FICO® Xpress Optimizer Python interface

User's manual

47.01

FICO® Xpress Optimization

FICO

©1983–2026 Fair Isaac Corporation. All rights reserved. This documentation is the property of Fair Isaac Corporation ("FICO"). Receipt or possession of this documentation does not convey rights to disclose, reproduce, make derivative works, use, or allow others to use it except solely for internal evaluation purposes to determine whether to purchase a license to the software described in this documentation, or as otherwise set forth in a written software license agreement between you and FICO (or a FICO affiliate). Use of this documentation and the software described in it must conform strictly to the foregoing permitted uses, and no other use is permitted.

The information in this documentation is subject to change without notice. If you find any problems in this documentation, please report them to us in writing. Neither FICO nor its affiliates warrant that this documentation is error-free, nor are there any other warranties with respect to the documentation except as may be provided in the license agreement. FICO and its affiliates specifically disclaim any warranties, express or implied, including, but not limited to, non-infringement, merchantability and fitness for a particular purpose. Portions of this documentation and the software described in it may contain copyright of various authors and may be licensed under certain third-party licenses identified in the software, documentation, or both.

In no event shall FICO or its affiliates be liable to any person for direct, indirect, special, incidental, or consequential damages, including lost profits, arising out of the use of this documentation or the software described in it, even if FICO or its affiliates have been advised of the possibility of such damage. FICO and its affiliates have no obligation to provide maintenance, support, updates, enhancements, or modifications except as required to licensed users under a license agreement.

FICO is a registered trademark of Fair Isaac Corporation in the United States and may be a registered trademark of Fair Isaac Corporation in other countries. Other product and company names herein may be trademarks of their respective owners.

Patent(s): www.fico.com/en/patents

Xpress Optimizer 47.01 (FICO® Xpress 9.9)

Deliverable Version: A

Last Revised: 15 May, 2026

Contents

1 Introduction **1**

1.1 Outline . . . 1
1.2 Installing the Python Xpress module . . . 1
1.2.1 Installation from the Python Package Index (PyPI) . . . 2
1.2.2 Installation from Conda . . . 2
1.2.3 Installation from a package file . . . 3
1.2.4 GPU installation guidelines for PDHG . . . 3
1.2.5 Troubleshooting the installation . . . 3

2 Modeling an optimization problem **5**

2.1 Getting started . . . 5
2.2 Creating a problem . . . 5
2.3 Variables . . . 5
2.3.1 Variable names and Python objects . . . 6
2.4 Constraints . . . 7
2.5 Objective function . . . 9
2.6 Special Ordered Sets (SOSs) . . . 10
2.7 Indicator constraints . . . 10
2.8 Piecewise linear functions . . . 10
2.9 General constraints . . . 11
2.10 Using `load*` functions for efficiency . . . 12
2.11 Modeling and solving nonlinear problems . . . 14
2.12 Solving a problem . . . 15
2.13 Querying a problem . . . 16
2.14 Reading and writing a problem . . . 18
2.15 Hints for building models efficiently . . . 18
2.16 Exceptions . . . 19
2.17 Warnings . . . 20

3 Using Python numerical libraries **21**

3.1 Using NumPy with the Xpress Python interface . . . 21
3.2 Products of NumPy arrays . . . 22
3.3 Products with SciPy sparse matrices . . . 23
3.4 Using Pandas with the Xpress Python interface . . . 23

4 Controls and Attributes **25**

4.1 Controls . . . 25
4.2 Examples . . . 26
4.3 Attributes . . . 26
4.4 Examples . . . 27
4.5 Accessing controls and attributes as object members . . . 27
4.6 Objective controls and attributes . . . 29

5 Using Callbacks 30

6 Examples of use 35

6.1 Modeling examples . . . 35
6.1.1 A simple model . . . 35
6.1.2 Using IIS to investigate an infeasible problem . . . 36
6.1.3 Modeling a problem using Python lists and vectors . . . 36
6.1.4 A knapsack problem . . . 37
6.1.5 A Min-cost-flow problem using NumPy . . . 37
6.1.6 A nonlinear model . . . 38
6.1.7 Finding the maximum-area *n*-gon . . . 38
6.1.8 Solving the *n*-queens problem . . . 39
6.1.9 Solving Sudoku problems . . . 40
6.2 Examples using NumPy . . . 41
6.2.1 Using NumPy multidimensional arrays to create variables . . . 41
6.2.2 Using the dot product to create arrays of expressions . . . 41
6.2.3 Using the Dot product to create constraints and quadratic functions . . . 42
6.2.4 Using NumPy to create quadratic optimization problems . . . 42
6.3 Advanced examples: callbacks and problem querying, modifying, and analysis . . . 43
6.3.1 Visualize the branch-and-bound tree of a problem . . . 43
6.3.2 Query and modify a simple problem . . . 45
6.3.3 Change a problem after solution . . . 45
6.3.4 Comparing the coefficients of two equally sized problems . . . 47
6.3.5 Combining modeling and API functions . . . 47
6.3.6 A simple Traveling Salesman Problem (TSP) solver . . . 48
6.3.7 Solving a nonconvex MIQCQP . . . 51
6.4 Creating simple problems using the low-level API functions . . . 59
6.4.1 Generating a small Linear Programming problem . . . 59
6.4.2 A Mixed Integer Quadratically Constrained Quadratic Programming problem . . . 60
6.5 Translated Mosel examples . . . 61

7 Migrating to the linked API 62

7.1 Creating linked variables . . . 62
7.2 Creating linked constraints . . . 63
7.3 Creating linked SOS constraints . . . 63
7.4 Creating linked problems . . . 64
7.5 Other API functions . . . 64
7.6 Summary . . . 64

8 Reference Manual 66

8.1 Using this chapter . . . 66
8.2 Format of the reference . . . 67
8.3 Classes of the Xpress module . . . 67
8.4 Global methods of the Xpress module . . . 67
xpress.setCheckedMode . . . 69
xpress.getCheckedMode . . . 70
xpress.getBanner . . . 71
xpress.getVersion . . . 72
xpress.getVersionNumbers . . . 73
xpress.getDaysLeft . . . 74
xpress.featureQuery . . . 75
xpress.setArchConsistency . . . 76

xpress.setComputeAllowed . . . 77
xpress.getComputeAllowed . . . 78
xpress.array . . . 79
xpress.evaluate . . . 80
xpress.examples . . . 82
xpress.free . . . 83
xpress.init . . . 84
xpress.setDefaults . . . 85
xpress.setDefaultControl . . . 86
xpress.getOutputEnabled . . . 87
xpress.setOutputEnabled . . . 88
xpress.getConstraintOperatorsEnabled . . . 89
xpress.setConstraintOperatorsEnabled . . . 90
xpress.isEqual . . . 91
xpress.eq . . . 92
xpress.leq . . . 93
xpress.geq . . . 94
8.5 Methods to be used within a callback of a problem object . . . 95
8.6 Xpress base classes . . . 95
xpress.attr . . . 96
xpress.branchobj . . . 97
xpress.constraint . . . 98
xpress.constraint . . . 98
xpress.ctrl . . . 100
xpress.expression . . . 101
xpress.linterm . . . 102
xpress.nonlin . . . 103
xpress.poolcut . . . 104
xpress.problem . . . 105
xpress.quadterm . . . 107
xpress.sos . . . 108
xpress.sos . . . 108
xpress.var . . . 109
xpress.var . . . 109
xpress.ndarray . . . 110
xpress.voidstar . . . 111
xpress.xprsobject . . . 112
8.7 Xpress object functions . . . 113
object.extractLinear . . . 114
object.extractQuadratic . . . 115
8.8 Xpress operators . . . 116
xpress.abs . . . 117
xpress.acos . . . 118
xpress.And . . . 119
xpress.asin . . . 120
xpress.atan . . . 121
xpress.cos . . . 122
xpress.Dot . . . 123
xpress.erf . . . 125
xpress.erfc . . . 126
xpress.exp . . . 127
xpress.log . . . 128
xpress.log10 . . . 129
xpress.max . . . 130
xpress.min . . . 131

xpress.Or . . . 132
xpress.pwl . . . 133
xpress.Prod . . . 134
xpress.sign . . . 135
xpress.sin . . . 136
xpress.sqrt . . . 137
xpress.Sum . . . 138
xpress.tan . . . 139
xpress.user . . . 140
8.9 Xpress `problem` object methods . . . 142
problem.setProbName . . . 158
problem.setLogFile . . . 159
problem.setDefaultControl . . . 160
problem.setDefaults . . . 161
problem.readProb . . . 162
problem.loadLP . . . 164
problem.loadQP . . . 166
problem.loadMIQP . . . 168
problem.fixMIPEntities . . . 170
problem.loadModelCuts . . . 171
problem.loadDelayedRows . . . 172
problem.loadDirs . . . 173
problem.loadBranchDirs . . . 174
problem.loadPresolveDirs . . . 175
problem.loadMIP . . . 176
problem.addNames . . . 178
problem.scale . . . 179
problem.readDirs . . . 180
problem.writeDirs . . . 181
problem.setIndicators . . . 182
problem.addPwlCons . . . 183
problem.getPwlCons . . . 184
problem.addGenCons . . . 185
problem.getGenCons . . . 186
problem.delPwlCons . . . 187
problem.delGenCons . . . 188
problem.dumpControls . . . 189
problem.getIndicators . . . 190
problem.delIndicators . . . 191
problem.lpOptimize . . . 192
problem.mipOptimize . . . 193
problem.optimize . . . 194
problem.readSlxSol . . . 195
problem.readBasis . . . 196
problem.readBinSol . . . 197
problem.getInfeas . . . 198
problem.getScaledInfeas . . . 199
problem.getUnbVec . . . 200
problem.crossoverLpSol . . . 201
problem.tune . . . 202
problem.tuneProbSetFile . . . 203
problem.tunerWriteMethod . . . 204
problem.tunerReadMethod . . . 205
problem.getLastBarSol . . . 206
problem.clearIIS . . . 207

problem.firstIIS . . . 208
problem.nextIIS . . . 209
problem.printIIS . . . 210
problem.IISStatus . . . 211
problem.IISAll . . . 212
problem.writeIIS . . . 213
problem.IISIsolations . . . 214
problem.getIISData . . . 215
problem.loadPresolveBasis . . . 217
problem.loadSecureVecs . . . 218
problem.addRows . . . 219
problem.delRows . . . 220
problem.addCols . . . 221
problem.delCols . . . 222
problem.chgColType . . . 223
problem.loadBasis . . . 224
problem.postSolve . . . 225
problem.delSets . . . 226
problem.addSets . . . 227
problem.strongBranch . . . 228
problem.strongBranchCB . . . 229
problem.estimateRowDualRanges . . . 230
problem.setMessageStatus . . . 231
problem.getMessageStatus . . . 232
problem.chgObjSense . . . 233
problem.chgGlbLimit . . . 234
problem.saveAs . . . 235
problem.restore . . . 236
problem.pivot . . . 237
problem.loadLPSol . . . 238
problem.repairWeightedInfeas . . . 239
problem.repairWeightedInfeasBounds . . . 241
problem.repairInfeas . . . 243
problem.basisStability . . . 245
problem.objSA . . . 246
problem.bndSA . . . 247
problem.rhsSA . . . 248
problem.addQMatrix . . . 249
problem.delQMatrix . . . 250
problem.loadQCQP . . . 251
problem.loadMIQCQP . . . 253
problem.addObj . . . 256
problem.chgObjN . . . 257
problem.delObj . . . 258
problem.copyControls . . . 259
problem.interrupt . . . 260
problem.getProbName . . . 261
problem.getControlInfo . . . 262
problem.getAttribInfo . . . 263
problem.getIndex . . . 264
problem.getQObj . . . 265
problem.getDirs . . . 266
problem.getScale . . . 267
problem.getPivotOrder . . . 268
problem.getPresolveMap . . . 269

problem.BTran . . . 270
problem.FTran . . . 271
problem.sparseBTran . . . 272
problem.sparseFTran . . . 273
problem.getObj . . . 274
problem.getObjN . . . 275
problem.getRHS . . . 276
problem.getRHSRange . . . 277
problem.getLB . . . 278
problem.getUB . . . 279
problem.getCols . . . 280
problem.getRows . . . 281
problem.getMIPEntities . . . 282
problem.getRowFlags . . . 283
problem.clearRowFlags . . . 284
problem.getCoef . . . 285
problem.getMQObj . . . 286
problem.writeBasis . . . 287
problem.writeSol . . . 288
problem.writeBinSol . . . 289
problem.writePrtSol . . . 290
problem.writeSlxSol . . . 291
problem.getPrimalRay . . . 292
problem.getDualRay . . . 293
problem.loadMipSol . . . 294
problem.getBasis . . . 295
problem.addManagedCuts . . . 296
problem.addCuts . . . 297
problem.delCuts . . . 298
problem.delCPCuts . . . 299
problem.getCutList . . . 300
problem.getCPCutList . . . 301
problem.getCPCuts . . . 302
problem.loadCuts . . . 303
problem.storeCuts . . . 304
problem.presolveRow . . . 305
problem.presolveSol . . . 307
problem.postSolveSol . . . 308
problem.getPivots . . . 309
problem.writeProb . . . 310
problem.calcSlacks . . . 311
problem.calcReducedCosts . . . 312
problem.calcObjective . . . 313
problem.calcObjN . . . 314
problem.calcSolInfo . . . 315
problem.getRowType . . . 316
problem.getPresolveBasis . . . 317
problem.getColType . . . 318
problem.getQRowCoeff . . . 319
problem.getQRowQMatrix . . . 320
problem.getQRowQMatrixTriplets . . . 321
problem.chgQRowCoeff . . . 322
problem.getQRows . . . 323
problem.chgBounds . . . 324
problem.triggerRestart . . . 325

problem.getNameList . . . 326
problem.addMipSol . . . 327
problem.getCutSlack . . . 328
problem.getCutMap . . . 329
problem.getPresolveSol . . . 330
problem.getLpSol . . . 331
problem.getCallbackPresolveSolution . . . 332
problem.getCallbackPresolveSlacks . . . 333
problem.getCallbackPresolveDuals . . . 334
problem.getCallbackPresolveRedCosts . . . 335
problem.chgObj . . . 336
problem.chgCoef . . . 337
problem.chgMCoef . . . 338
problem.chgMQObj . . . 339
problem.chgQObj . . . 340
problem.chgRHS . . . 341
problem.chgRHSRange . . . 342
problem.chgRowType . . . 343
problem.msAddJob . . . 344
problem.msAddPreset . . . 345
problem.msAddCustomPreset . . . 346
problem.nlpPrintEvalInfo . . . 347
problem.nlpValidate . . . 348
problem.nlpOptimize . . . 349
problem.nlpCurrentIV . . . 350
problem.nlpValidateRow . . . 351
problem.nlpValidateKKT . . . 352
problem.msClear . . . 353
problem.nlpEvaluateFormula . . . 354
problem.nlpValidateVector . . . 355
problem.nlpAddFormulas . . . 356
problem.nlpChgFormulaStr . . . 357
problem.nlpChgFormula . . . 358
problem.nlpGetFormula . . . 359
problem.nlpGetFormulaRows . . . 360
problem.nlpLoadFormulas . . . 361
problem.nlpDelFormulas . . . 362
problem.nlpGetFormulaStr . . . 363
problem.nlpSetInitVal . . . 364
problem.slpGetCoefFormula . . . 365
problem.slpGetCoefs . . . 366
problem.slpLoadCoefs . . . 367
problem.slpDelCoefs . . . 368
problem.slpGetCoefStr . . . 369
problem.slpSetDetRow . . . 370
problem.slpAddCoefs . . . 371
problem.slpChgCoefStr . . . 372
problem.slpChgCoef . . . 373
problem.slpGetColInfo . . . 374
problem.slpGetRowInfo . . . 375
problem.slpCascadeSol . . . 376
problem.slpCascadeOrder . . . 377
problem.slpChgRowStatus . . . 378
problem.slpChgRowWt . . . 379
problem.slpChgDeltaType . . . 380

problem.slpChgCascadeNLimit . . . 381
problem.slpConstruct . . . 382
problem.slpGetRowStatus . . . 383
problem.slpGetRowWT . . . 384
problem.slpEvaluateCoef . . . 385
problem.slpReInitialize . . . 386
problem.slpUnConstruct . . . 387
problem.slpUpdateLinearization . . . 388
problem.slpFixPenalties . . . 389
problem.nlpPostsolveProb . . . 390
problem.nlpCalcSlacks . . . 391
problem.slpSetInitStepBounds . . . 392
problem.addLplogCallback . . . 393
problem.removeLplogCallback . . . 394
problem.addMiplogCallback . . . 395
problem.removeMiplogCallback . . . 396
problem.addCutlogCallback . . . 397
problem.removeCutlogCallback . . . 398
problem.addBarlogCallback . . . 399
problem.removeBarlogCallback . . . 400
problem.addOptnodeCallback . . . 401
problem.removeOptnodeCallback . . . 402
problem.addPrenodeCallback . . . 403
problem.removePrenodeCallback . . . 404
problem.addInfnodeCallback . . . 405
problem.removeInfnodeCallback . . . 406
problem.addNodecutoffCallback . . . 407
problem.removeNodecutoffCallback . . . 408
problem.addIntsolCallback . . . 409
problem.removeIntsolCallback . . . 410
problem.addPreIntsolCallback . . . 411
problem.removePreIntsolCallback . . . 413
problem.addMessageCallback . . . 414
problem.removeMessageCallback . . . 415
problem.addMipThreadCallback . . . 416
problem.removeMipThreadCallback . . . 417
problem.addMipThreadDestroyCallback . . . 418
problem.removeMipThreadDestroyCallback . . . 419
problem.addBarIterationCallback . . . 420
problem.removeBarIterationCallback . . . 421
problem.addPresolveCallback . . . 422
problem.removePresolveCallback . . . 423
problem.addNewnodeCallback . . . 424
problem.removeNewnodeCallback . . . 425
problem.addChangeBranchObjectCallback . . . 426
problem.removeChangeBranchObjectCallback . . . 427
problem.addGapNotifyCallback . . . 428
problem.removeGapNotifyCallback . . . 429
problem.addUserSolNotifyCallback . . . 430
problem.removeUserSolNotifyCallback . . . 431
problem.addNodeLPSolvedCallback . . . 432
problem.removeNodeLPSolvedCallback . . . 433
problem.addComputeRestartCallback . . . 434
problem.removeComputeRestartCallback . . . 435
problem.addBeforeObjectiveCallback . . . 436

problem.removeBeforeObjectiveCallback . . . 437
problem.addAfterObjectiveCallback . . . 438
problem.removeAfterObjectiveCallback . . . 439
problem.addCheckTimeCallback . . . 440
problem.removeCheckTimeCallback . . . 441
problem.addCutRoundCallback . . . 442
problem.removeCutRoundCallback . . . 443
problem.addSlpCascadeEndCallback . . . 444
problem.removeSlpCascadeEndCallback . . . 445
problem.addSlpCascadeStartCallback . . . 446
problem.removeSlpCascadeStartCallback . . . 447
problem.addSlpCascadeVarCallback . . . 448
problem.removeSlpCascadeVarCallback . . . 449
problem.addSlpCascadeVarFailCallback . . . 450
problem.removeSlpCascadeVarFailCallback . . . 451
problem.addSlpConstructCallback . . . 452
problem.removeSlpConstructCallback . . . 453
problem.addSlpIntSolCallback . . . 454
problem.removeSlpIntSolCallback . . . 455
problem.addSlpIterEndCallback . . . 456
problem.removeSlpIterEndCallback . . . 457
problem.addSlpIterStartCallback . . . 458
problem.removeSlpIterStartCallback . . . 459
problem.addSlpIterVarCallback . . . 460
problem.removeSlpIterVarCallback . . . 461
problem.addSlpDrColCallback . . . 462
problem.removeSlpDrColCallback . . . 463
problem.addMsJobStartCallback . . . 464
problem.removeMsJobStartCallback . . . 465
problem.addMsJobEndCallback . . . 466
problem.removeMsJobEndCallback . . . 467
problem.addMsWinnerCallback . . . 468
problem.removeMsWinnerCallback . . . 469
problem.addNlpCoefEvalErrorCallback . . . 470
problem.removeNlpCoefEvalErrorCallback . . . 471
problem.addSlpPreUpdateLinearizationCallback . . . 472
problem.removeSlpPreUpdateLinearizationCallback . . . 473
problem.addConstraint . . . 474
problem.addIndicator . . . 475
problem.addObjective . . . 476
problem.addSOS . . . 477
problem.addVariable . . . 478
problem.addVariables . . . 479
problem.copy . . . 481
problem.copycallbacks . . . 482
problem.delConstraint . . . 483
problem.delSOS . . . 484
problem.delVariable . . . 485
problem.getAttrib . . . 486
problem.getCallbackDuals . . . 487
problem.getCallbackRedCosts . . . 488
problem.getCallbackSlacks . . . 489
problem.getCallbackSolution . . . 490
problem.getConstraint . . . 491
problem.getControl . . . 492

problem.getDuals . . . 493
problem.getRedCosts . . . 494
problem.getSlacks . . . 495
problem.getSolution . . . 496
problem.getSOS . . . 497
problem.getVariable . . . 498
problem.reset . . . 499
problem.save . . . 500
problem.setControl . . . 501
problem.setObjective . . . 502
problem.getOutputEnabled . . . 503
problem.setOutputEnabled . . . 504
8.10 Xpress `variable` object methods . . . 505
var.getSolution . . . 506
var.getRedCost . . . 507
8.11 Xpress `constraint` object methods . . . 508
constraint.getBody . . . 509
constraint.getSlack . . . 510
constraint.getDual . . . 511
8.12 Xpress `branchobj` object methods . . . 512
branchobj.store . . . 513
branchobj.addBranches . . . 514
branchobj.getBranches . . . 515
branchobj.setPriority . . . 516
branchobj.setPreferredBranch . . . 517
branchobj.addBounds . . . 518
branchobj.getBounds . . . 519
branchobj.addRows . . . 520
branchobj.getRows . . . 521
branchobj.addCuts . . . 522
branchobj.getID . . . 523
branchobj.validate . . . 524
8.13 Xpress constants and enumerations . . . 525
xpress.SolStatus . . . 527
xpress.SolveStatus . . . 528
xpress.LPStatus . . . 529
xpress.MIPStatus . . . 530
xpress.IISSolStatus . . . 531
xpress.SolAvailable . . . 532
xpress.OptimizeType . . . 533
xpress.BarOrder . . . 534
xpress.DefaultAlg . . . 535
xpress.StopType . . . 536
xpress.AlwaysNeverAutomatic . . . 537
xpress.OnOff . . . 538
xpress.BacktrackAlg . . . 539
xpress.BranchChoice . . . 540
xpress.CholeskyAlgorithm . . . 541
xpress.CrossoverDynamicReduction . . . 542
xpress.DualGradient . . . 543
xpress.DualStrategy . . . 544
xpress.FeasibilityPump . . . 545
xpress.HeuristicSearchSelect . . . 546
xpress.NodeSelectionCriteria . . . 547
xpress.OutputDetail . . . 548

xpress.PreProbing . . . 549
xpress.PresolveOperations . . . 550
xpress.PresolveState . . . 551
xpress.MipPresolve . . . 552
xpress.Presolve . . . 553
xpress.Pricing . . . 554
xpress.CutStrategy . . . 555
xpress.VariableSelection . . . 556
xpress.Scaling . . . 557
xpress.CutSelect . . . 558
xpress.RefineOps . . . 559
xpress.DualizeOps . . . 560
xpress.TreeDiagnostics . . . 561
xpress.BarPresolveOps . . . 562
xpress.MipRestart . . . 563
xpress.PresolveCoefElim . . . 564
xpress.PresolveDomRow . . . 565
xpress.PresolveDomColumn . . . 566
xpress.PrimalUnshift . . . 567
xpress.RepairIndefiniteQuadratic . . . 568
xpress.ObjSense . . . 569
xpress.ParameterType . . . 570
xpress.QConvexity . . . 571
xpress.SolInfo . . . 572
xpress.TunerMode . . . 573
xpress.TunerMethod . . . 574
xpress.TunerTarget . . . 575
xpress.TunerHistory . . . 576
xpress.TunerRootAlg . . . 577
xpress.LPFlags . . . 578
xpress.GenConsType . . . 579
xpress.Clamping . . . 580
xpress.RowFlag . . . 581
xpress.ObjControl . . . 582
xpress.AllowCompute . . . 583
xpress.ComputeLog . . . 584
xpress.Namespaces . . . 585
xpress.Globalboundingbox . . . 586
xpress.MultiObjOps . . . 587
xpress.IISOps . . . 588
xpress.UserSolStatus . . . 589
xpress.GlobalLSHEURStrategy . . . 590
xpress.BARHGOps . . . 591
xpress.BasisStatus . . . 592
xpress.SDPCutStrategy . . . 593
xpress.BarHGPrecision . . . 594
xpress.MIPStopStage . . . 595
xpress.constants . . . 596
8.14 Deprecated functionality . . . 603
problem.addcbbariteration . . . 614
problem.addcbbarlog . . . 616
problem.addcbchecktime . . . 617
problem.addcbchgbranchobject . . . 618
problem.addcbcutlog . . . 619
problem.addcbcutround . . . 620

problem.addcbdestroymt . . . 621
problem.addcbgapnotify . . . 622
problem.addcbmiplog . . . 624
problem.addcbinfnode . . . 625
problem.addcbintsol . . . 626
problem.addcblplog . . . 627
problem.addcbmessage . . . 628
problem.addcbmipthread . . . 629
problem.addcbnewnode . . . 630
problem.addcbnodecutoff . . . 631
problem.addcbnodelpsolved . . . 632
problem.addcboptnode . . . 633
problem.addcbpreintsol . . . 634
problem.addcbprenode . . . 635
problem.addcbusersolnotify . . . 636
problem.addcbbeforeobjective . . . 637
problem.addcbafterobjective . . . 638
problem.addcbpresolve . . . 639
problem.addcoefs . . . 640
problem.addcols . . . 642
problem.addcuts . . . 643
problem.adddfs . . . 644
problem.addgencons . . . 645
problem.addmipsol . . . 646
problem.addnames . . . 647
problem.addobj . . . 648
problem.addpwlcons . . . 649
problem.addqmatrix . . . 650
problem.addrows . . . 651
problem.addsetnames . . . 652
problem.addtolsets . . . 653
problem.addvars . . . 654
problem.basisstability . . . 655
problem.bndsa . . . 656
problem.btran . . . 657
problem.calcobjn . . . 658
problem.calcobjective . . . 659
problem.calcreducedcosts . . . 660
problem.calcslacks . . . 661
problem.calcsolinfo . . . 662
problem.cascade . . . 663
problem.cascadeorder . . . 664
problem.chgbounds . . . 665
problem.chgcoef . . . 666
problem.chgcoltype . . . 667
problem.chgcascadenlimit . . . 668
problem.setcbcascadeend . . . 669
problem.setcbcascadestart . . . 670
problem.setcbcascadevar . . . 671
problem.setcbcascadevarfail . . . 672
problem.setcbcoefevalerror . . . 673
problem.setcbconstruct . . . 674
problem.setcbdestroy . . . 675
problem.setcbdrcol . . . 676
problem.setcbintsol . . . 677

problem.setcbiterend . . . 678
problem.setcbiterstart . . . 679
problem.setcbitervar . . . 680
problem.setcbmessage . . . 681
problem.setcbmsjobend . . . 682
problem.setcbmsjobstart . . . 683
problem.setcbmswinner . . . 684
problem.setcboptnode . . . 685
problem.setcbprenode . . . 686
problem.setcbpreupdatelinearization . . . 687
problem.setcbslpend . . . 688
problem.setcbslpnode . . . 689
problem.setcbslpstart . . . 690
problem.slpchgcoefstr . . . 691
problem.chgccoef . . . 692
problem.chgdeltatype . . . 693
problem.chgdf . . . 694
problem.chgglblimit . . . 695
problem.chgmcoef . . . 696
problem.chgobjn . . . 697
problem.chgmqobj . . . 698
problem.chgnlcoef . . . 699
problem.slpchgcoef . . . 700
problem.chgobj . . . 701
problem.chgobjsense . . . 702
problem.chgqobj . . . 703
problem.chgqrowcoeff . . . 704
problem.chgrhs . . . 705
problem.chgrhsrange . . . 706
problem.chgrowstatus . . . 707
problem.chgrowtype . . . 708
problem.chgrowwt . . . 709
problem.chgtolset . . . 710
problem.chgvar . . . 711
problem.construct . . . 712
problem.copycontrols . . . 713
problem.crossoverlpsol . . . 714
problem.delcoefs . . . 715
problem.delcpcuts . . . 716
problem.delcuts . . . 717
problem.delgencons . . . 718
problem.delindicators . . . 719
problem.delpwlcons . . . 720
problem.delobj . . . 721
problem.delqmatrix . . . 722
problem.deltolsets . . . 723
problem.delvars . . . 724
problem.dumpcontrols . . . 725
problem.estimaterowdualranges . . . 726
problem.evaluatecoef . . . 727
problem.evaluateformula . . . 728
problem.fixmipentities . . . 729
problem.fixpenalties . . . 730
problem.ftran . . . 731
problem.getattribinfo . . . 732

problem.getbasis . . . 733
problem.getbasisval . . . 734
problem.getccoef . . . 735
problem.getcoef . . . 736
problem.getcoefformula . . . 737
problem.getcoefs . . . 738
problem.getcolinfo . . . 739
problem.getcols . . . 740
problem.getcoltype . . . 741
problem.getcontrolinfo . . . 742
problem.getcpcutlist . . . 743
problem.getcpcuts . . . 744
problem.getcutlist . . . 745
problem.getcutmap . . . 746
problem.getcutslack . . . 747
problem.getdirs . . . 748
problem.getdf . . . 749
problem.getDual . . . 750
problem.getdualray . . . 751
problem.getgencons . . . 752
problem.getmipentities . . . 753
problem.getiisdata . . . 754
problem.getIndexFromName . . . 756
problem.getindicators . . . 757
problem.getinfeas . . . 758
problem.getlastbarsol . . . 759
problem.getlasterror . . . 760
problem.getlb . . . 761
problem.getlpsol . . . 762
problem.getlpsolval . . . 763
problem.getmessagestatus . . . 764
problem.getmipsol . . . 765
problem.getmipsolval . . . 766
problem.getmqobj . . . 767
problem.getobjn . . . 768
problem.getnamelist . . . 769
problem.getobj . . . 770
problem.getObjVal . . . 771
problem.getpivotorder . . . 772
problem.getpivots . . . 773
problem.getpresolvebasis . . . 774
problem.getpresolvemap . . . 775
problem.getpresolvesol . . . 776
problem.getprimalray . . . 777
problem.getProbStatus . . . 778
problem.getProbStatusString . . . 779
problem.getpwlcons . . . 780
problem.getqobj . . . 781
problem.getqrowcoeff . . . 782
problem.getqrowqmatrix . . . 783
problem.getqrowqmatrixtriplets . . . 784
problem.getqrows . . . 785
problem.getRCost . . . 786
problem.getrhs . . . 787
problem.getrhsrange . . . 788

problem.getrowinfo . . . 789
problem.getrows . . . 790
problem.getrowstatus . . . 791
problem.getrowtype . . . 792
problem.getrowwt . . . 793
problem.getscaledinfeas . . . 794
problem.getSlack . . . 795
problem.getslpsol . . . 796
problem.gettolset . . . 797
problem.getub . . . 798
problem.getunbvec . . . 799
problem.getvar . . . 800
problem.hasdualray . . . 801
problem.hasprimalray . . . 802
problem.iisall . . . 803
problem.iisclear . . . 804
problem.iisfirst . . . 805
problem.iisisolations . . . 806
problem.iisnext . . . 807
problem.iisstatus . . . 808
problem.iiswrite . . . 809
problem.loadbasis . . . 810
problem.loadbranchdirs . . . 811
problem.loadcoefs . . . 812
problem.loadcuts . . . 814
problem.loaddelayedrows . . . 815
problem.loaddfs . . . 816
problem.loaddirs . . . 817
problem.loadlpsol . . . 818
problem.loadmipsol . . . 819
problem.loadmodelcuts . . . 820
problem.loadpresolvebasis . . . 821
problem.loadpresolvedirs . . . 822
problem.loadproblem . . . 823
problem.loadsecurevecs . . . 825
problem.loadtolsets . . . 826
problem.loadvars . . . 827
problem.lpoptimize . . . 828
problem.mipoptimize . . . 829
problem.msaddcustompreset . . . 830
problem.msaddjob . . . 831
problem.msaddpreset . . . 832
problem.msclear . . . 833
problem.name . . . 834
problem.nlpchgformula . . . 835
problem.nlpchgformulastr . . . 836
problem.nlpgetformula . . . 837
problem.slpgetcoefformula . . . 838
problem.nlpgetformulastr . . . 839
problem.nlpoptimize . . . 840
problem.nlpsetinitval . . . 841
problem.objsa . . . 842
problem.postsolve . . . 843
problem.presolve . . . 844
problem.presolverow . . . 845

problem.printmemory . . . 846
problem.printevalinfo . . . 847
problem.read . . . 848
problem.readbasis . . . 849
problem.readbinsol . . . 850
problem.readdirs . . . 851
problem.readslxsol . . . 852
problem.reinitialize . . . 853
problem.removecbbariteration . . . 854
problem.removecbbarlog . . . 855
problem.removecbchecktime . . . 856
problem.removecbchgbranchobject . . . 857
problem.removecbcutlog . . . 858
problem.removecbcutround . . . 859
problem.removecbdestroymt . . . 860
problem.removecbgapnotify . . . 861
problem.removecbmiplog . . . 862
problem.removecbinfnode . . . 863
problem.removecbintsol . . . 864
problem.removecblplog . . . 865
problem.removecbmessage . . . 866
problem.removecbmipthread . . . 867
problem.removecbnewnode . . . 868
problem.removecbnodecutoff . . . 869
problem.removecbnodelpsolved . . . 870
problem.removecboptnode . . . 871
problem.removecbpreintsol . . . 872
problem.removecbprenode . . . 873
problem.removecbusersolnotify . . . 874
problem.removecbbeforeobjective . . . 875
problem.removecbafterobjective . . . 876
problem.removecbpresolve . . . 877
problem.repairinfeas . . . 878
problem.repairweightedinfeas . . . 880
problem.repairweightedinfeasbounds . . . 882
problem.rhssa . . . 884
problem.scaling . . . 885
problem.setcurrentiv . . . 886
problem.setdefaultcontrol . . . 887
problem.setdefaults . . . 888
problem.setindicators . . . 889
problem.setlogfile . . . 890
problem.setmessagestatus . . . 891
problem.setprobname . . . 892
problem.slpgetcoefstr . . . 893
problem.storecuts . . . 894
problem.strongbranch . . . 895
problem.strongbranchcb . . . 896
problem.tuneprobsetfile . . . 897
problem.tunerreadmethod . . . 898
problem.tunerwritemethod . . . 899
problem.unconstruct . . . 900
problem.updatelinearization . . . 901
problem.validate . . . 902
problem.validatekkt . . . 903

problem.validaterow . . . 904
problem.validatevector . . . 905
problem.write . . . 906
problem.writebasis . . . 907
problem.writebinsol . . . 908
problem.writedirs . . . 909
problem.writeprtsol . . . 910
problem.writeslxsol . . . 911
problem.writesol . . . 912
branchobj.addbounds . . . 913
branchobj.addbranches . . . 914
branchobj.addcuts . . . 915
branchobj.addrows . . . 916
branchobj.getbounds . . . 917
branchobj.getbranches . . . 918
branchobj.getid . . . 919
branchobj.getlasterror . . . 920
branchobj.getrows . . . 921
branchobj.setpreferredbranch . . . 922
branchobj.setpriority . . . 923
xpress.addcbmsghandler . . . 924
xpress.featurequery . . . 925
xpress.getbanner . . . 926
xpress.getcomputeallowed . . . 927
xpress.getcheckedmode . . . 928
xpress.getdaysleft . . . 929
xpress.getlasterror . . . 930
xpress.getlicerrmsg . . . 931
xpress.getversion . . . 932
xpress.getversionnumbers . . . 933
xpress.is_equal . . . 934
xpress.removecbmsghandler . . . 935
xpress.setarchconsistency . . . 936
xpress.setcomputeallowed . . . 937
xpress.setcheckedmode . . . 938
xpress.setdefaultcontrol . . . 939
xpress.setdefaults . . . 940
xpress.vars . . . 941

Appendix 943

A Contacting FICO 943

FICO Customer Support . . . 943
Documentation . . . 943
FICO Learning . . . 944
Sales and maintenance . . . 944
About FICO . . . 944

Index 945

CHAPTER 1

Introduction

The Xpress Python interface allows for creating and solving optimization problems using the Python® programming language and the FICO Xpress Optimizer library. This manual describes how to use the Xpress Python interface.

1.1 Outline

The following chapters cover:

- Creating, handling, solving, and querying optimization problems (Chapter 2);
- Using Python numerical libraries such as NumPy to create optimization problems (Chapter 3);
- Setting and getting the value of parameters (controls and attributes) of a problem (Chapter 4);
- Using Python functions as callbacks for the Xpress Optimizer and the Xpress Nonlinear solver (Chapter 5);
- Several examples of usage of the Xpress Python interface (Chapter 6);
- A reference with all functions and parameters in the Python interface (Chapter 8).

