsiCpxSolverInterface](#), [OsiXprSolverInterface](#), [OsiGrbSolverInterface](#), and [OsiSpxSolverInterface](#).

7.31.3.123 `virtual void OsiSolverInterface::reset () [virtual]`

Reset the solver interface.

A call to [reset\(\)](#) returns the solver interface to the same state as it would have if it had just been constructed by calling the default constructor [OsiSolverInterface\(\)](#).

Reimplemented in [OsiGlpkSolverInterface](#), [OsiCpxSolverInterface](#), and [OsiGrbSolverInterface](#).

7.31.3.124 `virtual void OsiSolverInterface::applyRowCut (const OsiRowCut & rc) [protected],[pure virtual]`

Apply a row cut (append to the constraint matrix).

Implemented in [OsiCpxSolverInterface](#), [OsiGrbSolverInterface](#), [OsiGlpkSolverInterface](#), [OsiMskSolverInterface](#), [OsiSpxSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.125 `virtual void OsiSolverInterface::applyColCut (const OsiColCut & cc) [protected],[pure virtual]`

Apply a column cut (adjust the bounds of one or more variables).

Implemented in [OsiCpxSolverInterface](#), [OsiGrbSolverInterface](#), [OsiGlpkSolverInterface](#), [OsiMskSolverInterface](#), [OsiSpxSolverInterface](#), and [OsiXprSolverInterface](#).

7.31.3.126 `void OsiSolverInterface::setInitialData () [protected]`

Set [OsiSolverInterface](#) object state for default constructor.

This routine establishes the initial values of data fields in the [OsiSolverInterface](#) object when the object is created using the default constructor.

7.31.4 Friends And Related Function Documentation

7.31.4.1 `void OsiSolverInterfaceCommonUnitTest (const OsiSolverInterface * emptySi, const std::string & mpsDir, const std::string & netlibDir) [friend]`

A function that tests the methods in the [OsiSolverInterface](#) class.

Some time ago, if this method is compiled with optimization, the compilation took 10-15 minutes and the machine pages (has 256M core memory!)...

7.31.4.2 `void OsiSolverInterfaceMpsUnitTest (const std::vector< OsiSolverInterface * > & vecSiP, const std::string & mpsDir) [friend]`

A function that tests that a lot of problems given in MPS files (mostly the NETLIB problems) solve properly with all the specified solvers.

The routine creates a vector of NetLib problems (problem name, objective, various other characteristics), and a vector of solvers to be tested.

Each solver is run on each problem. The run is deemed successful if the solver reports the correct problem size after loading and returns the correct objective value after optimization.

If multiple solvers are available, the results are compared pairwise against the results reported by adjacent solvers in the solver vector. Due to limitations of the volume solver, it must be the last solver in vecEmptySiP.

7.31.5 Member Data Documentation

7.31.5.1 `OsiRowCutDebugger*` `OsiSolverInterface::rowCutDebugger_` `[mutable]`, `[protected]`

Pointer to row cut debugger object.

Mutable so that we can update the solution held in the debugger while maintaining const'ness for the Osi object.

Definition at line 2016 of file `OsiSolverInterface.hpp`.

7.31.5.2 `bool` `OsiSolverInterface::defaultHandler_` `[protected]`

Flag to say if the current handler is the default handler.

Indicates if the solver interface object is responsible for destruction of the handler (true) or if the client is responsible (false).

Definition at line 2025 of file `OsiSolverInterface.hpp`.

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

- `OsiSolverInterface.hpp`

7.32 OsiSolverResult Class Reference

Solver Result Class.

```
#include <OsiSolverBranch.hpp>
```

Public Member Functions

Add and Get methods

- void `createResult` (const `OsiSolverInterface` &solver, const double *lowerBefore, const double *upperBefore)
Create result.
- void `restoreResult` (`OsiSolverInterface` &solver) const
Restore result.
- const `CoinWarmStartBasis` & `basis` () const
Get basis.
- double `objectiveValue` () const
Objective value (as minimization)
- const double * `primalSolution` () const
Primal solution.
- const double * `dualSolution` () const
Dual solution.
- const `OsiSolverBranch` & `fixed` () const
Extra fixed.

Constructors and destructors

- `OsiSolverResult` ()
Default Constructor.
- `OsiSolverResult` (const `OsiSolverInterface` &solver, const double *lowerBefore, const double *upperBefore)

- Constructor from solver.*
- `OsiSolverResult` (const `OsiSolverResult` &rhs)
- Copy constructor.*
- `OsiSolverResult` & `operator=` (const `OsiSolverResult` &rhs)
- Assignment operator.*
- `~OsiSolverResult` ()
- Destructor.*

7.32.1 Detailed Description

Solver Result Class.

This provides information on a result as a set of tighter bounds on both ways

Definition at line 83 of file `OsiSolverBranch.hpp`.

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

- `OsiSolverBranch.hpp`

7.33 OsiSOS Class Reference

Define Special Ordered Sets of type 1 and 2.

```
#include <OsiBranchingObject.hpp>
```

Inheritance diagram for OsiSOS:

Collaboration diagram for OsiSOS:

Public Member Functions

- `OsiSOS` (const `OsiSolverInterface` *solver, int `numberMembers`, const int *which, const double *weights, int type=1)
- Useful constructor - which are indices and weights are also given.*
- virtual `OsiObject` * `clone` () const
- Clone.*
- virtual double `infeasibility` (const `OsiBranchingInformation` *info, int &whichWay) const
- Infeasibility - large is 0.5.*
- virtual double `feasibleRegion` (`OsiSolverInterface` *solver, const `OsiBranchingInformation` *info) const
- Set bounds to fix the variable at the current (integer) value.*
- virtual `OsiBranchingObject` * `createBranch` (`OsiSolverInterface` *solver, const `OsiBranchingInformation` *info, int way) const
- Creates a branching object.*
- virtual double `upEstimate` () const
- Return "up" estimate (default 1.0e-5)*
- virtual double `downEstimate` () const
- Return "down" estimate (default 1.0e-5)*
- virtual void `resetSequenceEtc` (int numberColumns, const int *originalColumns)
- Redoes data when sequence numbers change.*
- int `numberMembers` () const

- *Number of members.*
 • const int * `members` () const
 Members (indices in range 0 ... numberColumns-1)
- int `sosType` () const
 SOS type.
- int `setType` () const
 SOS type.
- const double * `weights` () const
 Array of weights.
- virtual bool `canDoHeuristics` () const
 Return true if object can take part in normal heuristics.
- void `setIntegerValued` (bool yesNo)
 Set whether set is integer valued or not.
- virtual bool `canHandleShadowPrices` () const
 Return true if knows how to deal with Pseudo Shadow Prices.
- void `setNumberMembers` (int value)
 Set number of members.
- int * `mutableMembers` () const
 Members (indices in range 0 ... numberColumns-1)
- void `setSosType` (int value)
 Set SOS type.
- double * `mutableWeights` () const
 Array of weights.

Protected Attributes

- int * `members_`
 data
- double * `weights_`
 Weights.
- int `numberMembers_`
 Number of members.
- int `sosType_`
 SOS type.
- bool `integerValued_`
 Whether integer valued.

7.33.1 Detailed Description

Define Special Ordered Sets of type 1 and 2.

These do not have to be integer - so do not appear in lists of integers.

which_ points columns of matrix

Definition at line 674 of file OsiBranchingObject.hpp.

7.33.2 Constructor & Destructor Documentation

7.33.2.1 `OsiSOS::OsiSOS (const OsiSolverInterface * solver, int numberMembers, const int * which, const double * weights, int type = 1)`

Useful constructor - which are indices and weights are also given.

If null then 0,1,2.. type is SOS type

7.33.3 Member Function Documentation

7.33.3.1 `virtual double OsiSOS::feasibleRegion (OsiSolverInterface * solver, const OsiBranchingInformation * info) const [virtual]`

Set bounds to fix the variable at the current (integer) value.

Given an integer value, set the lower and upper bounds to fix the variable. Returns amount it had to move variable.

Implements [OsiObject](#).

7.33.3.2 `virtual OsiBranchingObject* OsiSOS::createBranch (OsiSolverInterface * solver, const OsiBranchingInformation * info, int way) const [virtual]`

Creates a branching object.

The preferred direction is set by `way`, 0 for down, 1 for up.

Reimplemented from [OsiObject](#).

7.33.4 Member Data Documentation

7.33.4.1 `int* OsiSOS::members_ [protected]`

data

Members (indices in range 0 ... numberColumns-1)

Definition at line 774 of file `OsiBranchingObject.hpp`.

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

- `OsiBranchingObject.hpp`

7.34 OsiSOSBranchingObject Class Reference

Branching object for Special ordered sets.

`#include <OsiBranchingObject.hpp>`

Inheritance diagram for `OsiSOSBranchingObject`:

Collaboration diagram for `OsiSOSBranchingObject`:

Public Member Functions

- virtual [OsiBranchingObject](#) * [clone](#) () const
Clone.
- virtual double [branch](#) ([OsiSolverInterface](#) *solver)
Does next branch and updates state.
- virtual void [print](#) (const [OsiSolverInterface](#) *solver=NULL)
Print something about branch - only if log level high.

Additional Inherited Members

7.34.1 Detailed Description

Branching object for Special ordered sets.

Definition at line 789 of file [OsiBranchingObject.hpp](#).

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

- [OsiBranchingObject.hpp](#)

7.35 OsiSpxSolverInterface Class Reference

SoPlex Solver Interface Instantiation of [OsiSpxSolverInterface](#) for SoPlex.

```
#include <OsiSpxSolverInterface.hpp>
```

Inheritance diagram for [OsiSpxSolverInterface](#):

Collaboration diagram for [OsiSpxSolverInterface](#):

Public Types

- enum [keepCachedFlag](#) {
[KEEPCACHED_NONE](#) = 0, [KEEPCACHED_COLUMN](#) = 1, [KEEPCACHED_ROW](#) = 2, [KEEPCACHED_MATRIX](#) = 4,
[KEEPCACHED_RESULTS](#) = 8, [KEEPCACHED_PROBLEM](#) = [KEEPCACHED_COLUMN](#) | [KEEPCACHED_ROW](#) | [KEEPCACHED_MATRIX](#), [KEEPCACHED_ALL](#) = [KEEPCACHED_PROBLEM](#) | [KEEPCACHED_RESULTS](#),
[FREECACHED_COLUMN](#) = [KEEPCACHED_PROBLEM](#) & ~[KEEPCACHED_COLUMN](#),
[FREECACHED_ROW](#) = [KEEPCACHED_PROBLEM](#) & ~[KEEPCACHED_ROW](#), [FREECACHED_MATRIX](#) = [KEEPCACHED_PROBLEM](#) & ~[KEEPCACHED_MATRIX](#), [FREECACHED_RESULTS](#) = [KEEPCACHED_ALL](#) & ~[KEEPCACHED_RESULTS](#) }

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 `branchAndBound` ()
Invoke solver's built-in enumeration algorithm.

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.

- bool `setIntParam` (OsiIntParam key, int value)
Set an integer parameter.
- bool `setDbfParam` (OsiDbfParam key, double value)
Set a double parameter.
- bool `getIntParam` (OsiIntParam key, int &value) const
Get an integer parameter.
- bool `getDbfParam` (OsiDbfParam key, double &value) const
Get a double parameter.
- bool `getStrParam` (OsiStrParam key, std::string &value) const
Get a string parameter.
- void `setTimeLimit` (double value)
- double `getTimeLimit` () const

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 `isDualObjectiveLimitReached` () const
Is the given dual objective limit reached?
- virtual bool `isIterationLimitReached` () const
Iteration limit reached?
- virtual bool `isTimeLimitReached` () const
Time limit reached?