It is assumed here that the reader has basic understanding of the Python programming language. Ample documentation on Python is available at http://docs.python.org, including a tutorial and a reference manual. Unless specified otherwise, Python 3 is used in all of the examples and code samples throughout this manual. The current version of the Xpress Python interface works on Python 3.10 to 3.14.

Other components of the FICO-Xpress Optimization suite can interface with Python, albeit not the same Python versions. The Mosel module `python3`, for example, works with Python 3.5 or later. See the Mosel Language Reference Manual for specifics, and more in general the Xpress Insight Installation Guide, Appendix A: Supported Platforms for information on Python support.

”Python” is a registered trademark of the Python Software Foundation.

1.2 Installing the Python Xpress module

The Xpress Python module can be installed from the two main Python repositories: The Python Package Index (PyPI) and the Conda repository. Installing the Xpress Python interface does *not* require one to install the whole Xpress suite, as all necessary libraries are provided.

Xpress 9.9 will be the last release to be published on Anaconda. Subsequent releases of Xpress will be published on PyPI only.

The installation comes with a copy of the *community* license, which allows for solving linear and mixed-integer problems with up to 5000 total variables and constraints, and quadratic and general nonlinear problems with up to 200 variables and constraints. In addition, the number of nonlinear tokens (measure of the complexity for nonlinear expressions) is restricted to 1000, and the number of user functions (black-box optimization) is restricted to 1.

If you already have an Xpress license, please make sure to set the XPAUTH_PATH environment variable to the full path to the license file, `xpauth.xpr`. See also Section 1.2.5 below.

1.2.1 Installation from the Python Package Index (PyPI)

The Xpress Python interface is available on the PyPI server and can be installed with the following command:

```
pip install xpress
```

Packages for Python 3.10 to 3.14 are available, for Windows, Linux, and MacOS. The package contains the Python interface module, its documentation in PDF format, the Xpress Optimizer's libraries, various examples of use, and a copy of the community license (see https://www.fico.com/en/fico-xpress-community-license). Online documentation can be viewed at the FICO Xpress Optimization Help page.

Packages are also available for free-threading Python 3.14 in beta status. The Xpress Python interface does not yet support parallel threads, so Xpress will re-enable the GIL when it is imported. For more information about free- threading Python, see https://docs.python.org/3/howto/free-threading-python.html.

The above command installs the latest version of the Xpress Python module. Earlier versions of the module can be installed by appending a "==VERSION" string to the module name, for instance

```
pip install xpress==9.8.0
```

1.2.2 Installation from Conda

It is recommended to install the PyPI package in your Conda environment, using the `pip` command described above. Alternatively, you can specify the PyPI package in your Conda `environment.yaml` file as demonstrated below:

```
dependencies:
  - python==3.14.*
  - numpy<3
  - pip:
    - xpress==9.9.0
```

A Conda package is available for download with the following command:

```
conda install -c fico-xpress xpress
```

The content of the Conda package is the same as that of the PyPI package. Similar to the PyPI package, Conda packages for Python 3.10 to 3.14 are available, for Windows, Linux, and MacOS, but they are not available for free-threaded Python. Similar to PyPI, the Conda installer fetches the latest version of the package but allows for installing earlier versions as in the following example (note that the Conda installer only uses a single "="):

```
conda install -c fico-xpress xpress=9.8.0
```

Note that the Xpress Conda package requires the 'intel-openmp' package on Intel platforms, which is

available on the ‘main‘ and ‘intel‘ Conda channels.

1.2.3 Installation from a package file

Customers can download selected older versions of the Xpress Conda and PyPI packages from the Xpress client area site by clicking on the Archived Downloads link. Package files downloaded from this site can be installed with the following commands:

```
conda install /path/to/xpress-package-file.tar.bz2  # Conda
pip install /path/to/xpress-package-file.whl        # PyPI
```

1.2.4 GPU installation guidelines for PDHG

The primal-dual hybrid gradient (PDHG) linear optimization solver can now take advantage of a NVIDIA® CUDA®-capable GPU, if present. The GPU support for PDHG is available as a beta release with Xpress 9.9. The following platforms have been tested: Linux (both x86_64 and ARM64) and Windows (x86_64).

Software requirements:

- At least version 580 of the NVIDIA drivers must be installed. The latest version is available from https://www.nvidia.com/drivers.
- At least version 13.0 of the NVIDIA CUDA Runtime must be installed. The CUDA Toolkit, which includes the Runtime, can be downloaded from https://developer.nvidia.com/cuda-downloads.

When installing the Xpress Python package, the required CUDA Runtime dependencies can also be installed from PyPI or Conda:

- When installing Xpress from PyPI, the CUDA Runtime can be specified as an optional dependency:

```
pip install xpress[cuda]
```

- When installing Xpress from Conda, the CUDA Runtime can be installed using the following command:

```
conda install -c nvidia cuda-cudart libcusparse libcublas
```

1.2.5 Troubleshooting the installation

Whether the Xpress Python module is downloaded from PyPI or from the Conda server, there are a few remarks that might help ensure that the installation works right away. The advice below is independent of the Python platform (PyCharm, Spyder, etc.) that may be in use.

The Xpress Python interface uses the Python package NumPy for some operation, hence NumPy must be installed. It is usually installed if a Conda installation is used, nevertheless ensure that a recent-enough version is installed.

After installation, a license is not strictly necessary as the embedded Community license is used. If you already have a license (for example, a trial license, a full license, or one from the Academic Partnership Program), you can set the `XPAUTH_PATH` environment variable to the full path to the license file. For example, if the license file is `/home/brian/xpauth.xpr`, then `XPAUTH_PATH` should be set to `/home/brian/xpauth.xpr` in order for the module to pick the right license.

If you installed the Xpress Optimization suite before downloading the Xpress Conda or PyPI package, the Xpress Python interface will try to use the license file your Xpress installation automatically:

- On Windows, the Xpress installer sets the `XPRESSDIR` environment variable to the installation directory, and the Xpress Python interface will look for a license file at `%XPRESSDIR%\bin\xpauth.xpr`.
- On Linux and MacOS, the Xpress installer creates a script named `xpvars.sh` in the `bin` folder of the Xpress installation. This script sets `XPRESSDIR` to the installation directory, and sets `XPAUTH_PATH` to the location of the license file. If `xpvars.sh` has been properly sourced into the shell environment where Python is executed, the Xpress Python interface will use this `XPAUTH_PATH` value to locate the license from your Xpress installation. If for some reason `XPAUTH_PATH` is not set, the Xpress Python interface will look for a license file at `$XPRESSDIR/bin/xpauth.xpr`.

If you do not want to use the license file from your Xpress installation, you can override this behaviour by setting the `XPAUTH_PATH` environment variable to the full path to the license file that you want to use.

CHAPTER 2

Modeling an optimization problem

This chapter illustrates the modeling capabilities of the Xpress Python interface. It shows how to create variables, constraints of different types, add an objective function, and solving and retrieving a problem's solution. It also shows how to read or write a problem from/to a file.

2.1 Getting started

The Xpress Python module is imported as follows:

```
import xpress
```

A complete list of methods and constants available in the module is obtained by running the Python command `dir(xpress)`. Because all types and methods must be called by prepending "`xpress.`", it is advisable to alias the module name upon import:

```
import xpress as xp
```

We assume that this is the way the module is imported from now on. It is also possible to import all methods and types to avoid prepending the module name or its alias, but this practice is usually advised against:

```
from xpress import *
```

2.2 Creating a problem

Create an empty optimization problem `p` as follows:

```
p = xp.problem()
```

A name can be assigned to a problem upon creation:

```
p = xp.problem(name="My first problem")
```

2.3 Variables

The `problem.addVariable` function creates an optimization variable within the problem. All parameters are optional.

```
p.addVariable(name, lb, ub, threshold, vartype)
```

The parameters are:

1. `name` is a Python UTF-8 string containing the name of the variable (its ASCII version will be saved if written onto a file); a default name is assigned if the user does not specify it;
2. `lb` is the lower bound (0 by default);
3. `ub` is the upper bound (+inf is the default);
4. `threshold` is the threshold for semi-continuous, semi-integer, and partially integer variables; it must be between its lower and its upper bound; it has no default, so if a variable is defined as partially integer the threshold must be specified;
5. `vartype` is the variable type, one of the six following types:
   - `xpress.continuous` for continuous variables;
   - `xpress.binary` for binary variables (lower and upper bound are further restricted to 0 and 1);
   - `xpress.integer` for integer variables;
   - `xpress.semicontinuous` for semi-continuous variables;
   - `xpress.semiinteger` for semi-integer variables;
   - `xpress.partiallyinteger` for partially integer variables.

The features of each variable are accessible as members of the associated object: after declaring a variable with `x = prob.addVariable()`, its name, lower and upper bound can be accessed via `x.name`, `x.lb`, and `x.ub`.

By default, variables added to an Xpress problems are constrained to be nonnegative. In order to add a *free* variable, one must specify its lower bound to be $-\infty$ as follows:

```
v = p.addVariable(lb=-xp.infinity)
```

2.3.1 Variable names and Python objects

Variables and, as described below, constraints and other objects of the Xpress Python interface can have a name. Variable names and constraint names can be useful when saving a problem to a file and when querying the problem for the value of a variable in an optimal solution. If a variable is not given a name explicitly, it will be assigned a default name that is usually ”C” followed by a sequence number.

Python also uses these names when printing expressions, because the variables' `__str__` function is redirected to their name. Therefore, when querying Python for a variable or for an expression containing that variable, its name will be printed rather than the Python object used in the program, as in the following example:

```
>>> v = p.addVariable(lb=-1, ub=2)
>>> v
C1
>>> v.__str__()
'C1'
>>> x = p.addVariable(name='myvar')
>>> v + 2 * x
C1 + 2 myvar
>>>
```

This allows for querying a problem using both the variable object and its name, depending on what is more convenient. The following example prints twice an optimal solution to a simple problem:

```
import xpress as xp
p = xp.problem()
x = p.addVariable(name='var1')
```

```
y = p.addVariable(name='var2')
p.addConstraint(x + y >= 3)
p.setObjective(x + 2*y)
p.optimize()
print(p.getSolution([x, y]))
print(p.getSolution(['var1', 'var2']))
```

It can be therefore useful to create `xpress.var` objects with a meaningful `argument`, perhaps similar to the name they have in the Python program one is writing.

2.4 Constraints

Linear, quadratic, and nonlinear constraints can be specified as follows:

```
constraint (constraint, body, lb, ub, sense, rhs, name)
```

The parameters are:

1. `constraint` is the full-form constraint, such as `x1 + 2 * x2 <= 4`;
2. `body` is the body of the constraint, such as `3 * x1 + x2` (it may contain constants);
3. `lb` is the lower bound on the body of the constraint;
4. `ub` is the upper bound on the body of the constraint;
5. `sense` is the sense of the constraint, one among `xpress.leq`, `xpress.geq`, `xpress.eq`, and `xpress.rng`; in the first three cases, the parameter `rhs` must be specified; only in the fourth case must `lb` and `ub` be specified;
6. `rhs` is the right-hand side of the constraint;
7. `name` is the name of the constraint. Parameters `lb`, `ub`, and `rhs` must be constant.

A much more natural way to formulate a constraint is possible though:

```
myconstr = x1 + x2 * (x2 + 1) <= 4
myconstr2 = xp.exp(xp.sin(x1)) + x2 * (x2**5 + 1) <= 4
```

One or more constraints (or list of constraints) can be added to a problem via the `addConstraint` method:

```
p.addConstraint(myconstr, myconstr2)
p.addConstraint(v1 + xp.tan(v2) <= 3)
p.addConstraint(x[i] + y[i] <= 2 for i in range(10))
```

In order to help formulate compact problems, the `Sum` operator of the `xpress` module can be used to express sums of expressions. Its argument is a list of expressions:

```
p.addConstraint(xp.Sum([y[i] for i in range(10)]) <= 1)
p.addConstraint(xp.Sum([x[i]**5 for i in range(9)]) <= x[9])
```

When handling variables or expressions, it is advised to use the `Sum` operator in the Xpress module rather than the native Python operator, for reasons of efficiency.

As for variables, an object of type `constraint` allows for read/write access of its features via its members `name`, `body`, `lb`, and `ub`. The same caveat for variables holds here: any change to an object's members will only have an effect in the problems to which a constraint is added after the change.

A set of variables or constraint can also be created using Python's fundamental data structure: lists and dictionaries, as well as NumPy's arrays. As described in Section 2.15 below, one can for example create a list of variables `x[i]`, all with upper bound 10, indexed from 0 to `k-1` as follows:

```
k=24
x = [p.addVariable(ub=10) for _ in range(k)]
```

If a more elaborate indexing is required, dictionaries can be used. Suppose we want to create an integer variable `x` for each item in the list `['Seattle','Miami','Omaha','Charleston']`. Then

```
L = ['Seattle','Miami','Omaha','Charleston']
x = {i: p.addVariable(vartype=xpress.integer) for i in L}
```

This allows one to refer to such variables using the names in `L`, for instance `x['Seattle']`, `x['Charleston']`, etc.

Similarly, one can use lists and dictionaries to create constraints, like in the following example on lists:

```
import xpress as xp
p = xp.problem()
L = range(20)
x = [p.addVariable(ub=1) for i in L]
y = [p.addVariable(vartype=xp.binary) for i in L]
constr = [x[i] <= y[i] for i in L]
p.addConstraint(constr)
```

Below is an example with dictionaries. Note that Python allows for conditional indexing on the two parameters `i` and `j`, and each constraint can be referred to with pairs of names, e.g. `cliq['Seattle','Miami']`.

```
import xpress as xp
p = xp.problem()
L = ['Seattle','Miami','Omaha','Charleston']
x = {i: p.addVariable(vartype=xp.binary) for i in L}
cliq = {(i,j): x[i] + x[j] <= 1 for i in L for j in L if i != j}
p.addConstraint(cliq)
```

There is yet another function for creating an indexed set of variables: the function `problem.addVariables`. It takes one or more lists, sets, or ranges, and produces as many variables as can be indexed with all combinations from the provided lists/sets. This allows for creating a set of variables with the same bounds and type and a similar name, in case the problem is written onto an MPS or LP file. Its syntax is as follows:

```
p.addVariables(*indices, name='x', lb=0, ub=xp.infinity,
               threshold = -xp.infinity, vartype=xp.continuous)
```

The parameter `*indices` stands for one or more arguments, each a Python list, a Python set, or a positive integer. If `*indices` consists of one list, then the result contains one element for each element of the list. In case of more lists, sets, or ranges in `*indices`, the Cartesian product of these lists/sets provides the indexing space of the result. All other arguments are the same as for the declaration of a single variable. Here is an example of use:

```
myvar = p.addVariables(['a','b','c'], lb=-1, ub=+1)
```

The result is the three variables `myvar['a']`, `myvar['b']`, and `myvar['c']`, all with -1 as lower bound and +1 as upper bound. The following is an example of multi-indexed variables:

```
y = p.addVariables(['a','b','c','d'], [100, 120, 150], vartype=xp.integer)
```

The result is the 12 variables `y['a',100]`,`y['a',120]`,`y['a',150]`,`y['b',100]`,..., `y['d',150]`.

If argument `name` is not specified, a prefix "x" is used. The name of each variable resulting from a call to `problem.addVariables` is the given prefix and the comma-separated list of index values between brackets, for example it will be "x(a,100)", "x(a,120)", "x(a,150)" for the example above. The call

```
x = p.addVariables(['a','b','c','d'], [100, 120, 150], name='var')
```

produces variables `x['a',100]` whose name is "var(a,120)", etc.

In the `*indices` argument, in lieu of a list or a set one can also specify an integer positive number `k`, which is interpreted as the range of numbers `0,1,...,k-1`. Thus the call `x = p.addVariables(5, 7, vartype = xp.integer)` creates 35 variables `x[0,0]`,`x[0,1]`,`x[0,2]`,...,`x[4,6]`.

The `problem.addVariables` function, effectively, is a more readable way to create a Python dictionary of variables. The instruction

```
x = p.addVariables(['a','b','c','d'], [100, 120, 150], ub=20, name='newvar')
```

is equivalent to the following:

```
x = {(i,j): p.addVariable(ub=20, name='newvar({0},{1})'.format(i,j))
           for i in ['a','b','c','d']
           for j in [100, 120, 150]}
```

2.5 Objective function

The objective function is any expression, so it can be constructed as for constraints. The method `problem.setObjective` can be used to set (or replace if one has been specified before) the objective function of a problem. The definition of `setObjective` is as follows:

```
setObjective(objective, sense=xp.ObjSense.MINIMIZE)
```

where `objective` is the expression defining the new objective and `sense` is either `xpress.ObjSense.MINIMIZE` or `xpress.ObjSense.MAXIMIZE`. Examples follow; in the first, the objective function is to be minimized as per default, while the second example specifies the optimization sense as maximization.

```
p.setObjective(xp.Sum ([y[i]**2 for i in range (10)]))
p.setObjective (v1 + 3 * v2, sense=xp.ObjSense.MAXIMIZE)
```

Finally, a note on efficiency. For creating a large number of variables, one can obtain a Numpy multiarray of any dimension by just specifying numbers as the index arguments, as in the following example where a 4x7x5 multiarray of variables is created:

```
x = p.addVariables(4,7,5)
```

For added efficiency, one can drop variable naming if standard names (such as "C1", "C2", "C3") are acceptable. This is done by specifying the argument `name=""` as in the example below.

```
x = p.addVariables(4,7,5, name="")
```

2.6 Special Ordered Sets (SOSs)

A Special Order Set (SOS) is a modeling tool for constraining a small number of consecutive variables in a list to be nonzero. The `problem.addSOS` function can be used to add a SOS as follows:

```
p.addSOS(indices, weights, type, name)
```

The first argument, `indices`, is a list of variables, while `weights` is a list of floating point numbers. The type of SOS (either 1 or 2) is specified by `type`. While `indices` and `weights` are mandatory parameters, `type` and `name` are not; `type` is set to a default of 1 when not specified. Examples follow:

```
set1 = p.addSOS(x, [0.5 + i*0.1 for i in range(10)], type=2)
set2 = p.addSOS([y[i] for i in range(5)], [i+1 for i in range(5)])
set3 = p.addSOS([v1, v2], [2, 5], 2)
```

The `name` member of a SOS object can be read and written by the user.

2.7 Indicator constraints

Indicator constraints are defined by a binary variable, called the *indicator*, and a constraint. Depending on the value of the indicator, the constraint is enforced or relaxed.

For instance, if the constraint $x + y \geqslant 3$ should only be enforced if the binary variable *u* is equal to 1, then $(u = 1 \rightarrow x + y \geqslant 3)$ is an indicator constraint.

An indicator constraint in Python can be added to a problem with the `addIndicator` as follows (note the "==" as the symbol for equality):

```
p.addIndicator(vb == 1, v1 + v2 >= 4)
```

2.8 Piecewise linear functions

Other types of constraints are available for modelling. *Piecewise linear* constraints allow to define a variable as a piecewise linear function of another. The function does not have to be continuous, but please see the Optimizer's manual for information on how discontinuities are dealt with.

The most efficient way to model piecewise linear functions is through the API function `problem.addPwlCons`.

```
import xpress as xp
p = xp.problem()
x = p.addVariable(lb=-xp.infinity)
y = p.addVariable()
z1 = p.addVariable(lb=-xp.infinity)
z2 = p.addVariable(lb=-xp.infinity)

# Define z1 and z2 as a piecewise linear functions of x. Two functions
# are defined.
p.addPwlCons([x, x],   # input variable of each function
             [z1, z2], # created variables
             [0,4],    # index of the first breakpoints for z1 and z2
             [0,4, 4  7, -2,-1,1,2],   # x values of the breakpoints
             [4,12,11,20,-2,-2,2,2])   # y values
p.setObjective(z1 + 2*y)
p.addConstraint(z2 <= y)
p.optimize()
```

The above example creates variables `x`, `y`, `z1`, and `z2`, then constrains `z1` and `z2` to be (piecewise

linear) functions of x, to be used with y in other constraints and in the objective function.

The Xpress Python interface provides another, more intuitive way of specifying such a function with the method `xpress.pwl`, which is passed a dictionary associating intervals (defined as tuples of two elements) with linear functions. The code below exemplifies the use of `xpress.pwl` to construct two functions. The first, which is included into the objective of the problem, is the piecewise linear function $2x + 4$ for $x \in [0, 4]$ and $3x - 1$ for $x \in [4, 7]$; the second function is constant at -2 for $x \leqslant -1$, it is equal to $2x$ for $x \in [-1, 1]$, and is constant at 2 for $x \geqslant 2$:

```
import xpress as xp
p = xp.problem()
x = p.addVariable(lb=-xp.infinity)
y = p.addVariable()

# Create objective and constraint directly, without first creating
# piecewise linear functions.

p.setObjective(xp.pwl({(0, 4): 2*x + 4, (4, 7): 3*x - 1}) + 2*y)
p.addConstraint(xp.pwl({(-xp.infinity, -1):  -2,
                        (-1, 1):            2*x,
                        (1, xp.infinity):     2}) <= y)
p.optimize()
```

Here the definition of auxiliary variables `z1` and `z2` becomes redundant as the calls to `xpress.pwl` do not need any extra variable. The dictionary that is used in `xpress.pwl` has tuples of two elements each as keys and linear expressions (or constants) as values.

The tuples are treated as (pairwise disjoint) intervals, hence every tuple `(a,b)` in the set of keys must be such that $a \leqslant b$ and such that, for any two tuples `(a,b)` and `(c,d)` in the keys, either $b \leqslant c$ or $d \leqslant a$.

Piecewise linear functions should be defined over the whole domain of the input variable (*x* in the example above); with the syntax of `xpress.pwl`, it is possible to omit a portion of the domain of the input variable; in that case the value of the function is taken to be zero.

Piecewise linear functions can be used as operators when defining an optimization problem. For instance, one could write the constraint

```
y + 3*z**2 <= 3*xp.pwl({(0, 1): x + 4, (1, 3): 1})
```

Note that regardless of how a piecewise linear constraint is formulated, there must always be only one input variable, i.e., the piecewise linear function is always univariate. In addition, piecewise *constant* functions need a further specification as a variable does not appear in the values: for this case, one can specify the key-value pair `None: x` as in the example below.

```
# Set a piecewise CONSTANT objective
p.setObjective(xp.pwl({(0, 1): 4, (1, 2): 1, (2,3): 3, None: x})
```

2.9 General constraints

The Xpress Python interface allows the user to use the mathematical operators `min`, `max`, `abs`, and the logical operators `and`, `or` without having to explicitly introduce extra variables. The Xpress Optimizer handles such operators by automatically reformulating them as MIP constraints. These constraints are called *general constraints* by the Optimizer's library.

The `min` (resp. `max`) general operators impose that a variable be the minimum (resp. maximum) of two or more variables in a list of arguments. The `abs` constraints link a variable y to another variable x so that $y = |x|$.

The `And` and `Or` operators express a *logical* link between two or more binary variables $x_1, x_2, ..., x_k$. The result of this function is itself a binary expression that can take on value 0 (false) or 1 (true).

The most efficient way, in terms of modelling speed, to formulate a model using the aforementioned operator is through the function `problem.addGenCons`, which adds a general constraint. In the following example, variables `y1`, `y2`, and `y3` are constrained to be, respectively, the maximum among the set `{x[0], x[1], 46}`, the absolute value of `x[3]`, and the logical *and* of `x[4]`, `x[5]`, and `x[6]`.

```
import xpress as xp
p = xp.problem()
x = [p.addVariable(vartype=xp.binary) for _ in range(7)]
y1 = p.addVariable(vartype=xp.binary)
y2 = p.addVariable(vartype=xp.binary)
y3 = p.addVariable(vartype=xp.binary)
type = [xp.GenConsType.MAX, xp.GenConsType.ABS, xp.GenConsType.AND]
resultant = [y1, y2, y3]
colstart = [0, 2, 3]
col = [x[0], x[1], x[3], x[4], x[5], x[6]]
valstart = [0,1,1]
val = [46]
p.addGenCons(type, resultant, colstart, col, valstart, val)
p.optimize()
```

A more intuitive way to create problems containing these operators is by using the methods `max`, `min`, `abs`, `And`, and `Or` of the `xpress` module.

```
import xpress as xp
p = xp.problem()
x = [p.addVariable() for _ in range(4)]
y1 = p.addVariable()
y2 = p.addVariable()
p.addConstraint(y1 == xp.max(x[0], x[1], 46))  # max() accepts a tuple of arguments
p.addConstraint(y2 == xp.abs(x[3]))
p.addConstraint(y3 == xp.And(x[4], x[5], x[6]))
p.optimize()
```

The methods `And` and `Or` can be replaced by the Python binary operators & and |, as in the following example

```
y = [p.addVariable(vartype=xp.binary) for _ in range(5)]

p.addConstraint((y[0] & y[1]) + (y[2] | y[3]) + 2*y[4] >= 2)
```

Note that `And` and `Or` have a capital initial as the lower-case correspondents are reserved Python keywords, and that the & and | operators have a lower precedence than arithmetic operators such as $+$ and should hence be used with parentheses.

We also point out that because the & and | operator have lower *operator precedence* in Python than other arithmetic operators ($+$, $*$, etc.) and even comparison operators ($\leqslant$, etc.), all uses of & and | should be enclosed in brackets. as shown in the examples above.

2.10 Using `load*` functions for efficiency

The high-level functions `problem.addConstraint` and `problem.addVariables` allow for efficient, concise, and understandable modeling of any optimization problem. An even faster way to create a problem is through the `problem.loadLP`, `problem.loadMIP`, `problem.loadQP`, `problem.loadMIQP`, `problem.loadQCQP`, `problem.loadMIQCQP` functions, which use a more direct interface to the Optimizer's libraries and are hence preferable with very large problems and when efficiency in model creation is necessary.

The corresponding function can be used to create problems with linear and/or quadratic constraints and objective(s), a linear and/or quadratic objective function, and with continuous and/or discrete

variables. Their syntax with default parameter values allows for specifying only the components of interest. We refer the reader to the functions pages in Chapter 8, and present here a few examples of usages. More examples are shown in Chapter 6.

The first example uses `loadMIQP` to create a problem similar to that created earlier in this chapter. We first write the problem using standard modeling functions:

```
import xpress as xp
p = xp.problem(name='myexample')
x = p.addVariable(vartype=xp.integer, name='x1', lb=-10, ub=10)
y = p.addVariable(name='x2')
p.setObjective(x**2 + 2*y)
p.addConstraint(x + 3*y <= 4)
p.addConstraint(7*x + 4*y >= 8)
```

The following code creates a problem with the same features, including variable names and their types

```
import xpress as xp
p = xp.problem()
p.loadMIQP(probname='myexample',   # problem name
           rowtype=['L', 'G'],     # constraint senses
           rhs=[4, 8],             # right-hand sides
           rng=None,               # no range rows
           objcoef=[0, 2],         # linear obj. coeff.
           start=[0, 2, 4],        # start pos. of all columns
           collen=None,            # unused
           rowind=[0, 1, 0, 1],    # row index in each column
           rowcoef=[1, 7, 3, 4],   # coefficients
           lb=[-10,0],             # variable lower bounds
           ub=[10,xp.infinity],    # variable upper bounds
           objqcol1=[0],           # quadratic obj. terms, column 1
           objqcol2=[0],           # quadratic obj. terms, column 2
           objqcoef=[2],           # quadratic obj. terms, coeff
           coltype=['I'],          # variable types
           entind=[0])             # index of integer variable
```

Apart from the intuitive lists `rowtype` (for constraint types: `'L'` for "lesser-than", `'G'` for "greater-than", `'E'` for "equal-to"), `rhs` (constraints' right-hand sides), `objcoef` (objective linear coefficients), `lb` and `ub` (variables' lower and upper bounds), a few parameters deserve some attention. The three lists `start`, `colind`, `rowcoef` describe the coefficient matrix: `colind` and `rowcoef` contain, respectively, the *row* indices and the coefficients, while `start` is a list of $n+1$ integers (where n is the number of variables, i.e., the size of `objcoef`, `lb`, and `ub`); `start[i]` indicates the position, within `colind` and `rowcoef`, of the indices and coefficients of the `i`-th column. The last element `start[n+1]` indicates the number of nonzeros in the matrix.

The following shows two equivalent knapsack problems, again created first using the high-level modeling routines and then the lower-level `loadMIP` function.

```
import xpress as xp
N = 6
p = xp.problem(name='knapsack')
x = [p.addVariable(vartype=xp.binary) for _ in range(N)]
value = [1, 4, 6, 4, 7, 3]
weight = [1, 3, 5, 5, 8, 4]
p.setObjective(xp.Sum(value[i] * x[i] for i in range(N)), sense=xp.ObjSense.MAXIMIZE)
p.addConstraint(xp.Sum(weight[i] * x[i] for i in range(N)) <= 12)
```

Note that `problem.loadMIP` assumes that the optimization sense is minimization and hence a call to `problem.chgObjSense` is necessary to set the sense to maximization.

```
import xpress as xp
p = xp.problem()
N = 6
```

```
value = [1, 4, 6, 4, 7, 3]
weight = [1, 3, 5, 5, 8, 4]
p.loadMIP(probname='knapsack',     # problem name
            rowtype=['L'],         # constraint senses
            rhs=[12],              # right-hand sides
            rng=None,              # No range rows
            objcoef=value,         # linear obj. coeff.
            start=range(N+1),      # start pos. of all columns
            collen=None,           # (unused)
            rowind=[0] * N,        # row index in each column (always 0)
            rowcoef=weight,        # coefficients
            lb=[0] * N,            # variable lower bounds
            ub=[1] * N,            # variable upper bounds
            coltype=['B'] * N,     # variable types
            entind=range(N))       # indices of the N binary variables
p.chgObjSense(xp.ObjSense.MAXIMIZE)
```

2.11 Modeling and solving nonlinear problems

A nonlinear problem can be defined by creating one or more variables and then adding constraints and an objective function. This can be done using the same Python calls as one would do for other problems. The available operators are +, –, *, /, ** (which is the Python equivalent for the power operator, "^"). Univariate functions can also be used from the following list: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `exp`, `log`, `log10`, `abs`, `sign`, and `sqrt`. Multivariate functions are `min` and `max`, which can receive an arbitrary number of arguments.

Examples of nonlinear constraints are as follows:

```
import xpress as xp
import math

p = xp.problem()
x = p.addVariable()

# polynomial constraint
p.addConstraint(x**4 + 2 * x**2 - 5  >= 0)

# A terrible way to constrain x to be integer
p.addConstraint(xp.sin (math.pi * x) == 0)

p.addConstraint(x**2 * xp.sign (x) <= 4)
```

Note that non-native mathematical functions such as `log` and `sin` must be prefixed with `xpress` or its alias, `xp` in this case. This can be avoided by importing all symbols from xpress using the `import *` command as follows

```
from xpress import *
x = var()
a = sin(x)
```

but this hides namespaces and is usually frowned upon.

User functions are also accepted in the Python interface, and must be specified with the keyword `user` and the function as the first argument. They are handled in the Nonlinear solver in a transparent way, so all is needed is to define a Python function to be run as the user function and specify it in the problem with `user`, as in the following example:

```
import xpress as xp
import math

def mynorm(x1, x2):
  return (math.sqrt(x1**2 + x2**2) 2*x1, 2*x2)
```

```
def myfun(v1, v2, v3):
  return v1 / v2 + math.cos(v3)

p = xp.problem()
x, y = p.addVariable(), p.addVariable()

p.setObjective(xp.user (mynorm, x, y, derivatives=True))

p.addConstraint(x+y >= 2)
p.addConstraint(xp.user (myfun, x**2, x**3, 1/y) <= 3)
```

Note that user functions can be specified so that they can return derivatives. If we do not wish to return derivatives, a Python function in *k* variables must return a single number. If we want to provide the solver with derivatives, then the function must return a tuple of *k+1* numbers.

When defining a user function that provides derivatives (see `mynorm` in the example), we must set the `derivative=True` parameter in the `xpress.user` call. The `derivative` parameter is `False` by default. If a function returns a tuple of values but it is defined with `derivatives=False` or, viceversa, if it returns a single value but it is defined with `derivatives=True`, the behaviour is undefined.

As a final word of caution, solving nonlinear problem requires a preprocessing step that is transparent to the user except for two steps: first, if the objective function has a nonlinear component $f(x)$ then a new constraint (called *objective transfer row* or *objtransrow*) and a new variable, the *objective transfer column* or *objtranscol*) are called that are defined as follows:

$$\mathrm{objtransrow} : -\mathrm{objtranscol} + f(x) = 0$$

The resulting problem is equivalent in that the set of optimal (resp. feasible) solutions of this problem will be the same as those of the original problem. The user, however, will notice an increase by one of both the number of rows and of columns when a nonlinear objective function is set.

The second caveat is about yet another variable that may be added to the problem for reasons having to do with one of the Xpress Nonlinear solvers. This variable is called *equalscol* and it is fixed to 1. Its existence and value are therefore of no interest to the user.

It should also be noted that the control `xslp_postsolve` is set to 1 by default when the solver uses the SLP nonlinear solver. This is necessary to ensure that the solution retrieved after a `optimize()` or `nlpoptimize()` call refers to the original problem and not to a possible *reformulation*. The reader can find more information on this in the Xpress Nonlinear reference manual.

2.12 Solving a problem

Simply call `problem.optimize` to solve an optimization problem that was either built or read from a file. The type of solver is determined based on the type of problem: if at least one integer variable was declared, then the problem will be solved as a mixed integer (linear, quadratically constrained, or nonlinear) problem, while if all variables are continuous the problem is solved as a continuous optimization problem. If the problem is nonlinear in that it contains non-quadratic, non-conic nonlinear constraints, then the appropriate nonlinear solver of the Xpress Optimization suite will be called, either Xpress Global or Xpress Nonlinear, depending on available licenses. Note that non-convex quadratic problems are included in the base offering of the Xpress Optimizer license and will by default be solved with the Xpress Global technology.

```
p.optimize ()
```

The status of a problem after solving can be found via the `solvestatus` and `solstatus` attributes, and also in the return value of the optimize function, as follows:

```
import xpress as xp

p = xp.problem()
p.readProb("example3.lp")
solvestatus, solstatus = p.optimize()

if solvestatus == xp.SolveStatus.COMPLETED:
  print("Solve completed with solution status: ", solstatus.name)
else:
  print("Solve status: ", solvestatus.name)
```

The output of the solver when reading and solving a problem is the same as with other interfaces of the Xpress Optimizer. The verbosity level is determined by the control `outputlog`, which is 1 by default. To turn off the solver's output, it should be set to zero (see Chapter 4 for how to set a control).

2.13 Querying a problem

It is useful, after solving a problem, to obtain the value of an optimal solution. After solving a continuous or mixed integer problem, the two methods `problem.getSolution` and `problem.getSlacks` return the list (of portions thereof) of an optimal solution or the slack of the constraints, respectively. If an optimal solution was not found but a feasible solution is available, these methods will return data based on this solution.

Both `problem.getSolution` and `problem.getSlacks` can be used in multiple ways: if no argument is passed, the whole solution or slack list is returned. If a list of indices, variable/constraint objects, or names is passed, a list of values is returned corresponding to the range specified.

For `problem.getSolution`, there are more ways to call it: indices, strings, expressions are the basic types. An index `ind` will yield the value of the variable whose index in that problem (i.e. the order in which it was added to the problem) is `ind`; if the index is out of range, an error will occur. A string `str` will yield the value of the variable whose name is equal to `str`, if such variable exists, otherwise an error will occur. Finally, an expression, which can be just a variable, will yield the value of the expression given the current solution.

These basic types can be combined, even on multiple levels, with Python's fundamental aggregate types: `problem.getSolution` can be passed a list, a dictionary, a tuple, or any sequence, including NumPy arrays, of indices, strings, expressions, and other aggregate objects thereof. The result will have the same structure as the argument passed (list, dictionary, etc.) containing the value corresponding to the passed expressions, variable indices, or variable names.

The uses of `problem.getSolution` are exemplified in the following code:

```
import xpress as xp
import numpy as np

p = xp.problem()

v1 = p.addVariable(name='Var1')
x = [p.addVariable(lb=-1, ub=1, vartype=xp.integer) for i in range(10)]

[...] # add constraints and objective

p.optimize()

print(p.getSolution ())            # Prints a list with an optimal solution
print("v1 is", p.getSolution(v1))  # Only prints the value of v1
a = p.getSolution(x)               # Gets the values of all variables in the list x
b = p.getSolution(range(4))        # Gets the value of v1 and x[0], x[1], x[2], i.e.
                                   # the first four variables of the problem
c = p.getSolution('Var1')          # Gets the value of v1 by its name
d = p.getSolution({1: x, 2: 0,
                   3: 'Var1'})     # Returns a dictionary containing the same keys as
```

```
                                  # in the arguments and the values of the
                                  # variables/expressions passed
e = p.getSolution(v1 + 3*x)       # Gets the value of an expression under the
                                  # current solution
f = p.getSolution(np.array(x))    # Gets a NumPy array with the solution of x
```

Consider all lines after `p.optimize()`. The first of them returns a Python list of *ncol* floating point scalars, where *ncol* is the number of variables of the problem (*nrow* is the number of constraints, the size of the list returned by `problem.getSlacks`) containing the full solution. The second example retrieves the value of the single variable `v1`.

The third example returns an array of the same size as `x` with the value of all variables of the list `x`. The fourth example shows that a range of indices can be specified in order to obtain a list of values without specifying the corresponding variables. Recall that the column and row indices begin at 0. The fifth line shows that a variable can be passed by name, while the sixth line shows that passing a dictionary with variables, expression, indices, or variable names returns a dictionary with the same keys as the dictionary passed, but with its values set to the values of the corresponding variables/expressions.

The seventh line shows how to request the value of an expression when evaluated with the current solution found for the problem, and the eight line is equivalent to `p.getSolution(x)` but the returned object is a NumPy array with the solution (this can be useful when using NumPy with large vectors both for defining a problem and handling solution vectors).

The method `problem.getSlacks` works with indices, constraint names, constraint objects, and lists thereof. The following examples illustrate a few possible uses.

```
import xpress as xp
N = 10

p = xp.problem()
x = [p.addVariable(vartype=xp.binary) for i in range(N)]

con1 = xp.Sum(x[i] * i for i in range(N)) <= N)
con2 = (x[i] >= x[i+1] for i in range(N-1))

p.addConstraint(con1, con2)
p.setObjective(xp.Sum(x[i] for i in range(N))
p.optimize()

print(p.getSlacks())                  # prints a list of slacks for all N constraints
print("slack_1 is", p.getSlacks(con1)) # only prints the slack of con1

a = p.getSlacks(con2)                 # gets the slack of N-1 constraints con2 as a list of floats
b = p.getSlacks(range(2))             # gets the slack of con1 and con2[0]
```

In addition, for problems with only continuous variables, the two methods `problem.getDuals` and `problem.getRedCosts` return the list (or a portion thereof) of dual variables and reduced costs, respectively. Their usage is similar to that of `problem.getSlacks`.

Note that the inner workings of the Python interface obtain a copy of the *whole* solution, slack, dual, or reduced cost vectors, even if only one element is requested. It is therefore advisable that instead of repeated calls (for instance, in a loop) to `problem.getSolution`, `problem.getSlacks`, etc. only one call is made and the result is stored in a list to be consulted in the loop. Hence, in the following example:

```
import xpress as xp

n = 10000
N = range(n)

p = xp.problem()
x = [p.addVariable() for i in N]
```

```
p.addConstraint(xp.Sum(x[i] * i for i in N) <= n))
p.setObjective(xp.Sum(x[i] for i in N)
p.optimize()

for i in N:
    if p.getSolution(x[i]) > 1e-3:
        print(i)
```

the last three lines should be substituted as follows, as this will prevent repeatedly copying a large (10,000) list:

```
sol = p.getSolution()

for i in N:
    if sol[i] > 1e-3:
        print(i)
```

A very similar function of the class `problem` is `evaluate`, which allows for running all of the above evaluation functions while passing, rather than the solution currently available for the problem, any list or any dictionary assigning a `float` to the variables used in the expressions.

2.14 Reading and writing a problem

After creating an empty problem, one can read a problem from a file via the `read` method, which only takes the file name as its argument. An already-built problem can be written to a file with the `write` method. Its arguments are similar to those in the Xpress Optimizer API function `XPRSwriteprob`, to which we refer.

```
import xpress as xp

p = xp.problem()
p.readProb("example2.lp")
p.optimize()

print(p.getSolution())

p2 = xp.problem()
v1 = p.addVariable()
v2 = p.addVariable(vartype=xp.integer)

p2.addConstraint(v1 + v2 <= 4)
p2.setObjective(v1**2 + v2)

p2.writeProb("twovarsproblem", "lp")
```

2.15 Hints for building models efficiently

The Xpress Python interface allows for creating optimization models using methods described in this and other sections. As happens with other interpreted languages, using explicit loops may result in a slow Python script. When using the Xpress Python interface, this can be noticeable in large optimization models if multiple calls to `addVariable`, `addConstraint`, or `addSOS` are made. For this reason, the Xpress module allows for *generators* and list, dictionaries, and sequences as arguments to many methods, to ensure faster execution.

Let us consider an example:

```
import xpress as xp

N = 100000
S = range(N)
```

```
x = [p.addVariable()                  for i in S]
y = [p.addVariable(vartype=xp.binary) for i in S]

for i in S:
  p.addConstraint(x[i] <= y[i])

p.optimize()
```

Consider the loop which makes `N` calls to `addConstraint`. This method adds some overhead due to the conversion of Python objects into data that can be read by the Optimizer, and the total overhead can be large.

Most methods of the Xpress Python interface allow for passing sequences (lists, dictionaries, NumPy arrays, etc.) as parameters, and are automatically recognized as such. Hence the loop can be replaced with a single call to `addConstraint`:

```
p.addConstraint(x[i] <= y[i] for i in S)
```

This is much faster and more elegant.

The example also contains `2N` calls to `addVariable`, which can also result in a large overhead. This can be made more efficient (and compact) by creating `x` and `y` using `problem.addVariables`, which returns a NumPy array of variables:

```
import xpress as xp
import numpy as np

N = 100000

x = p.addVariables(N)
y = p.addVariables(N, vartype=xp.binary)

p.addConstraint(x <= y)

p.optimize()
```

See Chapter 3 for more information on how to use NumPy arrays in the Xpress Python interface.

2.16 Exceptions

The Xpress Python interface raises its own exceptions in the event of a modeling, interface, or solver issue. There are three types of exceptions:

- `xpress.ModelError`: raised in case of an issue in modelling a problem, for instance if an incorrect constraint sign is given or if a problem is amended an object that is neither a variable, a constraint, or a SOS;
- `xpress.InterfaceError`: raised when the issue can be ascribed to the API and the way it is used, for instance when not passing mandatory arguments or specifying incorrect ones in an API function;
- `xpress.SolverError`: raised when the Xpress Optimizer or Xpress-SLP returns an error that is given by the solver even though the model was specified correctly and the interface functions were used correctly.

As always with Python, one can use the `try/except` construct in order to analyze the raised exception as in the following example

```
import xpress as xp
p = xp.problem()
c = makeConstraint() # assume makeConstraint is defined elsewhere
try:
  p.addConstraint(c)
except xp.ModelError as e:
  print ("Modeling error:", repr(e))
```

2.17 Warnings

The Xpress Python interface can emit the following warnings:

- `xpress.DeprecationWarning`: emitted if the Xpress interface is used in a way which is deprecated and which will become unsupported in a future release.
- `xpress.LicenseWarning`: emitted if the path to the license file is not explicitly indicated with the `lic_path` argument of `xpress.init` or the `XPAUTH_PATH` environment variable.

Warnings can be suppressed using the Python `warnings` module:

```
import warnings
warnings.filterwarnings('ignore', 'Deprecated in Xpress 9.5')
warnings.filterwarnings('ignore', category=xp.LicenseWarning)
```

When suppressing deprecation warnings, it is recommended to add a separate filter for each Xpress version, to ensure that deprecations added in a future release will be identified.

CHAPTER 3

Using Python numerical libraries

The NumPy library allows for creating and using arrays of any order and size for efficiency and compactness purposes. This chapter shows how to take advantage of the features of NumPy in the creation of optimization problems. The Xpress Python interface supports NumPy versions 1.19 and later.

3.1 Using NumPy with the Xpress Python interface

NumPy arrays can be used as usual when creating variables, functions (linear and quadratic) of variables, and constraints. All functions described in this manual that take lists or tuples as arguments can take `array`'s, i.e., NumPy array objects, as well, as in the following example:

```
import numpy as np
import xpress as xp
N = 20
S = range(N)
p = xp.problem()
x = xp.array([p.addVariable()                  for i in S])
y = xp.array([p.addVariable(vartype=xp.binary) for i in S])
constr1 = x <= y
p.addConstraint(constr1)
```

The above script imports both NumPy and the Xpress Python interface, then declares two arrays of variables and creates the set of constraints $x_i \leqslant y_i$ for all *i* in the set *S*.

The arrays must be of class `xpress.ndarray` in order to use the matrix/vector forms of the comparison (<=, ==, >=) operators. This is to avoid the default behaviour of NumPy arrays, where an expression like `a <= b` will result in a Boolean array, instead of an array of constraint objects. Ways to create instances of this class are described at `xpress.ndarray`.

NumPy allows for multiarrays with one or more 0-based indices. Given that declaring a NumPy multiarray of variables can result in a long line of code, the `problem.addVariables` function in its simplest usage returns an `xpress.ndarray` of variables with one or more indices. Consider the following three array declarations:

```
import numpy as np
import xpress as xp
p = xp.problem()
x = xp.array([p.addVariable(name='v({0})'.format(i)) for i in range(20)]).reshape(5,4)
y = xp.array([p.addVariable(vartype=xp.binary) for i in range(27)]).reshape(3,3,3)
z = xp.array([p.addVariable(lb=-1, ub=1) for i in range(1000)])
```

These can be written equivalently in the compact form as

```
import numpy as np
import xpress as xp
p = xp.problem()
x = p.addVariables(5, 4, name='v')
```

```
y = p.addVariables(3, 3, 3, vartype=xp.binary)
z = p.addVariables(1000, lb=-1, ub=1)
```

The only side effect is that the assigned names change. In order to preserve the naming convention of the Xpress library, one can specify the parameter setting `name=''` in the call to `p.addVariables`. This also makes the creation of large arrays of variables much faster. We use this shorter notation in the remainder of this chapter.

The main advantage of using NumPy operations is the ability to replicate them on each element of an array, taking into account all *broadcasting* features. For example, the following script "broadcasts" the right-hand side 1 to all elements of the array, thus creating the set of constraints $x_i + y_i \leqslant 1$ for all *i* in the set *S*.

```
constr2 = x + y <= 1
```

All these operations can be carried out on arrays of any number of dimensions, and can be aggregated at any level. The following example shows two three-dimensional array of variables involved in two systems of constraints: the first has two variables per each of the 200 constraints, while the second has 10 constraints and 20 variables in each constraint.

```
z = p.addVariables(4, 5, 10)
t = p.addVariables(4, 5, 10, vartype=xp.binary)
p.addConstraint(z**2 <= 1 + t)
p.addConstraint(xp.Sum(z[i, j, k] for i in range(4) for j in range(5)) <= 4
                 for k in range(10))
```

Finally, a note on sums of multi-dimensional NumPy arrays: in keeping with the way NumPy arrays are handled, the sum of a multi-dimensional array results in a scalar expression with the `xpress.Sum` operator. The result of such a sum is exemplified by the following code:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> a.sum()
21
```

3.2 Products of NumPy arrays

The *dot product* is a useful operator for carrying out aggregate operations on vectors, matrices, and tensors. The dot operator in NumPy allows for reducing, along one axis of a multi-dimensional arrays, data such as floating points or integer values.

The application of the dot product of NumPy of two multi-dimensional arrays of dimensions $(i_1, i_2, ..., i_{k'})$ and $(j_1, j_2, ..., j_{k''})$, respectively, requires that $i_{k'} = j_{k''-1}$, i.e., the size of the last dimension of the first array must match the size of the penultimate dimension of the second vector. For instance, the following dot product is valid:

```
import numpy as np
a = np.random.random((4,6))
b = np.random.random((6,2))
c = np.dot(a,b)
```

and the result is a 4x2 matrix. The Xpress Python interface has its own dot product operator, which can be used for all similar operations on variables and expression. The rules for applying the Xpress dot operator are the same as for the native Python dot product, with one extra feature: there is no limit on the number of arguments, hence the following example is correct as per the restrictions on the dimensions, albeit it yields a nonconvex constraint.

```
coeff_pre = np.random.random((6,3,7))
p = xp.problem()
x = p.addVariables(4, 7, 5)
y = p.addVariables(2, 5, 8)
coeff_post = np.random.random((6, 8, 7))
p.addConstraint(xp.Dot(coeff_pre, x, y, coeff_post) >= 0)
```

Similar to the NumPy dot product, the Xpress dot product has an *out* parameter for suppyling an array in which to store the product.

The following script defines two constraints: the first restricts the squared norm $||z|| = z \cdot z$ of the vector z of variables to be at most one. It does so by applying the dot operator on the vector itself. The second constraint $(t - z)'Q(t - z) \leqslant 1$ restricts the quadratic form on the left-hand side to be at most 1.

```
p.addConstraint(xp.Dot(z, z) <= 1) # restrict norm of z to 1

Q = np.random.random(N, N)    # create a random 20x20 matrix
p.addConstraint(xp.Dot((t-z), Q, (t-z)) <= 1)
```

As for the `Sum` operator, when handling variables or expressions, it is advised to use the `Dot` operator in the Xpress module rather than the native Python operator, for reasons of efficiency.

3.3 Products with SciPy sparse matrices

The `xpress.Dot` operator supports the most common SciPy sparse matrix formats, allowing arrays of sparse expressions and constraints to be constructed efficiently. For more information, see the `xpress.Dot` documentation.

3.4 Using Pandas with the Xpress Python interface

When building a model, it is convenient to store data in Pandas data frames. It can also be helpful to store the objects representing the decision variables, expressions involving those variables, and constraints in the data frames. The Xpress Python interface provides a special Pandas dtype, `'xpressobj'` for use with series containing Xpress objects. This dtype is required to use the vector forms of the comparison (<=, ==, >=) operators and logical (`&`, `|`) operators. With the default dtype (i.e. `'object'`), an expression like `a <= b` will result in a Boolean series, instead of a series of constraint objects.

The following code shows how Pandas could be used to store and aggregate the data, decision variables and constraint for a problem with several knapsack constraints. Note the use of the `'xpressobj'` dtype, to ensure that the constraints are created correctly.

```
prob = xp.problem()

# Knapsack items
items = pd.DataFrame({
    'id': ['I1', 'I2', 'I3', 'I4'],
    'category': ['A', 'A', 'B', 'B'],
    'weight': [5, 12, 11, 2],
    'value': [9, 4, 10, 5],
})

# There is a different weight limit for each category of items
categories = pd.Series({
  'category': ['A', 'B'],
  'capacity': [10, 20]
})

# Create decision variables
```

```
items['pick'] = pd.Series(prob.addVariables(len(items), name='pick'), dtype='xpressobj')

# Create a constraint for each category, ensuring that the selected items in that category do not exceed i
prob.addConstraint(
  (items.weight * items.pick).groupby(items.category).sum() <= categories.capacity
)
```

This results in the following constraints:

```
5 * pick(0) + 12 * pick(1) <= 10
11 * pick(2) + 2 * pick(3) <= 20
```

CHAPTER 4

Controls and Attributes

A *control* is a parameter that can influence the performance and behavior of the Xpress Optimizer. For example, the MIP gap, the feasibility tolerance, or the type of root LP algorithms are controls that can be set. Controls can both be read from and written to an optimization problem.

An *attribute* is a feature of an optimization problem, such as the number of rows and columns or the number of quadratic elements of the objective function. They are read-only parameters in that they can only be modified, for example, by functions for adding constraints or variables, or functions for setting and modifying the objective function.

Both controls and attributes are of three types: integer, floating point, or string. The Xpress Python interface allows for setting and retrieving the value of all controls of an optimization problem, as well as getting the value of all of a problem's attributes.

This reference manual does *not* describe the meaning of controls and attributes in the Xpress Optimizer; for a detailed description of each, please refer to the Optimizer's reference manual.

Following Python's philosophy, one can set and obtain multiple controls/attributes with one function call. In other words, one can set either (i) a single control and its value; or (ii) a Python dictionary coupling a list of control names and their respective value. Similarly, with one function call one can obtain (i) the value of a single attribute or control by specifying it as a parameter; or (ii) a dictionary associating names to values for each of a list of controls or attributes given as an argument. See the examples below for more information.

You can see the available controls and attributes in the Optimizer Reference Manual, where each parameter is documented with its purpose and usage.

4.1 Controls

Use `problem.setControl` to set the value of one or more controls. Its synopsis is as follows:

```
setControl(ctrl, value)
setControl({ctrl1: value1, ctrl2: value2, ..., ctrlk: valuek})
```

The first form is for setting the value of the control `ctrl` to `value`. The second form is for setting `ctrl1` to `value1`, `ctrl2` to `value2`, ..., and `ctrlk` to `valuek`.

A list of all controls can be found on the Control Parameters page of the Optimizer Reference Manual. The control parameters to be passed in `setControl` are lower-case strings or upper-case strings (mixed lower- and upper-case will return an error), although in this manual we will only use lower-case:

```
p.setControl('miprelstop', 1e-9)
p.setControl({'miprelstop': 1e-3, 'feastol': 1e-6})
```

Alternatively, the control(s) to be changed can be identified by numeric id.

Use the method `getControl` to retrieve the value of one or more controls. Its synopsis is one of the following:

```
getControl(ctrl)
getControl([ctrl1, ctrl2, ..., ctrlk])
getControl(ctrl1, ctrl2, ..., ctrlk)
getControl()
```

The first form is for obtaining the value of the control `ctrl`. The output will be the value of the control. The second and third forms are for retrieving `ctrl1` , `ctrl2` , ..., and `ctrlk`. Whether the controls are declared in a list or a tuple does not matter. The result will be a dictionary coupling each control with its value. The last form is to obtain all controls; the result is a dictionary coupling all controls with their respective value.

Similar to `problem.setControl`, the control parameters to be passed in `getControl` are lower-case or upper-case strings. For a problem `p` the call will be as follows:

```
mrs = p.getControl('miprelstop')
someattr = p.getControl('miprelstop', 'feastol')
```

Alternatively, controls can be specified by their numeric id. In that case a returned dictionary will have that id as key for the requested control.

4.2 Examples

```
import xpress as xp

p = xp.problem()

p.setControl({'miprelstop': 1e-5, 'feastol': 1e-4})
p.setControl('miprelstop', 1e-5)

print(p.getControl('miprelstop') )           # print the current value of miprelstop
print(p.getControl('maxtime', 'feastol'))    # print a dictionary with the current
                                             # value of miprelstop and feastol
print(p.getControl(['presolve', 'miplog']))  # Same output
print(p.getControl())                        # print a dictionary with ALL control

# Initialize a dictionary with two controls and their value. Then
# change their value conditionally and set their new (possibly
# changed) value.

myctrl = p.getControl(['miprelstop', 'feastol'])

if (myctrl['miprelstop'] <= 1e-4):
    myctrl['miprelstop']  = 1e-3
    myctrl['feastol']     = 1e-3
else:
    myctrl['feastol']     = 1e-4

p.setControl(myctrl)
```

4.3 Attributes

Use the method `getAttrib` to retrieve the value of one or more attributes. Its synopsis is one of the following:

```
getAttrib(attr)
getAttrib([attr1, attr2, ..., attrk])
getAttrib(attr1, attr2, ..., attrk)
getAttrib()
```

The first form is for obtaining the value of the attribute `attr`. The output will be the value of the attribute. The second and third forms are for retrieving `attr1` , `attr2` , ..., and `attrk`. Whether the attributes are declared in a list or a tuple does not matter. The result will be a dictionary coupling each attribute with its value. The last form is to obtain all attributes; the result is a dictionary coupling all attributes with their respective value.

A list of all attributes can be found on the Problem Attributes page of the Optimizer Reference Manual. As for controls, the attribute parameters to be passed in `getAttrib` are lower-case or upper-case strings (mixed lower- and upper-case strings are, similar to controls, forbidden). For a problem `p` the call will be as follows:

```
nrows = p.getAttrib('rows')
problemsize = p.getAttrib('rows', 'cols')
```

Alternatively, attributes can be specified by their numeric id. In that case a returned dictionary will have that id as key for the requested attribute.

4.4 Examples

```
import xpress as xp

p = xp.problem()

p.readProb("example.lp")

print("The problem has",
    p.getAttrib('rows'), "rows and",
    p.getAttrib('cols'), "columns")

# Obtain dictionary with two entries: the number of rows and
# columns of the problem read

print(p.getAttrib(['rows', 'cols']))

# produce a Python dictionary with all attributes of problem m, and
# hence of LP file example.lp

attributes = p.getAttrib()
```

4.5 Accessing controls and attributes as object members

An alternative, more ”prompt-friendly” way to get controls and attributes is through their direct access in a problem or, in the case of controls, the Xpress module itself.

The Xpress module has an object, called `controls`, containing all controls of the Optimizer. Upon importing the Xpress module, these controls are initialized at their default value. The user can obtain their value at any point and can also set their value; this new value will be inherited by all problems created *after* the modification. They can be read and written as follows:

```
xpress.controls.<controlname>
xpress.controls.<controlname> = <new value>
```

For example, the object `xpress.controls.miprelstop` contains the value of the control *miprelstop*. Controls can be read (and, for example, printed) and set as follows:

```
import xpress as xp
print(xp.controls.heuremphasis)
```

```
xp.controls.feastol = 1e-4 # Set new default to 1e-4
```

These "global" controls are maintained throughout while the Xpress module is loaded. Note that the `controls` object of the Xpress module does not refer to any specific problem. All controls have default values that are determined by the Optimizer's library, except for the control `xslp_postsolve` that is set to 1, as opposed to its default value of 0 in the Xpress Optimizer's library.

In addition, every problem has a `controls` object that stores the controls related to the problem itself. This is the object the functions `getControl` and `setControl` refer to. Similar to the Xpress module's `controls` object, all members of a problem's object can be read and written. For a problem `p`, the following shows how to read and write a problem's control:

```
p.controls.<controlname>
p.controls.<controlname> = <new value>
```

A problem's controls are independent of the global `controls` object of the Xpress module. However, when a new problem is created its controls are copied from the current values in the global object. Note that after creating a new problem, changing the members in `xpress.controls` does not affect the problem's controls. The following examples should clarify this:

```
import xpress as xp

# create a new problem whose MIPRELSTOP is ten times smaller
# than the default value

p1 = xp.problem("problem1")
p1.controls.miprelstop = 0.1 * xp.controls.miprelstop
p1.controls.feastol = 1e-5
p1.readProb("example1.lp")

xp.controls.miprelstop = 1e-8 # Set new default

# The new problem will have a MIPRELSTOP of 1e-8

p2 = xp.problem("problem2")
p2.readProb("example2.lp")

# The next problem has a less restrictive feasibility tolerance
# (i.e. 1e-6) than problem 2

p2v = xp.problem("problem2 variant")
p2v.readProb("example2.lp")
p2v.controls.feastol = 100 * p2.controls.feastol

p1.optimize()
p2.optimize()

# solve "example2.lp" with a less restrictive
# feasibility tolerance
p2v.optimize()
```

Attributes can be handled similar as above through a member of the class `problem`, called `attributes`, with two exceptions: first, there is no "global" attribute object, as a set of attributes only makes sense when associated with a problem; second, an attribute cannot be set.

Once a problem `p` has been created (or read from a file), its attributes are available as `p.attribute.`*attribute_name*. The example in the previous section can be modified as follows:

```
import xpress as xp
p = xp.problem()
p.readProb("example.lp")
print("The problem has",
    p.attributes.rows, "rows and",
```

```
    p.attributes.cols, "columns")
```

When using the Python prompt in creating problems with the Xpress module, the name of controls and attributes can be auto-completed by pressing TAB (note: this only works in Python 3.4 and subsequent versions). For instance,

```
>>> import xpress
>>> p = xp.problem()
>>> p.readProb("example.lp")
>>> p.attributes.n<TAB>
p.attributes.namelength  p.attributes.nodedepth   p.attributes.nodes  p.attributes.numiis
>>> p.attributes.nodedepth
0
>>> p.attributes.ma<TAB>
p.attributes.matrixname          p.attributes.maxabsdualinfeas
p.attributes.maxabsprimalinfeas  p.attributes.maxprobnamelength
p.attributes.maxreldualinfeas    p.attributes.maxrelprimalinfeas
>>> p.attributes.matrixname
'noname'
>>> xp.controls.o<TAB>
xp.controls.oldnames             xp.controls.omniformat
xp.controls.optimalitytol        xp.controls.optimalitytoltarget
xp.controls.outputlog            xp.controls.outputmask
xp.controls.outputtol
>>> xp.controls.omniformat
0
```

4.6 Objective controls and attributes

For multi-objective problems, control values can be specified which will only be applied when solving the given objective. These control values can be set and queried by passing the `objidx` argument to `problem.setControl` and `problem.getControl`. They can also be set via the `problem.objcontrols` property. For example, to set a time limit for solving the second objective:

```
prob.objcontrols[1].timelimit = 20
```

After a multi-objective optimization, the attributes of each individual solve can be queried by passing the `solveidx` argument to `problem.getAttrib`. They can also be queried via the `problem.objattribs` property. For example, to print the number of MIP solutions found during the first solve:

```
print(prob.objattribs[1].mipsols)
```

For more information about objective controls and attributes, see the Multi-objective Optimization section in the Optimizer Reference Manual .

CHAPTER 5

Using Callbacks

This chapter shows how to define and use callback functions from the Xpress Python interface. The design of this part of the interface reflects as closely as possible the design of the callback functions defined in the C API of the Xpress Optimizer.

Callback functions are a useful tool for adapting the Xpress Optimizer to the solution of various classes of problems, in particular Mixed Integer Programming (MIP) problems, with linear or nonlinear constraints. Their main purpose is to provide the user with a point of entry into the branch-and-bound, which is the workhorse algorithm for MIPs.

Using callback functions is simple: the user first defines a function (say `myfunction`) that is to be run every time the branch-and-bound reaches a well-specified point; second, the user calls a function (such as `addPreIntsolCallback`) with `myfunction` as its argument. Finally, the user calls the `optimize` function that launches the branch-and-bound, the simplex solver, or the barrier solver; it is while these are run that `myfunction` is called.

A callback function, hence, is passed once as an argument and used possibly many times. It is called while a solver is running, and it is passed the following:

- a problem object, of the same class as an object declared with `p = xpress.problem();`
- a data object.

The data object is user-defined and is given to the problem when adding the callback function. It can be used to store information that the user can read and/or modify within the callback. For instance, the following code shows how to add a callback function, `preintsolcb`, that is called every time a new integer solution is found.

```
import xpress as xp

class foo:
    "Simple class"
    bar = 0
    def __init__(self):
        self.bar = 1
    def update(self):
        self.bar += 1

def preintsolcb(prob, data, isheuristic, cutoff):
   """
   Callback to be used when an integer solution is found. The
   "data" parameter is of class foo
   """

p = xp.problem()
p.readProb('myprob.lp')  # reads in a problem, let's say a MIP

baz = foo()
```

```
p.addPreIntsolCallback(preintsolcb, baz, 3)
p.optimize()
```

While the function argument is necessary for all `add*Callback` functions, the data object can be specified as `None`. In that case, the callback will be run with `None` as its data argument. The call also specifies a priority with which the callback should be called: the larger the (positive) priority, the more urgently it is called.

Any call to an `add*Callback` function, as the names imply, only *adds* a function to a list of callback functions for that specific point of the BB algorithm. For instance, two calls to `addPreIntsolCallback` with two functions `preint1` and `preint2`, respectively with priority 3 and 5, puts the two functions in a list. The two functions will be called (`preint2` first) whenever the BB algorithm finds an integer solution.

In order to remove a callback function that was added with `add*Callback`, a corresponding `remove*Callback` function is provided, for instance `removePreIntsolCallback`. This function takes two arguments, i.e., the callback function and the data object, and deletes all elements of the list of callbacks that were added with the corresponding `add*Callback` function that match the function **and** the data.

The `None` keyword acts as a wildcard that matches any function/data object: if `remove*Callback` is called with `None` as the function, then all callbacks matching the data will be deleted. If the data is also `None`, all callback functions of that type are deleted; this can be obtained by passing no argument to `remove*Callback`.

The arguments and return value of the callback functions reflect those in the C API, and this holds for parameter names as well. As for the other API functions of the Python interface, there are a few exceptions:

- If a function in the C API requires a parameter *n* to indicate the size of an array argument to follow, *n* is not required in the corresponding Python function;
- If a function in the C API uses passing by reference as a means to allow for modifying a value and returning it as an output, the Python counterpart will have this as the return value of the function. Where multiple output values are comprised in the list of parameters, the return value is a *tuple* composed of the returned values. Elements of this tuple can be `None` if no change was made to that output value.

Below is a list of all available callbacks on the `problem` class:

`problem.addAfterObjectiveCallback` Declares a callback which will be called after each objective in a multi-objective problem is solved.
[Callback, Multiobjective] p. 438

`problem.addBarIterationCallback` Declares a barrier iteration callback function, called after each iteration during the interior point algorithm, with the ability to access the current barrier solution/slack/duals or reduced cost values, and to ask barrier to stop. This callback function will be called in addition to any callbacks already added by problem.addBarIterationCallback. This functionality is not available for the hybrid gradient method.
[Barrier, Callback] p. 420

`problem.addBarlogCallback` Declares a barrier log callback function, called at each iteration during the interior point algorithm. This callback function will be called in addition to any barrier log callbacks already added by problem.addBarlogCallback. This functionality is not available for the hybrid gradient method.
[Barrier, Callback] p. 399

`problem.addBeforeObjectiveCallback` Declares a callback which will be called before each objective in a multi-objective problem is solved.
[Callback, Multiobjective] p. 436

`problem.addChangeBranchObjectCallback` Declares a callback function that will be called after the selection of a MIP entity to branch on.
[Branching, Callback] p. 426

`problem.addCheckTimeCallback` Declares a callback function which is called every time the Optimizer checks if the time limit has been reached. This callback function will be called in addition to any callbacks already added by problem.addCheckTimeCallback.
[Callback, Limits] p. 440

`problem.addComputeRestartCallback` Declares a callback to be called when a solve executed in compute mode needs to be restarted.
[Callback, Compute Interface] p. 434

`problem.addCutlogCallback` Declares a cut log callback function, called each time the cut log is printed.
[Callback, Logging] p. 397

`problem.addCutRoundCallback` Declares a callback function that is called when the Optimizer could separate cutting planes during the branch and bound search.
[Callback, Cuts] p. 442

`problem.addGapNotifyCallback` Declares a gap notification callback, to be called when a MIP solve reaches a predefined target, set using the `MIPRELGAPNOTIFY` (p.**??** Xpress Optimizer Reference Manual), `MIPABSGAPNOTIFY` (p.**??** Xpress Optimizer Reference Manual), `MIPABSGAPNOTIFYOBJ` (p.**??** Xpress Optimizer Reference Manual) and/or `MIPABSGAPNOTIFYBOUND` (p.**??** Xpress Optimizer Reference Manual) controls.
[Callback, Limits] p. 428

`problem.addInfnodeCallback` Declares a user infeasible node callback function, called after the current node has been found to be infeasible during the Branch and Bound search.
[Branch and Bound Search, Callback] p. 405

`problem.addIntsolCallback` Declares a user integer solution callback function, called every time an integer solution is found by heuristics or during the Branch and Bound search. This callback function will be called in addition to any callbacks already added by problem.addIntsolCallback.
[Callback, Solution] p. 409

`problem.addLplogCallback` Declares a simplex log callback function which is called after every `LPLOG` iterations of the simplex algorithm. This callback function will be called in addition to any callbacks already added by problem.addLplogCallback.
[Callback, LP] p. 393

`problem.addMessageCallback` Declares an output callback function, called every time a text line relating to the given problem is output by the Optimizer.
[Callback, Logging] p. 414

`problem.addMiplogCallback` Declares a MIP log callback function, called each time the MIP log is printed. This callback function will be called in addition to any callbacks already added by problem.addMiplogCallback.
[Callback, Logging] p. 395

`problem.addMipThreadCallback` Declares a MIP thread callback function, called every time a MIP worker problem is created by the parallel MIP code. This callback function will be called in addition to any callbacks already added by problem.addMipThreadCallback.
[Callback, Parallel] p. 416

`problem.addMipThreadDestroyCallback` Declares a destroy MIP thread callback function, called every time a MIP thread is destroyed by the parallel MIP code. This callback function will be called in addition to any callbacks already added by problem.addMipThreadDestroyCallback.
[Callback, Parallel] p. 418

`problem.addMsJobEndCallback` Add a user callback to be called every time a new multistart job finishes. Can be used to overwrite the default solution ranking function
[Callback, Multistart] p. 466

`problem.addMsJobStartCallback` Add a user callback to be called every time a new multistart job is created, and the pre-loaded settings are applied
[Callback, Multistart] p. 464

`problem.addMsWinnerCallback` Add a user callback to be called every time a multistart winner has been declared
[Callback, Multistart] p. 468

`problem.addNewnodeCallback` Declares a callback function that will be called every time a new node is created during the branch and bound search. This callback function will be called in addition to any callbacks already added by problem.addNewnodeCallback.
[Branch and Bound Search, Callback] p. 424

`problem.addNlpCoefEvalErrorCallback` Add a user callback to be called when an evaluation of a coefficient fails during the solve
[Callback, Numerics] p. 470

`problem.addNodecutoffCallback` Declares a user node cutoff callback function, called every time a node is cut off as a result of an improved integer solution being found during the branch and bound search. This callback function will be called in addition to any callbacks already added by problem.addNodecutoffCallback.
[Branch and Bound Search, Callback] p. 407

`problem.addNodeLPSolvedCallback` Declares a callback function, called during the branch and bound search, after the LP relaxation has been solved for the current node, but before any internal cuts and heuristics have been applied. This callback function will be called in addition to any callbacks already added by problem.addNodeLPSolvedCallback.
[Branch and Bound Search, Callback] p. 432

`problem.addOptnodeCallback` Declares an optimal node callback function, called during the branch and bound search, after the LP relaxation has been solved for the current node, and after any internal cuts and heuristics have been applied, but before the Optimizer checks if the current node should be branched. This callback function will be called in addition to any callbacks already added by problem.addOptnodeCallback.
[Branch and Bound Search, Callback] p. 401

`problem.addPreIntsolCallback` Declares a user integer solution callback function, called when an integer solution is found by heuristics or during the branch and bound search, but before it is accepted by the Optimizer.
[Callback, Solution] p. 411

`problem.addPrenodeCallback` Declares a preprocess node callback function, called before the LP relaxation of a node has been optimized, so the solution at the node will not be available. This callback function will be called in addition to any callbacks already added by problem.addPrenodeCallback.
[Branch and Bound Search, Callback] p. 403

`problem.addPresolveCallback` Declares a callback to be called after presolve has been performed.
[Callback, Presolve] p. 422

`problem.addSlpCascadeEndCallback` Add a user callback to be called at the end of the cascading process, after the last variable has been cascaded
[Callback, Cascading, SLP] p. 444

`problem.addSlpCascadeStartCallback` Add a user callback to be called at the start of the cascading process, before any variables have been cascaded
[Callback, Cascading, SLP] p. 446

`problem.addSlpCascadeVarCallback` Add a user callback to be called after each column has been cascaded
[Callback, Cascading, SLP] p. 448

`problem.addSlpCascadeVarFailCallback` Add a user callback to be called after cascading a column was not successful
[Callback, Cascading, SLP] p. 450

`problem.addSlpConstructCallback` Add a user callback to be called during the Xpress-SLP augmentation process
[Callback, SLP] p. 452

`problem.addSlpDrColCallback` Add a user callback used to override the update of variables with small determining column
[Callback, Cascading, SLP] p. 462

`problem.addSlpIntSolCallback` Add a user callback to be called during MISLP when an integer solution is obtained
[Callback, MISLP] p. 454

`problem.addSlpIterEndCallback` Add a user callback to be called at the end of each SLP iteration
[Callback, SLP] p. 456

`problem.addSlpIterStartCallback` Add a user callback to be called at the start of each SLP iteration
[Callback, SLP] p. 458

`problem.addSlpIterVarCallback` Add a user callback to be called after each column has been tested for convergence
[Callback, SLP, SLP-convergence] p. 460

`problem.addSlpPreUpdateLinearizationCallback` Add a user callback to be called before the linearization is updated
[Callback, SLP] p. 472

`problem.addUserSolNotifyCallback` Declares a callback function to be called each time a solution added by `problem.addMipSol` has been processed. This callback function will be called in addition to any callbacks already added by `problem.addUserSolNotifyCallback`.
[Callback, Solution] p. 430

CHAPTER 6

Examples of use

This chapter discusses some example Python scripts that are part of the Xpress Optimizer's Python interface. Most of them are well commented so the user can refer directly to the source for guidance.

Most of these scripts have an initial part in common, which we reproduce here but omit in all explanations below for compactness. These initial lines import the Xpress module itself and the NumPy module, which is used in some of the examples.

```
import xpress as xp
import numpy as np
```

6.1 Modeling examples

6.1.1 A simple model

This example demonstrates how variables and constraints, or lists/arrays thereof, can be added into a problem. The script then prints the solution and attributes of the problem.