WarmStart related methods

- `CoinWarmStart` * `getEmptyWarmStart` () const
Get empty warm start object.

- virtual **CoinWarmStart** * [getWarmStart](#) () const
Get warmstarting information.
- virtual bool [setWarmStart](#) (const **CoinWarmStart** *warmstart)
Set warmstarting information.

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.

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 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 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 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 -COIN_DBL_MAX for -infinity.*
- virtual void [setColUpper](#) (int elementIndex, double elementValue)
*Set a single column upper bound
Use COIN_DBL_MAX for infinity.*
- virtual void [setColBounds](#) (int elementIndex, double lower, double upper)
*Set a single column lower and upper bound
The default implementation just invokes setColLower and setColUpper*
- virtual void [setRowLower](#) (int elementIndex, double elementValue)
*Set a single row lower bound
Use -COIN_DBL_MAX for -infinity.*
- virtual void [setRowUpper](#) (int elementIndex, double elementValue)
*Set a single row upper bound
Use COIN_DBL_MAX for infinity.*
- virtual void [setRowBounds](#) (int elementIndex, double lower, double upper)
*Set a single row lower and upper bound
The default implementation just invokes setRowLower and setRowUpper*
- virtual void [setRowType](#) (int index, char sense, double rightHandSide, double range)
Set the type of a single row

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.

**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)
Add a column (primal variable) to the problem.
- virtual void [deleteCols](#) (const int num, const int *colIndices)
Remove a set of columns (primal variables) from the problem.
- virtual void [addRow](#) (const **CoinPackedVectorBase** &vec, 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)
Add a row (constraint) to the problem.
- virtual void [deleteRows](#) (const int num, const int *rowIndices)
Delete a set of rows (constraints) from the problem.

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 int numcols, const int numRows, const int *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 int *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 [readMps](#) (const char *filename, const char *extension="mps")
Read an mps file from the given filename.
- 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.

Constructors and destructor

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

Protected Member Functions

Protected methods

- virtual void [applyRowCut](#) (const [OsiRowCut](#) &rc)
Apply a row cut. Return true if cut was applied.
- virtual void [applyColCut](#) (const [OsiColCut](#) &cc)
Apply a column cut (bound adjustment).

Protected Attributes

Protected member data

- [soplex::SPxOut](#) * [spxout_](#)
SoPlex output object.
- [soplex::SoPlex](#) * [soplex_](#)
SoPlex solver object.

Friends

- void [OsiSpxSolverInterfaceUnitTest](#) (const std::string &mpsDir, const std::string &netlibDir)
A function that tests the methods in the [OsiSpxSolverInterface](#) class.

7.35.1 Detailed Description

SoPlex Solver Interface Instantiation of [OsiSpxSolverInterface](#) for SoPlex.

Definition at line 37 of file [OsiSpxSolverInterface.hpp](#).

7.35.2 Member Enumeration Documentation

7.35.2.1 enum [OsiSpxSolverInterface::keepCachedFlag](#)

Enumerator

- KEEPCACHED_NONE** discard all cached data (default)
- KEEPCACHED_COLUMN** column information: objective values, lower and upper bounds, variable types
- KEEPCACHED_ROW** row information: right hand sides, ranges and senses, lower and upper bounds for row
- KEEPCACHED_MATRIX** problem matrix: matrix ordered by column and by row
- KEEPCACHED_RESULTS** LP solution: primal and dual solution, reduced costs, row activities.
- KEEPCACHED_PROBLEM** only discard cached LP solution
- KEEPCACHED_ALL** keep all cached data (similar to [getMutableLpPtr\(\)](#))
- FREECACHED_COLUMN** free only cached column and LP solution information
- FREECACHED_ROW** free only cached row and LP solution information
- FREECACHED_MATRIX** free only cached matrix and LP solution information
- FREECACHED_RESULTS** free only cached LP solution information

Definition at line 618 of file [OsiSpxSolverInterface.hpp](#).

7.35.3 Member Function Documentation

7.35.3.1 `virtual bool OsiSpxSolverInterface::setWarmStart (const CoinWarmStart * warmstart)` `[virtual]`

Set warmstarting information.

Return true/false depending on whether the warmstart information was accepted or not.

Implements [OsiSolverInterface](#).

7.35.3.2 `virtual const char* OsiSpxSolverInterface::getRowSense () const` `[virtual]`

Get pointer to array[[getNumRows\(\)](#)] of row constraint senses.

- 'L': \leq constraint
- 'E': = constraint
- 'G': \geq constraint
- 'R': ranged constraint
- 'N': free constraint

Implements [OsiSolverInterface](#).

7.35.3.3 `virtual const double* OsiSpxSolverInterface::getRightHandSide () const` `[virtual]`

Get pointer to array[[getNumRows\(\)](#)] of rows right-hand sides.

- if `rowsense()[i] == 'L'` then `rhs()[i] == rowupper()[i]`
- if `rowsense()[i] == 'G'` then `rhs()[i] == rowlower()[i]`
- if `rowsense()[i] == 'R'` then `rhs()[i] == rowupper()[i]`
- if `rowsense()[i] == 'N'` then `rhs()[i] == 0.0`

Implements [OsiSolverInterface](#).

7.35.3.4 `virtual const double* OsiSpxSolverInterface::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 0.0

Implements [OsiSolverInterface](#).

7.35.3.5 `virtual int OsiSpxSolverInterface::getIterationCount () const` `[virtual]`

Get how many iterations it took to solve the problem (whatever "iteration" mean to the solver).

Implements [OsiSolverInterface](#).

7.35.3.6 `virtual std::vector<double*> OsiSpxSolverInterface::getDualRays (int maxNumRays, bool fullRay = false) const` [virtual]

Get as many dual rays as the solver can provide.

(In case of proven primal infeasibility there should be at least one.)

The first `getNumRows()` ray components will always be associated with the row duals (as returned by `getRowPrice()`). If `fullRay` is true, the final `getNumCols()` entries will correspond to the ray components associated with the nonbasic variables. If the full ray is requested and the method cannot provide it, it will throw an exception.

NOTE for implementers of solver interfaces:

The double pointers in the vector should point to arrays of length `getNumRows()` and they should be allocated via `new[]`.

NOTE for users of solver interfaces:

It is the user's responsibility to free the double pointers in the vector using `delete[]`.

Implements [OsiSolverInterface](#).

7.35.3.7 `virtual std::vector<double*> OsiSpxSolverInterface::getPrimalRays (int maxNumRays) const` [virtual]

Get as many primal rays as the solver can provide.

(In case of proven dual infeasibility there should be at least one.)

NOTE for implementers of solver interfaces:

The double pointers in the vector should point to arrays of length `getNumCols()` and they should be allocated via `new[]`.

NOTE for users of solver interfaces:

It is the user's responsibility to free the double pointers in the vector using `delete[]`.

Implements [OsiSolverInterface](#).

7.35.3.8 `virtual void OsiSpxSolverInterface::setColLower (int elementIndex, double elementValue)` [virtual]

Set a single column lower bound
Use `-COIN_DBL_MAX` for -infinity.

Implements [OsiSolverInterface](#).

7.35.3.9 `virtual void OsiSpxSolverInterface::setColUpper (int elementIndex, double elementValue)` [virtual]

Set a single column upper bound
Use `COIN_DBL_MAX` for infinity.

Implements [OsiSolverInterface](#).

7.35.3.10 `virtual void OsiSpxSolverInterface::setRowLower (int elementIndex, double elementValue)` [virtual]

Set a single row lower bound
Use `-COIN_DBL_MAX` for -infinity.

Implements [OsiSolverInterface](#).

7.35.3.11 `virtual void OsiSpxSolverInterface::setRowUpper (int elementIndex, double elementValue)` [virtual]

Set a single row upper bound
Use COIN_DBL_MAX for infinity.

Implements [OsiSolverInterface](#).

7.35.3.12 `virtual void OsiSpxSolverInterface::setColSolution (const double * colsol)` [virtual]

Set the primal solution column values.

`colsol[numcols()]` is an array of values of the problem column variables. These values are copied to memory owned by the solver object or the solver. They will be returned as the result of `colsol()` until changed by another call to `setColsol()` or by a call to any solver routine. Whether the solver makes use of the solution in any way is solver-dependent.

Implements [OsiSolverInterface](#).

7.35.3.13 `virtual void OsiSpxSolverInterface::setRowPrice (const double * rowprice)` [virtual]

Set dual solution vector.

`rowprice[numrows()]` is an array of values of the problem row dual variables. These values are copied to memory owned by the solver object or the solver. They will be returned as the result of `rowprice()` until changed by another call to `setRowprice()` or by a call to any solver routine. Whether the solver makes use of the solution in any way is solver-dependent.

Implements [OsiSolverInterface](#).

7.35.3.14 `virtual void OsiSpxSolverInterface::addCol (const CoinPackedVectorBase & vec, const double collb, const double colub, const double obj)` [virtual]

Add a column (primal variable) to the problem.

Implements [OsiSolverInterface](#).

7.35.3.15 `virtual void OsiSpxSolverInterface::deleteCols (const int num, const int * colIndices)` [virtual]

Remove a set of columns (primal variables) from the problem.

The solver interface for a basis-oriented solver will maintain valid warm start information if all deleted variables are nonbasic.

Implements [OsiSolverInterface](#).

7.35.3.16 `virtual void OsiSpxSolverInterface::addRow (const CoinPackedVectorBase & vec, const double rowlb, const double rowub)` [virtual]

Add a row (constraint) to the problem.

Implements [OsiSolverInterface](#).

7.35.3.17 `virtual void OsiSpxSolverInterface::addRow (const CoinPackedVectorBase & vec, const char rowsen, const double rowrhs, const double rowrng) [virtual]`

Add a row (constraint) to the problem.

Implements [OsiSolverInterface](#).

7.35.3.18 `virtual void OsiSpxSolverInterface::deleteRows (const int num, const int * rowIndices) [virtual]`

Delete a set of rows (constraints) from the problem.

The solver interface for a basis-oriented solver will maintain valid warm start information if all deleted rows are loose.

Implements [OsiSolverInterface](#).

7.35.3.19 `virtual void OsiSpxSolverInterface::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 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

Implements [OsiSolverInterface](#).

7.35.3.20 `virtual void OsiSpxSolverInterface::assignProblem (CoinPackedMatrix * & matrix, double * & collb, double * & colub, double * & obj, double * & rowlb, double * & rowub) [virtual]`

Load in an problem by assuming ownership of the arguments (the constraints on the rows are given by lower and upper bounds).

For default values see the previous method.

WARNING: The arguments passed to this method will be freed using the C++ `delete` and `delete[]` functions.

Implements [OsiSolverInterface](#).

7.35.3.21 `virtual void OsiSpxSolverInterface::loadProblem (const CoinPackedMatrix & matrix, const double * collb, const double * colub, const double * obj, const char * rowsen, const double * rowrhs, const double * rowrng) [virtual]`

Load in an problem by copying the arguments (the constraints on the rows are given by sense/rhs/range triplets).

If a pointer is 0 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

Implements [OsiSolverInterface](#).

```
7.35.3.22 virtual void OsiSpxSolverInterface::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.

Implements [OsiSolverInterface](#).

```
7.35.3.23 virtual void OsiSpxSolverInterface::loadProblem ( const int numcols, const int numRows, const int * 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).

```
7.35.3.24 virtual void OsiSpxSolverInterface::loadProblem ( const int numcols, const int numRows, const int * 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).

```
7.35.3.25 virtual void OsiSpxSolverInterface::writeMps ( const char * filename, const char * extension = "mps", double objSense
=0.0 ) const [virtual]
```

Write the problem into an mps file of the given filename.

If `objSense` is non zero then -1.0 forces the code to write a maximization objective and +1.0 to write a minimization one. If 0.0 then solver can do what it wants

Implements [OsiSolverInterface](#).