```
N = 4
S = range(N)

m = xp.problem()

v = [m.addVariable(name="y{0}".format(i), lb=0, ub=2*N) for i in S]

v1 = m.addVariable(name="v1", lb=0, ub=10, threshold=5, vartype=xp.continuous)
v2 = m.addVariable(name="v2", lb=1, ub=7, threshold=3, vartype=xp.semicontinuous)
vb = m.addVariable(name="vb", vartype=xp.binary)

c1 = v1 + v2 >= 5

m.addConstraint(c1,  # Adds a list of constraints: three single constraints...
                2*v1 + 3*v2 >= 5,
                v[0] + v[2] >= 1,
                # ... and a set of constraints indexed by all {i in
                # S: i<N-1}(recall that ranges in Python are from 0
                # to n-1)
                (v[i+1] >= v[i] + 1 for i in S if i < N-1))

# objective overwritten at each setObjective()
m.setObjective(xp.Sum([i*v[i] for i in S]), sense=xp.ObjSense.MINIMIZE)

solvestatus, solstatus = m.optimize()

print("solve status: ", solvestatus.name)
print("solution status: ", solstatus.name)

print("solution:", m.getSolution())
```

6.1.2 Using IIS to investigate an infeasible problem

The problem modeled below is infeasible,

```
import xpress as xp

minf = xp.problem("ex-infeas")

x0 = minf.addVariable()
x1 = minf.addVariable()
x2 = minf.addVariable(vartype=xp.binary)

c1 =     x0 + 2 * x1 >= 1
c2 = 2 * x0 +     x1 >= 1
c3 =     x0 +     x1 <= .5

c4 = 2 * x0 + 3 * x1 >= 0.1
```

The three constraints `c1`, `c2`, and `c3` above are incompatible as can be easily verified. Adding all of them to a problem will make it infeasible. We use the functions to retrieve the Irreducible Infeasible Subsystems (IIS).

```
minf.addConstraint(c1,c2,c3,c4)

minf.optimize()
minf.IISAll()
print("there are ", minf.attributes.numiis, " iis's")

# Get data for the first IIS

miisrow, miiscol, constrainttype, colbndtype, duals, rdcs, \
  isolationrows, isolationcols = minf.getIISData(1)

print("iis data:", miisrow, miiscol, constrainttype, colbndtype,
       duals, rdcs, isolationrows, isolationcols)

# Another way to check IIS isolations
print("iis isolations:", minf.IISIsolations(1))

niis, rowsizes, colsizes, suminfeas, numinfeas = minf.IISStatus()

print("number of IISs found:", niis)
print("vectors:", rowsizes, colsizes, suminfeas, numinfeas)
```

6.1.3 Modeling a problem using Python lists and vectors

We create a convex QCQP problem. We use a list of `N`=5 variables and sets constraints and objective. We define all constraints and the objective function using a Python aggregate type.

```
import xpress as xp

N = 5
S = range(N)

m = xp.problem("problem 1")

v = [m.addVariable(name="y{0}".format(i)) for i in S]

print("variable:", v)

m.addConstraint(v[i] + v[j] >= 1 for i in range(N-4) for j in range(i,i+4))
m.addConstraint(xp.Sum([v[i]**2 for i in range(N-1)]) <= N**2 * v[N-1]**2)
m.setObjective(xp.Sum([i*v[i] for i in S]) * (xp.Sum([i*v[i] for i in S])))

m.optimize()
```

```
print("solution: ", m.getSolution())
```

6.1.4 A knapsack problem

Here follows an example of a knapsack problem formulated using lists of numbers. All data in the problem are lists, and so are the variables.

```
import xpress as xp

S = range(5)          # that's the set {0,1,2,3,4}
value  = [102, 512, 218, 332, 41] # or just read them from file
weight = [ 21,  98,  44,  59,  9]

p = xp.problem("knapsack")

x = p.addVariables(5, vartype=xp.binary)
profit = xp.Sum(value[i] * x[i] for i in S)

p.addConstraint(xp.Sum(weight[i] * x[i] for i in S) <= 130)
p.setObjective(profit, sense=xp.ObjSense.MAXIMIZE)
p.optimize()
```

Note that the same result could have been achieved using NumPy arrays and the Xpress module's dot product as follows:

```
import xpress as xp
import numpy as np

value  = np.array([102, 512, 218, 332, 41])
weight = np.array([ 21,  98,  44,  59,  9])

p = xp.problem("knapsack")

x = p.addVariables(5, vartype=xp.binary)
profit = xp.Dot(value, x)

p.addVariable(x)
p.addConstraint(xp.Dot(weight, x) <= 130)
p.setObjective(profit, sense=xp.ObjSense.MAXIMIZE)
p.optimize()
```

6.1.5 A Min-cost-flow problem using NumPy

This example solves a min-cost-flow problem using NumPy and the incidence matrix of the graph.

```
import numpy as np
import xpress as xp

# digraph definition

V = [1, 2, 3, 4, 5]                                  # vertices
E = [[1, 2], [1, 4], [2, 3], [3, 4], [4, 5], [5, 1]] # arcs

n = len(V) # number of nodes
m = len(E) # number of arcs
```

We then generate the incidence matrix by creating a NumPy matrix with *n* rows and *m* columns, such that each column, which corresponds to an arc *(i,j)*, has a -1 at row *i* and a 1 at row *j*.

```
# Generate incidence matrix: begin with a NxM zero matrix
A = np.zeros((n,m))
```

```
# Then for each column i of the matrix, add a -1 in correspondence to
# the tail of the arc and a 1 for the head of the arc. Because Python
# uses 0-indexing, the row of A should be the node index minus one.
for i, edge in enumerate(E):
    A[edge[0] - 1][i] = -1
    A[edge[1] - 1][i] =  1
```

We use NumPy vectors and the Xpress interface's dot product, the `xpress.Dot` operator. Note that although NumPy has a dot operator, especially for large models it is strongly advised to use the Xpress interface's `Dot` function for reasons of efficiency.

```
demand = np.array([3, -5, 7, -2, -3])

cost = np.array([23, 62, 90, 5, 6, 8])

p = xp.problem('network flow')

flow = p.addVariables(m) # flow variables declared on arcs

p.addConstraint(xp.Dot(A, flow) == - demand)
p.setObjective(xp.Dot(cost, flow))

p.optimize()

for i in range(m):
    print('flow on', E[i], ':', p.getSolution(flow[i]))
```

6.1.6 A nonlinear model

Let's solve a classical nonlinear problem: finding the minimum of the Rosenbrock function. For parameters *a* and *b*, minimize $(a - x)^2 + b(y - x^2)^2$.

```
import xpress as xp

a,b = 1,100

p = xp.problem()

x = p.addVariable(lb=-xp.infinity)
y = p.addVariable(lb=-xp.infinity)

p.setObjective((a-x)**2 + b*(y-x**2)**2)

p.controls.xslp_solver = 0 # solve it with SLP, not Knitro

solvestatus, solstatus = p.optimize()

print("solve status: ", solvestatus.name)
print("solution status: ", solstatus.name)

print("solution:", p.getSolution())
```

6.1.7 Finding the maximum-area n-gon

The problem asks, given n, to find the n-sided polygon of largest area inscribed in the unit circle.

While it is natural to prove that all vertices of a global optimum reside on the unit circle, the problem is formulated so that every vertex i is at distance rho_i from the center, and at angle $theta_i$. We would expect that the local optimum found has all *rho*'s are equal to 1. The example file contains instructions for drawing the resulting polygon using matplotlib.

The objective function is the total area of the polygon. Considering the segment S[i] joining the center to the i-th vertex and A(i,j) the area of the triangle defined by the two segments S[i] and S[j], the objective

function is $A_{(0,1)} + A_{(1,2)} + ... + A_{(N-1,0)}$, where $A_{(i,j)} = 1/2 * rho_i * rho_j * sin(theta_i - theta_j)$. We first define the set `Vertices` as the set of integers from 0 to $n - 1$.

```
p = xp.problem()

rho = [p.addVariable(name='rho_{}'.format(i), lb=1e-5, ub=1.0) for i in Vertices]
theta = [p.addVariable(name='theta_{}'.format(i), lb=-math.pi, ub=math.pi)
        for i in Vertices]

p.setObjective(
  0.5*(xp.Sum(rho[i]*rho[i-1]*xp.sin(theta[i]-theta[i-1]) for i in Vertices if i != 0)
       +       rho[0]*rho[N-1]*xp.sin(theta[0]-theta[N-1])), sense=xp.ObjSense.MAXIMIZE)
```

We establish that the angles must be increasing in order to obtain a sensible solution:

```
p.addConstraint(theta[i] >= theta[i-1] + 1e-4 for i in Vertices if i != 0)
```

Note also that we enforce that the angles be different as otherwise they might form a local optimum where all of them are equal.

6.1.8 Solving the n-queens problem

In chess, the queen can move in all directions (even diagonally) and travel any distance. The problem of the *n* queens consists in placing *n* queens on an $n \times n$ chessboard so that none of them can be eaten in one move.

We first create a 2D array of variables, mapping each cell of the chessboard to one variable so that we can refer to it later. All variables are clearly binary as they indicate whether a given cell has a queen or not.

```
n = 10 # the size of the chessboard
N = range(n)

p = xp.problem()

# Create a 2D numpy array of (i,j) variables and link them to problem p
x = p.addVariables(N, N, vartype=xp.binary, name='q')

vertical   = [xp.Sum(x[i,j] for i in N) <= 1 for j in N]
horizontal = [xp.Sum(x[i,j] for j in N) <= 1 for i in N]

diagonal1  = [xp.Sum(x[k-j,j] for j in range(max(0,k-n+1), min(k+1,n))) <= 1
              for k in range(1,2*n-2)]
diagonal2  = [xp.Sum(x[k+j,j] for j in range(max(0,-k),    min(n-k,n))) <= 1
              for k in range(2-n,n-1)]

p.addConstraint(vertical, horizontal, diagonal1, diagonal2)

# Objective, to be maximized: number of queens on the chessboard
p.setObjective(xp.Sum(x), sense=xp.ObjSense.MAXIMIZE)

p.optimize()
```

As a rudimentary form of visualization, we print the solution on the chessboard with different symbols for variables at one or zero.

```
for i in N:
    for j in N:
        if p.getSolution(x[i,j]) == 1:
            print('@', sep='', end='')
        else:
            print('.', sep='', end='')
    print('')
```

6.1.9 Solving Sudoku problems

The well-known Sudoku puzzles ask one to place numbers from 1 to 9 into a 9×9 grid such that no number repeats in any row, in any column, and in any 3x3 sub-grid. For a more general version of the game, replace 3 with q and 9 with q^2.

We model this problem as an assignment problem where certain conditions must be met for all numbers in the columns, rows, and sub-grids.

These subgrids are lists of tuples with the coordinates of each subgrid. In a 9×9 sudoku, for instance, subgrids[0,1] has the 9 elements in the middle top square.

The input is a starting grid where the unknown numbers are replaced by zero. The example file contains a relatively hard 9×9 sudoku, which we show below, and also a 16×16 variant of the same game.

```
q = 3

starting_grid = \
 [[8,0,0,0,0,0,0,0,0],
  [0,0,3,6,0,0,0,0,0],
  [0,7,0,0,9,0,2,0,0],
  [0,5,0,0,0,7,0,0,0],
  [0,0,0,0,4,5,7,0,0],
  [0,0,0,1,0,0,0,3,0],
  [0,0,1,0,0,0,0,6,8],
  [0,0,8,5,0,0,0,1,0],
  [0,9,0,0,0,0,4,0,0]]

n = q**2 # the size must be the square of the size of the subgrids
N = range(n)

p = xp.problem()

x = p.addVariables(N, N, N, vartype=xp.binary)

# define all q^2 subgrids
subgrids = {(h,l): [(i,j) for i in range(q*h, q*h + q)
                          for j in range(q*l, q*l + q)]
                   for h in range(q) for l in range(q)}

vertical   = [xp.Sum(x[i,j,k] for i in N) == 1 for j in N for k in N]
horizontal = [xp.Sum(x[i,j,k] for j in N) == 1 for i in N for k in N]
subgrid    = [xp.Sum(x[i,j,k] for (i,j) in subgrids[h,l]) == 1
                              for (h,l) in subgrids.keys() for k in N]

# Assign exactly one number to each cell

assign = [xp.Sum(x[i,j,k] for k in N) == 1 for i in N for j in N]
```

Then we fix those variables that are non-zero in the input grid. We don't need an objective function as this is a feasibility problem. After computing the solution, we print it to the screen.

```
init = [x[i,j,k] == 1 for k in N for i in N for j in N
                      if starting_grid[i][j] == k+1]

p.addConstraint(vertical, horizontal, subgrid, assign, init)

p.optimize()

print('Solution:')

for i in N:
    for j in N:
        l = [k for k in N if p.getSolution(x[i,j,k]) >= 0.5]
```

```
        assert(len(l) == 1)
        print('{0:2d}'.format(1 + l[0]), end='', sep='')
    print('')
```

6.2 Examples using NumPy

6.2.1 Using NumPy multidimensional arrays to create variables

Use NumPy arrays for creating a 3-dimensional array of variables, then use it to create a mode.

```
S1 = range(2)
S2 = range(3)
S3 = range(4)

m = xp.problem()

h = xp.array([[[m.addVariable(vartype=xp.binary)
                for i in S1]
                for j in S2]
                for k in S3])

m.setObjective (h[0][0][0] * h[0][0][0] +
                h[1][0][0] * h[0][0][0] +
                h[1][0][0] * h[1][0][0] +
                xp.Sum(h[i][j][k] for i in S3 for j in S2 for k in S1))

cons00 = - h[0][0][0] ** 2 +
         xp.Sum(i * j * k * h[i][j][k]for i in S3 for j in S2 for k in S1) >= 11

m.addConstraint(cons00)

m.optimize()
```

The final part of the code retrieves the matrix representation of the quadratic part of the only constraint.

```
mstart1, mclind1, dqe1 = m.getQRowQMatrix(cons00, h[0][0][0], h[3][2][1])
print("row 0:", mstart1, mclind1, dqe1)
```

6.2.2 Using the dot product to create arrays of expressions

Here we use NumPy arrays to print the product of a matrix by a random vector, and the `xpress.Dot` function on a matrix and a vector. Note that the NumPy dot operator works perfectly fine here, but should be avoided for reasons of performance, especially when handling large arrays where at least one contains optimization variables or expressions.

```
p = xp.problem()

x = p.addVariables(5)
p.addConstraint(xp.Sum(x) >= 2)

p.setObjective(xp.Sum(x[i]**2 for i in range(5)))
p.optimize()

A   = np.array(range(30)).reshape(6,5) # A is a 6x5 matrix
sol = np.array(p.getSolution()) # a vector of size 5
columns = A*sol         # not a matrix-vector product!
v = np.dot(A,sol)      # an array: matrix-vector product A*sol
w = xp.Dot(A,x)        # an array of expressions

print(v,w)
```

6.2.3 Using the Dot product to create constraints and quadratic functions

This is an example of a problem formulation that uses the `xpress.Dot` operator to formulate constraints in a concise fashion. Note that the NumPy dot operator is not suitable here as the result is an expression in the Xpress variables.

```
A = np.random.random(30).reshape(6,5) # A is a 6x5 matrix
Q = np.random.random(25).reshape(5,5) # Q is a 5x5 matrix

p = xp.problem()

# Add a NumPy array of variables
x = p.addVariables(5)
x0 = np.random.random(5) # random vector

Q += 4 * np.eye(5) # add 5 * the identity matrix

Lin_sys = xp.Dot(A,x)   <= np.array([3,4,1,4,8,7])  # 6 constraints (rows of A)
Conv_c  = xp.Dot(x,Q,x) <= 1                        # one quadratic constraint

p.addConstraint(Lin_sys, Conv_c)
p.setObjective(xp.Dot(x-x0, x-x0)) # minimize distance from x0

p.optimize()
```

6.2.4 Using NumPy to create quadratic optimization problems

This example creates and solves a simple quadratic optimization problem. Given an $n \times n$ matrix Q and a point x_0, minimize the quadratic function $x^T(Q + n^3 I)x$ subject to the linear system $(x - x_0)^T Q + e = 0$, where e is the vector of all ones, the inequalities $Qx \geqslant 0$, and nonnegativity on all variables. Report solution if available.

```
N = 10

p = xp.problem()

Q = np.arange(1, N**2 + 1).reshape(N, N)
x = p.addVariables(N)
x0 = np.random.random(N)

c1 = xp.Dot((x - x0), Q) + 1 == 0
c2 = xp.Dot(Q, x) >= 0

p.addConstraint(c1,c2)
p.setObjective(xp.Dot(x, Q + N**3 * np.eye(N), x))

p.optimize('')

print("nrows, ncols:", p.attributes.rows, p.attributes.cols)
print("solution:", p.getSolution())

p.writeProb("test5-qp", "lp")
```

6.3 Advanced examples: callbacks and problem querying, modifying, and analysis

6.3.1 Visualize the branch-and-bound tree of a problem

This example shows how to visualize the BB tree of a problem after (partially) solving it. It is assumed here that all branches are binary.

We first define a message callback for running code whenever the Optimizer wants to print a message. The callback receives four arguments: the problem and callback data and, most importantly, the message to be printed and an information number. The callback prints the output message prefixed by a time stamp related to the creation of the problem. As the message could be on multiple lines, it is split into multiple substrings, one per line.

```
import networkx as nx
import time
from matplotlib import pyplot as plt

def message_addtime (prob, data, msg, info):
    """Message callback example: print a timestamp before the message from the optimizer"""
    if msg:
        for submsg in msg.split('\n'):
            print("{0:6.3f}: [{2:+4d}] {1}".format(time.time() - start_time, submsg, info))
```

We then define a recursive function that computes the cardinality of a subtree rooted at a node i. This is necessary as the visualization of the BB tree is more balanced when the subtree size is taken into account. The `card_subtree` array, which is filled here, is used then for computing the width of each visualized subtree.

```
def postorder_count(node):
    """
    Recursively count nodes to compute the cardinality of a subtree for
    each node
    """

    card = 0

    if node in left.keys(): # see if node has a left key
        postorder_count(left[node])
        card += card_subtree[left[node]]

    if node in right.keys():
        postorder_count(right[node])
        card += card_subtree[right[node]]

    card_subtree[node] = 1 + card
```

We also define a function that determines the position of each node depending on the cardinality of the subtree rooted at the node.

```
def setpos(T, node, curpos, st_width, depth):

    """
    Set position depending on cardinality of each subtree
    """

    # Special condition: we are at the root
    if node == 1:
        T.add_node(node, pos=(0.5, 1))

    alpha = .1 # use a convex combination of subtree comparison and
```

```
                # depth to assign a width to each subtree

    if node in left.keys():

        # X position in the graph should not just depend on depth,
        # otherwise we'd see a long and thin subtree and it would just
        # look like a path

        leftwidth = st_width * (alpha * .5 + (1 - alpha) * card_subtree[left[node]]
                    / card_subtree[node])
        leftpos = curpos - (st_width - leftwidth) / 2

        T.add_node(left[node], pos=(leftpos, - depth))
        T.add_edge(node, left[node])
        setpos(T, left[node], leftpos, leftwidth, depth + 1)

    if node in right.keys():

        rightwidth = st_width * (alpha * .5 + (1 - alpha) * card_subtree[right[node]]
                    / card_subtree[node])
        rightpos = curpos + (st_width - rightwidth) / 2

        T.add_node(right[node], pos=(rightpos, - depth))
        T.add_edge(node, right[node])
        setpos(T, right[node], rightpos, rightwidth, depth + 1)
```

This is the only operation we need to be carried out at every node: given a node number, `newnode`, and its parent, `parent`, we store the information in the `left` and `right` arrays so that at the end of the BB we have an explicit BB tree stored in these arrays.

```
def storeBBnode(prob, Tree, parent, newnode, branch):
    # Tree is the callback data, and it's equal to T

    if branch == 0:
        left[parent] = newnode
    else:
        right[parent] = newnode
```

We now set up the BB tree data and create a problem. We read it from a local file, but any user problem can be read and analyzed. We set the node callback with `addcbnewnode` so that we can collect information at each new node. We also save the initial time for use by message_addtime, the function that is called every time the problem prints out a message.

```
T = nx.Graph()

left  = {}
right = {}
card_subtree = {}
pos = {}

start_time = time.time()

p = xp.problem()
p.addcbmessage(message_addtime)

p.readProb('sampleprob.mps.gz')
p.addcbnewnode(storeBBnode, T, 100)
p.controls.maxnode=40000 # Limit the number of nodes inserted in the graph
p.optimize()

postorder_count(1)      # assign card_subtree to each node
setpos(T, 1, 0.5, 1, 0) # determine the position of each node
                         # depending on subtree cardinalities

pos = nx.get_node_attributes(T, 'pos')
```

```
nx.draw(T, pos) # create BB tree representation
plt.show()      # display it; you can zoom indefinitely and see all subtrees
```

6.3.2 Query and modify a simple problem

This example shows how to change an optimization problem using the Xpress Python interface.

```
p = xp.problem()

x = p.addVariable()
y = p.addVariable()

cons1    =   x + y >= 2
upperlim = 2*x + y <= 3

p.setObjective((x-4)**2 + (y-1)**2)
p.addConstraint(cons1, upperlim)

p.writeProb('original', 'lp')
```

After saving the problem to a file, we change two of its coefficients. Note that the same operations can be carried out with a single call to `p.chgMCoef([cons1,1],[x,0],[3,4])`.

```
p.chgCoef(cons1, x, 3) # coefficient of x in cons1    becomes 3
p.chgCoef(1, 0, 4)     # coefficient of y in upperlim becomes 4

p.writeProb('changed', 'lp')
```

6.3.3 Change a problem after solution

Construct a problem using addVariable and addConstraint, then use the Xpress API routines to amend the problem with rows and quadratic terms.

```
import xpress as xp

p = xp.problem()
N = 5
S = range(N)

x = [p.addVariable(vartype=xp.binary) for i in S]

# Vectors of variables can be used whole or addressed with an index or
# index range

c0 = xp.Sum(x) <= 10
cc = [x[i]/1.1 <= x[i+1]*2 for i in range(N-1)]

p.addConstraint(c0, cc)

p.setObjective(3 - x[0])

mysol = [0, 0, 1, 1, 1, 1.4]

# add a variable with its coefficients

p.addCols([4], [0,3], [c0,4,2], [-3, 2.4, 1.4], [0], [2])
p.addNames(xp.Namespaces.COLUMN, ['Y'], 5, 5)
p.chgColType([5], ['B'])
p.writeProb("problem1", "lp")

# load a MIP solution
p.loadMipSol([0,0,1,1,1,1.4])
```

We now add a quadratic term $x_0^2 - 2x_0x_3 + x_1^3$ to the second constraint. Note that the -2 coefficient for an off-diagonal element must be passed divided by two.

```
p.addQMatrix(cc[0], [x[0],x[3],x[3]], [x[0],x[0],x[3]], [1,-1,1])
```

As constraint list `cc` was added after `c0`, it is the latter which has index 0 in the problem, while `cc[0]` has index 1.

We then add the seventh and eighth constraints:

subject to:	$x_0 + 2\,x_1 + 3x_2$	$\geqslant$	4
	$4x_0 + 5x_1 + 6x_2 + 7\,x_3 + 8\,x_4$ -3 y	$\leqslant$	4.4

Note the new column named 'Y' is added with its index 5 (variables' indices begin at 0). The same would happen if 5 were substituted by `Y`.

```
p.addQMatrix(1, [x[0],x[3],x[3]], [x[0],x[0],x[3]], [1,-1,1])

p.addRows(rowtype=['G', 'L'],
          rhs=[4, 4.4],
          start=[0, 3, 9],
          colind=[x[0],x[1],x[2], x[0],x[1],x[2],x[3],x[4], 5],
          rowcoef=[1,2,3,4,5,6,7,8,-3])
p.addNames(xp.Namespaces.ROW, ['newcon1', 'newcon2'], 5, 6)

p.optimize()
p.writeProb("amended", "lp")

slacks = p.calcSlacks(solution=mysol)

print("slacks:", slacks)
```

The code below first adds five columns, then solves the problem and prints the solution, if one has been found.

```
p.addCols([4], [0, 3], [c0, 4, 2], [-3, -2, 1], [0], [2])
p.addCols([4], [0, 3], [c0, 4, 2], [-3, 2.4, 1.4], [0], [10])
p.addCols([4], [0, 3], [c0, 4, 2], [-3, 2, 1], [0], [1])
p.addCols([4], [0, 3], [c0, 4, 2], [-3, 2.4, 4], [0], [2])
p.addCols([4], [0, 3], [c0, 4, 2], [-3, 2, 1], [0], [2])

p.chgColType(range(6, 11), ['I','C','S','P','R'])

p.optimize()

try:
    print("new solution:", p.getSolution())
except:
    print("could not get solution, perhaps problem is infeasible")
```

Note that the single command below has the same effect as the four `addCols` calls above, and is to be preferred when adding a large number of columns for reasons of efficiency.

```
p.addCols([4, 4, 4, 4, 4],
              [0, 3, 6, 9, 12, 15],
              [c0, 4, 2, c0, 4, 2, c0, 4, 2, c0, 4, 2, c0, 4, 2],
              [3,  -2,  1,  -3,  2.4,  1.4, 3, 2, 1, -3, 2.4, 4, 3, 2, 1],
              [0,0,0,0,0],
              [2,10,1,2,2])
```

6.3.4 Comparing the coefficients of two equally sized problems

Given two problems with the same number of variables, we read their coefficient matrices into Scipy so as to compare each row for discrepancies in the coefficients. We begin by creating two Xpress problems and reading them from two files, `prob1.lp` and `prob2.lp`, though `p1` and `p2` might have been created with the module's modeling features.

```
import xpress as xp
import scipy.sparse

p1 = xp.problem()
p2 = xp.problem()

p1.readProb('prob1.lp')
p2.readProb('prob2.lp')
```

Next we obtain the matrix representation of the coefficient matrix for both problems.

```
beg1, ind1, coef1 = p1.getRows(0, p1.attributes.rows - 1)
beg2, ind2, coef2 = p2.getRows(0, p2.attributes.rows - 1)
```

The next step is to create a Compressed Sparse Row (CSR) format matrix, defined in the `scipy.sparse` module, using the data from `problem.getRows` plus the numerical indices.

Then we convert the CSR matrix to a NumPy array of arrays, so that each row is a (non-compressed) array to be compared in the loop below.

```
A1 = scipy.sparse.csr_matrix((coef1, ind1n, beg1))
A2 = scipy.sparse.csr_matrix((coef2, ind2n, beg2))
```

```
M1 = A1.toarray()
M2 = A2.toarray()

for i in range(min(p1.attributes.rows, p2.attributes.rows)):
    print(M1[i] != M2[i])
```

The result is a few vectors of size `COLS` with an element-wise comparison of the coefficient vector of each row, with `True` indicating discrepancies. A more meaningful representation can be given using other functions in NumPy.

```
[False False  True False False]
[False False False False False]
[False False False False  True]
[ True  True False False False]
[False False False False False]
```

6.3.5 Combining modeling and API functions

This is an example where a problem is loaded from a file, solved, then modified by adding a Global Upper Bound (GUB) constraint. Note that we do not know the structure of the problem when reading it, yet we can simply extract the list of variables and use them to add a constraint.

```
import xpress
p = xpress.problem()

p.readProb("example.lp")
p.optimize()
print("solution of the original problem: ", p.getVariable(), "==>", p.getSolution())
```

After solving the problem, we obtain its variables through `getVariable` and add a constraints so that

their sum cannot be more than 1.1.

```
x = p.getVariable()
p.addConstraint(xpress.Sum(x) <= 1.1)
p.optimize()
print("New solution: ", p.getSolution())
```

6.3.6 A simple Traveling Salesman Problem (TSP) solver

A classical example of use of callbacks is the development of a simple solver for the well-known TSP problem. The aim here is not to create an efficient solver (there are far better implementations), but rather a simple solver where the user only needs to specify two callbacks: one for checking whether a given solution forms a Hamiltonian tour and one for separating a subtour elimination constraint from the current node solution.

After a successful solve (or an interrupted one with a feasible solution), the best Hamiltonian tour is displayed. Note that this section omits unnecessary details (checks of return values, exceptions, etc.) of the actual code, which can be found in the `Examples/` directory.

```
import networkx as nx
import xpress   as xp
import re, math, sys

from matplotlib import pyplot as plt

import urllib.request as ul

filename = 'dj38.tsp'

ul.urlretrieve('http://www.math.uwaterloo.ca/tsp/world/' + filename, filename)

instance = open(filename, 'r')
coord_section = False
points = {}

G = nx.Graph()
```

We have downloaded an instance of the TSP and now it must be read and interpreted as it does not have a format that we know. We save in `cx` and `cy` the coordinates of all nodes in the graph, which is assumed to be *complete*, i.e., all nodes are connected to one another.

```
for line in instance.readlines():

    if re.match('NODE_COORD_SECTION.*', line):
        coord_section = True
        continue
    elif re.match('EOF.*', line):
        break

    if coord_section:
        coord = line.split(' ')
        index = int(coord[0])
        cx    = float(coord[1])
        cy    = float(coord[2])
        points[index] = (cx, cy)
        G.add_node(index, pos=(cx, cy))
```

The next step is to define a callback function for checking if the solution forms a Hamiltonian tour, i.e., if it connects all nodes of the graph. The callback will be passed with the method `addcbpreintsol`, therefore it needs to return a tuple of two values: the first value is `True` if the solution should be rejected, and the second is the new cutoff in case it has to be changed. This is not the case here, so `None` can be safely returned.

After obtaining the integer solution to be checked, the function scans the graph from node 1 to see if the solutions at one form a tour.

```
def check_tour(prob, G, isheuristic, cutoff):

    s = prob.getCallbackSolution()

    orignode = 1
    nextnode = 1
    card     = 0

    while nextnode != orignode or card == 0:

        FS = [j for j in V if j != nextnode
              and s[prob.getIndex(x[nextnode,j])] == 1] # forward star
        card += 1

        if len(FS) < 1:
            return (True, None) # reject solution if we can't close the loop

        nextnode = FS[0]

    # If there are n arcs in the loop, the solution is feasible

    return (card < n, None) # accept the cutoff: return second element as None
```

The second callback to be defined is a separator for subtour elimination constraints. It must return a nonzero value if the node is deemed infeasible by the function, zero otherwise. The function `addCuts` is used to insert a subtour elimination constraint.

The function works as follows: Starting from node 1, gather all connected nodes of a loop in `connset`. If this set contains all nodes, then the solution is valid if integer, otherwise the function adds a subtour elimination constraint in the form of a clique constraint with all arcs (i, j) for all i, j in `connset`.

```
def eliminate_subtour(prob, G):

    s = prob.getCallbackSolution()

    orignode = 1
    nextnode = 1

    connset  = []

    while nextnode != orignode or len(connset) == 0:

        connset.append(nextnode)

        FS = [j for j in V if j != nextnode
              and s[prob.getIndex(x[nextnode, j])] == 1] # forward star

        if len(FS) < 1:
            return 0

        nextnode = FS[0]

    if len(connset) < n:

        # Add a subtour elimination using the nodes in connset (or, if
        # card(connset) > n/2, its complement)


        if len(connset) <= n/2:
            columns = [x[i,j] for i in connset for j in connset
                       if i != j]
            nArcs  = len(connset)
        else:
            columns = [x[i,j] for i in       V for j in       V
```

```
                        if not i in connset and not j in connset and i != j]
            nArcs  = n - len(connset)

        nTerms = len(columns)

        prob.addCuts([1], ['L'], [nArcs - 1], [0, nTerms], columns, [1] * nTerms)

    return 0
```

We now formulate the problem with the degree constraints on each node and the objective function (the cost of each arc (i, j) is assumed to be the Euclidean distance between i and j).

```
n = len(points)                              # number of nodes
V = range(1, n+1)                             # set of nodes
A = [(i,j) for i in V for j in V if i != j]  # set of arcs (i.e. all pairs)

p = xp.problem()

x = {(i, j): p.addVariable(name='x_{0}_{1}'.format(i, j),
                           vartype=xp.binary) for (i, j) in A}

conservation_in  = [xp.Sum(x[i,j] for j in V if j != i) == 1 for i in V]
conservation_out = [xp.Sum(x[j,i] for j in V if j != i) == 1 for i in V]

p.addConstraint(conservation_in, conservation_out)

xind = {(i,j): p.getIndex(x[i,j]) for (i,j) in x.keys()}

# Objective function: total distance travelled
p.setObjective(xp.Sum(math.sqrt((points[i][0] - points[j][0])**2 +
                                (points[i][1] - points[j][1])**2) *
                                 x[i,j]
                      for (i,j) in A))

p.controls.timelimit = -20 # negative for "stop even if no solution is found"

p.addcboptnode(eliminate_subtour, G, 1)
p.addcbpreintsol(check_tour,        G, 1)
```

We now solve the problem, and if a solution is found it is displayed using the Python library `matplotlib`.

```
p.optimize()

sol = p.getSolution()

# Read solution and store it in the graph

for (i,j) in A:
    if sol[p.getIndex(x[i,j])] > 0.5:
        G.add_edge(i,j)

# Display best tour found

pos = nx.get_node_attributes(G, 'pos')

nx.draw(G, points) # create a graph with the tour
plt.show()         # display it interactively
```

Another solver for TSP problems is available in `example_tsp_numpy.py`. The two main differences consist in the problem generation, which is now random, and in the fact that most data structures are NumPy vectors and matrices: the optimization variables, the LP solution obtained from the Branch-and-Bound, and the data used to check feasibility of the solutions.

6.3.7 Solving a nonconvex MIQCQP

In this example we turn the Xpress Optimizer into a solver for nonconvex MIQCQPs, i.e. problems with nonconvex quadratic objective and/or nonconvex quadratic constraints.

In order to handle nonconvex quadratic constraints, we have to reformulate the problem to a MILP so that the simplest nonlinear terms, i.e. the products of variables, are transformed into new, so-called *auxiliary* variables.

Product $x_i x_j$ is assigned to a new variable w_{ij} so that every occurrence of that product in the problem is replaced by w_{ij}. Assuming l_i and u_i are the lower and upper bound on x_i, respectively, we add the linear *McCormick inequalities*:

- $w_{ij} \geqslant l_j x_i + l_i x_j - l_j l_i$
- $w_{ij} \geqslant u_j x_i + u_i x_j - u_j u_i$
- $w_{ij} \leqslant l_j x_i + u_i x_j - l_j u_i$
- $w_{ij} \leqslant u_j x_i + l_i x_j - u_j l_i$

The bounds on the new auxiliary variable w_{ij} are a function of the bounds on x_i and x_j.

Below is the code that takes care of reformulating the problem. We first have to identify all terms $x_i x_j$ and create a dictionary linking each pair *(i,j)* to an auxiliary variable w_{ij}. The dictionary `aux` is used throughout the solver and contains this information. The function `create_prob` checks all bilinear terms and creates `aux` and the McCormick inequalities.

```
def create_prob(filename):

    [...]

    x = p.getVariable()

    aux = {}  # Dictionary containing the map (x_i,x_j) --> y_ij

    [...]

    p.addConstraint(
        [aux[i, j] >= lb[j]*x[i] + lb[i]*x[j] - lb[i] * lb[j]
         for (i, j) in aux.keys() if max(-lb[i], -lb[j]) < xp.infinity],
        [aux[i, j] >= ub[j]*x[i] + ub[i]*x[j] - ub[i] * ub[j]
         for (i, j) in aux.keys() if max(ub[i], ub[j]) < xp.infinity],
        [aux[i, j] <= ub[j]*x[i] + lb[i]*x[j] - lb[i] * ub[j]
         for (i, j) in aux.keys() if max(-lb[i], ub[j]) < xp.infinity],
        [aux[i, j] <= lb[j]*x[i] + ub[i]*x[j] - ub[i] * lb[j]
         for (i, j) in aux.keys() if max(ub[i], -lb[j]) < xp.infinity])
```

We also needs to tell the Optimizer that the newly created auxiliary variables and the variables that used to appear in bilinear terms should be protected against deletion by the presolver.

```
    securecols = list(aux.values())
    secureorig = set()

    for i, j in aux.keys():
        secureorig.add(i)
        secureorig.add(j)

    securecols += list(secureorig)

    p.loadSecureVecs(rowind=None, colind=securecols)
```

The creation of a single auxiliary variable is done in `addaux`, where its bounds are created and,

depending on whether it is the product of two variables or the square of one, it receives a different treatment.

```
def addaux(aux, p, i, j, lb, ub, vtype):

    # Find bounds of auxiliary first
    if i != j:
        # bilinear term
        l, u = bdprod(lb[i], ub[i], lb[j], ub[j])
    elif lb[i] >= 0:
        l, u = lb[i]**2, ub[i]**2
    elif ub[i] <= 0:
        l, u = ub[i]**2, lb[i]**2
    else:
        l, u = 0, max([lb[i]**2, ub[i]**2])
```

After setting the bounds on w_{ij}, we determine its type and create the corresponing `xp.var` object.

```
    if vtype[i] == 'B' and vtype[j] == 'B':
        t = xp.binary
    elif (vtype[i] == 'B' or vtype[i] == 'I') and \
         (vtype[j] == 'B' or vtype[j] == 'I'):
        t = xp.integer
    else:
        t = xp.continuous

    # Add auxiliaries
    aux[i, j] = p.addVariable(lb=l, ub=u, vartype=t,
                        name='aux_{0}_{1}'.format(
                            p.getVariable(i).name,
                            p.getVariable(j).name))

    return aux[i, j]
```

Quadratic constraints and the quadratic objective (if any) are converted in `convQaux`, where they are replaced by a linear expression containing auxiliary variables.

```
def convQaux(p, aux, mstart, ind, coef, row, lb, ub, vtype):

    rcols = []
    rrows = []
    rcoef = []

    for i,__ms in enumerate(mstart[:-1]):
        for j in range(mstart[i], mstart[i+1]):

            J = p.getIndex(ind[j])

            if (i, J) not in aux.keys():
                y = addaux(aux, p, i, J, lb, ub, vtype)
            else:
                y = aux[i, J]

            if row < 0:  # objective
                mult = .5
            else:
                mult = 1

            if i != J:
                coe = 2 * mult * coef[j]
            else:
                coe =     mult * coef[j]

            if row < 0:
                p.chgObj([y], [coe])
            else:
                rcols.append(y)
```

```
                rrows.append(row)
                rcoef.append(coe)

    if row >= 0:

        # This is a quadratic constraint, not the objective function
        # Add linear coefficients for newly introduced variables
        p.chgMCoef(rrows, rcols, rcoef)
        # Remove quadratic matrix
        p.delQMatrix(row)

    else:

        # Objective: Remove quadratic part
        indI = []
        for i in range(len(mstart) - 1):
            indI.extend([i] * (mstart[i+1] - mstart[i]))
        # Set all quadratic elements to zero
        p.chgMQObj(indI, ind, [0] * mstart[-1])
```

The new problem, called a *reformulation*, is then solved as a MILP with a few callbacks. Given that the problem is nonconvex, we need to branch on continuous variables, those that appear in bilinear terms, and we also need to keep adding McCormick inequalities when the bounds change. This is because in branch-and-bound algorithms for nonconvex problems the linear relaxation should be exact at the extremes of the variable bound ranges.

Another callback is to decide whether to accept or not a solution that was found by the branch-and-bound: because the constraints linking w to x are missing, we must make sure that they are satisfied by a solution, and must refuse a solution that does not satisfy $w_{ij} = x_i x_j$.

```
def solveprob(p, aux):

    p.addcbpreintsol(cbchecksol, aux, 1)
    p.addcboptnode(cbaddcuts, aux, 3)
    p.addcbchgbranchobject(cbbranch, aux, 1)

    p.mipOptimize()
```

The callback functions are fundamental. The branch callback checks whether the auxiliary variables w_{ij} are satisfied, and if not it creates a branching object on either x_i or x_j. Due to the presolved nature of the problem at this point in the branch-and-bound, care must be applied in handling the variable indices, as they might have changed by the presolver to allow for a smaller problem.

```
def cbbranch(prob, aux, branch):

    lb, ub = getCBbounds(prob, len(sol))

    x = prob.getVariable()  # presolved variables

    rowmap, colmap = prob.getPresolveMap()

    invcolmap = [-1 for _ in lb]

    for i, m in enumerate(colmap):
        invcolmap[m] = i

    # Check if all auxiliaries are equal to their respective bilinear
    # term. If so, we have a feasible solution

    sol = np.array(sol)

    discr = sol[Aux_ind] - sol[Aux_i] * sol[Aux_j]
    discr[Aux_i == Aux_j] = np.maximum(0, discr[Aux_i == Aux_j])
    maxdiscind = np.argmax(np.abs(discr))
```

```
    if abs(discr[maxdiscind]) < eps:
        return branch

    i,j = Aux_i[maxdiscind], Aux_j[maxdiscind]

    yind = prob.getIndex(aux[i, j])
```

For terms of the form $w_{ii} = x_i^2$, branching might still be necessary as the curve defining it is a nonconvex set.

```
    if i == j:

        # Test of violation is done on the original
        # space. However, the problem variables are scrambled with invcolmap

        if sol[i] > lb[i] + eps and \
           sol[i] < ub[i] - eps and \
           sol[yind] > sol[i]**2 + eps and \
           sol[yind] - lb[i]**2 <= (ub[i] + lb[i]) * (sol[i] - lb[i]) - eps:

            # Can't separate, must branch. Otherwise OA or secant
            # cut separated above should be enough

            brvarind = invcolmap[i]
            brpoint = sol[i]
            brvar = x[brvarind]
            brleft = brpoint
            brright = brpoint

            assert(brvarind >= 0)

            if brvar.vartype in [xp.integer, xp.binary]:
                brleft = math.floor(brpoint + 1e-5)
                brright = math.ceil(brpoint - 1e-5)

            b = xp.branchobj(prob, isoriginal=False)

            b.addbranches(2)

            addrowzip(prob, b, 0, 'L', brleft,  [i], [1])
            addrowzip(prob, b, 1, 'G', brright, [i], [1])

            # New variable bounds are not enough, add new McCormick
            # inequalities for y = x**2: suppose x0,y0 are the current
            # solution values for x,y, yp = x0**2 and xu,yu = xu**2 are their
            # upper bound, and similar for lower bound. Then these two
            # rows must be added, one for each branch:
            #
            # y - yp <= (yl-yp)/(xl-x0) * (x - x0)  <===>
            # (yl-yp)/(xl-x0) * x - y >= (yl-yp)/(xl-x0) * x0 - yp
            #
            # y - yp <= (yu-yp)/(xu-x0) * (x - x0)  <===>
            # (yu-yp)/(xu-x0) * x - y >= (yu-yp)/(xu-x0) * x0 - yp
            #
            # Obviously do this only for finite bounds

            ypl = brleft**2
            ypr = brright**2

            if lb[i] > -1e7 and sol[i] > lb[i] + eps:

                yl = lb[i]**2
                coeff = (yl - ypl) / (lb[i] - sol[i])

                if coeff != 0:
                    addrowzip(prob, b, 0, 'G', coeff*sol[i] - ypl,
                              [i, yind], [coeff, -1])
```

```
            if ub[i] < 1e7 and sol[i] < ub[i] - eps:

                yu = ub[i]**2
                coeff = (yu - ypr) / (ub[i] - sol[i])

                if coeff != 0:
                    addrowzip(prob, b, 1, 'G', coeff*sol[i] - ypr,
                              [i, yind], [coeff, -1])

            return b
```

Similarly for bilinear terms, we must choose where to branch and on which variable.

```
    else:

        lbi0, ubi0 = lb[i], ub[i]
        lbi1, ubi1 = lb[i], ub[i]

        lbj0, ubj0 = lb[j], ub[j]
        lbj1, ubj1 = lb[j], ub[j]

        # No cut violated, must branch
        if min(sol[i] - lb[i], ub[i] - sol[i]) / (1 + ub[i] - lb[i]) > \
           min(sol[j] - lb[j], ub[j] - sol[j]) / (1 + ub[j] - lb[j]):
            lbi1 = sol[i]
            ubi0 = sol[i]
            brvar = i
        else:
            lbj1 = sol[j]
            ubj0 = sol[j]
            brvar = j

        alpha = 0.2

        brvarind = invcolmap[brvar]
        brpoint = sol[brvar]
        brleft = brpoint
        brright = brpoint

        if x[brvarind].vartype in [xp.integer, xp.binary]:
            brleft = math.floor(brpoint + 1e-5)
            brright = math.ceil(brpoint - 1e-5)

        b = xp.branchobj(prob, isoriginal=False)

        b.addbranches(2)

        addrowzip(prob, b, 0, 'L', brleft,  [brvar], [1])
        addrowzip(prob, b, 1, 'G', brright, [brvar], [1])

        # As for the i==j case, the variable branch is
        # insufficient, so add updated McCormick inequalities.
        # There are two McCormick inequalities per changed bound:
        #
        # y >= lb[j] * x[i] + lb[i] * x[j] - lb[j] * lb[i] ---> add to branch 1
        # y >= ub[j] * x[i] + ub[i] * x[j] - ub[j] * ub[i] ---> add to branch 0
        # y <= lb[j] * x[i] + ub[i] * x[j] - lb[j] * ub[i] ---> add to branch 1 if x[brvarind] == j, 0 if
        # y <= ub[j] * x[i] + lb[i] * x[j] - ub[j] * lb[i] ---> add to branch 1 if x[brvarind] == i, 0 if

        addrowzip(prob, b, 0, 'G', - ubi0 * ubj0, [yind, i, j], [1, -ubj0, -ubi0])
        addrowzip(prob, b, 1, 'G', - lbi1 * lbj1, [yind, i, j], [1, -lbj1, -lbi1])

        if brvarind == i:
            addrowzip(prob, b, 0, 'L', - lbj0 * ubi0, [yind, i, j], [1, -lbj0, -ubi0])
            addrowzip(prob, b, 1, 'L', - ubj1 * lbi1, [yind, i, j], [1, -ubj1, -lbi1])
        else:
            addrowzip(prob, b, 0, 'L', - ubj0 * lbi0, [yind, i, j], [1, -ubj0, -lbi0])
            addrowzip(prob, b, 1, 'L', - lbj1 * ubi1, [yind, i, j], [1, -lbj1, -ubi1])
```

```
        return b

    # If no branching rule was found, return none
    return branch
```

The callback for checking a solution is straightforward: for all pairs *ij*, check if the corresponding identity $w_{ij} = x_i x_j$ is satisfied, and if not, simply reject the solution.

```
def cbchecksol(prob, aux, soltype, cutoff):

    global Aux_i, Aux_j, Aux_ind

    if (prob.attributes.presolvestate & 128) == 0:
        return (1, cutoff)

    sol = []

    # Retrieve node solution
    try:
        sol = prob.getCallbackSolution()
    except:
        return (1, cutoff)

    sol = np.array(sol)

    # Check if all auxiliaries are equal to their respective bilinear
    # term. If so, we have a feasible solution

    refuse = 1 if np.max(np.abs(sol[Aux_i] * sol[Aux_j] - sol[Aux_ind])) > eps else 0

    # Return with refuse != 0 if solution is rejected, 0 otherwise;
    # and same cutoff
    return (refuse, cutoff)
```

An important part of this nonconvex solver is a function that computes a new feasible solution. The one we attempt here is rather trivial and probably not able to find good solutions, but one could add other algorithms, which for example might just use an alternative solver, and find a feasible solution, regardless of how good.

```
def cbfindsol(prob, aux):

    try:
        sol = prob.getCallbackSolution()
    except:
        return 0

    xnew = sol[:]

    # Round solution to nearest integer
    for i,t in enumerate(var_type):
        if t == 'I' or t == 'B' and \
           xnew[i] > math.floor(xnew[i] + prob.controls.miptol) + prob.controls.miptol:
            xnew[i] = math.floor(xnew[i] + .5)

    for i, j in aux.keys():
        yind = prob.getIndex(aux[i, j])
        xnew[yind] = xnew[i] * xnew[j]

    prob.addMipSol(xnew)

    return 0
```

The function for adding McCormick inequalities is perhaps the most important as it allows for the lower bound in the branch-and-bound to get tighter at every node. All violated inequalities are added for all pairs *ij*.

```
def cbaddmccormickcuts(prob, aux, sol):
    lb, ub = getCBbounds(prob, len(sol))

    cuts = []

    # Check if all auxiliaries are equal to their respective bilinear
    # term. If so, we have a feasible solution
    for i, j in aux.keys():

        yind = prob.getIndex(aux[i, j])

        if i == j:

            # Separate quadratic term

            if sol[yind] < sol[i]**2 - eps and \
               abs(sol[i]) < xp.infinity / 2:

                xk = sol[i]

                ox = xk
                oy = ox ** 2

                # Add Outer Approximation cut y >= xs^2 + 2xs*(x-xs)
                # <===> y - 2xs*x >= -xs^2
                cuts.append((TYPE_OA, 'G', - ox**2, [yind, i],
                             [1, -2*ox]))

            # Otherwise, check if secant can be of help: y0 - xl**2 >
            # (xu**2 - xl**2) / (xu - xl) * (x0 - xl)
            elif sol[yind] > sol[i]**2 + eps and \
                 sol[yind] - lb[i]**2 > (ub[i] + lb[i]) * (sol[i] - lb[i]) \
                 + eps and abs(lb[i] + ub[i]) < xp.infinity / 2:
                cuts.append((TYPE_SECANT, 'L',
                             lb[i]**2 - (ub[i] + lb[i]) * lb[i],
                             [yind, i], [1, - (lb[i] + ub[i])]))

        elif abs(sol[yind] - sol[i]*sol[j]) > eps:

            # Separate bilinear term, where i != j.  There might be at
            # least one cut violated

            if sol[yind] < lb[j]*sol[i] + lb[i]*sol[j] - lb[i]*lb[j] - eps:
                if lb[i] > -xp.infinity / 2 and lb[j] > -xp.infinity / 2:
                    cuts.append((TYPE_MCCORMICK, 'G', - lb[i] * lb[j],
                                 [yind, i, j], [1, -lb[j], -lb[i]]))
            elif sol[yind] < ub[j]*sol[i] + ub[i]*sol[j] - ub[i]*ub[j] - eps:
                if ub[i] < xp.infinity / 2 and ub[j] < xp.infinity / 2:
                    cuts.append((TYPE_MCCORMICK, 'G', - ub[i] * ub[j],
                                 [yind, i, j], [1, -ub[j], -ub[i]]))
            elif sol[yind] > lb[j]*sol[i] + ub[i]*sol[j] - ub[i]*lb[j] + eps:
                if ub[i] < xp.infinity / 2 and lb[j] > -xp.infinity / 2:
                    cuts.append((TYPE_MCCORMICK, 'L', - ub[i] * lb[j],
                                 [yind, i, j], [1, -lb[j], -ub[i]]))
            elif sol[yind] > ub[j]*sol[i] + lb[i]*sol[j] - lb[i]*ub[j] + eps:
                if lb[i] > -xp.infinity / 2 and ub[j] < xp.infinity / 2:
                    cuts.append((TYPE_MCCORMICK, 'L', - lb[i] * ub[j],
                                 [yind, i, j], [1, -ub[j], -lb[i]]))

    # Done creating cuts. Add them to the problem

    for (t, s, r, I, C) in cuts:  # cuts might be the empty list
        mcolsp, dvalp, drhsp, status = prob.presolveRow(s, I, C, r)
        if status >= 0:
            prob.addCuts([t], [s], [drhsp], [0, len(mcolsp)], mcolsp, dvalp)

    return 0
```

Another useful component of any nonconvex solver is a procedure to tighten the variable bounds based

on information that is known on other variables. For example, if new bounds are inferred on w_{ij}, possible tighter lower or upper bounds can be deduced on x_i and/or x_j.

```
def cbboundreduce(prob, aux, sol):
    cuts = []

    lb, ub = getCBbounds(prob, len(sol))

    # Check if bounds on original variables can be reduced based on
    # bounds on auxiliary ones. The other direction is already taken
    # care of by McCormick and tangent/secant cuts.

    feastol = prob.controls.feastol

    for (i,j),a in aux.items():

        auxind = prob.getIndex(a)

        lbi = lb[i]
        ubi = ub[i]
        lba = lb[auxind]
        uba = ub[auxind]

        if i == j:  # check if upper bound is tight w.r.t. bounds on
                    # x[i]

            # Forward propagation for term x[i]^2: from new bounds on x[i],
            # infer new bound for x[i]^2.

            if uba > max(lbi**2, ubi**2) + feastol:
                cuts.append((TYPE_BOUNDREDUCE, 'L', max(lbi**2, ubi**2), [auxind], [1]))

            if lbi > 0 and lba < lbi**2 - feastol:
                cuts.append((TYPE_BOUNDREDUCE, 'G', lbi**2, [auxind], [1]))
            elif ubi < 0 and lba < ubi**2 - feastol:
                cuts.append((TYPE_BOUNDREDUCE, 'G', ubi**2, [auxind], [1]))

            if uba < -feastol:
                return 1  # infeasible node
            else:
                if uba < lbi**2 - feastol:
                    if lbi > 0:
                        return 1  # infeasible node
                    else:
                        cuts.append((TYPE_BOUNDREDUCE, 'G', -math.sqrt(uba), [i], [1]))
                if uba < ubi**2 - feastol:
                    if ubi < - feastol:
                        return 1
                    else:
                        cuts.append((TYPE_BOUNDREDUCE, 'L', math.sqrt(uba), [i], [1]))

            if lba > prob.controls.feastol and lbi > 0 and lbi**2 < lba - feastol:
                cuts.append((TYPE_BOUNDREDUCE, 'G', math.sqrt(lba), [i], [1]))

        else:

            tlb, tub = bdprod(lb[i], ub[i], lb[j], ub[j])

            if lba < tlb - feastol:
                cuts.append((TYPE_BOUNDREDUCE, 'G', tlb, [auxind], [1]))

            if uba > tub + feastol:
                cuts.append((TYPE_BOUNDREDUCE, 'L', tub, [auxind], [1]))

            # For simplicity let's just assume lower bounds are nonnegative

            lbj = lb[j]
            ubj = ub[j]
```

```
            if lbj >= 0 and lbi >= 0:

                if lbi*ubj < lba - feastol:
                    cuts.append((TYPE_BOUNDREDUCE, 'G', lba / ubj, [i], [1]))
                if lbj*ubi < lba - feastol:
                    cuts.append((TYPE_BOUNDREDUCE, 'G', lba / ubi, [j], [1]))

                if lbi*ubj > uba + feastol:
                    cuts.append((TYPE_BOUNDREDUCE, 'L', uba / lbi, [j], [1]))
                if lbj*ubi > uba + feastol:
                    cuts.append((TYPE_BOUNDREDUCE, 'L', uba / lbj, [i], [1]))

    # Done creating cuts. Add them to the problem

    for (t, s, r, I, C) in cuts:  # cuts might be the empty list

        mcolsp, dvalp, drhsp, status = prob.presolveRow(s, I, C, r)
        if status >= 0:

            if len(mcolsp) == 0:
                continue
            elif len(mcolsp) == 1:
                if s == 'G':
                    btype = 'L'
                elif s == 'L':
                    btype = 'U'
                else:  # don't want to add an equality bound reduction
                    continue

                assert(dvalp[0] > 0)

                prob.chgBounds(mcolsp,[btype],[drhsp/dvalp[0]])
            else:
                prob.addCuts([t], [s], [drhsp], [0, len(mcolsp)], mcolsp, dvalp)

    return 0
```

There are a few other functions not shown here that are used in the example. These are functions for retrieving bounds withing a callback and other service functions. The example file provides commented code that can be used to improve the solver.

6.4 Creating simple problems using the low-level API functions

Below are a few examples on how to create simple LP, MIP, MIQP, and similar problems. Note that they make use of API functions that resemble the C API functions for creating problems, and are used similarly here.

6.4.1 Generating a small Linear Programming problem

In this example, we create a problem and load a matrix of coefficients, a rhs, and an objective coefficient list with the `loadLP` function. We also assign names to both rows and columns (both are optional). These data correspond to the following problem with three variables and four constraints:

minimize:	$3x_1 + 4x_2 + 5x_3$		
subject to:	$x_1 + x_3$	$\geqslant$	-2.4
	$2x_1 + 3x_3$	$\geqslant$	-3
	$2x_2 + 3x_3$	=	4
	$x_2 + x_3$	$\leqslant$	5
	$-1 \leqslant x_1 \leqslant 3$		
	$-1 \leqslant x_1 \leqslant 5$		
	$-1 \leqslant x_1 \leqslant 8$		

```
p = xp.problem()

p.loadLP("",                  # probname
              ['G','G','E', 'L'], # rowtype
              [-2.4, -3, 4, 5],   # rhs
               None,               # rng
              [3,4,5],            # objcoef
              [0,2,4,8],          # start
               None,               # collen
              [0,1,2,3,0,1,2,3],  # rowind
              [1,2,2,1,1,3,3,1],  # rowcoef
              [-1,-1,-1],         # lb
              [3,5,8])            # ub

p.writeProb("loadlp", "lp")
p.optimize()
```

We then create another variable and add it to the problem, then modify the objective function. Note that the objective function is replaced by, not amended with, the new expression. After solving the problem, it saves it into a file called `update.lp`.

```
x = p.addVariable()
p.setObjective(x**2 + 2*x + 444)
p.optimize()
p.writeProb("updated", "lp")
```

6.4.2 A Mixed Integer Quadratically Constrained Quadratic Programming problem

This example uses `loadMIQCQP` to create a Mixed Integer Quadratically Constrained Quadratic Programming problem with two Special Ordered Sets. Note that data that is not needed is simply set as `None`.

The `Examples` directory provides similar examples for different types of problems.

```
p = xp.problem()

p.loadMIQCQP("",                  # probname
              ['G','G','L', 'L'], # rowtype
              [-2.4, -3, 4, 5],   # rhs
              [0,2,4,8],          # start
              [0,1,2,3,0,1,2,3],  # rowind
              [1,1,1,1,1,1,1,1],  # rowcoef
              [0,0,0,1,1,2],      # objqcol1
              [0,1,2,1,2,2],      # objqcol2
              [2,1,1,2,1,2],      # objqcoef
              [2,3],              # qrowind
              [2,3],              # nrowqcoefs
              [1,2,0,0,2],        # rowqcol1
              [1,2,0,2,2],        # rowqcol2
```

```
                [3,4,1,1,1],        # rowqcoef
                ['I','S'],          # coltype
                [0,1],              # entind
                None,               # rng
                [3,4,5],            # objcoef
                None,               # collen
                [-1,-1,-1],         # lb
                [3,5,8],            # ub
                [0,2],              # limit
                ['1','1'],          # settype
                [0,2,4],            # setstart
                [0,1,0,2],          # setind
                [1.1,1.2,1.3,1.4])  # refval

p.optimize()
```

6.5 Translated Mosel examples

The subdirectory `modeling_examples` of the Python examples directory contains a few examples from the Mosel distribution that were adapted to the Xpress Python interface:

- `blend.py`, `blend2.py`: variants of an oil blending optimization model;
- `burglari.py`, `burglar.py`, `burglarl.py`, `burglar_rec.py`: several variants of the knapsack problem
- `catenary.py`: optimization model for finding the position of all elements of a hanging chain
- `chess.py`, `chess2.py`: two variants on the simple problem of production management;
- `coco.py`: Multiperiod production planning problem;
- `complex_test.py`: an example of complex numbers (a native type in Python
- `fstns.py`: the problem of firestation siting;
- `date_test.py`: an example of dates using the `datatime` module;
- `pplan.py`: a production planning example;
- `trans.py`: a transportation problem.

CHAPTER 7

Migrating to the linked API

Prior to Xpress 9.4, variables and constraints in the Python API were global objects that could be shared among several different problems at the same time. As a consequence, the attributes of variable and constraint objects could not be used to inspect or modify the optimization problem held in the solver after being added to it:

```
import xpress as xp
x = xp.var(ub=1)
p1 = xp.problem(x)
x.ub = 10           # Does not modify problem p1
p2 = xp.problem(x)
# Upper bound of x is 1 in p1 and 10 in p2
```

Xpress 9.4 introduces a new way to create variables and constraints that are associated with a single problem. These variables and constraints are called *linked objects*, to distinguish them from the *unlinked objects* that were used in previous releases. The attributes of linked objects are synchronized with the optimization problem:

```
import xpress as xp
p = xp.problem()
x = p.addVariable(ub=1)
x.ub = 10   # Problem p is updated with the new upper bound
```

Unlinked variables and constraints are deprecated and will be removed in a future release.

7.1 Creating linked variables

Linked variables are created using `problem.addVariable`:

```
import xpress as xp
p = xp.problem()
x = p.addVariable(name='x', lb=0, ub=5, vartype=xp.integer)
```

`addVariable` takes exactly the same arguments as the deprecated `xpress.var` constructor, so migrating code that uses unlinked variables is simple:

```
import xpress as xp

# Before (unlinked)
p = xp.problem()              # Create problem
x = xp.var(name=...)          # Create unlinked variable
p.addVariable(x)              # Add variable to problem

# After (linked)
p = xp.problem()              # Create problem
x = p.addVariable(name=...)   # Create linked variable within the problem
```

Note that the `addVariable` function was previously used to add existing unlinked variables to a problem. This is no longer necessary. The calls to `xpress.var(var_attributes...)` and `problem.addVariable(myvar)` should be replaced with a single call to `problem.addVariable(var_attributes...)`.

Several linked variables can be created at once using `problem.addVariables`:

```
import xpress as xp
p = xp.problem(x)
x = p.addVariables(10)  # Returns a NumPy array of 10 linked variables
```

`addVariables` takes exactly the same arguments as the deprecated `xpress.vars` function, so migrating code that uses unlinked variables is again simple:

```
import xpress as xp

# Before (unlinked)
p = xp.problem()        # Create problem
x = xp.vars(10)         # Create 10 unlinked variables
p.addVariable(x)        # Add variables to problem

# After (linked)
p = xp.problem()        # Create problem
x = p.addVariables(10)  # Create 10 linked variables within the problem
```

7.2 Creating linked constraints

Linked constraints are created using the same syntax as unlinked constraints: using an inequality operator or by explicitly calling the `xpress.constraint` constructor. As in previous releases, constraints must be explicitly added to the problem by calling `problem.addConstraint`. The attributes of linked constraints are synchronized with the problem:

```
import xpress as xp
p = xp.problem()
x, y = p.addVariables(2, vartype=xp.binary)
c = (x + y <= 1)
p.addConstraint(c)
c.rhs = 2     #  Problem p is updated with the new right-hand side
```

A constraint involving linked variables can only be added to the problem which owns those variables.

7.3 Creating linked SOS constraints

Linked special ordered set (SOS) constraints are created using `problem.addSOS`:

```
import xpress as xp
p = xp.problem()
x, y = p.addVariables(2)
s = p.addSOS([x, y], [1, 2])  # Create SOS1 constraint with given weights
```

`addSOS` takes exactly the same arguments as the deprecated `xpress.sos` constructor, so migrating code that uses unlinked SOS constraints is simple:

```
import xpress as xp

# Before (unlinked)
p = xp.problem()              # Create problem
x, y = xp.vars(2)             # Create unlinked variables
p.addVariable(x, y)           # Add variables
```

```
s = xp.sos([x, y], [1, 2])    # Create unlinked SOS1 constraint
p.addSOS(s)                   # Add SOS to problem

# After (linked)
p = xp.problem()              # Create problem
x, y = p.addVariables(2)      # Create linked variables
s = p.addSOS([x, y], [1, 2])  # Create linked SOS1 constraint within the problem
```

Note that the `addSOS` function was previously used to add existing unlinked SOS constraints to a problem. This is no longer necessary. The calls to `xpress.sos(sos_attributes...)` and `problem.addSOS(mysos)` should be replaced with a single call to `problem.addSOS(sos_attributes...)`.

7.4 Creating linked problems

When working with unlinked objects, the variables, constraints and objective could be provided as problem constructor arguments:

```
import xpress as xp
x, y = xp.vars(2)
p = xp.problem(x, y, x + y <= 1, 2 * x)
```

When working with linked objects, the problem must be created first, before the variables and constraints. Only the optional `name` argument may be provided in the constructor. When migrating code that uses unlinked objects, it may be necessary to move the call to the problem constructor earlier in your code, and to add explicit calls to `problem.addConstraint` and `problem.setObjective`.

```
import xpress as xp
p = xp.problem('myprob')
x, y = p.addVariables(2)
p.addConstraint(x + y <= 1)
p.setObjective(2 * x)
```

7.5 Other API functions

The following legacy functions can create either linked or unlinked objects, depending on the context: `problem.addrows`, `problem.addcols`, `problem.read`, `problem.loadproblem`. If the problem already contains unlinked objects, these functions will continue to create unlinked objects. If not, the default behaviour is now to create linked objects. The previous behaviour of creating unlinked objects can be achieved by passing the argument `unlinked=True`:

```
import xpress as xp
p = xp.problem()
p.readprob('myprob.mps', unlinked=True)   # Creates unlinked objects
```

7.6 Summary

The following usages are deprecated and will be removed in a future release:

`xpress.var` Replace these calls with `problem.addVariable`

`xpress.vars` Replace these calls with `problem.addVariables`

`xpress.sos` Replace these calls with `problem.addSOS`

`problem.addVariable(var1, var2, ...)` Remove these calls

`problem.addSOS(sos1, sos2, ...)` Remove these calls

`xpress.problem(vars, constraints, objective)` Remove constructor arguments and add explicit calls to `problem.addConstraint` and `problem.setObjective`

CHAPTER 8

Reference Manual

8.1 Using this chapter

This chapter provides a list of functions available through the Xpress Python interface. For each function, the synopsis and an example are given.

In keeping with the Xpress Optimizer's C API, the name and order of the parameters used in these functions has been retained. However, in order to make optimal use of the greater flexibility provided by Python, the argument lists and the return value of some functions has been modified so as to obtain a more compact notation.

For example, for functions with a list as an argument, the number of elements of the list is not part of the arguments. Compare the call to the C function `XPRSaddRows`, where the parameters `newrow` and `newnz` must be passed, to its Python counterpart:

```
(C)      result = XPRSaddrows (prob, n, nnz, type,
                               rhs, NULL, mstart, indices, values);
(Python) p.addRows(type, rhs, None, mstart, indices, values)
```

In the Python version, the `prob` pointer is not provided as obviously `addRows` is a method of the `problem` class. The C variables `n` and `nnz`, which are assigned to arguments `newrow` and `newnz`, respectively, of the call to `XPRSaddrows`, are not necessary in the Python call as the length of `rhs`, `mstart`, etc. is inferred from the passed lists. If the lengths of all lists passed as arguments are not consistent with one another, an error will be returned.

Because lists (or tuples, generators, iterators, sequences) can be used as parameters of all functions in this manual, their size does not need to be passed explicitly as it is detected from the parameter itself. The interface will check the consistency and the content if the vector is referred to the variables or constraints, and will return an error in case of a mismatch.

When passing (lists, arrays, dictionaries of) variables, constraints, or SOSs, there are three ways of referring to these entities: by numerical index, by Python object, and by name. For instance, consider the `problem.getSolution` method, which admits both an empty argument and one or a list of variables. If we define a variable with a name as follows

```
p = xpress.problem()
x = p.addVariable(name='myvar')
```

then we can refer to its index (which will be 0 here as it is the first variable added to the problem), by its object name, i.e., `x`, and by its given name `"myvar"`, in the three following (equivalent) ways:

```
print('x is ', p.getSolution(x))
print('x is ', p.getSolution(0))
print('x is ', p.getSolution('myvar'))
```

Another difference between the Python methods and their C API counterpart is that some *output* arguments are no longer passed (by reference) as arguments to the Python functions but rather are (part of) the value returned by the function. Where multiple scalar output parameters are returned by the C API function, some Python functions return a *tuple* containing all such output values.

The non-scalar parameters can instead be specified as lists, NumPy arrays, sequences, or generators when applicable. The output non-scalar parameters are stored as lists.

Optional parameters can be specified as `None` or skipped, provided the subsequent arguments are explicitly declared with their parameter name as Python allows:

```
p.addRows(rowtype=type, rhs=rhs, start=mstart,
          colind=indices, rowcoef=values)
```

Because the Python interface relies on the Xpress Optimizer C Application Program Interface, it is advisable to complement the knowledge in this reference manual with that of the Xpress Optimizer reference manual.

8.2 Format of the reference

The descriptions in the following pages report, for each function:

- Name;
- A short description of its purpose;
- Its synopsis, i.e., how it must be called. If it returns a value, then it will be presented as a Python assignment statement, otherwise it will be just shown as a call without a returned value; also, if it is a module function rather than a problem-specific function, it will be prefixed by `xpress`;
- A description of its arguments and whether each argument is optional;
- Error values;
- Associated controls;
- A sample usage of the function;
- Further useful information about the function;
- Related functions, parameters.

Note that all arguments defined in the remainder of this chapter as ”array” or ”vector” can be many other Python non-scalar objects: lists, generators, and NumPy arrays are admissible as parameters, except when specified (e.g. `getControl`). However, for simplicity we refer to non-scalar arguments as array.

Finally, some attributes and controls are referred to as uppercase words for clarity. For example, `ROWS` indicates the attribute ”rows” of a problem, hence it is equivalent to `problem.attributes.rows`.

8.3 Classes of the Xpress module

Below is a list of classes used in all operations of the `xpress` module. While for a few of these classes an explicit constructor exists (for instance, `xpress.problem` and `xpress.sos`), objects of other classes, like `xpress.linterm` and `xpress.expression`, cannot be created with a constructor methods but are created using algebraic operators applied to constants, variables, and other expressions.

`xpress.attr`	`xpress.branchobj`	`xpress.ctrl`
`xpress.constraint`	`xpress.expression`	`xpress.linterm`
`xpress.nonlin`	`xpress.problem`	`xpress.poolcut`
`xpress.quadterm`	`xpress.sos`	`xpress.var`
`xpress.voidstar`	`xpress.xprsobject`	

8.4 Global methods of the Xpress module

Below is a list of functions that are invoked from the Xpress module, i.e., they are not methods of the `problem` or the `branchobj` class and can be invoked after the `import` statement. The invocation is therefore as in the example that follows:

```
import xpress as xp
print(xp.getVersion())
```

`xpress.array`	Creates an instance of `xpress.ndarray`.	p. 79
`xpress.eq`	Creates an equality constraint from two expressions.	p. 92
`xpress.evaluate`	Returns the evaluation of one or more expressions for a given assignment of values to optimization variables.	p. 80
`xpress.examples`	Returns the full path to the directory of examples of the Xpress Python interface module.	p. 82
`xpress.featureQuery`	Checks if the provided feature is available in the current license used by the optimizer. [Licensing]	p. 75
`xpress.free`	Releases the Xpress environment, thus freeing up one license.	p. 83
`xpress.geq`	Creates a greater than or equal to constraint from two expressions.	p. 94
`xpress.getBanner`	Returns the banner and copyright message. [Misc]	p. 71
`xpress.getCheckedMode`	You can use this function to interrogate whether checking and validation of all Optimizer function calls is enabled for the current process. Checking and validation is enabled by default but can be disabled by `xpress.setCheckedMode`. [Misc]	p. 70
`xpress.getComputeAllowed`	Query whether the current application is allowed to use the Insight Compute interface. [Compute Interface, Global Environment]	p. 78
`xpress.getConstraintOperatorsEnabled`	Returns True if comparing two expressions using the ==, <= or >= operators creates a new constraint.	p. 89
`xpress.getDaysLeft`	Returns the number of days left until the license expires. [Licensing]	p. 74
`xpress.getOutputEnabled`	Returns True if Optimizer messages will be written to the Python output stream, False otherwise.	p. 87
`xpress.getVersion`	Returns the full Optimizer version number in the form 15.10.03, where 15 is the major release, 10 is the minor release, and 03 is the build number. [Misc]	p. 72
`xpress.getVersionNumbers`	Returns the Optimizer version numbers split into major, minor, and build number. [Misc]	p. 73
`xpress.init`	Initializes the Xpress environment prior to creating or reading a problem.	p. 84
`xpress.isEqual`	Compares two Xpress objects for equality.	p. 91
`xpress.leq`	Creates a less than or equal to constraint from two expressions.	p. 93
`xpress.setCheckedMode`	You can use this function to disable some of the checking and validation of function calls and function call parameters for calls to the Xpress Optimizer API. [Misc]	p. 69
`xpress.setComputeAllowed`	Set whether the current application is allowed to use the Insight Compute interface. [Compute Interface, Global Environment]	p. 77
`xpress.setConstraintOperatorsEnabled`	Enables or disables the behaviour where comparing two expressions using the ==, <= or >= operators creates a new constraint.	p. 90
`xpress.setDefaultControl`	Sets one of the module's controls to its default value.	p. 86
`xpress.setDefaults`	Sets the module's controls to their default values.	p. 85
`xpress.setOutputEnabled`	Enables or disables writing Optimizer messages to the Python output stream.	p. 88

xpress.setCheckedMode

Purpose

You can use this function to disable some of the checking and validation of function calls and function call parameters for calls to the Xpress Optimizer API. This checking is relatively lightweight but disabling it can improve performance in cases where non-intensive Xpress Optimizer functions are called repeatedly in a short space of time.

Please note: after disabling function call checking and validation, invalid usage of Xpress Optimizer functions may not be detected and may cause the Xpress Optimizer process to behave unexpectedly or crash. It is not recommended that you disable function call checking and validation during application development.

Topic area

Misc

Synopsis

```
xpress.setCheckedMode(checkedmode)
```

Argument

`checkedmode: bool` Pass as 0 to disable much of the validation for all Xpress function calls from the current process. Pass 1 to re-enable validation. By default, validation is enabled.

Related topics

`xpress.getCheckedMode`.

xpress.getCheckedMode

Purpose

You can use this function to interrogate whether checking and validation of all Optimizer function calls is enabled for the current process. Checking and validation is enabled by default but can be disabled by `xpress.setCheckedMode`.

Topic area

Misc

Synopsis

```
checkedmode = xpress.getCheckedMode()
```

Return value

`checkedmode: bool` 0 If checking and validation of Optimizer function calls is disabled for the current process, non-zero otherwise.

Related topics

`xpress.setCheckedMode`.

xpress.getBanner

Purpose

Returns the banner and copyright message.

Topic area

Misc

Synopsis

```
banner = xpress.getBanner()
```

Return value

`banner: str` The None terminated banner string.

Further information

This function can most usefully be employed to return extra information if a problem occurs with `xpress.init`.

Related topics

`xpress.init`.

xpress.getVersion

Purpose

Returns the full Optimizer version number in the form 15.10.03, where 15 is the major release, 10 is the minor release, and 03 is the build number.

Topic area

Misc

Synopsis

```
version = xpress.getVersion()
```

Return value

`version: str` The version string. This should be at least 16 characters.

Further information

1. This function supersedes the `VERSION` (p.**??** Xpress Optimizer Reference Manual) control, which only returns the first two parts of the version number. Release 2004 versions of the Optimizer have a three-part version number.
2. This function can be called without calling `xpress.init()` first.

Related topics

`xpress.init`, `xpress.getVersionNumbers`.

xpress.getVersionNumbers

Purpose

Returns the Optimizer version numbers split into major, minor, and build number.

Topic area

Misc

Synopsis