```
7.35.3.26 virtual void OsiSpxSolverInterface::applyColCut ( const OsiColCut & cc ) [protected],[virtual]
```

Apply a column cut (bound adjustment).

Return true if cut was applied.

Implements [OsiSolverInterface](#).

7.35.4 Friends And Related Function Documentation

7.35.4.1 `void OsiSpxSolverInterfaceUnitTest (const std::string & mpsDir, const std::string & netlibDir) [friend]`

A function that tests the methods in the [OsiSpxSolverInterface](#) class.

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

- `OsiSpxSolverInterface.hpp`

7.36 OsiTwoWayBranchingObject Class Reference

This just adds two-wayness to a branching object.

```
#include <OsiBranchingObject.hpp>
```

Inheritance diagram for `OsiTwoWayBranchingObject`:

Collaboration diagram for `OsiTwoWayBranchingObject`:

Public Member Functions

- [OsiTwoWayBranchingObject](#) ()
Default constructor.
- [OsiTwoWayBranchingObject](#) ([OsiSolverInterface](#) *solver, const [OsiObject](#) *originalObject, int way, double value)
Create a standard two-way branch object.
- [OsiTwoWayBranchingObject](#) (const [OsiTwoWayBranchingObject](#) &)
Copy constructor.
- [OsiTwoWayBranchingObject](#) & operator= (const [OsiTwoWayBranchingObject](#) &rhs)
Assignment operator.
- virtual [~OsiTwoWayBranchingObject](#) ()
Destructor.
- virtual double [branch](#) ([OsiSolverInterface](#) *solver)=0
Sets the bounds for the variable according to the current arm of the branch and advances the object state to the next arm.
- int [way](#) () const
Way returns -1 on down +1 on up.

Protected Attributes

- int [firstBranch_](#)
Which way was first branch -1 = down, +1 = up.

7.36.1 Detailed Description

This just adds two-wayness to a branching object.

Definition at line 467 of file `OsiBranchingObject.hpp`.

7.36.2 Constructor & Destructor Documentation

7.36.2.1 `OsiTwoWayBranchingObject::OsiTwoWayBranchingObject (OsiSolverInterface * solver, const OsiObject * originalObject, int way, double value)`

Create a standard two-way branch object.

Specifies a simple two-way branch. Specify way = -1 to set the object state to perform the down arm first, way = 1 for the up arm.

7.36.3 Member Function Documentation

7.36.3.1 `virtual double OsiTwoWayBranchingObject::branch (OsiSolverInterface * solver) [pure virtual]`

Sets the bounds for the variable according to the current arm of the branch and advances the object state to the next arm.

state. Returns change in guessed objective on next branch

Implements [OsiBranchingObject](#).

Implemented in [OsiLotsizeBranchingObject](#), [OsiSOSBranchingObject](#), and [OsiIntegerBranchingObject](#).

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

- [OsiBranchingObject.hpp](#)

7.37 OsiXprSolverInterface Class Reference

XPRESS-MP Solver Interface.

```
#include <OsiXprSolverInterface.hpp>
```

Inheritance diagram for OsiXprSolverInterface:

Collaboration diagram for OsiXprSolverInterface:

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 [branchAndBound](#) ()

Invoke solver's built-in enumeration algorithm.

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.

- bool [setIntParam](#) (OsiIntParam key, int value)
Set an integer parameter.
- bool [setDbParam](#) (OsiDbParam key, double value)
Set a double parameter.
- bool [setStrParam](#) (OsiStrParam key, const std::string &value)
Set a string parameter.
- bool [getIntParam](#) (OsiIntParam key, int &value) const
Get an integer parameter.
- bool [getDbParam](#) (OsiDbParam key, double &value) const
Get a double parameter.
- bool [getStrParam](#) (OsiStrParam key, std::string &value) const
Get a string parameter.
- void **setMipStart** (bool value)
- bool **getMipStart** () const

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

- **CoinWarmStart** * [getEmptyWarmStart](#) () const
Get empty warm start object.
- virtual **CoinWarmStart** * [getWarmStart](#) () const
Get warmstarting information.
- virtual bool [setWarmStart](#) (const **CoinWarmStart** *warmstart)
Set warmstarting information.

**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.

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 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 colIndex) const
Return true if variable is continuous.
- 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 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 -COIN_DBL_MAX for -infinity.
- virtual void [setColUpper](#) (int elementIndex, double elementValue)
Set a single column upper bound
Use COIN_DBL_MAX for infinity.
- virtual void [setColBounds](#) (int elementIndex, double lower, double upper)
Set a single column lower and upper bound
The default implementation just invokes [setColLower\(\)](#) and [setColUpper\(\)](#)
- 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 -COIN_DBL_MAX for -infinity.
- virtual void [setRowUpper](#) (int elementIndex, double elementValue)
Set a single row upper bound
Use COIN_DBL_MAX for infinity.
- virtual void [setRowBounds](#) (int elementIndex, double lower, double upper)
Set a single row lower and upper bound
The default implementation just invokes [setRowLower\(\)](#) and [setRowUpper\(\)](#)
- 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.

Integrality related changing methods

- virtual void [setContinuous](#) (int index)

- virtual void [setInteger](#) (int index)
Set the index-th variable to be a continuous variable.
- virtual void [setContinuous](#) (const int *indices, int len)
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.

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)
Add a 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)
Add a set of columns (primal variables) to the problem.
- virtual void [deleteCols](#) (const int num, const int *colIndices)
Remove a set of columns (primal variables) from the problem.
- virtual void [addRow](#) (const **CoinPackedVectorBase** &vec, 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)
Add a row (constraint) to the problem.
- virtual void [addRows](#) (const int numRows, const **CoinPackedVectorBase** *const *rows, const double *rowlb, const double *rowub)
Add a set of rows (constraints) to the problem.
- virtual void [addRows](#) (const int numRows, const **CoinPackedVectorBase** *const *rows, const char *rowsen, const double *rowrhs, const double *rowrng)
Add a set of rows (constraints) to the problem.
- virtual void [deleteRows](#) (const int num, const int *rowIndices)
Delete a set of rows (constraints) from the problem.

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 int numcols, const int numRows, const int *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 int *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 [readMps](#) (const char *filename, const char *extension="mps")
Read an mps file from the given filename.
- 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.

Message handling

- void [passInMessageHandler](#) ([CoinMessageHandler](#) *handler)
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.

Constructors and destructors

- [OsiXprSolverInterface](#) (int newrows=50, int newnz=100)
Default Constructor.
- virtual [OsiSolverInterface](#) * [clone](#) (bool copyData=true) const
Clone.
- [OsiXprSolverInterface](#) (const [OsiXprSolverInterface](#) &)
Copy constructor.
- [OsiXprSolverInterface](#) & [operator=](#) (const [OsiXprSolverInterface](#) &rhs)
Assignment operator.
- virtual [~OsiXprSolverInterface](#) ()
Destructor.

Static Public Member Functions

- static int [version](#) ()
Return XPRESS-MP Version number.

Protected Member Functions

Protected methods

- virtual void [applyRowCut](#) (const [OsiRowCut](#) &rc)
Apply a row cut. Return true if cut was applied.
- virtual void [applyColCut](#) (const [OsiColCut](#) &cc)
Apply a column cut (bound adjustment).

Friends

- void [OsiXprSolverInterfaceUnitTest](#) (const std::string &mpsDir, const std::string &netlibDir)
A function that tests the methods in the [OsiXprSolverInterface](#) class.

Static instance counter methods

- XPRSProb [getLpPtr](#) ()
Return a pointer to the XPRESS problem.
- static void [incrementInstanceCounter](#) ()
XPRESS has a context that must be created prior to all other XPRESS calls.
- static void [decrementInstanceCounter](#) ()
XPRESS has a context that should be deleted after XPRESS calls.
- static unsigned int [getNumInstances](#) ()
Return the number of instances of instantiated objects using XPRESS services.

Log File

- static int [iXprCallCount](#) _
- static FILE * [getLogFilePtr](#) ()
*Get logfile FILE *.*
- static void [setLogFileName](#) (const char *filename)
Set logfile name.

Additional Inherited Members

7.37.1 Detailed Description

XPRESS-MP Solver Interface.

Instantiation of [OsiSolverInterface](#) for XPRESS-MP

Definition at line 21 of file OsiXprSolverInterface.hpp.

7.37.2 Member Function Documentation

7.37.2.1 `virtual bool OsiXprSolverInterface::setWarmStart (const CoinWarmStart * warmstart) [virtual]`

Set warmstarting information.

Return true/false depending on whether the warmstart information was accepted or not.

Implements [OsiSolverInterface](#).

7.37.2.2 `virtual const char* OsiXprSolverInterface::getRowSense () const [virtual]`

Get pointer to array[[getNumRows\(\)](#)] of row constraint senses.

- 'L': <= constraint
- 'E': = constraint
- 'G': >= constraint
- 'R': ranged constraint
- 'N': free constraint

Implements [OsiSolverInterface](#).

7.37.2.3 `virtual const double* OsiXprSolverInterface::getRightHandSide () const [virtual]`

Get pointer to array[[getNumRows\(\)](#)] of rows right-hand sides.

- if `rowsense()[i] == 'L'` then `rhs()[i] == rowupper()[i]`
- if `rowsense()[i] == 'G'` then `rhs()[i] == rowlower()[i]`
- if `rowsense()[i] == 'R'` then `rhs()[i] == rowupper()[i]`
- if `rowsense()[i] == 'N'` then `rhs()[i] == 0.0`

Implements [OsiSolverInterface](#).

7.37.2.4 `virtual const double* OsiXprSolverInterface::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 0.0

Implements [OsiSolverInterface](#).

7.37.2.5 `virtual int OsiXprSolverInterface::getIterationCount () const [virtual]`

Get how many iterations it took to solve the problem (whatever "iteration" mean to the solver).

Implements [OsiSolverInterface](#).

7.37.2.6 `virtual std::vector<double*> OsiXprSolverInterface::getDualRays (int maxNumRays, bool fullRay = false) const [virtual]`

Get as many dual rays as the solver can provide.

(In case of proven primal infeasibility there should be at least one.)

The first [getNumRows\(\)](#) ray components will always be associated with the row duals (as returned by [getRowPrice\(\)](#)). If `fullRay` is true, the final [getNumCols\(\)](#) entries will correspond to the ray components associated with the nonbasic variables. If the full ray is requested and the method cannot provide it, it will throw an exception.

NOTE for implementers of solver interfaces:

The double pointers in the vector should point to arrays of length [getNumRows\(\)](#) and they should be allocated via `new[]`.

NOTE for users of solver interfaces:

It is the user's responsibility to free the double pointers in the vector using `delete[]`.

Implements [OsiSolverInterface](#).

7.37.2.7 `virtual std::vector<double*> OsiXprSolverInterface::getPrimalRays (int maxNumRays) const [virtual]`

Get as many primal rays as the solver can provide.

(In case of proven dual infeasibility there should be at least one.)

NOTE for implementers of solver interfaces:

The double pointers in the vector should point to arrays of length [getNumCols\(\)](#) and they should be allocated via `new[]`.

NOTE for users of solver interfaces:

It is the user's responsibility to free the double pointers in the vector using `delete[]`.

Implements [OsiSolverInterface](#).

7.37.2.8 `virtual void OsiXprSolverInterface::setColLower (int elementIndex, double elementValue)` [virtual]

Set a single column lower bound
Use `-COIN_DBL_MAX` for -infinity.

Implements [OsiSolverInterface](#).

7.37.2.9 `virtual void OsiXprSolverInterface::setColUpper (int elementIndex, double elementValue)` [virtual]

Set a single column upper bound
Use `COIN_DBL_MAX` for infinity.

Implements [OsiSolverInterface](#).

7.37.2.10 `virtual void OsiXprSolverInterface::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, indexLast</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

Reimplemented from [OsiSolverInterface](#).

7.37.2.11 `virtual void OsiXprSolverInterface::setRowLower (int elementIndex, double elementValue)` [virtual]

Set a single row lower bound
Use `-COIN_DBL_MAX` for -infinity.

Implements [OsiSolverInterface](#).

7.37.2.12 `virtual void OsiXprSolverInterface::setRowUpper (int elementIndex, double elementValue)` [virtual]

Set a single row upper bound
Use `COIN_DBL_MAX` for infinity.

Implements [OsiSolverInterface](#).

7.37.2.13 `virtual void OsiXprSolverInterface::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>index</i> ↔ <i>First, indexLast</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

Reimplemented from [OsiSolverInterface](#).

7.37.2.14 `virtual void OsiXprSolverInterface::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>index</i> ↔ <i>First, indexLast</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

Reimplemented from [OsiSolverInterface](#).

7.37.2.15 `virtual void OsiXprSolverInterface::setColSolution (const double * colsol) [virtual]`

Set the primal solution column values.

`colsol[numcols()]` is an array of values of the problem column variables. These values are copied to memory owned by the solver object or the solver. They will be returned as the result of `colsol()` until changed by another call to `setColsol()` or by a call to any solver routine. Whether the solver makes use of the solution in any way is solver-dependent.

Implements [OsiSolverInterface](#).

7.37.2.16 `virtual void OsiXprSolverInterface::setRowPrice (const double * rowprice) [virtual]`

Set dual solution vector.

`rowprice[numrows()]` is an array of values of the problem row dual variables. These values are copied to memory owned by the solver object or the solver. They will be returned as the result of `rowprice()` until changed by another call to `setRowprice()` or by a call to any solver routine. Whether the solver makes use of the solution in any way is solver-dependent.

Implements [OsiSolverInterface](#).

7.37.2.17 `virtual void OsiXprSolverInterface::addCol (const CoinPackedVectorBase & vec, const double collb, const double colub, const double obj) [virtual]`

Add a column (primal variable) to the problem.

Implements [OsiSolverInterface](#).

7.37.2.18 `virtual void OsiXprSolverInterface::addCols (const int numcols, const CoinPackedVectorBase *const * cols, const double * collb, const double * colub, const double * obj) [virtual]`

Add a set of columns (primal variables) to the problem.

The default implementation simply makes repeated calls to [addCol\(\)](#).

Reimplemented from [OsiSolverInterface](#).

7.37.2.19 `virtual void OsiXprSolverInterface::deleteCols (const int num, const int * colIndices) [virtual]`

Remove a set of columns (primal variables) from the problem.

The solver interface for a basis-oriented solver will maintain valid warm start information if all deleted variables are nonbasic.

Implements [OsiSolverInterface](#).

7.37.2.20 `virtual void OsiXprSolverInterface::addRow (const CoinPackedVectorBase & vec, const double rowlb, const double rowub) [virtual]`

Add a row (constraint) to the problem.

Implements [OsiSolverInterface](#).

7.37.2.21 `virtual void OsiXprSolverInterface::addRow (const CoinPackedVectorBase & vec, const char rowSen, const double rowrhs, const double rowrng) [virtual]`

Add a row (constraint) to the problem.

Implements [OsiSolverInterface](#).

7.37.2.22 `virtual void OsiXprSolverInterface::addRows (const int numrows, const CoinPackedVectorBase *const * rows, const double * rowlb, const double * rowub) [virtual]`

Add a set of rows (constraints) to the problem.

The default implementation simply makes repeated calls to [addRow\(\)](#).

Reimplemented from [OsiSolverInterface](#).

7.37.2.23 `virtual void OsiXprSolverInterface::addRows (const int numrows, const CoinPackedVectorBase *const * rows, const char * rowSen, const double * rowrhs, const double * rowrng) [virtual]`

Add a set of rows (constraints) to the problem.

The default implementation simply makes repeated calls to [addRow\(\)](#).

Reimplemented from [OsiSolverInterface](#).

7.37.2.24 `virtual void OsiXprSolverInterface::deleteRows (const int num, const int * rowIndices) [virtual]`

Delete a set of rows (constraints) from the problem.

The solver interface for a basis-oriented solver will maintain valid warm start information if all deleted rows are loose.

Implements [OsiSolverInterface](#).

7.37.2.25 `virtual void OsiXprSolverInterface::loadProblem (const CoinPackedMatrix & matrix, const double * collb, const double * colub, const double * obj, const double * rowlb, const double * rowub) [virtual]`

Load in a problem by copying the arguments (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

Implements [OsiSolverInterface](#).

7.37.2.26 `virtual void OsiXprSolverInterface::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.

Implements [OsiSolverInterface](#).

7.37.2.27 `virtual void OsiXprSolverInterface::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 0 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

Implements [OsiSolverInterface](#).

7.37.2.28 `virtual void OsiXprSolverInterface::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.

Implements [OsiSolverInterface](#).

7.37.2.29 `virtual void OsiXprSolverInterface::loadProblem (const int numcols, const int numRows, const int * 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).

7.37.2.30 `virtual void OsiXprSolverInterface::loadProblem (const int numcols, const int numRows, const int * 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).

7.37.2.31 `virtual void OsiXprSolverInterface::writeMps (const char * filename, const char * extension = "mps", double objSense = 0.0) const [virtual]`

Write the problem into an mps file of the given filename.

If objSense is non zero then -1.0 forces the code to write a maximization objective and +1.0 to write a minimization one.

If 0.0 then solver can do what it wants

Implements [OsiSolverInterface](#).

7.37.2.32 `static void OsiXprSolverInterface::incrementInstanceCounter () [static]`

XPRESS has a context that must be created prior to all other XPRESS calls.

This method:

- Increments by 1 the number of uses of the XPRESS environment.
- Creates the XPRESS context when the number of uses is changed to 1 from 0.

7.37.2.33 `static void OsiXprSolverInterface::decrementInstanceCounter () [static]`

XPRESS has a context that should be deleted after XPRESS calls.

This method:

- Decrements by 1 the number of uses of the XPRESS environment.
- Deletes the XPRESS context when the number of uses is change to 0 from 1.

7.37.2.34 `static unsigned int OsiXprSolverInterface::getNumInstances () [static]`

Return the number of instances of instantiated objects using XPRESS services.

7.37.2.35 `static void OsiXprSolverInterface::setLogFileName (const char * filename) [static]`

Set logfile name.

The logfile is an attempt to capture the calls to Xpress functions for debugging.

7.37.2.36 `virtual void OsiXprSolverInterface::applyColCut (const OsiColCut & cc) [protected], [virtual]`

Apply a column cut (bound adjustment).

Return true if cut was applied.

Implements [OsiSolverInterface](#).

7.37.3 Friends And Related Function Documentation

7.37.3.1 `void OsiXprSolverInterfaceUnitTest (const std::string & mpsDir, const std::string & netlibDir) [friend]`

A function that tests the methods in the [OsiXprSolverInterface](#) class.

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

- [OsiXprSolverInterface.hpp](#)

7.38 OsiUnitTest::TestOutcome Class Reference

A single test outcome record.