```
major, minor, build = xpress.getVersionNumbers()
```

Return value

`major: int` The major version number.

`minor: int` The minor version number. This is a number from 0 to 99.

`build: int` The build number. This is a number from 0 to 99.

Further information

1. This function always succeeds.
2. This function supersedes the `VERSION` (p.**??** Xpress Optimizer Reference Manual) control, which only returns the first two parts of the version number. Release 2004 versions of the Optimizer have a three-part version number.
3. This function can be called without calling `xpress.init()` first.

Related topics

`xpress.init`.

xpress.getDaysLeft

Purpose

Returns the number of days left until the license expires.

Topic area

Licensing

Synopsis

```
daysleft = xpress.getDaysLeft()
```

Return value

`daysleft: int` The number of days. For a permanent license, the return value will be $2^{32}-1$

Related topics

`xpress.getBanner`.

xpress.featureQuery

Purpose

Checks if the provided feature is available in the current license used by the optimizer.

Topic area

Licensing

Synopsis

```
status = xpress.featureQuery(feature)
```

Argument

`feature: str` The feature string to be checked in the license.

Return value

`status: int` Return status of the check, a value of 1 indicates the feature is available.

xpress.setArchConsistency

Purpose

This subroutine is deprecated and will be removed in a future release.
Sets whether to force the same execution path on various x64 CPU architecture extensions, in particular (pre-)AVX and AVX2.

Topic area

Global Environment

Synopsis

```
xpress.setArchConsistency(consistent)
```

Argument

`consistent: int`	Whether to force the same execution path: `0` Do not force the same execution path; `1` Force the same execution path (default behavior).

Further information

1. Note that, using this general environment API function is different from setting the `CPUPLATFORM` (p.**??** Xpress Optimizer Reference Manual) control. Setting `CPUPLATFORM` selects a vectorization instruction set for the barrier method.
2. Note that this setting only affects x64 platforms, it does not guarantee consistent math operations between x64 and aarch64 and neither between different aarch64 platforms.

xpress.setComputeAllowed

Purpose

Set whether the current application is allowed to use the Insight Compute interface.

Topic areas

Global Environment, Compute Interface

Synopsis

```
xpress.setComputeAllowed(allow)
```

Argument

`allow: AllowCompute` Whether the Insight Compute interface may be used; must be one of the following constants:

`xpress.AllowCompute.ALWAYS` Always allow solves to be sent to Compute.

`xpress.AllowCompute.NEVER` Never allow solves to be sent to Compute.

`xpress.AllowCompute.DEFAULT` Allow solves to be sent to Compute only from non-OEM applications.

Further information

1. This function controls whether this process would be allowed to use the Insight Compute Interface if the user tries to enable it.
2. If the user enables the Insight Compute Interface but the value specified through this function does not allow the Insight Compute Interface to be used, any solves will terminate with an immediate error. This function can be used to prevent solves being sent to Insight Compute but cannot be used to force solves to be performed locally. The purpose of this function is to allow an application to prevent the optimization model being sent to the Insight Compute Interface.

Related topics

`xpress.getComputeAllowed`.

xpress.getComputeAllowed

Purpose

Query whether the current application is allowed to use the Insight Compute interface.

Topic areas

Global Environment, Compute Interface

Synopsis

```
allow = xpress.getComputeAllowed()
```

Return value

`allow: AllowCompute` The value. Value will equal one of the following constants.

`xpress.AllowCompute.ALWAYS` Always allow solves to be sent to Compute.

`xpress.AllowCompute.NEVER` Never allow solves to be sent to Compute.

`xpress.AllowCompute.DEFAULT` Allow solves to be sent to Compute only from non-OEM applications.

Further information

This value supplied by this function describes whether this process would be allowed to use Insight 5 Compute Interface - this is not affected by whether or not the user has tried to enable use of the Insight 5 Compute interface.

Related topics

`xpress.setComputeAllowed`.

xpress.array

Purpose

Creates an instance of `xpress.ndarray`.

Synopsis

```
a = xpress.array(object, dtype=None, *, copy=True, order='K', subok=False, ndmin=0,
       like=None)
```

Further information

This is a convenience function which is equivalent to `numpy.array(args...).view(xpress.ndarray)`. See the NumPy documentation for more information.

Related topics

Other ways to create instances of this class are described at `xpress.ndarray`.

xpress.evaluate

Purpose

Returns the evaluation of one or more expressions for a given assignment of values to optimization variables.

Synopsis

```
v = xpress.evaluate(*args, problem=None, solution=None)
```

Arguments

`args` One or more objects to be evaluated. These can be variables, linear or nonlinear expressions; they can also be tuples, lists, dictionaries, or NumPy arrays of variables and expressions.

`problem` If this argument is provided and `solution` is `None`, the current solution from the problem is used when evaluating the variables, and the result will be the same as `problem.getSolution(*args)`.

`solution` A list, NumPy array, or dictionary of values, to be used when evaluating the variables. If a dictionary is provided, it can be keyed by `xpress.variable` objects or by variable indices.

Further information

1. Variable assignments do not have to correspond to a feasible solution.
2. At least one of the arguments `problem` and `solution` must be specified. For unlinked problems, both arguments must be provided whenever the indices of the variables are needed (i.e., if the solution is provided as a list, NumPy array, or dictionary keyed by variable index). This is because unlinked variables can belong to multiple problems, and they can have a different index in each one, so the `problem` argument is needed to resolve ambiguity.
3. When using `evaluate` with piecewise linear functions that have a step discontinuity, for example with the constraint `y == xp.pwl({(0,3): x, (3,5): 10*x})`, if at an optimal solution `x=3`the Optimizer library will compute a value for `y` that is anywhere between 3 and 30, because of numerical issues associated with discontinuities. In such cases, `evaluate` is unaware of the link between the function and `y` and, by convention, will return a value of the function that correspond to the second interval, i.e., the function will be evaluated at 30. In order to obtain the value of the piecewise linear function, evaluate should be run on `y` instead.

Example

The following examples are valid uses of `xpress.evaluate`:

```
import xpress as xp

p = xp.problem()  # Create a problem

x = p.addVariable()
y = p.addVariable(vartype=xp.binary)

# Uses evaluate without a problem but by assigning the variables
# explicitly. The result should be [5.4, 124.71633781453677].

v1 = xp.evaluate([x + y, x**3 - xp.cos(x)], solution={x:5, y:0.4})


# Similar to the computation of v1 but with a vector of numbers; the
# order in which the variables were added to p means that x=2 and
# y=3 here. The result should be {'exp1':11, 'exp2':6, 'exp3':9}.

v2 = xp.evaluate({'exp1':x + 3*y, 'exp2':x*y, 'exp3':y**2},
                 solution=[2,3])

p.addConstraint(x + y >= 3)
p.setObjective(x + 2*y)

p.optimize()

l = xp.array([x**2 * y, x * y**2, x**3])

# No solution is passed, so the solution of p as computed with optimize()
# above is used. It is easy to show that the solution is x=3, y=0, so
```

```
# the result is np.array([0, 0, 27]).

v3 = xp.evaluate(1, problem=p)
```

Related topics

problem.getSolution.

xpress.examples

Purpose

Returns the full path to the directory of examples of the Xpress Python interface module.

Synopsis

```
xpress.examples()
```

Further information

The `modeling_examples/` subdirectory contains some of the Mosel examples translated into their Python counterpart.

xpress.free

Purpose

Releases the Xpress environment, thus freeing up one license. The subsequent creation of a problem automatically triggers a call to `xpress.init`.

Synopsis

```
xpress.free ()
```

Example

The following example shows how to call `xpress.free` and a possible use:

```
p = xp.problem() # This would imply a call to xp.init()
x = p.addVariable()
y = p.addVariable()
p.addConstraint(x+y <= 1)
p.setObjective(x+2*y, sense=xp.ObjSense.MAXIMIZE)
p.optimize()
xp.free() # from this point on, the license
          # can be claimed by other users
```

Further information

1. Similar to a call to `XPRSfree()` of the C API, calling `xpress.free` cleans the Xpress environment. Any problem created prior to a call to `xpress.free` is no longer valid, and attempting to use it will raise an `xpress.ModelError`.
2. Instead of using this function, it is recommended to call `xpress.init` in a `with` statement, which implicitly calls `xpress.free` at the end of the statement.

Related topics

`xpress.init`

xpress.init

Purpose

Initializes the Xpress environment prior to creating or reading a problem.
Note that it is **not** necessary to call this function after importing the Xpress module and before creating or solving a problem, since the environment will be automatically initialized when it is needed for the first time. However, you may want to call `xpress.init` in a `with` statement, which allows you to:

- detect initialization errors (which will be raised as an `xpress.ModelError`);
- specify the path to your license file;
- explicitly acquire the Xpress license;
- automatically release the Xpress license at the end of the `with` statement.

Synopsis

```
xpress.init(lic_path=None)
```

Argument

`lic_path` (optional) Path to the Xpress license file.

Example

The following example shows how to call `xpress.init` and why it could be useful:

```
try:
  with xp.init():     # Acquire the Xpress license
    p = xp.problem()  # This would imply a call to xp.init() if it had not already been called
    x = p.addVariable()
    y = p.addVariable()
    p.addConstraint(x+y <= 1)
    p.setObjective(x+2*y, sense=xp.ObjSense.MAXIMIZE)
    p.optimize()
  # Xpress license is implicitly released here
catch xp.ModelError:
  print('Failed to initialize Xpress')
```

Further information

`xpress.init` may emit a `xpress.LicenseWarning` if the path to the license file is not explicitly indicated with the `lic_path` argument or the `XPAUTH_PATH` environment variable. This warning can be suppressed by adding the following code before the call to `xpress.init`:

```
import warnings
warnings.simplefilter('ignore', xpress.LicenseWarning)
```

Related topics

`xpress.free`

xpress.setDefaults

Purpose

Sets the module's controls to their default values. This affects all problems created after calling `setDefaults`, not before.

Synopsis

```
xpress.setDefaults()
```

Example

The following creates two problems, one before and one after calling `setDefaults()`:

```
xpress.controls.presolve = 0
p1 = xpress.problem()
xpress.setDefaults()
p2 = xpress.problem()
print('Check p1.controls.presolve is 0:          ', p1.controls.presolve)
print('Check p2.controls.presolve is its default:', p2.controls.presolve)
```

Related topics

`xpress.setDefaultControl`, `problem.setDefaults`, `problem.setDefaultControl`.

xpress.setDefaultControl

Purpose

Sets one of the module's controls to its default value. This affects all problems created after calling `setDefaults`, not before.

Synopsis

```
xpress.setDefaultControl(control)
```

Argument

`control` Name of the control to be set to default.

Example

The following creates two problems, one before and one after calling `setDefaultControl('presolve')`:

```
xpress.controls.presolve = 0
p1 = xpress.problem()
xpress.setDefaultControl('presolve')
p2 = xpress.problem()
print('I bet p1.controls.presolve is 0:          ', p1.controls.presolve)
print('I bet p2.controls.presolve is its default:', p2.controls.presolve)
```

Related topics

`xpress.setDefaults`, `problem.setDefaults`, `problem.setDefaultControl`.

xpress.getOutputEnabled

Purpose

Returns True if Optimizer messages will be written to the Python output stream, False otherwise.

Synopsis

```
enabled = xpress.getOutputEnabled()
```

Related topics

`xpress.setOutputEnabled`, `problem.getOutputEnabled`, `problem.setOutputEnabled`.

xpress.setOutputEnabled

Purpose

Enables or disables writing Optimizer messages to the Python output stream.

Synopsis

```
xpress.setOutputEnabled(enabled)
```

Argument

`enabled` True if Optimizer messages should be written to the Python output stream, False otherwise.

Further information

By default, Optimizer messages are printed to `stdout`. This behaviour can be disabled for the whole Xpress interface by calling `xpress.setOutputEnabled` with argument `False`. This does not affect user message callbacks added with `problem.addMessageCallback`.

Related topics

`xpress.getOutputEnabled`, `problem.setOutputEnabled`, `problem.addMessageCallback`, `problem.removeMessageCallback`.

xpress.getConstraintOperatorsEnabled

Purpose

Returns True if comparing two expressions using the ==, <= or >= operators creates a new constraint.

Synopsis

```
enabled = xpress.getConstraintOperatorsEnabled()
```

Return value

`enabled` True if constraint operators are enabled, otherwise False.

Related topics

`xpress.setConstraintOperatorsEnabled`.

xpress.setConstraintOperatorsEnabled

Purpose

Enables or disables the behaviour where comparing two expressions using the ==, <= or >= operators creates a new constraint.

Synopsis

```
xpress.setConstraintOperatorsEnabled(enabled)
```

Argument

`enabled` True if comparing two expressions should create a new constraint, False if comparing two expressions should return a Boolean.

Example

The example below disables constraint operators until a subsequent call to `xpress.setConstraintOperatorsEnabled(True)`:

```
xpress.setConstraintOperatorsEnabled(False)
```

The example below disables constraint operators for the duration of the following code block, and then automatically re-enables them afterwards:

```
with xpress.setConstraintOperatorsEnabled(False):
  run_unit_tests()
```

Further information

1. By default, an expression like `x == y`, where `x` and `y` are `xpress.var` objects creates a new constraint, fixing `x` and `y` to the same value. This is sometimes not the desired behaviour, for example during unit testing, where you might want to determine whether two expressions have the same algebraic form.
2. Whether or not constraint operators are enabled, constraints can always be created using the `xpress.constraint` constructors (or `xpress.eq`, `xpress.leq` and `xpress.geq` functions), and expressions can always be compared for equality using `xpress.isEqual`.

Related topics

`xpress.getConstraintOperatorsEnabled`, `xpress.isEqual`, `xpress.constraint`, `xpress.eq`, `xpress.leq`, `xpress.geq`.

xpress.isEqual

Purpose

Compares two Xpress objects for equality.

Synopsis

```
result = xpress.isEqual(arg1, arg2)
```

Arguments

`arg1` First object to compare.

`arg2` Second object to compare.

Return value

`result` True if `arg1` and `arg2` represent the same expression, otherwise False.

Further information

When comparing variables, linear expressions, quadratic expressions, and constant values, `xpress.isEqual` returns True for all algebraically equivalent objects. However, when comparing nonlinear expressions, `xpress.isEqual` only returns True for expressions with the same algebraic form. For example, the linear expressions `3 * x + 2 * y` and `2 * y + 3 * x` are considered to be equal, whereas the nonlinear expressions `cos(x) + sin(x)` and `sin(x) + cos(x)` are not considered to be equal, because the operands are ordered differently. This is for performance reasons.

Related topics

`xpress.setConstraintOperatorsEnabled`.

xpress.eq

Purpose

Creates an equality constraint from two expressions.

Synopsis

```
constraint = xpress.eq(lhs, rhs)
```

Arguments

`lhs` Left-hand side expression.

`rhs` Right-hand side expression.

Return value

`constraint` An equality constraint that enforces `lhs == rhs`.

Related topics

`xpress.constraint`.

xpress.leq

Purpose

Creates a less than or equal to constraint from two expressions.

Synopsis

```
constraint = xpress.leq(lhs, rhs)
```

Arguments

`lhs` Left-hand side expression.

`rhs` Right-hand side expression.

Return value

`constraint` A constraint that enforces `lhs <= rhs`.

Related topics

`xpress.constraint`.

xpress.geq

Purpose

Creates a greater than or equal to constraint from two expressions.

Synopsis

```
constraint = xpress.geq(lhs, rhs)
```

Arguments

`lhs` Left-hand side expression.

`rhs` Right-hand side expression.

Return value

`constraint` A constraint that enforces `lhs >= rhs`.

Related topics

`xpress.constraint`.

8.5 Methods to be used within a callback of a problem object

The following methods can be called from within a callback function that has been passed in one of the `problem.add*Callback` methods. Calling these functions outside of a callback may result in an error and trigger termination of the optimization process.

`problem.copycallbacks`	`problem.delCPCuts`	`problem.delCuts`
`problem.getCPCutList`	`problem.getCPCuts`	`problem.getCutList`
`problem.getCutMap`	`problem.getCutSlack`	`problem.loadCuts`
`problem.storeCuts`	`problem.strongBranchCB`	`problem.addCuts`

8.6 Xpress base classes

`xpress.attr`	Internal object class used for the attributes of an xpress.problem.	p. 96
`xpress.branchobj`	Class for branching objects.	p. 97
`xpress.ctrl`	Internal object class used for the controls of an xpress.problem.	p. 100
`xpress.expression`	Class for linear and quadratic expressions.	p. 101
`xpress.linterm`	Internal class for a linear term.	p. 102
`xpress.nonlin`	Internal class representing functions which are neither quadratic nor linear nor constant.	p. 103
`xpress.problem`	Class for all optimization problems solved by the Xpress Optimizer.	p. 105
`xpress.quadterm`	Internal class for a quadratic term.	p. 107
`xpress.voidstar`	Internal class for unspecified objects in the Xpress Optimizer library.	p. 111
`xpress.xprsobject`	Internal class for Xpress objects used within an optimization problem.	p. 112

xpress.attr

Purpose

Internal object class used for the attributes of an `xpress.problem`. The user can read attributes from a problem, but cannot create objects of this class. Also, an attribute of a problem may be read, but it cannot be set.

Example

The following example creates a problem and then prints one of its attributes:

```
import xpress as xp

p = xp.problem(x)

x = [p.addVariable() for _ in range(10)]

print(p.attributes.cols, "variables")  # will print "10 variables"
```

Related topics

`problem.getAttrib`.

xpress.branchobj

Purpose

Class for branching objects. These objects are created by the user within a callback when directing a branch-and-bound solve toward different branching decisions.

Synopsis

```
b = xpress.branchobj(prob, branches=None, isoriginal=True)
```

Arguments

`prob` Problem object.

`branches` List or tuple of branching decisions. If it is a tuple, its members are constraints of distinct branches; if it is a list, its members must be either tuples of branching constraints, each tuple for a single branch (deprecated).

`isoriginal`

`False`	Column indices should refer to the current (presolved) node problem;
`True`	Column indices should refer to the original matrix.

Further information

The `branches` argument is deprecated. Branches should be added after creating the `branchobj` using `branchobj.addBranches` and `branchobj.addRows`.

xpress.constraint

Description

Class representing linear, quadratic, and nonlinear constraints.

Constructors

```
c = xpress.constraint(constraint=None, body=None, lb=-xpress.infinity,
ub=xpress.infinity, type=None, rhs=None, name='', rhsrange=None)
```

Constructor detail

xpress.constraint

Synopsis

```
c = xpress.constraint(constraint=None, body=None, lb=-xpress.infinity,
        ub=xpress.infinity, type=None, rhs=None, name='', rhsrange=None)
```

Arguments

`constraint` The constraint, written as a ==, <=, or >= condition between two expressions. Variables can appear on either or both sides of the sign. Example: x1 + 2 * x2 <= 4

`body` An expression indicating the function to be constrained between `lb` and `ub` or by `rhs` with an assigned `type`. It should not be used when `constraint` is defined. Example: 3 * x1 + x2

`lb` Lower bound on `body`. May not be specified at the same time as `rhs`.

`ub` Upper bound on `body`. May not be specified at the same time as `rhs`.

`type` Type of the constraint, one of:
`xpress.leq` for $\leqslant$ constraints;
`xpress.eq` for equality constraints;
`xpress.geq` for $\geqslant$ constraints;
`xpress.rng` for range constraints.

`rhs` Right-hand side of the constraint if `type` is defined. May not be specified at the same time as `lb` or `ub`.

`name` Name of the constraint.

`rhsrange` Right-hand side range of the constraint. `rhs` argument must also be specified. May not be specified at the same time as `lb` or `ub`.

Description

Creates a new constraint. Constraints can be specified in several different ways:

- Passing an existing constraint with the `constraint` argument. This is typically used to assign a name to a constraint that was created using `body <= rhs` syntax.
- Passing the `body`, `type` and `rhs` arguments. This is convenient when creating constraints of different types from arrays of data.
- Passing the `body`, `lb` and `ub` arguments. This is convenient when defining range constraints, but can also be used to define inequalities by passing an infinite upper or lower bound.

Example

Constraint declared without the explicit constructor:

```
myconstr = x1 + x2 * (x2 + 1) <= 4
```

One or more constraints (or lists of constraints) can be added to a problem via the addConstraint method:

```
m.addConstraint(myconstr)
m.addConstraint(v1 + v2 <= 3)
m.addConstraint(x[i] + y[i] <= 2 for i in range(10))
```

The `constraint` constructor argument is provided so that constraints defined with an inequality can be assigned a name:

```
myconstr = xp.constraint(x1 + x2 * (x2 + 1) <= 4, name='myconstr')
```

In order to help formulate compact problems, the Sum operator of the xpress module can be used to express sums of expressions. Its argument is a list of expressions (linear or quadratic):

```
m.addConstraint(xp.Sum([y[i]    for i in range(10)]) <= 1)
m.addConstraint(xp.Sum([x[i]**2 for i in range(9)])  <= x[9])
```

For range constraints, the `lb` and `ub` arguments are provided for convenience:

```
m.addConstraint(v1 + v2, lb=2, ub=4)  # 2 <= v1 + v2 <= 4
```

See also `problem.addConstraint`, `xpress.eq`, `xpress.leq`, `xpress.geq`.

xpress.ctrl

Purpose

Internal object class used for the controls of an `xpress.problem`. The user can read and write controls for a problem, but cannot create objects of this class.

Example

The following example creates a problem and then reads and sets a few of its controls:

```
import xpress as xp

p = xp.problem()

x = [p.addVariable() for _ in range(10)]

p.addConstraint(xp.Sum(x) >= 1)

print('miprelstop is currently", p.controls.miprelstop)

p.controls.miprelstop = 1e-7
p.controls.xslp_solver = 0

# An equivalent way to do the two lines above
p.setControl({'miprelstop': 1e-7, 'xslp_solver': 0})
```

Related topics

`problem.setControl`, `problem.getControl`.

xpress.expression

Purpose

Class for linear and quadratic expressions. These can be used and combined to create constraints and objective function of an optimization problem. The user cannot explicitly create an object of this class, but applying sum, multiplication, and squaring of variables and constants gives rise to an object of this type.

Example

An expression can be created as follows:

```
import xpress as xp

p = xp.problem()

x = p.addVariable()
y = p.addVariable()

e = x**2 + 2*y - 5
```

xpress.linterm

Purpose

Internal class for a linear term, i.e., the product of a constant by a variable. It can be used and combined to create constraints and objective function of an optimization problem. The user cannot explicitly create an object of this class.

Example

Example declaration:

```
import xpress as xp

p = xp.problem()

x = p.addVariable()

l = 2*x  # l is of type xpress.linterm
```

xpress.nonlin

Purpose

Internal class representing functions which are neither quadratic nor linear nor constant. It can be used and combined to create constraints and objective function of an optimization problem. The user cannot explicitly create an object of this class.

Example

The following creates a nonlinear expression and sets it as the objective function of a problem:

```
import xpress as xp

p = xp.problem()

x = p.addVariable()

obj = x**4 + xp.exp(x)

p.setObjective(obj)
```

xpress.poolcut

Description

Class for poolcut objects. These are used by callback functions when creating cuts within a Branch-and-bound.

These objects are created by the Optimizer within callbacks and can be used by Python callback functions to store and pass pool cuts.

xpress.problem

Purpose

Class for all optimization problems solved by the Xpress Optimizer.

Synopsis

```
p = xpress.problem(*elements=None, name='', sense=xpress.ObjSense.MINIMIZE)
```

Arguments

`elements` Variables, constraints, SOSs, or objective function of the problem. These can be specified as single objects or lists and arrays thereof. They can be listed in the same order as would be added to the problem through `problem.addVariable`, `problem.addConstraint`, `problem.addSOS`, `problem.setObjective`, i.e. by making sure that the variables appearing in a constraint or objective function appear beforehand in the list.

`name` Name of the problem, displayed on solve log or saved in the `.lp` or `.mps` file when saved with `problem.writeProb`.

`sense` Optimization sense. Can be `xpress.ObjSense.MINIMIZE` (default) or `xpress.ObjSense.MAXIMIZE`.

Example 1

An object of class `xpress.problem` can be created from scratch or read from a file. It contains a set of variables and constraints, and may have an objective function. An empty optimization problem is created as follows:

```
myproblem = xp.problem()
```

A name can be assigned to a problem upon creation:

```
myproblem = xp.problem(name='My first problem')
```

The problem has no variables or constraint at this point.

Example 2

Simply call optimize() to solve an optimization problem that was either built or read from a file. The type of solver is determined based on the type of problem: if at least one integer variable was declared, then the problem will be solved as a mixed integer (linear or quadratically constrained) problem, while if all variables are continuous the problem is solved as a linear or quadratic optimization problem.

```
m.optimize()
```

The status of a problem after solution can be found via the `solvestatus` and `solstatus` attributes, and also in the return value of the optimize function, as follows:

```
import xpress as xp

m = xp.problem()
m.read("example3.lp")
solvestatus, solstatus = m.optimize()

print("solve status:", solvestatus)
print("solution status:", solstatus)

print("solution:", m.getSolution())
```

Example 3

It is useful, after solving a problem, to obtain the value of an optimal solution. After solving a continuous or mixed integer problem, the two methods `problem.getSolution` and `problem.getSlacks` return the vector (of portions thereof) of an optimal solution or the slack of the constraints. If an optimal solution was not found but a feasible solution is available, these methods will return data based on this solution. They can be used in multiple ways as shown in the following examples:

```
import xpress as xp

m  = xp.problem()

v1 = m.addVariable()
x  = [m.addVariable(lb=-1, ub=1, vartype=xp.integer) for i in range(10)]
```

```
[...] # add constraints and objective

m.optimize()

print(m.getSolution ())            # prints a list with an optimal solution
print("v1 is", m.getSolution(v1))  # only prints the value of v1
a = m.getSolution(x)               # gets the values of all variables in the vector x
b = m.getSolution(0:4)             # gets the value of v1 and x[0], x[1], x[2]
```

After creating an empty problem, one can read a problem from a file via the read method, which only takes the file name as its argument. An already-built problem can be written to a file with the write method. Its arguments are similar to those in the Xpress-Optimizer API function XPRSwriteprob, to which we refer.

xpress.quadterm

Purpose

Internal class for a quadratic term, i.e., the product of two variables and a constant. It can be used and combined to create constraints and objective function of an optimization problem. The user cannot explicitly create an object of this class.

Example

Example declaration:

```
import xpress as xp

p  = xp.problem()

x = p.addVariable()
y = p.addVariable()

q1 = 2*x*y  # bilinear term
q2 = 3*x**2  # quadratic term
```

xpress.sos

Description

Class representing special ordered set (SOS) constraints. A SOS constraint is a modeling tool for constraining a small number of consecutive variables in a vector to be nonzero.

Constructors

`s = xpress.sos(indices, weights, type=1, name='')`
(deprecated)

Related topics

`problem.addSOS`.

Constructor detail

xpress.sos

Synopsis

`s = xpress.sos(indices, weights, type=1, name='')` *(deprecated)*
Use `problem.addSOS` to add linked SOS constraints to the problem instead.

Arguments

`indices` List of variables composing the SOS.

`weights` List of weights (one per variable). These define the order for SOS2 constraints and may be used in branching for both types.

`type` Type of SOS. Can be 1 (default) or 2.

`name` Name of the SOS.

Description

1. Weights must be sufficiently distinct (see the `SOSREFTOL` control in the Optimizer manual).
2. Unlinked SOS constraints are not tied to a problem but may exist globally in a Python program. In order for them to be included into a problem, they have to be explicitly added to that problem using `problem.addSOS`. Unlinked SOS constraints are deprecated.

Example

The following are example declarations of SOS:

```
x = [xp.var() for _ in range(10)]
set1 = xp.sos(x, [0.5 + i*0.1 for i in range(10)], type=2)
v1, v2 = xp.vars(2)
set2 = xp.sos([v1, v2], [2, 5], 2, "mysos")
```

One or more SOS can be added to a problem via the addSOS method:

```
n = 10
w = [xp.var() for i in range(n)]
p = xp.problem(w)
p.addSOS([
  xp.sos([w[i], w[i+1]], [2,3])
  for i in range(n-1)
])
```

xpress.var

Description

Class representing optimization variables.

Variables are created using `problem.addVariable` or `problem.addVariables`.

Constructors

`x = xpress.var(name=None, lb=0, ub=xpress.infinity, threshold=1, vartype=xpress.continuous)`
(deprecated)

Related topics

`problem.addVariable`, `problem.addVariables`.

Constructor detail

xpress.var

Synopsis

`x = xpress.var(name=None, lb=0, ub=xpress.infinity, threshold=1, vartype=xpress.continuous)` *(deprecated)*
Use `problem.addVariable` to add linked variables to the problem instead.

Arguments

`name` A Python UTF-8 string containing the name of the variable; a default name is assigned if the user does not specify it.

`lb` Lower bound (0 by default).

`ub` Upper bound (+infinity by default).

`threshold` The threshold for semi-continuous, semi-integer, and partially integer variables.

`vartype` Type of the variable, one among the following:
`xpress.continuous` for continuous variables;
`xpress.binary` for binary variables;
`xpress.integer` for integer variables;
`xpress.semicontinuous` for semi-continuous variables;
`xpress.semiinteger` for semi-integer variables;
`xpress.partiallyinteger` for partially integer variables.

Description

Unlinked variables are not tied to a problem but may exist globally in a Python program. In order for them to be included into a problem, they have to be explicitly added to that problem using `problem.addVariable`. Unlinked variables are deprecated.

Example

One or more variables (or vectors of variables) can be added to a problem with the addVariable method:

```
v = xp.var(lb=-1, ub=2)
m.addVariable (v)

x = [xp.var(ub=10) for i in range(10)]
y = [xp.var(ub=10, vartype=xp.integer) for i in range(10)]
m.addVariable(x,y)
```

See also

`xpress.vars`, `problem.addVariable`, `problem.addVariables`.

xpress.ndarray

Description

Subclass of `numpy.ndarray` which overrides the comparison operators to support implementations of these operators which return non-Boolean objects.

The purpose of this class is to allow expressions like `a <= b`, where `a` and/or `b` are arrays of `xpress.expression` objects, to produce an array of `xpress.constraint` objects. This is in contrast to the behaviour of `numpy.ndarray`, where comparisons with <=, == or >= always produce Boolean arrays.

`xpress.ndarray` provides the same API as `numpy.ndarray`. For more information, see the NumPy documentation.

Instances of `xpress.ndarray` can be created using the `xpress.array` function, by calling `myarray.view(xpress.ndarray)` on an existing NumPy array, or by using the `xpress.ndarray` constructor, which has the same signature as `numpy.ndarray`, as described in the NumPy documentation. In addition, all functions in the Xpress Python interface which return NumPy arrays, such as `problem.addVariables`, `xpress.Dot` and `xpress.Sum`, return instances of `xpress.ndarray`.

The <=, == or >= operators provided by `xpress.ndarray` apply these operators element-wise, delegating to the implementations of these operators provided by each element. This means that an expression like `a <= b`, where `a` and/or `b` contain a mixture of `xpress.expression` objects and numbers, will produce a result array containing a mixture of `xpress.constraint` objects (where one or both elements were expressions) and Boolean objects (where both elements were numbers).

xpress.voidstar

Purpose

Internal class for unspecified objects in the Xpress Optimizer library. This is an internal class and the user cannot create an object of this class.

xpress.xprsobject

Purpose

Internal class for Xpress objects used within an optimization problem solved by the Xpress Optimizer. The user cannot explicitly create an object of this class.

8.7 Xpress object functions

The functions in the following table are available on all Xpress objects which are a kind of expression: `xpress.var`, `xpress.expression`, `xpress.linterm`, `xpress.quadterm` and `xpress.nonlin`.

`object.extractLinear` Returns the variables and coefficients of the linear part of any expression. p. 114

`object.extractQuadratic` Returns the variables and coefficients of the quadratic part of any expression. p. 115

object.extractLinear

Purpose

Returns the variables and coefficients of the linear part of any expression.

Synopsis

```
vars, coef = a.extractLinear()
```

Arguments

`a`	An expression or variable.
`vars`	A list containing the variable objects composing the linear expression in `a`.
`coef`	A list containing the corresponding coefficients in the linear expression.

Example

The following code snippets show what is the expected result of applying `extractLinear`:

```
import xpress as xp

p  = xp.problem()

x = p.addVariable()
y = p.addVariable(name='myvar')

a = x + 2*y
b = 3*x
c = y**2 + x**2 - 6*x
d = x**5 - 7*x  # nonlinear expression

print (a.extractLinear())  # will print "([C1, myvar], [1, 2])"
assert (a.extractLinear() == ([x, y], [1, 2]))

print (b.extractLinear())  # will print "([C1], [3])"
print (c.extractLinear())  # will print "([C1], [-6])"
print (d.extractLinear())  # will print "([C1], [-7])"
```

Further information

1. Note that this operator returns variable objects, not indices, in the `vars` portion of the output tuple. To obtain indices, use the `problem.getIndex` function. Printing these lists will show the name of the associated variables, as determined by the user when creating the variable with the `name` argument or, if `name` was not provided, it will show the name as determined by the Optimizer's library (default variable names are "C"+index). See also the Modelling chapter.
2. This operator is most useful only for linear expressions with more than one element. For nonlinear expressions, the function attempts to extract as much linear information it can, but will not be able to infer linearity apart from the most obvious cases. For example, for the expression `x**4 + xp.log(xp.exp(y))`, which contains the linear term `y`, the function will return `([],[])`.

object.extractQuadratic

Purpose

Returns the variables and coefficients of the quadratic part of any expression.

Synopsis

```
vars1, vars2, coef = a.extractQuadratic()
```

Arguments

`a`	An expression or variable.
`vars1`	A list containing the first variables of each bilinear term composing the quadratic expression in `a`.
`vars2`	A list containing the second variables of each bilinear term of the quadratic expression in `a`.
`coef`	A list containing the corresponding coefficients in the quadratic expression.

Example

The following code snippets show what is the expected result of applying `extractQuadratic`:

```
import xpress as xp

p  = xp.problem()

x = p.addVariable()
y = p.addVariable()
z = p.addVariable()

a = x + 2*y + x*y + 8 * x**2
b = 3*x**2 + z + 4
c = y**2 + x**2 - 6*x*y
d = x**5 - 7*x*y - 4*x*y*z  # nonlinear expression
e = x*y + y*x  # note: same bilinear term added twice. This is compressed to 2*x*y

print (a.extractQuadratic())  # will print "([C1, C1], [C2,C1], [1,8])"
assert (a.extractQuadratic() == ([x,x], [y,x], [1,8]))

print (b.extractQuadratic())  # will print "([C1], [C1], [3])"
print (c.extractQuadratic())  # will print "([C2, C1], [C2, C1], [1, 1])"
print (d.extractQuadratic())  # will print "([C1], [C2], [-7])"
print (e.extractQuadratic())  # will print "([C1], [C2], [2])"
```

Further information

1. Similar to `object.extractLinear`, this operator returns variable objects, not indices, in the `vars` portion of the output tuple. To obtain indices, use the `problem.getIndex` function. Printing these lists will show the name of the associated variables, as determined by the user when creating the variable with the `name` argument or, if `name` is not provided, it will show the name as determined by the Optimizer's library (default variable names are "C"+index). See also the Modelling chapter.
2. This operator is most useful only for quadratic expressions with more than one element. For nonlinear, non-quadratic expressions, the function attempts to extract as much quadratic information it can, but will not be able to detect quadratic/bilinear expressions apart from the most obvious cases. For example, for the expression `x**4 + xp.sqrt(y**4)`, which contains the quadratic term `y**2`, the function will return `([],[])`.

8.8 Xpress operators

The functions in the table below provide various operations that can be used when building expressions to be used in constraints. They typically accept any Xpress objects which are a kind of expression, i.e., `xpress.var`, `xpress.expression`, `xpress.linterm`, `xpress.quadterm` and `xpress.nonlin`, and they return another expression object. For example:

```
p = xpress.problem()
x, y = p.addVariables(2)
expr = xpress.cos(x) - xpress.sin(y)
p.addConstraint(expr == 1)
```

`xpress.abs`	Returns the absolute value of a given expression	p. 117
`xpress.acos`	Returns the arccosine of a given expression.	p. 118
`xpress.And`	Returns a logical AND of two or more binary variables or expressions.	p. 119
`xpress.asin`	Returns the arcsine of a given expression.	p. 120
`xpress.atan`	Returns the arctangent of a given expression.	p. 121
`xpress.cos`	Returns the cosine of a given expression.	p. 122
`xpress.Dot`	Alternative dot-product operator for an arbitrary number of NumPy single- or multi-dimensional arrays.	p. 123
`xpress.erf`	Returns the error function with an expression as its argument.	p. 125
`xpress.erfc`	Returns the complementary error function with an expression as its argument.	p. 126
`xpress.exp`	Returns the exponential of a given expression.	p. 127
`xpress.log`	Returns the natural logarithm of a given expression.	p. 128
`xpress.log10`	Returns the base-10 logarithm of a given expression.	p. 129
`xpress.max`	Returns the maximum of one or more expressions.	p. 130
`xpress.min`	Returns the minimum of one or more expressions.	p. 131
`xpress.Or`	Returns a logical OR of two or more binary variables or expressions.	p. 132
`xpress.Prod`	Returns the product of a sequence of one or more expressions.	p. 134
`xpress.pwl`	Returns a piecewise linear function over a variable.	p. 133
`xpress.sign`	Returns the sign of an expression: 1 if positive, -1 if negative, 0 if zero.	p. 135
`xpress.sin`	Returns the sine of a given expression.	p. 136
`xpress.sqrt`	Returns the square root of an expression.	p. 137
`xpress.Sum`	Alternative sum operator for an arbitrary number of objects created by a list, tuple, generator, NumPy array, dictionary, etc.	p. 138
`xpress.tan`	Returns the tangent of a given expression.	p. 139
`xpress.user`	Creates an expression that is computed by means of a user-specified function.	p. 140

xpress.abs

Purpose

Returns the absolute value of a given expression

Synopsis