```
#include <OsiUnitTests.hpp>
```

Collaboration diagram for OsiUnitTest::TestOutcome:

Public Types

- enum [SeverityLevel](#)

Test result.

Public Member Functions

- [TestOutcome](#) (const std::string &comp, const std::string &tst, const char *cond, [SeverityLevel](#) sev, const char *file, int line, bool exp=false)

Standard constructor.

- void [print](#) () const

Print the test outcome.

Public Attributes

- std::string [component](#)
Name of component under test.
- std::string [testname](#)
Name of test.
- std::string [testcond](#)
Condition being tested.
- [SeverityLevel](#) [severity](#)
Test result.
- bool [expected](#)
Set to true if problem is expected.
- std::string [filename](#)
Name of code file where test executed.
- int [linenumber](#)
Line number in code file where test executed.

Static Public Attributes

- static std::string [SeverityLevelName](#) [LAST]
Print strings for SeverityLevel.

7.38.1 Detailed Description

A single test outcome record.

Definition at line 166 of file OsiUnitTests.hpp.

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

- [OsiUnitTests.hpp](#)

7.39 OsiUnitTest::TestOutcomes Class Reference

Utility class to maintain a list of test outcomes.

```
#include <OsiUnitTests.hpp>
```

Inheritance diagram for OsiUnitTest::TestOutcomes:

Collaboration diagram for OsiUnitTest::TestOutcomes:

Public Member Functions

- void [add](#) (std::string comp, std::string tst, const char *cond, [TestOutcome::SeverityLevel](#) sev, const char *file, int line, bool exp=false)
Add an outcome to the list.
- void [add](#) (const [OsiSolverInterface](#) &si, std::string tst, const char *cond, [TestOutcome::SeverityLevel](#) sev, const char *file, int line, bool exp=false)
Add an outcome to the list.

- void [print](#) () const
Print the list of outcomes.
- void [getCountBySeverity](#) ([TestOutcome::SeverityLevel](#) sev, int &total, int &expected) const
Count total and expected outcomes at given severity level.

7.39.1 Detailed Description

Utility class to maintain a list of test outcomes.

Definition at line 204 of file [OsiUnitTests.hpp](#).

7.39.2 Member Function Documentation

7.39.2.1 void [OsiUnitTest::TestOutcomes::add](#) (const [OsiSolverInterface](#) & *si*, std::string *tst*, const char * *cond*, [TestOutcome::SeverityLevel](#) *sev*, const char * *file*, int *line*, bool *exp* = false)

Add an outcome to the list.

Get the component name from the solver interface.

7.39.2.2 void [OsiUnitTest::TestOutcomes::getCountBySeverity](#) ([TestOutcome::SeverityLevel](#) *sev*, int & *total*, int & *expected*) const

Count total and expected outcomes at given severity level.

Given a severity level, walk the list of outcomes and count the total number of outcomes at this severity level and the number expected.

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

- [OsiUnitTests.hpp](#)

Chapter 8

File Documentation

8.1 OsiRowCutDebugger.hpp File Reference

Provides a facility to validate cut constraints to ensure that they do not cut off a given solution.

```
#include <string>
#include "OsiCuts.hpp"
#include "OsiSolverInterface.hpp"
Include dependency graph for OsiRowCutDebugger.hpp:
```

8.2 OsiUnitTests.hpp File Reference

Utility methods for OSI unit tests.

```
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <string>
#include <sstream>
#include <vector>
#include <list>
#include <map>
Include dependency graph for OsiUnitTests.hpp:
```

Classes

- class [OsiUnitTest::TestOutcome](#)
A single test outcome record.
- class [OsiUnitTest::TestOutcomes](#)
Utility class to maintain a list of test outcomes.

Namespaces

- [OsiUnitTest](#)
A namespace so we can define a few 'global' variables to use during tests.

Macros

- #define [OSIUNITTEST_QUOTEME_\(x\)](#) #x
Convert parameter to a string (stringification)
- #define [OSIUNITTEST_QUOTEME\(x\)](#) [OSIUNITTEST_QUOTEME_\(x\)](#)
Convert to string with one level of expansion of the parameter.
- #define [OSIUNITTEST_ADD_OUTCOME](#)(component, testname, testcondition, severity, expected)
Add a test outcome to the list held in [OsiUnitTest::outcomes](#).
- #define [OSIUNITTEST_ASSERT_SEVERITY_EXPECTED](#)(condition, failurecode, component, testname, severity, expected)
Test for a condition and record the result.
- #define [OSIUNITTEST_ASSERT_ERROR](#)(condition, failurecode, component, testname)
Perform a test with severity [OsiUnitTest::TestOutcome::ERROR](#), failure not expected.
- #define [OSIUNITTEST_ASSERT_WARNING](#)(condition, failurecode, component, testname)
Perform a test with severity [OsiUnitTest::TestOutcome::WARNING](#), failure not expected.
- #define [OSIUNITTEST_CATCH_SEVERITY_EXPECTED](#)(trycode, catchcode, component, testname, severity, expected)
Perform a test surrounded by a try/catch block.
- #define [OSIUNITTEST_CATCH_ERROR](#)(trycode, catchcode, component, testname) [OSIUNITTEST_CATCH_↵](#)
[_SEVERITY_EXPECTED](#)(trycode, catchcode, component, testname, [OsiUnitTest::TestOutcome::ERROR](#), false)
Perform a try/catch test with severity [OsiUnitTest::TestOutcome::ERROR](#), failure not expected.
- #define [OSIUNITTEST_CATCH_WARNING](#)(trycode, catchcode, component, testname) [OSIUNITTEST_CAT_↵](#)
[CH_SEVERITY_EXPECTED](#)(trycode, catchcode, component, testname, [OsiUnitTest::TestOutcome::WARNING](#), false)
Perform a try/catch test with severity [OsiUnitTest::TestOutcome::WARNING](#), failure not expected.

Functions

- void [OsiSolverInterfaceMpsUnitTest](#) (const std::vector< [OsiSolverInterface](#) * > &vecEmptySiP, const std::string &mpsDir)
A function that tests that a lot of problems given in MPS files (mostly the NETLIB problems) solve properly with all the specified solvers.
- void [OsiSolverInterfaceCommonUnitTest](#) (const [OsiSolverInterface](#) *emptySi, const std::string &mpsDir, const std::string &netlibDir)
A function that tests the methods in the [OsiSolverInterface](#) class.
- void [OsiColCutUnitTest](#) (const [OsiSolverInterface](#) *baseSiP, const std::string &mpsDir)
A function that tests the methods in the [OsiColCut](#) class.
- void [OsiRowCutUnitTest](#) (const [OsiSolverInterface](#) *baseSiP, const std::string &mpsDir)
A function that tests the methods in the [OsiRowCut](#) class.
- void [OsiRowCutDebuggerUnitTest](#) (const [OsiSolverInterface](#) *siP, const std::string &mpsDir)
A function that tests the methods in the [OsiRowCutDebugger](#) class.
- void [OsiCutsUnitTest](#) ()
A function that tests the methods in the [OsiCuts](#) class.
- void [OsiUnitTest::failureMessage](#) (const std::string &solverName, const std::string &message)
Print an error message.
- void [OsiUnitTest::failureMessage](#) (const [OsiSolverInterface](#) &si, const std::string &message)
- void [OsiUnitTest::failureMessage](#) (const std::string &solverName, const std::string &testname, const std::string &testcond)

Print an error message, specifying the test name and condition.

- void [OsiUnitTest::failureMessage](#) (const [OsiSolverInterface](#) &si, const std::string &testname, const std::string &testcond)
- void [OsiUnitTest::testingMessage](#) (const char *const msg)

Print a message.

- bool [OsiUnitTest::equivalentVectors](#) (const [OsiSolverInterface](#) *si1, const [OsiSolverInterface](#) *si2, double tol, const double *v1, const double *v2, int size)

Utility method to check equality.

- bool [OsiUnitTest::compareProblems](#) ([OsiSolverInterface](#) *osi1, [OsiSolverInterface](#) *osi2)

Compare two problems for equality.

- bool [OsiUnitTest::isEquivalent](#) (const **CoinPackedVectorBase** &pv, int n, const double *fv)

Compare a packed vector with an expanded vector.

- bool [OsiUnitTest::processParameters](#) (int argc, const char **argv, std::map< std::string, std::string > &parms, const std::map< std::string, int > &ignorekeywords=std::map< std::string, int >())

Process command line parameters.

Variables

- unsigned int [OsiUnitTest::verbosity](#)
Verbosity level of unit tests.
- unsigned int [OsiUnitTest::haltonerror](#)
Behaviour on failing a test.
- TestOutcomes [OsiUnitTest::outcomes](#)
Test outcomes.

8.2.1 Detailed Description

Utility methods for OSI unit tests.

8.2.2 Macro Definition Documentation

8.2.2.1 `#define OSIUNITTEST_ASSERT_SEVERITY_EXPECTED(condition, failurecode, component, testname, severity, expected)`

Value:

```
{ \
  if (!OsiUnitTestAssertSeverityExpected(condition, #condition, \
    __FILE__, __LINE__, component, testname, severity, expected)) { \
    failurecode; \
  } \
}
```

Test for a condition and record the result.

Test `condition` and record the result in [OsiUnitTest::outcomes](#). If it succeeds, record the result as `OsiUnitTest::↵TestOutcome::PASSED` and print a message for [OsiUnitTest::verbosity](#) ≥ 2 . If it fails, record the test as failed with `severity` and `expected` and react as specified by [OsiUnitTest::haltonerror](#).

`failurecode` is executed when failure is not fatal.

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

8.2.2.2 `#define OSIUNITTEST_CATCH_SEVERITY_EXPECTED(trycode, catchcode, component, testname, severity, expected)`

Perform a test surrounded by a try/catch block.

`trycode` is executed in a try/catch block; if there's no throw the test is deemed to have succeeded and is recorded in `OsiUnitTest::outcomes` with status `OsiUnitTest::TestOutcome::PASSED`. If the `trycode` throws a **CoinError**, the failure is recorded with status `severity` and `expected` and the value of `OsiUnitTest::haltonerror` is consulted. If the failure is not fatal, `catchcode` is executed. If any other error is thrown, the failure is recorded as for a **CoinError** and `catchcode` is executed (`haltonerror` is not consulted).

Definition at line 314 of file `OsiUnitTests.hpp`.

8.2.3 Function Documentation

8.2.3.1 `void OsiSolverInterfaceMpsUnitTest (const std::vector< OsiSolverInterface * > & vecEmptySiP, const std::string & mpsDir)`

A function that tests that a lot of problems given in MPS files (mostly the NETLIB problems) solve properly with all the specified solvers.

The routine creates a vector of NetLib problems (problem name, objective, various other characteristics), and a vector of solvers to be tested.

Each solver is run on each problem. The run is deemed successful if the solver reports the correct problem size after loading and returns the correct objective value after optimization.

If multiple solvers are available, the results are compared pairwise against the results reported by adjacent solvers in the solver vector. Due to limitations of the volume solver, it must be the last solver in `vecEmptySiP`.

8.2.3.2 `void OsiSolverInterfaceCommonUnitTest (const OsiSolverInterface * emptySi, const std::string & mpsDir, const std::string & netlibDir)`

A function that tests the methods in the `OsiSolverInterface` class.

Some time ago, if this method is compiled with optimization, the compilation took 10-15 minutes and the machine pages (has 256M core memory!)...

8.2.3.3 `void OsiColCutUnitTest (const OsiSolverInterface * baseSiP, const std::string & mpsDir)`

A function that tests the methods in the `OsiColCut` class.

8.2.3.4 `void OsiRowCutUnitTest (const OsiSolverInterface * baseSiP, const std::string & mpsDir)`

A function that tests the methods in the `OsiRowCut` class.

8.2.3.5 `void OsiRowCutDebuggerUnitTest (const OsiSolverInterface * siP, const std::string & mpsDir)`

A function that tests the methods in the `OsiRowCutDebugger` class.

8.2.3.6 `void OsiCutsUnitTest ()`

A function that tests the methods in the `OsiCuts` class.

Index

- activate
 - OsiRowCutDebugger, [129](#), [130](#)
- activateRowCutDebugger
 - OsiSolverInterface, [165](#)
- add
 - OsiUnitTest::TestOutcomes, [202](#)
- addCol
 - OsiCpxSolverInterface, [51](#)
 - OsiGlpkSolverInterface, [71](#)
 - OsiGrbSolverInterface, [88](#)
 - OsiMskSolverInterface, [110](#)
 - OsiSolverInterface, [156](#)
 - OsiSpxSolverInterface, [182](#)
 - OsiXprSolverInterface, [196](#)
- addCols
 - OsiCpxSolverInterface, [51](#)
 - OsiGlpkSolverInterface, [71](#)
 - OsiGrbSolverInterface, [88](#)
 - OsiMskSolverInterface, [111](#)
 - OsiSolverInterface, [156](#), [157](#)
 - OsiXprSolverInterface, [196](#)
- addObjects
 - OsiSolverInterface, [165](#)
- addRow
 - OsiCpxSolverInterface, [52](#)
 - OsiGlpkSolverInterface, [72](#)
 - OsiGrbSolverInterface, [88](#), [89](#)
 - OsiMskSolverInterface, [111](#)
 - OsiSolverInterface, [157](#), [158](#)
 - OsiSpxSolverInterface, [182](#)
 - OsiXprSolverInterface, [197](#)
- addRows
 - OsiCpxSolverInterface, [52](#)
 - OsiGlpkSolverInterface, [72](#)
 - OsiGrbSolverInterface, [89](#)
 - OsiMskSolverInterface, [111](#)
 - OsiSolverInterface, [158](#)
 - OsiXprSolverInterface, [197](#)
- applyColCut
 - OsiCpxSolverInterface, [54](#)
 - OsiGlpkSolverInterface, [75](#)
 - OsiGrbSolverInterface, [92](#)
 - OsiMskSolverInterface, [114](#)
 - OsiSolverInterface, [169](#)
 - OsiSpxSolverInterface, [184](#)
 - OsiXprSolverInterface, [200](#)
- applyCuts
 - OsiGrbSolverInterface, [91](#)
 - OsiSolverInterface, [159](#)
- applyRowCut
 - OsiSolverInterface, [169](#)
- applyRowCuts
 - OsiSolverInterface, [160](#)
- assignProblem
 - OsiCpxSolverInterface, [53](#)
 - OsiGlpkSolverInterface, [73](#)
 - OsiGrbSolverInterface, [89](#), [90](#)
 - OsiMskSolverInterface, [112](#), [113](#)
 - OsiSolverInterface, [160](#), [161](#)
 - OsiSpxSolverInterface, [183](#), [184](#)
 - OsiXprSolverInterface, [198](#)
- basisIsAvailable
 - OsiSolverInterface, [166](#)
- branch
 - OsiBranchingObject, [31](#)
 - OsiIntegerBranchingObject, [95](#)
 - OsiLotsizeBranchingObject, [99](#)
 - OsiTwoWayBranchingObject, [186](#)
- branchIndex_
 - OsiBranchingObject, [31](#)
- canDoSimplexInterface
 - OsiSolverInterface, [166](#)
- clone
 - OsiSolverInterface, [168](#)
- columnType
 - OsiSolverInterface, [149](#)
- compareProblems
 - OsiUnitTest, [19](#)
- consistent
 - OsiColCut, [39](#)
 - OsiCut, [56](#)
 - OsiRowCut, [125](#), [126](#)
- copyParameters
 - OsiSolverInterface, [147](#)
- createBranch
 - OsiLotsize, [97](#)
 - OsiObject, [117](#)
 - OsiSOS, [173](#)
 - OsiSimpleInteger, [132](#)

- decrementInstanceCounter
 - OsiGlpkSolverInterface, [75](#)
 - OsiGrbSolverInterface, [91](#)
 - OsiMskSolverInterface, [113](#)
 - OsiXprSolverInterface, [199](#)
- defaultHandler_
 - OsiSolverInterface, [170](#)
- deleteColNames
 - OsiSolverInterface, [155](#)
- deleteCols
 - OsiCpxSolverInterface, [52](#)
 - OsiGlpkSolverInterface, [72](#)
 - OsiGrbSolverInterface, [88](#)
 - OsiMskSolverInterface, [111](#)
 - OsiSolverInterface, [157](#)
 - OsiSpxSolverInterface, [182](#)
 - OsiXprSolverInterface, [197](#)
- deleteRowNames
 - OsiSolverInterface, [154](#)
- deleteRows
 - OsiCpxSolverInterface, [52](#)
 - OsiGlpkSolverInterface, [72](#)
 - OsiGrbSolverInterface, [89](#)
 - OsiMskSolverInterface, [112](#)
 - OsiSolverInterface, [158](#)
 - OsiSpxSolverInterface, [183](#)
 - OsiXprSolverInterface, [197](#)
- dfltRowColName
 - OsiSolverInterface, [154](#)
- differentModel
 - OsiSolverInterface, [164](#)
- disableFactorization
 - OsiSolverInterface, [166](#)
- doStrongBranching
 - OsiChooseStrong, [33](#)
- dumpCuts
 - OsiCuts, [60](#)
- enableFactorization
 - OsiSolverInterface, [166](#)
- enableSimplexInterface
 - OsiSolverInterface, [168](#)
- equivalentVectors
 - OsiUnitTest, [18](#)
- eraseAndDumpCuts
 - OsiCuts, [60](#)
- extraCharacteristics
 - OsiBabSolver, [26](#)
- extraCharacteristics_
 - OsiBabSolver, [27](#)
- FREECACHED_COLUMN
 - OsiCpxSolverInterface, [47](#)
 - OsiGlpkSolverInterface, [66](#)
 - OsiGrbSolverInterface, [82](#)
 - OsiMskSolverInterface, [107](#)
 - OsiSpxSolverInterface, [179](#)
- FREECACHED_MATRIX
 - OsiCpxSolverInterface, [47](#)
 - OsiGlpkSolverInterface, [67](#)
 - OsiGrbSolverInterface, [83](#)
 - OsiMskSolverInterface, [107](#)
 - OsiSpxSolverInterface, [179](#)
- FREECACHED_RESULTS
 - OsiCpxSolverInterface, [47](#)
 - OsiGlpkSolverInterface, [67](#)
 - OsiGrbSolverInterface, [83](#)
 - OsiMskSolverInterface, [107](#)
 - OsiSpxSolverInterface, [179](#)
- FREECACHED_ROW
 - OsiCpxSolverInterface, [47](#)
 - OsiGlpkSolverInterface, [67](#)
 - OsiGrbSolverInterface, [82](#)
 - OsiMskSolverInterface, [107](#)
 - OsiSpxSolverInterface, [179](#)
- failureMessage
 - OsiUnitTest, [18](#)
- feasibleRegion
 - OsiLotsize, [97](#)
 - OsiObject, [116](#)
 - OsiSOS, [173](#)
 - OsiSimpleInteger, [132](#)
- findIntegers
 - OsiSolverInterface, [164](#)
- findIntegersAndSOS
 - OsiSolverInterface, [165](#)
- findRange
 - OsiLotsize, [97](#)
- forceFeasible
 - OsiSolverInterface, [165](#)
- getBlvARow
 - OsiSolverInterface, [167](#)
- getBasics
 - OsiSolverInterface, [168](#)
- getBasisStatus
 - OsiSolverInterface, [167](#)
- getColName
 - OsiSolverInterface, [155](#)
- getColNames
 - OsiSolverInterface, [155](#)
- getColType
 - OsiSolverInterface, [149](#)
- getCountBySeverity
 - OsiUnitTest::TestOutcomes, [202](#)
- getDualRays
 - OsiCpxSolverInterface, [48](#)
 - OsiGlpkSolverInterface, [68](#)
 - OsiGrbSolverInterface, [85](#)

- OsiMskSolverInterface, 108
- OsiSolverInterface, 150
- OsiSpxSolverInterface, 180
- OsiXprSolverInterface, 193
- getEmptyWarmStart
 - OsiCpxSolverInterface, 47
 - OsiGlpkSolverInterface, 67
 - OsiGrbSolverInterface, 83
 - OsiMskSolverInterface, 107
 - OsiSolverInterface, 147
- getFractionalIndices
 - OsiSolverInterface, 151
- getHintParam
 - OsiGrbSolverInterface, 83
 - OsiSolverInterface, 146, 147
- getIntegerTolerance
 - OsiSolverInterface, 147
- getIterationCount
 - OsiCpxSolverInterface, 48
 - OsiGlpkSolverInterface, 68
 - OsiGrbSolverInterface, 84
 - OsiMskSolverInterface, 108
 - OsiSpxSolverInterface, 180
 - OsiXprSolverInterface, 193
- getMutableMatrixByCol
 - OsiSolverInterface, 150
- getMutableMatrixByRow
 - OsiSolverInterface, 149
- getNumInstances
 - OsiXprSolverInterface, 199
- getObjSense
 - OsiSolverInterface, 149
- getPointerToWarmStart
 - OsiSolverInterface, 148
- getPrimalRays
 - OsiCpxSolverInterface, 48
 - OsiGlpkSolverInterface, 68
 - OsiGrbSolverInterface, 85
 - OsiMskSolverInterface, 108
 - OsiSolverInterface, 150
 - OsiSpxSolverInterface, 181
 - OsiXprSolverInterface, 193
- getReducedGradient
 - OsiSolverInterface, 167
- getRightHandSide
 - OsiCpxSolverInterface, 47
 - OsiGlpkSolverInterface, 67
 - OsiGrbSolverInterface, 84
 - OsiMskSolverInterface, 108
 - OsiSolverInterface, 148
 - OsiSpxSolverInterface, 180
 - OsiXprSolverInterface, 192
- getRowActivity
 - OsiSolverInterface, 150
- getRowCutDebugger
 - OsiSolverInterface, 165
- getRowCutDebuggerAlways
 - OsiSolverInterface, 166
- getRowName
 - OsiSolverInterface, 154
- getRowNames
 - OsiSolverInterface, 154
- getRowRange
 - OsiCpxSolverInterface, 48
 - OsiGlpkSolverInterface, 68
 - OsiGrbSolverInterface, 84
 - OsiMskSolverInterface, 108
 - OsiSolverInterface, 149
 - OsiSpxSolverInterface, 180
 - OsiXprSolverInterface, 193
- getRowSense
 - OsiCpxSolverInterface, 47
 - OsiGlpkSolverInterface, 67
 - OsiGrbSolverInterface, 84
 - OsiMskSolverInterface, 107
 - OsiSolverInterface, 148
 - OsiSpxSolverInterface, 180
 - OsiXprSolverInterface, 192
- getWarmStart
 - OsiSolverInterface, 147
- glp_prob, 22
- gutsOfDestroy
 - OsiPresolve, 121
- haltonerror
 - OsiUnitTest, 19
- hasSolution
 - OsiBabSolver, 26
- incrementInstanceCounter
 - OsiGlpkSolverInterface, 74
 - OsiGrbSolverInterface, 91
 - OsiMskSolverInterface, 113
 - OsiXprSolverInterface, 199
- infeasibility
 - OsiObject, 116
- infeasibility_
 - OsiObject, 117
- infeasible
 - OsiColCut, 39
 - OsiCut, 56
 - OsiRowCut, 126
- insert
 - OsiCuts, 59
- insertIfNotDuplicate
 - OsiCuts, 59
- invalidCut
 - OsiRowCutDebugger, 129
- isEquivalent

- OsiUnitTest, [19](#)
- isInteger
 - OsiSolverInterface, [149](#)
- KEEPCACHED_ALL
 - OsiCpxSolverInterface, [47](#)
 - OsiGlpkSolverInterface, [66](#)
 - OsiGrbSolverInterface, [82](#)
 - OsiMskSolverInterface, [107](#)
 - OsiSpxSolverInterface, [179](#)
- KEEPCACHED_COLUMN
 - OsiCpxSolverInterface, [47](#)
 - OsiGlpkSolverInterface, [66](#)
 - OsiGrbSolverInterface, [82](#)
 - OsiMskSolverInterface, [107](#)
 - OsiSpxSolverInterface, [179](#)
- KEEPCACHED_MATRIX
 - OsiCpxSolverInterface, [47](#)
 - OsiGlpkSolverInterface, [66](#)
 - OsiGrbSolverInterface, [82](#)
 - OsiMskSolverInterface, [107](#)
 - OsiSpxSolverInterface, [179](#)
- KEEPCACHED_NONE
 - OsiCpxSolverInterface, [47](#)
 - OsiGlpkSolverInterface, [66](#)
 - OsiGrbSolverInterface, [82](#)
 - OsiMskSolverInterface, [107](#)
 - OsiSpxSolverInterface, [179](#)
- KEEPCACHED_PROBLEM
 - OsiCpxSolverInterface, [47](#)
 - OsiGlpkSolverInterface, [66](#)
 - OsiGrbSolverInterface, [82](#)
 - OsiMskSolverInterface, [107](#)
 - OsiSpxSolverInterface, [179](#)
- KEEPCACHED_RESULTS
 - OsiCpxSolverInterface, [47](#)
 - OsiGlpkSolverInterface, [66](#)
 - OsiGrbSolverInterface, [82](#)
 - OsiMskSolverInterface, [107](#)
 - OsiSpxSolverInterface, [179](#)
- KEEPCACHED_ROW
 - OsiCpxSolverInterface, [47](#)
 - OsiGlpkSolverInterface, [66](#)
 - OsiGrbSolverInterface, [82](#)
 - OsiMskSolverInterface, [107](#)
 - OsiSpxSolverInterface, [179](#)
- keepCachedFlag
 - OsiCpxSolverInterface, [46](#)
 - OsiGlpkSolverInterface, [66](#)
 - OsiGrbSolverInterface, [82](#)
 - OsiMskSolverInterface, [107](#)
 - OsiSpxSolverInterface, [179](#)
- loadFromCoinModel
 - OsiSolverInterface, [162](#)
- loadProblem
 - OsiCpxSolverInterface, [53, 54](#)
 - OsiGlpkSolverInterface, [73, 74](#)
 - OsiGrbSolverInterface, [89, 90](#)
 - OsiMskSolverInterface, [112, 113](#)
 - OsiSolverInterface, [160, 161](#)
 - OsiSpxSolverInterface, [183, 184](#)
 - OsiXprSolverInterface, [198, 199](#)
- members_
 - OsiSOS, [173](#)
- model
 - OsiPresolve, [120](#)
- OSIUNITTEST_ASSERT_SEVERITY_EXPECTED
 - OsiUnitTests.hpp, [205](#)
- OSIUNITTEST_CATCH_SEVERITY_EXPECTED
 - OsiUnitTests.hpp, [205](#)
- onOptimalPath
 - OsiRowCutDebugger, [129](#)
- OsiAuxInfo, [23](#)
- OsiBabSolver, [23](#)
 - extraCharacteristics, [26](#)
 - extraCharacteristics_, [27](#)
 - hasSolution, [26](#)
 - setExtraCharacteristics, [26](#)
 - setSolution, [26](#)
 - setSolverType, [26](#)
 - solverType, [26](#)
 - solverType_, [27](#)
- OsiBranchingInformation, [27](#)
 - stateOfSearch_, [29](#)
- OsiBranchingObject, [29](#)
 - branch, [31](#)
 - branchIndex_, [31](#)
- OsiChooseStrong, [32](#)
 - doStrongBranching, [33](#)
 - setupList, [33](#)
- OsiChooseVariable, [34](#)
 - setupList, [37](#)
- OsiColCut, [37](#)
 - consistent, [39](#)
 - infeasible, [39](#)
 - OsiColCutUnitTest, [39](#)
 - violated, [39](#)
- OsiColCutUnitTest
 - OsiColCut, [39](#)
 - OsiUnitTests.hpp, [206](#)
- OsiCpxSolverInterface, [40](#)
 - addCol, [51](#)
 - addCols, [51](#)
 - addRow, [52](#)
 - addRows, [52](#)
 - applyColCut, [54](#)
 - assignProblem, [53](#)

- deleteCols, 52
- deleteRows, 52
- FREECACHED_COLUMN, 47
- FREECACHED_MATRIX, 47
- FREECACHED_RESULTS, 47
- FREECACHED_ROW, 47
- getDualRays, 48
- getEmptyWarmStart, 47
- getIterationCount, 48
- getPrimalRays, 48
- getRightHandSide, 47
- getRowRange, 48
- getRowSense, 47
- KEEPCACHED_ALL, 47
- KEEPCACHED_COLUMN, 47
- KEEPCACHED_MATRIX, 47
- KEEPCACHED_NONE, 47
- KEEPCACHED_PROBLEM, 47
- KEEPCACHED_RESULTS, 47
- KEEPCACHED_ROW, 47
- keepCachedFlag, 46
- loadProblem, 53, 54
- OsiCpxSolverInterfaceUnitTest, 54
- setColLower, 49
- setColSetBounds, 49
- setColSolution, 51
- setColUpper, 49
- setRowLower, 49
- setRowPrice, 51
- setRowSetBounds, 49
- setRowSetTypes, 51
- setRowUpper, 49
- setWarmStart, 47
- writeMps, 54
- OsiCpxSolverInterfaceUnitTest
 - OsiCpxSolverInterface, 54
- OsiCut, 55
 - consistent, 56
 - infeasible, 56
 - violated, 56
- OsiCuts, 57
 - dumpCuts, 60
 - eraseAndDumpCuts, 60
 - insert, 59
 - insertIfNotDuplicate, 59
 - OsiCutsUnitTest, 60
- OsiCuts::const_iterator, 22
- OsiCuts::iterator, 22
- OsiCutsUnitTest
 - OsiCuts, 60
 - OsiUnitTests.hpp, 206
- OsiGlpkSolverInterface, 60
 - addCol, 71
 - addCols, 71
 - addRow, 72
 - addRows, 72
 - applyColCut, 75
 - assignProblem, 73
 - decrementInstanceCounter, 75
 - deleteCols, 72
 - deleteRows, 72
 - FREECACHED_COLUMN, 66
 - FREECACHED_MATRIX, 67
 - FREECACHED_RESULTS, 67
 - FREECACHED_ROW, 67
 - getDualRays, 68
 - getEmptyWarmStart, 67
 - getIterationCount, 68
 - getPrimalRays, 68
 - getRightHandSide, 67
 - getRowRange, 68
 - getRowSense, 67
 - incrementInstanceCounter, 74
 - KEEPCACHED_ALL, 66
 - KEEPCACHED_COLUMN, 66
 - KEEPCACHED_MATRIX, 66
 - KEEPCACHED_NONE, 66
 - KEEPCACHED_PROBLEM, 66
 - KEEPCACHED_RESULTS, 66
 - KEEPCACHED_ROW, 66
 - keepCachedFlag, 66
 - loadProblem, 73, 74
 - OsiGlpkSolverInterfaceUnitTest, 75
 - setColLower, 69
 - setColName, 74
 - setColSetBounds, 69
 - setColSolution, 71
 - setColUpper, 69
 - setHintParam, 67
 - setRowLower, 69
 - setRowName, 74
 - setRowPrice, 71
 - setRowSetBounds, 69
 - setRowSetTypes, 71
 - setRowUpper, 69
 - setWarmStart, 67
 - writeMps, 74
- OsiGlpkSolverInterfaceUnitTest
 - OsiGlpkSolverInterface, 75
- OsiGrbSolverInterface, 75
 - addCol, 88
 - addCols, 88
 - addRow, 88, 89
 - addRows, 89
 - applyColCut, 92
 - applyCuts, 91
 - assignProblem, 89, 90
 - decrementInstanceCounter, 91

- deleteCols, 88
- deleteRows, 89
- FREECACHED_COLUMN, 82
- FREECACHED_MATRIX, 83
- FREECACHED_RESULTS, 83
- FREECACHED_ROW, 82
- getDualRays, 85
- getEmptyWarmStart, 83
- getHintParam, 83
- getIterationCount, 84
- getPrimalRays, 85
- getRightHandSide, 84
- getRowRange, 84
- getRowSense, 84
- incrementInstanceCounter, 91
- KEEPCACHED_ALL, 82
- KEEPCACHED_COLUMN, 82
- KEEPCACHED_MATRIX, 82
- KEEPCACHED_NONE, 82
- KEEPCACHED_PROBLEM, 82
- KEEPCACHED_RESULTS, 82
- KEEPCACHED_ROW, 82
- keepCachedFlag, 82
- loadProblem, 89, 90
- OsiGrbSolverInterfaceUnitTest, 92
- setColLower, 85
- setColSetBounds, 85
- setColSolution, 88
- setColUpper, 85
- setHintParam, 83
- setRowLower, 87
- setRowPrice, 88
- setRowSetBounds, 87
- setRowSetTypes, 87
- setRowUpper, 87
- setWarmStart, 84
- switchToLP, 91
- writeMps, 90
- OsiGrbSolverInterfaceUnitTest
 - OsiGrbSolverInterface, 92
- OsiHotInfo, 92
 - updateInformation, 94
- OsiIntegerBranchingObject, 94
 - branch, 95
 - OsiIntegerBranchingObject, 95
- OsiLotsize, 95
 - createBranch, 97
 - feasibleRegion, 97
 - findRange, 97
 - resetBounds, 97
- OsiLotsizeBranchingObject, 97
 - branch, 99
 - OsiLotsizeBranchingObject, 98
- OsiMskSolverInterface, 99
- addCol, 110
- addCols, 111
- addRow, 111
- addRows, 111
- applyColCut, 114
- assignProblem, 112, 113
- decrementInstanceCounter, 113
- deleteCols, 111
- deleteRows, 112
- FREECACHED_COLUMN, 107
- FREECACHED_MATRIX, 107
- FREECACHED_RESULTS, 107
- FREECACHED_ROW, 107
- getDualRays, 108
- getEmptyWarmStart, 107
- getIterationCount, 108
- getPrimalRays, 108
- getRightHandSide, 108
- getRowRange, 108
- getRowSense, 107
- incrementInstanceCounter, 113
- KEEPCACHED_ALL, 107
- KEEPCACHED_COLUMN, 107
- KEEPCACHED_MATRIX, 107
- KEEPCACHED_NONE, 107
- KEEPCACHED_PROBLEM, 107
- KEEPCACHED_RESULTS, 107
- KEEPCACHED_ROW, 107
- keepCachedFlag, 107
- loadProblem, 112, 113
- setColLower, 109
- setColSetBounds, 109
- setColSolution, 110
- setColUpper, 109
- setRowLower, 109
- setRowPrice, 110
- setRowSetBounds, 110
- setRowSetTypes, 110
- setRowUpper, 109
- setWarmStart, 107
- writeMps, 113
- OsiNameVec
 - OsiSolverInterface, 146
- OsiObject, 114
 - createBranch, 117
 - feasibleRegion, 116
 - infeasibility, 116
 - infeasibility_, 117
 - resetBounds, 117
 - setWhichWay, 117
 - whichWay, 117
- OsiObject2, 117
- OsiPresolve, 118
 - gutsOfDestroy, 121

- model, 120
- postsolve, 120, 121
- presolve, 121
- presolvedModel, 120
- setNonLinearValue, 121
- setPresolveActions, 121
- OsiPseudoCosts, 122
- OsiRowCut, 123
 - consistent, 125, 126
 - infeasible, 126
 - OsiRowCut, 125
 - OsiRowCutUnitTest, 126
 - violated, 126
- OsiRowCut2, 126
- OsiRowCutDebugger, 127
 - activate, 129, 130
 - invalidCut, 129
 - onOptimalPath, 129
 - OsiRowCutDebugger, 129
 - OsiRowCutDebuggerUnitTest, 130
 - redoSolution, 130
 - validateCuts, 129
- OsiRowCutDebugger.hpp, 203
- OsiRowCutDebuggerUnitTest
 - OsiRowCutDebugger, 130
 - OsiUnitTests.hpp, 206
- OsiRowCutUnitTest
 - OsiRowCut, 126
 - OsiUnitTests.hpp, 206
- OsiSOS, 171
 - createBranch, 173
 - feasibleRegion, 173
 - members_, 173
 - OsiSOS, 173
- OsiSOSBranchingObject, 173
- OsiSimpleInteger, 130
 - createBranch, 132
 - feasibleRegion, 132
 - resetBounds, 132
- OsiSolverBranch, 132
- OsiSolverInterface, 133
 - activateRowCutDebugger, 165
 - addCol, 156
 - addCols, 156, 157
 - addObjects, 165
 - addRow, 157, 158
 - addRows, 158
 - applyColCut, 169
 - applyCuts, 159
 - applyRowCut, 169
 - applyRowCuts, 160
 - assignProblem, 160, 161
 - basisIsAvailable, 166
 - canDoSimplexInterface, 166
 - clone, 168
 - columnType, 149
 - copyParameters, 147
 - defaultHandler_, 170
 - deleteColNames, 155
 - deleteCols, 157
 - deleteRowNames, 154
 - deleteRows, 158
 - dfltRowColName, 154
 - differentModel, 164
 - disableFactorization, 166
 - enableFactorization, 166
 - enableSimplexInterface, 168
 - findIntegers, 164
 - findIntegersAndSOS, 165
 - forceFeasible, 165
 - getBInvARow, 167
 - getBasics, 168
 - getBasisStatus, 167
 - getColName, 155
 - getColNames, 155
 - getColType, 149
 - getDualRays, 150
 - getEmptyWarmStart, 147
 - getFractionalIndices, 151
 - getHintParam, 146, 147
 - getIntegerTolerance, 147
 - getMutableMatrixByCol, 150
 - getMutableMatrixByRow, 149
 - getObjSense, 149
 - getPointerToWarmStart, 148
 - getPrimalRays, 150
 - getReducedGradient, 167
 - getRightHandSide, 148
 - getRowActivity, 150
 - getRowCutDebugger, 165
 - getRowCutDebuggerAlways, 166
 - getRowName, 154
 - getRowNames, 154
 - getRowRange, 149
 - getRowSense, 148
 - getWarmStart, 147
 - isInteger, 149
 - loadFromCoinModel, 162
 - loadProblem, 160, 161
 - OsiNameVec, 146
 - OsiSolverInterfaceCommonUnitTest, 169
 - OsiSolverInterfaceMpsUnitTest, 169
 - passInMessageHandler, 164
 - pivot, 168
 - primalPivotResult, 168
 - readGMPL, 162
 - readLp, 164
 - readMps, 162

- reducedCostFix, [153](#)
- replaceMatrix, [159](#)
- replaceMatrixOptional, [158](#)
- reset, [168](#)
- resolve, [146](#)
- restoreBaseModel, [159](#)
- rowCutDebugger_, [170](#)
- saveBaseModel, [159](#)
- setApplicationData, [164](#)
- setAuxiliaryInfo, [164](#)
- setBasisStatus, [167](#)
- setColBounds, [152](#)
- setColLower, [151](#)
- setColName, [155](#)
- setColNames, [155](#)
- setColSetBounds, [152](#)
- setColSolution, [153](#)
- setColUpper, [151](#), [152](#)
- setHintParam, [146](#)
- setInitialData, [169](#)
- setObjSense, [151](#)
- setObjective, [151](#)
- setRowBounds, [152](#)
- setRowColNames, [155](#), [156](#)
- setRowLower, [152](#)
- setRowName, [154](#)
- setRowNames, [154](#)
- setRowPrice, [153](#)
- setRowSetBounds, [153](#)
- setRowSetTypes, [153](#)
- setRowUpper, [152](#)
- setWarmStart, [148](#)
- writeLp, [163](#)
- writeLpNative, [163](#), [164](#)
- writeMps, [162](#)
- writeMpsNative, [163](#)
- OsiSolverInterface::ApplyCutsReturnCode, [21](#)
- OsiSolverInterfaceCommonUnitTest
 - OsiSolverInterface, [169](#)
 - OsiUnitTests.hpp, [206](#)
- OsiSolverInterfaceMpsUnitTest
 - OsiSolverInterface, [169](#)
 - OsiUnitTests.hpp, [206](#)
- OsiSolverResult, [170](#)
- OsiSpxSolverInterface, [174](#)
 - addCol, [182](#)
 - addRow, [182](#)
 - applyColCut, [184](#)
 - assignProblem, [183](#), [184](#)
 - deleteCols, [182](#)
 - deleteRows, [183](#)
 - FREECACHED_COLUMN, [179](#)
 - FREECACHED_MATRIX, [179](#)
 - FREECACHED_RESULTS, [179](#)
 - FREECACHED_ROW, [179](#)
 - getDualRays, [180](#)
 - getIterationCount, [180](#)
 - getPrimalRays, [181](#)
 - getRightHandSide, [180](#)
 - getRowRange, [180](#)
 - getRowSense, [180](#)
 - KEEPCACHED_ALL, [179](#)
 - KEEPCACHED_COLUMN, [179](#)
 - KEEPCACHED_MATRIX, [179](#)
 - KEEPCACHED_NONE, [179](#)
 - KEEPCACHED_PROBLEM, [179](#)
 - KEEPCACHED_RESULTS, [179](#)
 - KEEPCACHED_ROW, [179](#)
 - keepCachedFlag, [179](#)
 - loadProblem, [183](#), [184](#)
 - OsiSpxSolverInterfaceUnitTest, [185](#)
 - setColLower, [181](#)
 - setColSolution, [182](#)
 - setColUpper, [181](#)
 - setRowLower, [181](#)
 - setRowPrice, [182](#)
 - setRowUpper, [181](#)
 - setWarmStart, [180](#)
 - writeMps, [184](#)
- OsiSpxSolverInterfaceUnitTest
 - OsiSpxSolverInterface, [185](#)
- OsiTwoWayBranchingObject, [185](#)
 - branch, [186](#)
 - OsiTwoWayBranchingObject, [186](#)
- OsiUnitTest, [17](#)
 - compareProblems, [19](#)
 - equivalentVectors, [18](#)
 - failureMessage, [18](#)
 - haltonerror, [19](#)
 - isEquivalent, [19](#)
 - outcomes, [19](#)
 - processParameters, [19](#)
 - testingMessage, [18](#)
 - verbosity, [19](#)
- OsiUnitTest::TestOutcome, [200](#)
- OsiUnitTest::TestOutcomes, [201](#)
 - add, [202](#)
 - getCountBySeverity, [202](#)
- OsiUnitTests.hpp, [203](#)
 - OSIUNITTEST_ASSERT_SEVERITY_EXPECTED, [205](#)
 - OSIUNITTEST_CATCH_SEVERITY_EXPECTED, [205](#)
 - OsiColCutUnitTest, [206](#)
 - OsiCutsUnitTest, [206](#)
 - OsiRowCutDebuggerUnitTest, [206](#)
 - OsiRowCutUnitTest, [206](#)
 - OsiSolverInterfaceCommonUnitTest, [206](#)

- OsiSolverInterfaceMpsUnitTest, 206
- OsiXprSolverInterface, 186
 - addCol, 196
 - addCols, 196
 - addRow, 197
 - addRows, 197
 - applyColCut, 200
 - assignProblem, 198
 - decrementInstanceCounter, 199
 - deleteCols, 197
 - deleteRows, 197
 - getDualRays, 193
 - getIterationCount, 193
 - getNumInstances, 199
 - getPrimalRays, 193
 - getRightHandSide, 192
 - getRowRange, 193
 - getRowSense, 192
 - incrementInstanceCounter, 199
 - loadProblem, 198, 199
 - OsiXprSolverInterfaceUnitTest, 200
 - setColLower, 194
 - setColSetBounds, 194
 - setColSolution, 196
 - setColUpper, 194
 - setLogFileName, 200
 - setRowLower, 194
 - setRowPrice, 196
 - setRowSetBounds, 194
 - setRowSetTypes, 196
 - setRowUpper, 194
 - setWarmStart, 192
 - writeMps, 199
- OsiXprSolverInterfaceUnitTest
 - OsiXprSolverInterface, 200
- outcomes
 - OsiUnitTest, 19
- passInMessageHandler
 - OsiSolverInterface, 164
- pivot
 - OsiSolverInterface, 168
- postsolve
 - OsiPresolve, 120, 121
- presolve
 - OsiPresolve, 121
- presolvedModel
 - OsiPresolve, 120
- primalPivotResult
 - OsiSolverInterface, 168
- processParameters
 - OsiUnitTest, 19
- readGMPL
 - OsiSolverInterface, 162
- readLp
 - OsiSolverInterface, 164
- readMps
 - OsiSolverInterface, 162
- redoSolution
 - OsiRowCutDebugger, 130
- reducedCostFix
 - OsiSolverInterface, 153
- replaceMatrix
 - OsiSolverInterface, 159
- replaceMatrixOptional
 - OsiSolverInterface, 158
- reset
 - OsiSolverInterface, 168
- resetBounds
 - OsiLotsize, 97
 - OsiObject, 117
 - OsiSimpleInteger, 132
- resolve
 - OsiSolverInterface, 146
- restoreBaseModel
 - OsiSolverInterface, 159
- rowCutDebugger_
 - OsiSolverInterface, 170
- saveBaseModel
 - OsiSolverInterface, 159
- setApplicationData
 - OsiSolverInterface, 164
- setAuxiliaryInfo
 - OsiSolverInterface, 164
- setBasisStatus
 - OsiSolverInterface, 167
- setColBounds
 - OsiSolverInterface, 152
- setColLower
 - OsiCpxSolverInterface, 49
 - OsiGlpkSolverInterface, 69
 - OsiGrbSolverInterface, 85
 - OsiMskSolverInterface, 109
 - OsiSolverInterface, 151
 - OsiSpxSolverInterface, 181
 - OsiXprSolverInterface, 194
- setColName
 - OsiGlpkSolverInterface, 74
 - OsiSolverInterface, 155
- setColNames
 - OsiSolverInterface, 155
- setColSetBounds
 - OsiCpxSolverInterface, 49
 - OsiGlpkSolverInterface, 69
 - OsiGrbSolverInterface, 85
 - OsiMskSolverInterface, 109
 - OsiSolverInterface, 152

- OsiXprSolverInterface, 194
- setColSolution
 - OsiCpxSolverInterface, 51
 - OsiGlpkSolverInterface, 71
 - OsiGrbSolverInterface, 88
 - OsiMskSolverInterface, 110
 - OsiSolverInterface, 153
 - OsiSpxSolverInterface, 182
 - OsiXprSolverInterface, 196
- setColUpper
 - OsiCpxSolverInterface, 49
 - OsiGlpkSolverInterface, 69
 - OsiGrbSolverInterface, 85
 - OsiMskSolverInterface, 109
 - OsiSolverInterface, 151, 152
 - OsiSpxSolverInterface, 181
 - OsiXprSolverInterface, 194
- setExtraCharacteristics
 - OsiBabSolver, 26
- setHintParam
 - OsiGlpkSolverInterface, 67
 - OsiGrbSolverInterface, 83
 - OsiSolverInterface, 146
- setInitialData
 - OsiSolverInterface, 169
- setLogFileName
 - OsiXprSolverInterface, 200
- setNonLinearValue
 - OsiPresolve, 121
- setObjSense
 - OsiSolverInterface, 151
- setObjective
 - OsiSolverInterface, 151
- setPresolveActions
 - OsiPresolve, 121
- setRowBounds
 - OsiSolverInterface, 152
- setRowColNames
 - OsiSolverInterface, 155, 156
- setRowLower
 - OsiCpxSolverInterface, 49
 - OsiGlpkSolverInterface, 69
 - OsiGrbSolverInterface, 87
 - OsiMskSolverInterface, 109
 - OsiSolverInterface, 152
 - OsiSpxSolverInterface, 181
 - OsiXprSolverInterface, 194
- setRowName
 - OsiGlpkSolverInterface, 74
 - OsiSolverInterface, 154
- setRowNames
 - OsiSolverInterface, 154
- setRowPrice
 - OsiCpxSolverInterface, 51
- OsiGlpkSolverInterface, 71
- OsiGrbSolverInterface, 88
- OsiMskSolverInterface, 110
- OsiSolverInterface, 153
- OsiSpxSolverInterface, 182
- OsiXprSolverInterface, 196
- setRowSetBounds
 - OsiCpxSolverInterface, 49
 - OsiGlpkSolverInterface, 69
 - OsiGrbSolverInterface, 87
 - OsiMskSolverInterface, 110
 - OsiSolverInterface, 153
 - OsiXprSolverInterface, 194
- setRowSetTypes
 - OsiCpxSolverInterface, 51
 - OsiGlpkSolverInterface, 71
 - OsiGrbSolverInterface, 87
 - OsiMskSolverInterface, 110
 - OsiSolverInterface, 153
 - OsiXprSolverInterface, 196
- setRowUpper
 - OsiCpxSolverInterface, 49
 - OsiGlpkSolverInterface, 69
 - OsiGrbSolverInterface, 87
 - OsiMskSolverInterface, 109
 - OsiSolverInterface, 152
 - OsiSpxSolverInterface, 181
 - OsiXprSolverInterface, 194
- setSolution
 - OsiBabSolver, 26
- setSolverType
 - OsiBabSolver, 26
- setWarmStart
 - OsiCpxSolverInterface, 47
 - OsiGlpkSolverInterface, 67
 - OsiGrbSolverInterface, 84
 - OsiMskSolverInterface, 107
 - OsiSolverInterface, 148
 - OsiSpxSolverInterface, 180
 - OsiXprSolverInterface, 192
- setWhichWay
 - OsiObject, 117
- setupList
 - OsiChooseStrong, 33
 - OsiChooseVariable, 37
- solverType
 - OsiBabSolver, 26
- solverType_
 - OsiBabSolver, 27
- stateOfSearch_
 - OsiBranchingInformation, 29
- switchToLP
 - OsiGrbSolverInterface, 91

- testingMessage
 - OsiUnitTest, [18](#)
- updateInformation
 - OsiHotInfo, [94](#)
- validateCuts
 - OsiRowCutDebugger, [129](#)
- verbosity
 - OsiUnitTest, [19](#)
- violated
 - OsiColCut, [39](#)
 - OsiCut, [56](#)
 - OsiRowCut, [126](#)
- whichWay
 - OsiObject, [117](#)
- writeLp
 - OsiSolverInterface, [163](#)
- writeLpNative
 - OsiSolverInterface, [163](#), [164](#)
- writeMps
 - OsiCpxSolverInterface, [54](#)
 - OsiGlpkSolverInterface, [74](#)
 - OsiGrbSolverInterface, [90](#)
 - OsiMskSolverInterface, [113](#)
 - OsiSolverInterface, [162](#)
 - OsiSpxSolverInterface, [184](#)
 - OsiXprSolverInterface, [199](#)
- writeMpsNative
 - OsiSolverInterface, [163](#)