```
a = xpress.abs(arg)
```

Argument

`arg` Argument of the abs() function.

Further information

Python's native `abs` operator is equivalent to `xpress.abs` for arguments that are functions of variables.

xpress.acos

Purpose

Returns the arccosine of a given expression.

Synopsis

```
a = xpress.acos(arg)
```

Argument

`arg` Argument of the arccosine function.

Further information

Using Python's `math` library operator `math.acos` is only advisable when the argument is not an expression that depends on variables.

Related topics

`xpress.sin`, `xpress.cos`, `xpress.tan`, `xpress.asin`, `xpress.atan`.

xpress.And

Purpose

Returns a logical AND of two or more binary variables or expressions.

Synopsis

`xpress.And(variables)`

Argument

`variables` A list/array of binary variables or binary expressions

Example

The following example shows how to use `and` to model various logical constraints:

```
N = 10

p = xp.problem()  # Creates a problem

x = p.addVariables(N, vartype=xp.binary)  # Creates N binary variables

c = [1, 4, 7, 3, 5, 7, 8, 4, 4, 9]

# Sets a linear objective
p.setObjective (xp.Sum(c[i] * x[i] for i in range(N)))

# Linear constraint
p.addConstraint (xp.Sum(x) <= 6)

# Constrains the first x variable to be the conjunction of all other x's
p.addConstraint (x[0] == xp.And(x[1:]))

# Forces the logical AND between some logical expressions to
# be zero, i.e., at least one of them must be zero

p.addConstraint (xp.And([x[1] | x[4], x[2] | x[1], x[3] | x[6]]) == 0)
```

Further information

1. For `AND` functions, all variables and expressions must be binary; an error will be generated otherwise.
2. A function call `xpress.And(x1,x2,...,xk)` is equivalent to `x1 and (x2 and (x3 and ... xk))...)`.
3. Note that since `x1`, `x2`, ..., `xk`, are binary variables, `xpress.And(x1,x2,...,xk)` is equivalent to `xpress.min(x1,x2,...,xk)`.

Related topics

`problem.addGenCons`, `problem.delGenCons`, `problem.getGenCons`, `xpress.Or`.

xpress.asin

Purpose

Returns the arcsine of a given expression.

Synopsis

```
a = xpress.asin(arg)
```

Argument

`arg` Argument of the arcsine function.

Further information

Using Python's `math` library operator `math.asin` is only advisable when the argument is not an expression that depends on variables.

Related topics

`xpress.sin`, `xpress.cos`, `xpress.tan`, `xpress.acos`, `xpress.atan`.

xpress.atan

Purpose

Returns the arctangent of a given expression.

Synopsis

```
a = xpress.atan(arg)
```

Argument

`arg` Argument of the arctangent function.

Further information

Using Python's `math` library operator `math.atan` is only advisable when the argument is not an expression that depends on variables.

Related topics

`xpress.sin`, `xpress.cos`, `xpress.tan`, `xpress.asin`, `xpress.acos`.

xpress.cos

Purpose

Returns the cosine of a given expression.

Synopsis

```
a = xpress.cos(arg)
```

Argument

`arg` Argument of the cosine function.

Further information

Using Python's `math` library operator `math.cos` is only advisable when the argument is not an expression that depends on variables.

Related topics

`xpress.sin`, `xpress.tan`, `xpress.asin`, `xpress.acos`, `xpress.atan`.

xpress.Dot

Purpose

Alternative dot-product operator for an arbitrary number of NumPy single- or multi-dimensional arrays. Following the convention for dot-product, the result of `Dot` for a list of k objects $T_1, T_2, ..., T_k$ of $d_1, d_2, ..., d_k$ dimensions is an object of $d_1 + d_2 + ... + d_k - 2(k-1)$ dimensions. For each i-th factor in [1,2,...,$k-1$], the arity of the last dimension of T_i must match the arity of the penultimate dimension of T_{i+1} (or its arity if T_{i+1} is single-dimensional, i.e., a vector).

Synopsis

```
a = xpress.Dot(*terms, out)
```

Arguments

`terms` NumPy arrays whose product should be taken.

`out` (optional) NumPy array of the correct dimension and arity where the result is stored. If not provided, the dot product is returned.

Example

The following code shows some possible uses of the `Dot` operator:

```
import numpy as np
import xpress as xp

N = 10
M = 20
S = range(N)

p = xp.problem()

x  = xp.array([p.addVariable() for i in S])
x0 = np.random.random(N) # creates an N-vector of random numbers

# objective function is the squared Euclidean distance of the
# variable vector x from a fixed point x0
p.setObjective(xp.Dot((x-x0),(x-x0)))

A = np.random.random((M,N))
b = np.random.random(M)

# constraint Ax = b, random MxN matrix A and M-vector b
p.addConstraint(xp.Dot(A, x) == b)

# Create a single quadratic constraint with
# a positive semidefinite matrix Q + N^3 * I

Q = np.random.random((N,N))
p.addConstraint(xp.Dot(x, Q + N**3 * np.eye(N), x) <= 1)

# Create four quadratic constraints using an order-three
# tensor, i.e., a three-dimensional array.

k = 4

T = np.random.random((k,N,N))
q = np.random.random(k)
p.addConstraint(xp.Dot(x, T, x) <= q)
```

Further information

1. From an operational standpoint, the dot product of k multi-dimensional arrays is the result of $k-1$ dot products of two factors each, and proceeds as in the following Python code:

```
result = T[0]
for i in range(1,k):
    result = xpress.Dot(result, T[i])
```

The dot product of two multi-dimensional array T' and T'' of dimensions d' and d'' and of arities $(n_1, n_2, ..., n_{d'})$ and $(m_1, m_2, ..., m_{d''})$, respectively, is a multi-dimensional array of dimension $d' + d'' - 2$, whose arity vector is $(n_1, n_2, ..., n_{d'-1}, m_1, m_2, ..., m_{d''-2}, m_{d''})$ and whose generic element is

$v_{i_1,i_2,...,i_{d'-1},j_1,j_2,...,j_{d''-2},j_{d''}} = \sum_{1 \leqslant h \leqslant n_{d'}} t'_{i_1,i_2,...,i_{d'-1},h} \cdot t''_{j_1,j_2,...,j_{d''-2},h,j_{d''}}$.

It is assumed here that $n_{d'} = m_{d''-1}$. Two simple cases may help understand the behavior of the operator: for two single-dimensional arrays v' and v'' of size n, the result is the inner product

$\sum_{1 \leqslant h \leqslant n} v'_h \cdot v''_h$.

For two matrices A and B of sizes $m \times n$ and $n \times p$ respectively, the result is the $m \times p$ matrix C whose generic element is

$C_{ij} = \sum_{1 \leqslant h \leqslant n} A_{ih} \cdot B_{hj}$.

2. The `Dot` operator is functionally equivalent to Python's `dot` operator from the `NumPy` package. However, the Xpress `Dot` operator is the only one that can work on variables and expressions containing variables.

3. The `Dot` operator supports the most common SciPy sparse matrix formats, allowing arrays of sparse expressions and constraints to be constructed efficiently:

```
x = p.addVariables(1000)
A = scipy.sparse.csr_array(scipy.sparse.random(1000, 1000, density=0.1))
b = np.random.random(1000)
p.addConstraint(xp.Dot(A, x) <= b)
```

`xpress.Dot` can compute the product of a 1-dimensional NumPy array of variables or expressions with a sparse matrix of type `int32`, `int64`, `float32` or `float64` in CSR or CSC format. Computing the product of an N-dimensional array with a sparse matrix is not currently supported when `N > 1`.

See https://docs.scipy.org/doc/scipy/reference/sparse.html for more information about SciPy sparse matrices.

xpress.erf

Purpose

Returns the error function with an expression as its argument.

Synopsis

```
a = xpress.erf(arg)
```

Argument

`arg` Argument of the function.

Further information

For reasons related to compilers and math libraries, on Windows machines this function can only be used with Python 3.

Related topics

`xpress.erfc`.

xpress.erfc

Purpose

Returns the complementary error function with an expression as its argument.

Synopsis

```
a = xpress.erfc(arg)
```

Argument

`arg` Argument of the function.

Further information

For reasons related to compilers and math libraries, on Windows machines this function can only be used with Python 3.

Related topics

`xpress.erf`.

xpress.exp

Purpose

Returns the exponential of a given expression.

Synopsis

```
a = xpress.exp(arg)
```

Argument

`arg` Exponent.

Further information

Using Python's `math` library operator `math.exp` is only advisable when the argument is not an expression that depends on variables.

xpress.log

Purpose

Returns the natural logarithm of a given expression.

Synopsis

```
a = xpress.log(arg)
```

Argument

`arg` Argument of the log function.

Further information

Using Python's `math` library operator `math.log` is only advisable when the argument is not an expression that depends on variables.

xpress.log10

Purpose

Returns the base-10 logarithm of a given expression.

Synopsis

```
a = xpress.log10(arg)
```

Argument

`arg` Argument of the log function.

Further information

Using Python's `math` library operator `math.log10` is only advisable when the argument is not an expression that depends on variables.

Related topics

`xpress.log`.

xpress.max

Purpose

Returns the maximum of one or more expressions.

Synopsis

```
a = xpress.max(*terms, tn)
```

Argument

`terms` Expressions whose maximum value is to be taken.

Further information

Using Python's operator `max` is only advisable when the argument is not an expression that depends on variables.

Related topics

`xpress.min`.

xpress.min

Purpose

Returns the minimum of one or more expressions.

Synopsis

```
a = xpress.min(*terms, tn)
```

Argument

`terms`	Expressions whose minimum value is to be taken.

Further information

Using Python's operator `min` is only advisable when the argument is not an expression that depends on variables.

Related topics

`xpress.max`.

xpress.Or

Purpose

Returns a logical OR of two or more binary variables or expressions.

Synopsis

`xpress.Or(variables)`

Argument

`variables`	A list/array of binary variables or binary expressions

Example

The following example shows how to use `or` to model various logical constraints:

```
N = 10

p = xp.problem()  # Creates a problem

x = p.addVariables(N, vartype=xp.binary)  # Creates N binary variables

c = [1, 4, 7, 3, 5, 7, 8, 4, 4, 9]

# Sets a linear objective
p.setObjective (xp.Sum(c[i] * x[i] for i in range(N)))

# Linear constraint
p.addConstraint (xp.Sum(x) <= 6)

# Constrains the first x variable to be the conjunction of all other x's
p.addConstraint (x[0] == xp.Or(x[1:]))

# Forces the logical OR between some logical expressions to
# be one, i.e., at least one of them must be one

p.addConstraint (xp.Or([x[1] & x[4], x[2] & x[1], x[3] & x[6]]) == 1)
```

Further information

1. For `OR` functions, all variables and expressions must be binary; an error will be generated otherwise.
2. A function call `xpress.Or(x1,x2,...,xk)` is equivalent to `x1 or (x2 or (x3 or ... xk))...)`.
3. Note that since `x1`, `x2`, ..., `xk`, are binary variables, `xpress.Or(x1,x2,...,xk)` is equivalent to `xpress.max(x1,x2,...,xk)`.

Related topics

`problem.addGenCons`, `problem.delGenCons`, `problem.getGenCons`, `xpress.And`.

xpress.pwl

Purpose

Returns a piecewise linear function over a variable.

Synopsis

`xpress.pwl(dict)`

Argument

`dict`	Python dictionary containing, as keys, two-elements tuples, and, as values, linear expressions in a variable. If the piecewise linear function has only constant values (i.e. it is a piecewise constant function), the input variable can be specified with the key-value pair `None: x`.

Example

The following example shows various usages of `xpress.pwl` to model nonlinear functions as piecewise-linear functions :

```
p = xp.problem()

x = p.addVariable()  # Nonnegative variable
y = p.addVariable(lb=-xp.infinity)  # dependent variable, unrestricted
t = p.addVariable()
w = p.addVariable()

# Sets a piecewise linear objective: a ramp function
p.setObjective (xp.pwl({(-xp.infinity, -1):  -2,
                        (-1, 1):             2*x,
                        (1, xp.infinity):    2}))

p.addConstraint (t == xp.pwl({(1,2): 4*x, (2,4): 2, (4,5): -1}))

# Piecewise CONSTANT function: add a key-value pair None: x to specify
# input variable
p.addConstraint (t == xp.pwl({(1,2): 4, (2,4): 2, (4,5): -1, None: x}))

p.addConstraint (xp.pwl({(-1,0): x, (0,1): 2*x, (1,10): 2}) <=
                 xp.pwl({(0,10): 2*z, (10,20): z+2, (20,xp.infinity): 4}))
```

Further information

1. A piecewise linear function must use only one variable in all of the dictionary's values;
2. All values in the dictionary must be either constants or linear functions;
3. The intervals, specified as two-element tuples in the dictionary's keys, must be pairwise disjoint, i.e., they must not overlap.
4. Discontinuities in the function are allowed, i.e., one can declare a function as follows: `xp.pwl({(1, 2): 2*x + 4, (2,3): x - 1})`, which is obviously discontinuous at 2. The value of the function if the optimal solution has `x=2` will be then either 8 or 1.

Related topics

`problem.addPwlCons`, `problem.delPwlCons`, `problem.getPwlCons`.

xpress.Prod

Purpose

Returns the product of a sequence of one or more expressions.

Synopsis

```
a = xpress.Prod(*terms)
```

Argument

`terms`	Expressions whose product is to be taken.

Example

The following are allowed uses of the `Prod` operator:

```
n = 10
x = [p.addVariable() for i in range(n)]
prod = xp.Prod(x)
polynomial = xp.Sum(i * xp.Prod(x[i:i+4]) for i in range(n-4))
```

Further information

While n-ary product operators may exist in Python and/or NumPy, it is advisable to use `xpress.Prod` when creating products of many expressions as it is the most efficient alternative.

xpress.sign

Purpose

Returns the sign of an expression: 1 if positive, -1 if negative, 0 if zero.

Synopsis

```
a = xpress.sign(arg)
```

Argument

`arg` Argument of the sign function.

xpress.sin

Purpose

Returns the sine of a given expression.

Synopsis

```
a = xpress.sin(arg)
```

Argument

`arg` Argument of the sine function.

Further information

Using Python's `math` library operator `math.sin` is only advisable when the argument is not an expression that depends on variables.

Related topics

`xpress.cos`, `xpress.tan`, `xpress.asin`, `xpress.acos`, `xpress.atan`.

xpress.sqrt

Purpose

Returns the square root of an expression.

Synopsis

```
a = xpress.sqrt(arg)
```

Argument

`arg` Radicand of the function.

Further information

Using Python's `math` library operator `math.sqrt` is only advisable when the argument is not an expression that depends on variables.

xpress.Sum

Purpose

Alternative sum operator for an arbitrary number of objects created by a list, tuple, generator, NumPy array, dictionary, etc.

Synopsis

```
a = xpress.Sum(*terms)
```

Argument

`terms`	Expressions whose sum is to be taken.

Example

The following are allowed uses of the `Sum` operator:

```
import math
N = 20
S = range(S)
p = xpress.problem()
x = [p.addVariable() for i in S]
y = [p.addVariable(vartype=xpress.binary) for i in S]
p.setObjective(x[0] + xpress.Sum(x[i]**2 for i in S))
p.addConstraint(xpress.Sum(x,y) <= 100)
p.addConstraint(xpress.Sum(x[:i]) + xpress.Sum(y[:i])
                <= math.log(10 + i) for i in S)
```

Further information

The `Sum` operator is functionally equivalent to NumPy's `ndarray.sum` operator but it works with Xpress variables and expression objects. It reduces an n-dimensional array to a scalar expression.

xpress.tan

Purpose

Returns the tangent of a given expression.

Synopsis

```
a = xpress.tan(arg)
```

Argument

`arg` Argument of the tangent function.

Further information

Using Python's `math` library operator `math.tan` is only advisable when the argument is not an expression that depends on variables.

Related topics

`xpress.sin`, `xpress.cos`, `xpress.asin`, `xpress.acos`, `xpress.atan`.

xpress.user

Purpose

Creates an expression that is computed by means of a user-specified function. The user function can be defined to either provide or not provide the value of all derivatives w.r.t. the variables.

Synopsis

```
def f(a1, a2, ..., an[, *deltas]):
[...]
a = xpress.user(f, t1, t2, ..., tn, derivatives="never")
```

Arguments

`f`	User function; must be a Python function with as many (possibly optional) arguments as specified in the declaration.	
`t1,...,tn`	Arguments of the user function.	
`derivatives`	`"never"`	`f` does not return derivatives;
	`"always"`	`f` always returns derivatives;
	`"ondemand"`	`f` returns derivatives when they are requested by the solver (see notes below).

Example

The following code shows how to define user functions and use them in an optimization problem:

```
import math

def mynorm(*v):
  return math.sqrt(sum(e**2 for e in v))

def weighted_sum(t1, t2, t3):
  return (2*t1 + 3*t2 + 4*t1*t3,
  2 + 4*t3, 3, 4*t1)

def ondemand_derivatives(t1, t2, *deltas):
  val = 2*t1 + 4*t1*t2
  if not deltas:
    # No derivatives needed
    return val
  else:
    # Calculate whichever derivatives are needed
    d1, d2 = deltas
    return (val,
      2 + 4*t2 if d1 != 0 else None,
      4*t1 if d2 != 0 else None)

p = xp.problem()

x = [p.addVariable() for i in range(20)]

f1 = xp.user(mynorm, *x)
f2 = xp.user(weighted_sum, x[4], x[5], x[6], derivatives="always")
f3 = xp.user(ondemand_derivatives, x[0], x[1], derivatives="ondemand")

p.addConstraint(f3 >= 4)
p.addConstraint(f2 == 1)
p.setObjective(f1)
p.optimize()
print('solution:', p.getSolution(x))
```

Further information

1. User functions **must** produce a Float, as the behaviour is otherwise undefined. If the `derivatives` parameter is set to `"never"` (the default), then the function should simply return the function value. If `derivatives="always"`, the function must return a tuple consisting of the function value and the derivatives of the function w.r.t. all variables in the list of arguments. If `derivatives="ondemand"`, the function will either be called with `numArgs` arguments, or with `2 * numArgs` arguments, depending on whether derivatives are required by the solver. When derivatives are not required, only the input values will be passed to the function, and the function can simply return the function value. When derivatives are required, the function will be passed the input value arguments followed by a delta argument for every input argument, and the function must return a tuple, as when `derivatives="always"`. Note that for this reason, the delta arguments must be declared as optional, either by providing default values, or by using variable-length arguments syntax (`*deltas`). The function only needs to populate the tuple with a derivative where the corresponding delta argument is nonzero. (Where the delta argument is zero, the function should provide some placeholder value such as `None` or zero.) The delta can be used as a suggested perturbation for numerical differentiation (a negative sign indicates that if a one-sided derivative is calculated, then a backward one is preferred).
2. The variables on which the function is defined cannot be passed as lists or NumPy arrays. Lists of variables can be passed by unpacking the list: `xpress.user(lambda *y: sum(y), *list_of_vars)` Note that in this case the wrapped function must be variadic.

Related topics

`problem.addSlpPreUpdateLinearizationCallback`.

8.9 Xpress `problem` object methods

The tables below show all methods of the class `problem` of the Xpress Python interface, with the exception of callbacks, which are listed separately. Their invocation is therefore to be preceded by a problem object (the class prefix `problem.` is omitted in the table for compactness), as follows:

```
import xpress as xp
p = xp.problem()
x = p.addVariable()
p.setObjective(x + 3 * x**2 + 2)
```

`problem.addCols`	Adds columns to the optimizer matrix. [Optimizer, Problem Creation]	p. 221
`problem.addCuts`	Adds cuts directly to the matrix at the current node. [Cuts, Optimizer]	p. 297
`problem.addGenCons`	Adds one or more general constraints to the problem. [Optimizer, Piecewise Linear and General Constraints, Problem Creation]	p. 185
`problem.addManagedCuts`	Adds cuts to the Optimizer's internal cut pool from within the cutround callback set by problem.addCutRoundCallback. [Callback, Cuts, Optimizer]	p. 296
`problem.addMipSol`	Adds a new feasible, infeasible or partial MIP solution for the problem to the Optimizer. [Optimizer, Solution]	p. 327
`problem.addNames`	When a model is loaded, the rows, columns, sets, piecewise linear and general constraints of the model may not have names associated with them. [Optimizer, Problem Creation]	p. 178
`problem.addObj`	Appends an objective function with the given coefficients to a multi-objective problem. The weight and priority of the objective are set to the given values. [Multiobjective, Optimizer, Problem Creation]	p. 256
`problem.addPwlCons`	Adds one or more piecewise linear constraints to the problem. [Optimizer, Piecewise Linear and General Constraints, Problem Creation]	p. 183
`problem.addQMatrix`	Adds a new quadratic matrix into a row defined by triplets. [Optimizer, Problem Creation, Quadratic]	p. 249
`problem.addRows`	Adds rows to the optimizer matrix. [Optimizer, Problem Creation]	p. 219
`problem.addSets`	Allows sets to be added to the problem after passing it to the Optimizer using the input routines. [MIP Entities, Optimizer, Problem Creation]	p. 227
`problem.basisStability`	Calculates various measures for the stability of the current basis, including the basis condition number. [Linear Algebra, LP, Optimizer]	p. 245
`problem.bndSA`	Returns upper and lower sensitivity ranges for specified variables' lower and upper bounds. [LP, Optimizer, Sensitivity Analysis]	p. 247
`problem.BTran`	Post-multiplies a (row) vector provided by the user by the inverse of the current basis. [Linear Algebra, LP, Optimizer]	p. 270
`problem.calcObjective`	Calculates the objective value of a given solution. [Optimizer, Solution]	p. 313
`problem.calcObjN`	Calculates the objective value of the given objective function in a multi-objective problem. [Multiobjective, Optimizer, Solution]	p. 314

`problem.calcReducedCosts` Calculates the reduced cost values for a given (row) dual solution.
[Linear Algebra, Optimizer, Solution] p. 312

`problem.calcSlacks` Calculates the row slack values for a given solution.
[Optimizer, Solution] p. 311

`problem.calcSolInfo` Calculates the required property of a solution, like maximum infeasibility of a given primal and dual solution.
[Optimizer, Solution] p. 315

`problem.chgBounds` Used to change the bounds on columns in the matrix.
[Optimizer, Problem Modification] p. 324

`problem.chgCoef` Used to change a single coefficient in the matrix.
[Optimizer, Problem Modification] p. 337

`problem.chgColType` Used to change the type of a specified set of columns in the matrix.
[MIP Entities, Optimizer, Problem Modification] p. 223

`problem.chgGlbLimit` Used to change semi-continuous or semi-integer lower bounds, or upper limits on partial integers.
[MIP Entities, Optimizer, Problem Modification] p. 234

`problem.chgMCoef` Used to change multiple coefficients in the matrix.
[Optimizer, Problem Modification] p. 338

`problem.chgMQObj` Used to change multiple quadratic coefficients in the objective function. If any of the coefficients does not exist already, new coefficients will be added to the objective function.
[Optimizer, Problem Modification, Quadratic] p. 339

`problem.chgObj` Used to change the objective function coefficients.
[Optimizer, Problem Modification] p. 336

`problem.chgObjN` Modifies one or more coefficients of an objective function in a multi-objective problem.
[Multiobjective, Optimizer, Problem Modification] p. 257

`problem.chgObjSense` Changes the problem's objective function sense to minimize or maximize.
[Optimizer, Problem Modification] p. 233

`problem.chgQObj` Used to change a single quadratic coefficient in the objective function corresponding to the variable pair `(objqcol1,objqcol2)` of the Hessian matrix.
[Optimizer, Problem Modification, Quadratic] p. 340

`problem.chgQRowCoeff` Changes a single quadratic coefficient in a row.
[Optimizer, Problem Modification, Quadratic] p. 322

`problem.chgRHS` Used to change right–hand side values of the problem.
[Optimizer, Problem Modification] p. 341

`problem.chgRHSRange` Used to change the range for a row of the problem matrix.
[Optimizer, Problem Modification] p. 342

`problem.chgRowType` Used to change the type of a row in the matrix.
[Optimizer, Problem Modification] p. 343

`problem.clearIIS` Resets the search for Irreducible Infeasible Sets (IIS).
[Infeasibility, Optimizer] p. 207

`problem.clearRowFlags` Clears extra information attached to a range of rows.
[Optimizer, Problem Modification] p. 284

`problem.copyControls` Copies controls defined for one problem to another.
[Controls and Attributes, Optimizer] p. 259

`problem.crossoverLpSol` Provides a basic optimal solution for a given solution of an LP problem.
[Barrier, LP, Optimizer] p. 201

`problem.delCols` Delete columns from a matrix.
[Optimizer, Problem Creation] p. 222

`problem.delCPCuts` During the branch and bound search, cuts are stored in the cut pool to be applied at descendant nodes.
[Cuts, Optimizer] p. 299

`problem.delCuts` Deletes cuts from the matrix at the current node.
[Cuts, Optimizer] p. 298

`problem.delGenCons` Delete general constraints from a problem.
[Optimizer, Piecewise Linear and General Constraints, Problem Creation] p. 188

`problem.delIndicators` Delete indicator constraints.
[MIP Entities, Optimizer, Problem Creation] p. 191

`problem.delObj` Removes an objective function from the problem.
[Optimizer, Problem Creation] p. 258

`problem.delPwlCons` Delete piecewise linear constraints from a problem.
[Optimizer, Piecewise Linear and General Constraints, Problem Creation] p. 187

`problem.delQMatrix` Deletes the quadratic part of a row or of the objective function.
[Optimizer, Problem Creation, Quadratic] p. 250

`problem.delRows` Delete rows from a matrix.
[Optimizer, Problem Creation] p. 220

`problem.delSets` Delete sets from a problem.
[MIP Entities, Optimizer, Problem Creation] p. 226

`problem.dumpControls` Displays the list of controls and their current value for those controls that have been set to a non default value.
[Controls and Attributes, Optimizer] p. 189

`problem.estimateRowDualRanges` Performs a dual side range sensitivity analysis, i.e.
[LP, Optimizer, Sensitivity Analysis] p. 230

`problem.firstIIS` Initiates a search for an Irreducible Infeasible Set (IIS) in an infeasible problem.
[Infeasibility, Optimizer] p. 208

`problem.fixMIPEntities` Fixes all the MIP entities to the values of the last found MIP solution.
[MIP Entities, Optimizer, Problem Modification] p. 170

`problem.FTran` Pre-multiplies a (column) vector provided by the user by the inverse of the current matrix.
[Linear Algebra, LP, Optimizer] p. 271

`problem.getAttribInfo` Accesses the id number and the type information of an attribute given its name.
[Controls and Attributes, Optimizer] p. 263

`problem.getBasis` Returns the current basis into the user's data arrays.
[Linear Algebra, LP, Optimizer] p. 295

`problem.getCallbackPresolveDuals` Returns the dual values from the solution to the presolved problem associated with the current callback.
[Callback, Optimizer, Solution] p. 334

`problem.getCallbackPresolveRedCosts` Returns the reduced costs from the solution to the presolved problem associated with the current callback.
[Callback, Optimizer, Solution] p. 335

`problem.getCallbackPresolveSlacks` Returns the slack values from the solution to the presolved problem associated with the current callback.
[Callback, Optimizer, Solution] p. 333

`problem.getCallbackPresolveSolution` Returns the solution to the presolved problem associated with the current callback.
[Callback, Optimizer, Solution] p. 332

`problem.getCoef` Returns a single coefficient in the constraint matrix.
[Optimizer, Problem Information] p. 285

`problem.getCols` Returns the nonzeros in the constraint matrix for the columns in a given range.
[Optimizer, Problem Information] p. 280

`problem.getColType` Returns the column types for the columns in a given range.
[MIP Entities, Optimizer, Problem Information] p. 318

`problem.getControlInfo` Accesses the id number and the type information of a control given its name.
[Controls and Attributes, Optimizer] p. 262

`problem.getCPCutList`	Returns a list of cut indices from the cut pool. [Cuts, Optimizer] p. 301
`problem.getCPCuts`	Returns cuts from the cut pool. [Cuts, Optimizer] p. 302
`problem.getCutList`	Retrieves a list of cut objects for the cuts active at the current node. [Cuts, Optimizer] p. 300
`problem.getCutMap`	Used to return in which rows a list of cuts are currently loaded into the Optimizer. This is useful for example to retrieve the duals associated with active cuts. [Cuts, Optimizer] p. 329
`problem.getCutSlack`	Used to calculate the slack value of a cut with respect to the current LP relaxation solution. [Cuts, Optimizer] p. 328
`problem.getDirs`	Used to return the directives that have been loaded into a matrix. [Branching, Optimizer, Problem Information] p. 266
`problem.getDualRay`	Retrieves a dual ray (dual unbounded direction) for the current problem, if the problem is found to be infeasible. [LP, Optimizer, Solution] p. 293
`problem.getGenCons`	Returns the general constraints `y = f(x1, ..., xn, c1, ..., cm)` in a given range. [Optimizer, Piecewise Linear and General Constraints, Problem Information] p. 186
`problem.getIISData`	Returns information for an Irreducible Infeasible Set: size, variables and constraints (row and column vectors), and conflicting sides of the variables. For pure linear problems there is also information on duals, reduced costs and isolations. [Infeasibility, Optimizer] p. 215
`problem.getIndex`	Returns the index for a specified row or column name. [Optimizer, Problem Information] p. 264
`problem.getIndicators`	Returns the indicator constraint condition (indicator variable and complement flag) associated to the rows in a given range. [Optimizer, Problem Information] p. 190
`problem.getInfeas`	Returns a list of infeasible primal and dual variables. [LP, Optimizer, Solution] p. 198
`problem.getLastBarSol`	Used to obtain the last barrier solution values following optimization that used the barrier solver. [Barrier, Optimizer, Solution] p. 206
`problem.getLB`	Returns the lower bounds for the columns in a given range. [Optimizer, Problem Information] p. 278
`problem.getLpSol`	Used to obtain the LP solution values following optimization. [LP, Optimizer, Solution] p. 331
`problem.getMessageStatus`	Retrieves the current suppression status of a message. [Misc, Optimizer] p. 232
`problem.getMIPEntities`	Retrieves integr and entity information about a problem. It must be called before `problem.mipOptimize` if the presolve option is used. [MIP Entities, Optimizer, Problem Information] p. 282
`problem.getMQObj`	Returns the nonzeros in the quadratic objective coefficients matrix for the columns in a given range. [Optimizer, Problem Information, Quadratic] p. 286
`problem.getNameList`	Returns the names for the rows, columns, sets, piecewise linear constraints, general constraints or objectives in a given range. [Names Manager, Optimizer] p. 326
`problem.getObj`	Returns the objective function coefficients for the columns in a given range. [Optimizer, Problem Information] p. 274
`problem.getObjN`	For a given objective function, returns the objective coefficients for the columns in a given range. [Multiobjective, Optimizer, Problem Information] p. 275

`problem.getPivotOrder` Returns the pivot order of the basic variables.
[LP, Optimizer, Simplex] p. 268

`problem.getPivots` Returns a list of potential leaving variables if a specified variable enters the basis.
[LP, Optimizer, Simplex] p. 309

`problem.getPresolveBasis` Returns the current basis from memory into the user's data areas.
[LP, Optimizer, Simplex] p. 317

`problem.getPresolveMap` Returns the mapping of the row and column numbers from the presolve problem back to the original problem.
[Optimizer, Presolve, Problem Information] p. 269

`problem.getPresolveSol` Returns the solution for the presolved problem from memory.
[Optimizer, Presolve, Solution] p. 330

`problem.getPrimalRay` Retrieves a primal ray (primal unbounded direction) for the current problem, if the problem is found to be unbounded.
[LP, Optimizer, Solution] p. 292

`problem.getProbName` Returns the current problem name.
[Optimizer, Problem Information] p. 261

`problem.getPwlCons` Returns the piecewise linear constraints `y = f(x)` in a given range.
[Optimizer, Piecewise Linear and General Constraints, Problem Information] p. 184

`problem.getQObj` Returns a single quadratic objective function coefficient corresponding to the variable pair `(objqcol1, objqcol2)` of the Hessian matrix.
[Optimizer, Problem Information, Quadratic] p. 265

`problem.getQRowCoeff` Returns a single quadratic constraint coefficient corresponding to the variable pair (`rowqcol1`, `rowqcol2`) of the Hessian of a given constraint.
[Optimizer, Problem Information, Quadratic] p. 319

`problem.getQRowQMatrix` Returns the nonzeros in a quadratic constraint coefficients matrix for the columns in a given range.
[Optimizer, Problem Information, Quadratic] p. 320

`problem.getQRowQMatrixTriplets` Returns the nonzeros in a quadratic constraint coefficients matrix as triplets (index pairs with coefficients).
[Optimizer, Problem Information, Quadratic] p. 321

`problem.getQRows` Returns the list indices of the rows that have quadratic coefficients.
[Optimizer, Problem Information, Quadratic] p. 323

`problem.getRHS` Returns the right hand side elements for the rows in a given range.
[Optimizer, Problem Information] p. 276

`problem.getRHSRange` Returns the right hand side range values for the rows in a given range.
[Optimizer, Problem Information] p. 277

`problem.getRowFlags` Retrieve if a range of rows have been set up as special rows.
[Optimizer, Problem Information] p. 283

`problem.getRows` Returns the nonzeros in the constraint matrix for the rows in a given range.
[Optimizer, Problem Information] p. 281

`problem.getRowType` Returns the row types for the rows in a given range.
[Optimizer, Problem Information] p. 316

`problem.getScale` Returns the the current scaling of the matrix.
[Numerics, Optimizer] p. 267

`problem.getScaledInfeas` Returns a list primal and dual variables that are infeasible for the scaled original problem. If the problem is currently presolved, it is postsolved before the function returns.
[LP, Numerics, Optimizer] p. 199

`problem.getUB` Returns the upper bounds for the columns in a given range.
[Optimizer, Problem Information] p. 279

`problem.getUnbVec` Returns the index vector which causes the primal simplex or dual simplex algorithm to determine that a matrix is primal or dual unbounded respectively.
[LP, Optimizer, Solution] p. 200

`problem.IISAll` Performs an automated search for independent Irreducible Infeasible Sets (IIS) in an infeasible problem.
[Infeasibility, Optimizer] p. 212

`problem.IISIsolations` Performs the isolation identification procedure for an Irreducible Infeasible Set (IIS). This function applies only to linear problems.
[Infeasibility, Optimizer] p. 214

`problem.IISStatus` Returns statistics on the Irreducible Infeasible Sets (IIS) found so far by `problem.firstIIS`, `problem.nextIIS` or `problem.IISAll`.
[Infeasibility, Optimizer] p. 211

`problem.interrupt` Interrupts the Optimizer algorithms.
[Optimizer, Solution Process] p. 260

`problem.loadBasis` Loads a basis from the user's areas.
[Data Input, LP, Optimizer] p. 224

`problem.loadBranchDirs` Loads directives into the current problem to specify which MIP entities the Optimizer should continue to branch on when a node solution is integer feasible.
[Branching, Data Input, Optimizer] p. 174

`problem.loadCuts` Loads cuts from the cut pool into the matrix.
[Cuts, Data Input, Optimizer] p. 303

`problem.loadDelayedRows` Specifies that a set of rows in the matrix will be treated as delayed rows during a tree search.
[Cuts, Optimizer, Problem Input] p. 172

`problem.loadDirs` Loads directives into the matrix.
[Data Input, Optimizer] p. 173

`problem.loadLP` Enables the user to pass a matrix directly to the Optimizer, rather than reading the matrix from a file.
[LP, Optimizer, Problem Input] p. 164

`problem.loadLPSol` Loads an LP solution for the problem into the Optimizer.
[Data Input, LP, Optimizer, Solution] p. 238

`problem.loadMIP` Used to load a MIP problem into the Optimizer data structures.
[MIP Entities, Optimizer, Problem Input] p. 176

`problem.loadMipSol` Loads a starting MIP solution for the problem into the Optimizer.
[Data Input, MIP Entities, Optimizer, Solution] p. 294

`problem.loadMIQCQP` Used to load a mixed integer quadratic problem with quadratic constraints into the Optimizer data structure.
[Optimizer, Problem Input, Quadratic] p. 253

`problem.loadMIQP` Used to load a MIQP problem, hence a MIP with quadratic objective coefficients, into the Optimizer data structures.
[Optimizer, Problem Input, Quadratic] p. 168

`problem.loadModelCuts` Specifies that a set of rows in the matrix will be treated as model cuts.
[Cuts, Data Input, Optimizer] p. 171

`problem.loadPresolveBasis` Loads a presolved basis from the user's areas.
[Data Input, LP, Optimizer] p. 217

`problem.loadPresolveDirs` Loads directives into the presolved matrix.
[Branching, Data Input, Optimizer] p. 175

`problem.loadQCQP` Used to load a quadratic problem with quadratic side constraints into the Optimizer data structure.
[Optimizer, Problem Input, Quadratic] p. 251

`problem.loadQP` Used to load a quadratic problem into the Optimizer data structure.
[Optimizer, Problem Input, Quadratic] p. 166

`problem.loadSecureVecs` Allows the user to mark rows and columns in order to prevent the presolve removing these rows and columns from the matrix.
[Data Input, Optimizer, Presolve] p. 218

`problem.lpOptimize` This function begins a search for the optimal continuous (LP) solution.
[LP, Optimizer, Solution Process] p. 192

`problem.mipOptimize` This function begins a tree search for the optimal MIP solution.
[MIP Entities, Optimizer, Solution Process] p. 193

`problem.nextIIS` Continues the search for further Irreducible Infeasible Sets (IIS), or calls `problem.firstIIS` if no IIS has been identified yet.
[Infeasibility, Optimizer] p. 209

`problem.objSA` Returns upper and lower sensitivity ranges for specified objective function coefficients.
[LP, Optimizer, Sensitivity Analysis] p. 246

`problem.optimize` This function begins a search for the optimal solution of the problem.
[Optimizer, Solution Process] p. 194

`problem.pivot` Performs a simplex pivot by bringing variable `enter` into the basis and removing `leave`.
[Linear Algebra, Optimizer, Simplex] p. 237

`problem.postSolve` Postsolve the current matrix when it is in a presolved state.
[Optimizer, Presolve, Solution Process] p. 225

`problem.postSolveSol` Postsolves a primal solution formulated in the presolved space into the corresponding solution formulated in the input space. The problem itself is unchanged.
[Optimizer, Presolve, Solution] p. 308

`problem.presolveRow` Presolves a row formulated in terms of the original variables such that it can be added to a presolved matrix.
[Misc, Optimizer, Presolve] p. 305

`problem.presolveSol` Presolves a primal solution formulated in the input space into the corresponding solution formulated in the presolved space. The problem itself is unchanged.
[Optimizer, Presolve, Solution] p. 307

`problem.printIIS` Prints a given Irreducible Infeasible Set (IIS) in the log.
[Infeasibility, Logging, Optimizer] p. 210

`problem.readBasis` Instructs the Optimizer to read in a previously saved basis from a file.
[File IO, LP, Optimizer] p. 196

`problem.readBinSol` Reads a solution from a binary solution file.
[File IO, Optimizer, Solution] p. 197

`problem.readDirs` Reads a directives file to help direct the tree search.
[Branching, File IO, Optimizer] p. 180

`problem.readProb` Reads an (X)MPS or LP format matrix from file.
[File IO, Optimizer, Problem Creation] p. 162

`problem.readSlxSol` Reads an ASCII solution file (`.slx`) created by the `problem.writeSlxSol` function.
[File IO, Optimizer, Solution] p. 195

`problem.repairInfeas` Provides a simplified interface for `problem.repairWeightedInfeas`.
[Infeasibility, Optimizer] p. 243

`problem.repairWeightedInfeas` By relaxing a set of selected constraints and bounds of an infeasible problem, it attempts to identify a 'solution' that violates the selected set of constraints and bounds minimally, while satisfying all other constraints and bounds. Among such solution candidates, it selects one that is optimal regarding the original objective function.
[Infeasibility, Optimizer] p. 239

`problem.repairWeightedInfeasBounds` An extended version of `problem.repairWeightedInfeas` that allows for bounding the level of relaxation allowed.
[Infeasibility, Optimizer] p. 241

`problem.restore` Restores the Optimizer's data structures from a file created by `problem.saveAs`. Optimization may then recommence from the point at which the file was created.
[File IO, Optimizer, Save Restore] p. 236

`problem.rhsSA` Returns upper and lower sensitivity ranges for specified right hand side (RHS) function coefficients.
[LP, Optimizer, Sensitivity Analysis] p. 248

`problem.saveAs` Saves the current data structures to file and terminates the run
[File IO, Optimizer, Save Restore] p. 235

`problem.scale` Re-scales the current matrix.
[Numerics, Optimizer] p. 179

`problem.setDefaultControl` Sets a single control to its default value.
[Controls and Attributes, Optimizer] p. 160

`problem.setDefaults` Sets all controls to their default values.
[Controls and Attributes, Optimizer] p. 161

`problem.setIndicators` Specifies that a set of rows in the matrix will be treated as indicator constraints during a tree search.
[Optimizer, Problem Creation] p. 182

`problem.setLogFile` This directs all Optimizer output to a log file.
[File IO, Logging, Optimizer] p. 159

`problem.setMessageStatus` Manages suppression of messages.
[Misc, Optimizer] p. 231

`problem.setProbName` Sets the current problem name.
[Optimizer, Problem Information] p. 158

`problem.sparseBTran` Post-multiplies a (row) vector provided by the user by the inverse of the current matrix. Sparse version of `problem.BTran`.
[Linear Algebra, Optimizer, Simplex] p. 272

`problem.sparseFTran` Pre-multiplies a (column) vector provided by the user by the inverse of the current matrix. Sparse version of `problem.FTran`.
[Linear Algebra, Optimizer, Simplex] p. 273

`problem.storeCuts` Stores cuts into the cut pool, but does not apply them to the current node.
[Cuts, Optimizer] p. 304

`problem.strongBranch` Performs strong branching iterations on all specified bound changes.
[Branching, Optimizer] p. 228

`problem.strongBranchCB` Performs strong branching iterations on all specified bound changes.
[Branching, Optimizer] p. 229

`problem.triggerRestart` Triggers a restart of the MIP search process.
[Optimizer, Presolve, Solution Process] p. 325

`problem.tune` This function begins a tuner session for the current problem.
[Optimizer, Tuner] p. 202

`problem.tuneProbSetFile` This function begins a tuner session for a set of problems.
[Optimizer, Tuner] p. 203

`problem.tunerReadMethod` This function loads a user defined tuner method from the given file.
[File IO, Optimizer, Tuner] p. 205

`problem.tunerWriteMethod` This function writes the current tuner method to a given file or prints it to the console.
[File IO, Optimizer, Tuner] p. 204

`problem.writeBasis` Writes the current basis to a file for later input into the Optimizer.
[File IO, LP, Optimizer] p. 287

`problem.writeBinSol` Writes the current MIP or LP solution to a binary solution file for later input into the Optimizer.
[File IO, Optimizer, Solution] p. 289

`problem.writeDirs` Writes the tree search directives from the current problem to a directives file.
[Branching, File IO, Optimizer] p. 181

`problem.writeIIS` Writes an LP/MPS/CSV file containing a given Irreducible Infeasible Set (IIS).
[File IO, Infeasibility, Optimizer] p. 213

`problem.writeProb` Writes the current problem to an MPS or LP file.
[File IO, Optimizer] p. 310

`problem.writePrtSol` Writes the current solution to a fixed format ASCII file, *problem_name* `.prt`.
[File IO, Optimizer, Solution] p. 290

`problem.writeSlxSol` Creates an ASCII solution file (.slx) using a similar format to MPS files.
[File IO, Optimizer, Solution] p. 291

`problem.writeSol` Writes the current solution to a CSV format ASCII file, *problem_name* `.asc` (and `.hdr`).
[File IO, Optimizer, Solution] p. 288

The following table contains the problem functions to be called for nonlinear problems.

`problem.msAddCustomPreset` A combined version of problem.msAddJob and problem.msAddPreset.
[Data Input, Multistart, Nonlinear] p. 346

`problem.msAddJob` Adds a multistart job to the multistart pool
[Data Input, Multistart, Nonlinear] p. 344

`problem.msAddPreset` Loads a preset of jobs into the multistart job pool.
[Data Input, Multistart, Nonlinear] p. 345

`problem.msClear` Removes all scheduled jobs from the multistart job pool
[Multistart, Nonlinear] p. 353

`problem.nlpAddFormulas` Add non-linear formulas to the SLP problem.
[Nonlinear, Problem Modification] p. 356

`problem.nlpCalcSlacks` Calculate the slack values for the provided solution in the non-linear problem
[Nonlinear, Solution] p. 391

`problem.nlpChgFormula` Add or replace a single matrix formula using a parsed or unparsed formula
[Nonlinear, Problem Modification] p. 358

`problem.nlpChgFormulaStr` Add or replace a single matrix formula using a character string for the formula.
[Nonlinear, Problem Modification] p. 357

`problem.nlpCurrentIV` Transfer the current solution to initial values
[Data Input, Nonlinear] p. 350

`problem.nlpDelFormulas` Delete nonlinear formulas from the current problem
[Nonlinear, Problem Modification] p. 362

`problem.nlpEvaluateFormula` Evaluate a formula using the current values of the variables
[Nonlinear, Solution] p. 354

`problem.nlpGetFormula` Retrieve a single matrix formula as a formula split into tokens.
[Nonlinear, Problem Information] p. 359

`problem.nlpGetFormulaRows` Retrieve the list of positions of the nonlinear formulas in the problem
[Nonlinear, Problem Information] p. 360

`problem.nlpGetFormulaStr` Retrieve a single matrix formula in a character string.
[Nonlinear, Problem Information] p. 363

`problem.nlpLoadFormulas` Load non-linear formulas into the SLP problem
[Nonlinear, Problem Information] p. 361

`problem.nlpOptimize` Maximize or minimize an SLP problem
[Nonlinear, Solution Process] p. 349

`problem.nlpPostsolveProb` Restores the problem to its pre-solve state
[Nonlinear, Presolve] p. 390

`problem.nlpPrintEvalInfo` Print a summary of any evaluation errors that may have occurred during solving a problem
[Logging, Nonlinear] p. 347

`problem.nlpSetInitVal` Set the initial value of columns
[Data Input, Nonlinear] p. 364

`problem.nlpValidate` Validate the feasibility of constraints in a converged solution
[Nonlinear, Solution] p. 348

`problem.nlpValidateKKT` Validates the first order optimality conditions also known as the Karush-Kuhn-Tucker (KKT) conditions versus the currect solution
[Nonlinear, Solution] p. 352

`problem.nlpValidateRow` Prints an extensive analysis on a given constraint of the SLP problem
[Nonlinear, Solution] p. 351

`problem.nlpValidateVector` Validate the feasibility of constraints for a given solution
[Nonlinear, Solution] p. 355

`problem.slpAddCoefs` Add non-linear coefficients to the SLP problem.
[Nonlinear, Problem Modification, SLP] p. 371

`problem.slpCascadeOrder` Establish a re-calculation sequence for SLP variables with determining rows.
[Data Input, Nonlinear, SLP] p. 377

`problem.slpCascadeSol` Re-calculate consistent values for SLP variables based on the current values of the remaining variables.
[Cascading, Nonlinear, SLP, Solution Process] p. 376

`problem.slpChgCascadeNLimit` Set a variable specific cascade iteration limit
[Data Input, Nonlinear, SLP] p. 381

`problem.slpChgCoef` Add or change a single matrix coefficient using a parsed or unparsed formula.
[Nonlinear, Problem Modification, SLP] p. 373

`problem.slpChgCoefStr` Add or change a single matrix coefficient using a character string for the formula.
[Nonlinear, Problem Modification, SLP] p. 372

`problem.slpChgDeltaType` Changes the type of the delta assigned to a nonlinear variable
[Nonlinear, Problem Modification, SLP] p. 380

`problem.slpChgRowStatus` Change the status setting of a constraint
[Bit-vector, Data Input, Nonlinear, SLP] p. 378

`problem.slpChgRowWt` Set or change the initial penalty error weight for a row
[Data Input, Nonlinear, SLP] p. 379

`problem.slpConstruct` Create the full augmented SLP matrix and data structures, ready for optimization
[Nonlinear, SLP, Solution Process] p. 382

`problem.slpDelCoefs` Delete coefficients from the current problem.
[Nonlinear, Problem Modification, SLP] p. 368

`problem.slpEvaluateCoef` Evaluate a coefficient using the current values of the variables
[Nonlinear, Solution] p. 385

`problem.slpFixPenalties` Fixe the values of the error vectors
[Nonlinear, Solution Process] p. 389

`problem.slpGetCoefFormula` Retrieve a single matrix coefficient as a formula split into tokens.
[Nonlinear, Problem Information, SLP] p. 365

`problem.slpGetCoefs` Retrieve the list of positions of the nonlinear coefficients in the problem.
[Nonlinear, Problem Information, SLP] p. 366

`problem.slpGetCoefStr` Retrieve a single matrix coefficient as a formula in a character string.
[Nonlinear, Problem Information, SLP] p. 369

`problem.slpGetColInfo` Get current column information.
[Nonlinear, SLP, Solution] p. 374

`problem.slpGetRowInfo` Get current row information.
[Nonlinear, SLP, Solution] p. 375

`problem.slpGetRowStatus` Retrieve the status setting of a constraint
[Bit-vector, Nonlinear, SLP, Solution] p. 383

`problem.slpGetRowWT` Get the initial penalty error weight for a row
[Data Information, Nonlinear, SLP] p. 384

`problem.slpLoadCoefs` Load non-linear coefficients into the SLP problem.
[Nonlinear, Problem Information, SLP] p. 367

`problem.slpReInitialize` Reset the SLP problem to match a just augmented system
[Nonlinear, SLP, Solution Process] p. 386

`problem.slpSetDetRow` Set the determining row of a variable
[Cascading, Data Input, Nonlinear, SLP] p. 370

`problem.slpSetInitStepBounds` Set the initial step bounds of columns
[Data Input, Nonlinear] p. 392

`problem.slpUnConstruct` Removes the augmentation and returns the problem to its pre-linearization state
[Nonlinear, SLP, Solution Process] p. 387

`problem.slpUpdateLinearization` Updates the current linearization
[Nonlinear, SLP, Solution Process] p. 388

The following table contains methods that can be called from a problem **before** optimization has started, to add or remove callbacks.

`problem.addAfterObjectiveCallback` Declares a callback which will be called after each objective in a multi-objective problem is solved.
[Callback, Multiobjective] p. 438

`problem.addBarIterationCallback` Declares a barrier iteration callback function, called after each iteration during the interior point algorithm, with the ability to access the current barrier solution/slack/duals or reduced cost values, and to ask barrier to stop. This callback function will be called in addition to any callbacks already added by problem.addBarIterationCallback. This functionality is not available for the hybrid gradient method.
[Barrier, Callback] p. 420

`problem.addBarlogCallback` Declares a barrier log callback function, called at each iteration during the interior point algorithm. This callback function will be called in addition to any barrier log callbacks already added by problem.addBarlogCallback. This functionality is not available for the hybrid gradient method.
[Barrier, Callback] p. 399

`problem.addBeforeObjectiveCallback` Declares a callback which will be called before each objective in a multi-objective problem is solved.
[Callback, Multiobjective] p. 436

`problem.addChangeBranchObjectCallback` Declares a callback function that will be called after the selection of a MIP entity to branch on.
[Branching, Callback] p. 426

`problem.addCheckTimeCallback` Declares a callback function which is called every time the Optimizer checks if the time limit has been reached. This callback function will be called in addition to any callbacks already added by problem.addCheckTimeCallback.
[Callback, Limits] p. 440

`problem.addComputeRestartCallback` Declares a callback to be called when a solve executed in compute mode needs to be restarted.
[Callback, Compute Interface] p. 434

`problem.addCutlogCallback` Declares a cut log callback function, called each time the cut log is printed.
[Callback, Logging] p. 397

`problem.addCutRoundCallback` Declares a callback function that is called when the Optimizer could separate cutting planes during the branch and bound search.
[Callback, Cuts] p. 442

`problem.addGapNotifyCallback` Declares a gap notification callback, to be called when a MIP solve reaches a predefined target, set using the `MIPRELGAPNOTIFY`, `MIPABSGAPNOTIFY`, `MIPABSGAPNOTIFYOBJ` and/or `MIPABSGAPNOTIFYBOUND` controls.
[Callback, Limits] p. 428

`problem.addInfnodeCallback` Declares a user infeasible node callback function, called after the current node has been found to be infeasible during the Branch and Bound search.
[Branch and Bound Search, Callback] p. 405

`problem.addIntsolCallback` Declares a user integer solution callback function, called every time an integer solution is found by heuristics or during the Branch and Bound search. This callback function will be called in addition to any callbacks already added by problem.addIntsolCallback.
[Callback, Solution] p. 409

`problem.addLplogCallback` Declares a simplex log callback function which is called after every `LPLOG` iterations of the simplex algorithm. This callback function will be called in addition to any callbacks already added by problem.addLplogCallback.
[Callback, LP] p. 393

`problem.addManagedCuts` Adds cuts to the Optimizer's internal cut pool from within the cutround callback set by problem.addCutRoundCallback.
[Callback, Cuts, Optimizer] p. 296

`problem.addMessageCallback` Declares an output callback function, called every time a text line relating to the given problem is output by the Optimizer.
[Callback, Logging] p. 414

`problem.addMiplogCallback` Declares a MIP log callback function, called each time the MIP log is printed. This callback function will be called in addition to any callbacks already added by problem.addMiplogCallback.
[Callback, Logging] p. 395

`problem.addMipThreadCallback` Declares a MIP thread callback function, called every time a MIP worker problem is created by the parallel MIP code. This callback function will be called in addition to any callbacks already added by problem.addMipThreadCallback.
[Callback, Parallel] p. 416

`problem.addMipThreadDestroyCallback` Declares a destroy MIP thread callback function, called every time a MIP thread is destroyed by the parallel MIP code. This callback function will be called in addition to any callbacks already added by problem.addMipThreadDestroyCallback.
[Callback, Parallel] p. 418

`problem.addMsJobEndCallback` Add a user callback to be called every time a new multistart job finishes. Can be used to overwrite the default solution ranking function
[Callback, Multistart] p. 466

`problem.addMsJobStartCallback` Add a user callback to be called every time a new multistart job is created, and the pre-loaded settings are applied
[Callback, Multistart] p. 464

`problem.addMsWinnerCallback` Add a user callback to be called every time a multistart winner has been declared
[Callback, Multistart] p. 468

`problem.addNewnodeCallback` Declares a callback function that will be called every time a new node is created during the branch and bound search. This callback function will be called in addition to any callbacks already added by problem.addNewnodeCallback.
[Branch and Bound Search, Callback] p. 424

`problem.addNlpCoefEvalErrorCallback` Add a user callback to be called when an evaluation of a coefficient fails during the solve
[Callback, Numerics] p. 470

`problem.addNodecutoffCallback` Declares a user node cutoff callback function, called every time a node is cut off as a result of an improved integer solution being found during the branch and bound search. This callback function will be called in addition to any callbacks already added by problem.addNodecutoffCallback.
[Branch and Bound Search, Callback] p. 407

`problem.addNodeLPSolvedCallback` Declares a callback function, called during the branch and bound search, after the LP relaxation has been solved for the current node, but before any internal cuts and heuristics have been applied. This callback function will be called in addition to any callbacks already added by problem.addNodeLPSolvedCallback.
[Branch and Bound Search, Callback] p. 432

`problem.addOptnodeCallback` Declares an optimal node callback function, called during the branch and bound search, after the LP relaxation has been solved for the current node, and after any internal cuts and heuristics have been applied, but before the Optimizer checks if the current node should be branched. This callback function will be called in addition to any callbacks already added by problem.addOptnodeCallback.
[Branch and Bound Search, Callback] p. 401

`problem.addPreIntsolCallback` Declares a user integer solution callback function, called when an integer solution is found by heuristics or during the branch and bound search, but before it is accepted by the Optimizer.
[Callback, Solution] p. 411

`problem.addPrenodeCallback` Declares a preprocess node callback function, called before the LP relaxation of a node has been optimized, so the solution at the node will not be available. This callback function will be called in addition to any callbacks already added by problem.addPrenodeCallback.
[Branch and Bound Search, Callback] p. 403

`problem.addPresolveCallback` Declares a callback to be called after presolve has been performed.
[Callback, Presolve] p. 422

`problem.addSlpCascadeEndCallback` Add a user callback to be called at the end of the cascading process, after the last variable has been cascaded
[Callback, Cascading, SLP] p. 444

`problem.addSlpCascadeStartCallback` Add a user callback to be called at the start of the cascading process, before any variables have been cascaded
[Callback, Cascading, SLP] p. 446

`problem.addSlpCascadeVarCallback` Add a user callback to be called after each column has been cascaded
[Callback, Cascading, SLP] p. 448

`problem.addSlpCascadeVarFailCallback` Add a user callback to be called after cascading a column was not successful
[Callback, Cascading, SLP] p. 450

`problem.addSlpConstructCallback` Add a user callback to be called during the Xpress-SLP augmentation process
[Callback, SLP] p. 452

`problem.addSlpDrColCallback` Add a user callback used to override the update of variables with small determining column
[Callback, Cascading, SLP] p. 462

`problem.addSlpIntSolCallback` Add a user callback to be called during MISLP when an integer solution is obtained
[Callback, MISLP] p. 454

`problem.addSlpIterEndCallback` Add a user callback to be called at the end of each SLP iteration
[Callback, SLP] p. 456

`problem.addSlpIterStartCallback` Add a user callback to be called at the start of each SLP iteration
[Callback, SLP] p. 458

`problem.addSlpIterVarCallback` Add a user callback to be called after each column has been tested for convergence
[Callback, SLP, SLP-convergence] p. 460

`problem.addSlpPreUpdateLinearizationCallback` Add a user callback to be called before the linearization is updated
[Callback, SLP] p. 472

`problem.addUserSolNotifyCallback` Declares a callback function to be called each time a solution added by `problem.addMipSol` has been processed. This callback function will be called in addition to any callbacks already added by `problem.addUserSolNotifyCallback`.
[Callback, Solution] p. 430

`problem.getCallbackPresolveDuals` Returns the dual values from the solution to the presolved problem associated with the current callback.
[Callback, Optimizer, Solution] p. 334

`problem.getCallbackPresolveRedCosts` Returns the reduced costs from the solution to the presolved problem associated with the current callback.
[Callback, Optimizer, Solution] p. 335

`problem.getCallbackPresolveSlacks` Returns the slack values from the solution to the presolved problem associated with the current callback.
[Callback, Optimizer, Solution] p. 333

`problem.getCallbackPresolveSolution` Returns the solution to the presolved problem associated with the current callback.
[Callback, Optimizer, Solution] p. 332

`problem.removeAfterObjectiveCallback` Removes a callback function previously added by problem.addAfterObjectiveCallback.
[Callback, Multiobjective] p. 439

`problem.removeBarIterationCallback` Removes a barrier iteration callback function previously added by `problem.addBarIterationCallback`. The specified callback function will no longer be called after it has been removed.
[Barrier, Callback] p. 421

`problem.removeBarlogCallback` Removes a Newton barrier log callback function previously added by `problem.addBarlogCallback`. The specified callback function will no longer be called after it has been removed.
[Barrier, Callback] p. 400

`problem.removeBeforeObjectiveCallback` Removes a callback function previously added by problem.addBeforeObjectiveCallback.
[Callback, Multiobjective] p. 437

`problem.removeChangeBranchObjectCallback` Removes a callback function previously added by problem.addChangeBranchObjectCallback.
[Branching, Callback] p. 427

`problem.removeCheckTimeCallback` Removes a callback function previously added by problem.addCheckTimeCallback.
[Callback, Limits] p. 441

`problem.removeComputeRestartCallback` Removes a computerestart callback function previously added by `problem.addComputeRestartCallback`. The specified callback function will no longer be called after it has been removed.
[Callback, Compute Interface] p. 435

`problem.removeCutlogCallback` Removes a cut log callback function previously added by problem.addCutlogCallback.
[Callback, Logging] p. 398

`problem.removeCutRoundCallback` Removes a cut round callback function previously added by problem.addCutRoundCallback.
[Callback, Cuts] p. 443

`problem.removeGapNotifyCallback` Removes a callback function previously added by problem.addGapNotifyCallback.
[Callback, Limits] p. 429

`problem.removeInfnodeCallback` Removes a user infeasible node callback function previously added by `problem.addInfnodeCallback`. The specified callback function will no longer be called after it has been removed.
[Branch and Bound Search, Callback] p. 406

`problem.removeIntsolCallback` Removes an integer solution callback function previously added by `problem.addIntsolCallback`. The specified callback function will no longer be called after it has been removed.
[Callback, Solution] p. 410

`problem.removeLplogCallback` Removes a simplex log callback function previously added by `problem.addLplogCallback`. The specified callback function will no longer be called after it has been removed.
[Callback, Logging] p. 394

`problem.removeMessageCallback` Removes a message callback function previously added by problem.addMessageCallback.
[Callback, Logging] p. 415

`problem.removeMiplogCallback` Removes a MIP log callback function previously added by `problem.addMiplogCallback`. The specified callback function will no longer be called after it has been removed.
[Callback, Logging] p. 396

`problem.removeMipThreadCallback` Removes a callback function previously added by problem.addMipThreadCallback.
[Callback, Parallel] p. 417

`problem.removeMipThreadDestroyCallback` Removes a child thread destruction callback function previously added by `problem.addMipThreadDestroyCallback`. The specified callback function will no longer be called after it has been removed.
[Callback, Parallel] p. 419

`problem.removeMsJobEndCallback` Removes a callback function previously added by problem.addMsJobEndCallback.
[Callback, Multistart] p. 467

`problem.removeMsJobStartCallback` Removes a callback function previously added by problem.addMsJobStartCallback.
[Callback, Multistart] p. 465

`problem.removeMsWinnerCallback` Removes a callback function previously added by problem.addMsWinnerCallback.
[Callback, Multistart] p. 469

`problem.removeNewnodeCallback` Removes a new-node callback function previously added by problem.addNewnodeCallback.
[Branch and Bound Search, Callback] p. 425

`problem.removeNlpCoefEvalErrorCallback` Removes a callback function previously added by `problem.addNlpCoefEvalErrorCallback`. The specified callback function will no longer be called after it has been removed.
[Callback, Numerics] p. 471

`problem.removeNodecutoffCallback` Removes a node-cutoff callback function previously added by `problem.addNodecutoffCallback`. The specified callback function will no longer be called after it has been removed.
[Branch and Bound Search, Callback] p. 408

`problem.removeNodeLPSolvedCallback` Removes a node callback function previously added by problem.addNodeLPSolvedCallback.
[Branch and Bound Search, Callback] p. 433

`problem.removeOptnodeCallback` Removes a node-optimal callback function previously added by `problem.addOptnodeCallback`. The specified callback function will no longer be called after it has been removed.
[Branch and Bound Search, Callback] p. 402

`problem.removePreIntsolCallback` Removes a pre-integer solution callback function previously added by `problem.addPreIntsolCallback`. The specified callback function will no longer be called after it has been removed.
[Callback, Solution] p. 413

`problem.removePrenodeCallback` Removes a preprocess node callback function previously added by `problem.addPrenodeCallback`. The specified callback function will no longer be called after it has been removed.
[Branch and Bound Search, Callback] p. 404

`problem.removePresolveCallback` Removes a presolve callback function previously added by problem.addPresolveCallback.
[Callback, Presolve] p. 423

`problem.removeSlpCascadeEndCallback` Removes a callback function previously added by `problem.addSlpCascadeEndCallback`. The specified callback function will no longer be called after it has been removed.
[Callback, Cascading, SLP] p. 445

`problem.removeSlpCascadeStartCallback` Removes a callback function previously added by `problem.addSlpCascadeStartCallback`. The specified callback function will no longer be called after it has been removed.
[Callback, Cascading, SLP] p. 447

`problem.removeSlpCascadeVarCallback` Removes a callback function previously added by `problem.addSlpCascadeVarCallback`. The specified callback function will no longer be called after it has been removed.
[Callback, Cascading, SLP] p. 449

`problem.removeSlpCascadeVarFailCallback` Removes a callback function previously added by `problem.addSlpCascadeVarFailCallback`. The specified callback function will no longer be called after it has been removed.
[Callback, Cascading, SLP] p. 451

`problem.removeSlpConstructCallback` Removes a callback function previously added by `problem.addSlpConstructCallback`. The specified callback function will no longer be called after it has been removed.
[Callback, SLP] p. 453

`problem.removeSlpDrColCallback` Removes a callback function previously added by problem.addSlpDrColCallback.
[Callback, Cascading, SLP] p. 463

`problem.removeSlpIntSolCallback` Removes a callback function previously added by problem.addSlpIntSolCallback.
[Callback, MISLP] p. 455

`problem.removeSlpIterEndCallback` Removes a callback function previously added by problem.addSlpIterEndCallback.
[Callback, SLP] p. 457

`problem.removeSlpIterStartCallback` Removes a callback function previously added by `problem.addSlpIterStartCallback`. The specified callback function will no longer be called after it has been removed.
[Callback, SLP] p. 459

`problem.removeSlpIterVarCallback` Removes a callback function previously added by problem.addSlpIterVarCallback.
[Callback, SLP, SLP-convergence] p. 461

`problem.removeSlpPreUpdateLinearizationCallback` Removes a callback function previously added by `problem.addSlpPreUpdateLinearizationCallback`. The specified callback function will no longer be called after it has been removed.
[Callback, SLP] p. 473

`problem.removeUserSolNotifyCallback` Removes a user solution notification callback previously added by `problem.addUserSolNotifyCallback`. The specified callback function will no longer be called after it has been removed.
[Callback, Solution] p. 431

problem.setProbName

Purpose

Sets the current problem name.

Topic areas

Problem Information, Optimizer

Synopsis

`problem.setProbName(probname)`

Argument

`probname: str` A string of up to `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual) characters containing the problem name.

Further information

This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.

Related topics

`problem.readProb`, `problem.getProbName`, `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual).

problem.setLogFile

Purpose

This directs all Optimizer output to a log file.

Topic areas

File IO, Logging, Optimizer

Synopsis

```
problem.setLogFile(filename)
```

Argument

`filename: str` A string of up to `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual) characters containing the file name to which all logging output should be written. If set to `None`, redirection of the output will stop.

Further information

1. It is recommended that a log file be set up for each problem being worked on, since it provides a means for obtaining any errors or warnings output by the Optimizer during the solution process.
2. Alternatively, an output callback can be defined using `problem.addMessageCallback`, which will be called every time a line of text is output. To discard all output messages the `OUTPUTLOG` (p.**??** Xpress Optimizer Reference Manual) integer control can be set to `0`.

Related topics

`problem.addMessageCallback`.

problem.setDefaultControl

Purpose

Sets a single control to its default value.

Topic areas

Controls and Attributes, Optimizer

Synopsis

```
problem.setDefaultControl(control)
```

Argument

`control: str | int` Integer, double or string control parameter whose default value is to be set.

Further information

1. A full list of all controls may be found in the Optimizer Reference Manual, or from the list in the `xprs.h` header file.
2. This function also reset any objective-specific value previously set for a solver control . But it cannot be used to reset objective priorities, weights or tolerances.

Related topics

`problem.setDefaults`

problem.setDefaults

Purpose

Sets all controls to their default values.

Topic areas

Controls and Attributes, Optimizer

Synopsis

```
problem.setDefaults()
```

Further information

1. This function also resets all objective-specific solver controls . But it does not reset objective priorities, weights or tolerances.
2. The value of the `COMPUTE` (p.**??** Xpress Optimizer Reference Manual) control is not reset by this function. This can be reset by calling `problem.setDefaultControl`.
3. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.

Related topics

`problem.setDefaultControl`

problem.readProb

Purpose

Reads an (X)MPS or LP format matrix from file.

Topic areas

File IO, Problem Creation, Optimizer

Synopsis

```
problem.readProb(filename, flags="")
```

Arguments

`filename: str` The path and file name from which the problem is to be read. Limited to `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual) characters. If omitted, the default *problem_name* is used with various extensions - see below.

`flags: str` Flags to be passed:

- `l` only `filename.lp` is searched for;
- `v` use the provided filename verbatim, without appending the `.mps`, `.mat` or `.lp` extension;
- `z` read a compressed input file.

Related controls

Integer

`EXTRACOLS`	Number of extra columns to be allowed for.
`EXTRAELEMS`	Number of extra matrix elements to be allowed for.
`EXTRAMIPENTS`	Number of extra MIP entities to be allowed for.
`EXTRAROWS`	Number of extra rows to be allowed for.
`KEEPNROWS`	Status for nonbinding rows.
`MPSECHO`	Whether MPS comments are to be echoed.
`MPSFORMAT`	Specifies format of MPS files.
`SCALING`	Type of scaling.

Double

`MATRIXTOL`	Tolerance on matrix elements.
`SOSREFTOL`	Minimum gap between reference row entries.

String

`MPSBOUNDNAME`	The active bound name.
`MPSOBJNAME`	Name of objective function row.
`MPSRANGENAME`	Name of range.
`MPSRHSNAME`	Name of right hand side.

Further information

1. If no flags are given, file types are searched for in the order: `.mps`, `.mat`, `.lp`. Matrix files are assumed to be in XMPS or MPS format unless their file extension is `.lp` in which case they must be LP files. Files with compressed extensions are also searched for, e.g., `.mps.gz`.
2. The Optimizer can read matrix files that have been compressed using one of the following formats, as long as the command-line tool necessary to decompress the file can be located in the path: `bzip2`, `xz`, `lzma`, `Z`, `zip`, `tar`, `tgz`. The Optimizer has built-in support for the `gz` format: no external tools are necessary to read gzipped matrix files.
3. If `filename` has been specified, the problem name is changed to `filename`, ignoring any extension.
4. `problem.readProb` will take as the objective function the first `N` type row in the matrix, unless the string parameter `MPSOBJNAME` (p.**??** Xpress Optimizer Reference Manual) has been set, in which case the objective row sought will be the one named by `MPSOBJNAME`. Similarly, if non-blank, the string parameters `MPSRHSNAME` (p.**??** Xpress Optimizer Reference Manual), `MPSBOUNDNAME` (p.**??** Xpress Optimizer Reference Manual) and `MPSRANGENAME` (p.**??** Xpress Optimizer Reference Manual) specify the right hand side, bound and range sets to be taken. For example:
   ```
   MPSOBJNAME="Cost"
   MPSRHSNAME="RHS 1"
   READPROB
   ```
   The treatment of `N` type rows other than the objective function depends on the `KEEPNROWS` (p.**??** Xpress Optimizer Reference Manual) control. If `KEEPNROWS` is `1` the rows and their elements are kept in memory; if it is `0` the rows are retained, but their elements are removed; and if it is `−1` the rows are deleted entirely. The performance impact of retaining such `N` type rows will be small unless the presolve has been disabled by setting `PRESOLVE` (p.**??** Xpress Optimizer Reference Manual) to `0` prior to optimization.
5. The Optimizer checks that the matrix file is in a legal format and displays error messages if it detects errors. When the Optimizer has read and verified the problem, it will display summary problem statistics.
6. By default, the `MPSFORMAT` (p.**??** Xpress Optimizer Reference Manual) control is set to `−1` and `problem.readProb` determines automatically whether the MPS files are in free or fixed format. If `MPSFORMAT` is set to `0`, fixed format is assumed and if it is set to `1`, free format is assumed. Fields in free format MPS files are delimited by one or more blank characters. The keywords NAME, ROWS, COLUMNS, QUADOBJ / QMATRIX, QCMATRIX, DELAYEDROWS, MODELCUTS, SETS, RHS, RANGES, BOUNDS and ENDATAmust start in column one and no vector name may contain blanks. If a special ordered set is specified with a reference row, its name may not be the same as that of a column. Note that numeric values which contain embedded spaces (for example after unary minus sign) will not be read correctly unless `MPSFORMAT` is set to `0`.
7. If the problem is not to be scaled automatically, set the parameter `SCALING` (p.**??** Xpress Optimizer Reference Manual) to `0` before issuing the `problem.readProb` command.

Related topics

`problem.loadMIP`, `problem.loadLP`, `problem.loadMIQP`, `problem.loadQP`, `problem.writeProb`.

problem.loadLP

Purpose

Enables the user to pass a matrix directly to the Optimizer, rather than reading the matrix from a file.

Topic areas

Problem Input, LP, Optimizer

Synopsis

```
problem.loadLP(probname="", rowtype=None, rhs=None, rng=None, objcoef=None,
       start=None, collen=None, rowind=None, rowcoef=None, lb=None, ub=None)
```

Arguments

`probname: str` A string of up to `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual) characters containing a name for the problem. If `probname` is `None`, the problem name will be an empty string.

`rowtype: list[str]` Character array containing the row types:

`L`	indicates a $\leqslant$ constraint;
`E`	indicates an = constraint;
`G`	indicates a $\geqslant$ constraint;
`R`	indicates a range constraint;
`N`	indicates a nonbinding constraint.

May be `None` if the problem contains no rows.

`rhs: list[float]` Double array containing the right hand side coefficients of the rows. The right hand side value for a range row gives the upper bound on the row. May be `None` if the problem contains no rows.

`rng: list[float]` Double array containing the range values for range rows. Values for all other rows will be ignored. May be `None` if not required. The lower bound on a range row is the right hand side value minus the range value. The sign of the range value is ignored - the absolute value is used in all cases.

`objcoef: list[float]` Double array containing the objective function coefficients. This can be `None` to set all objective coefficients to 0 (zero).

`start: list[int]` Integer array containing the offsets in the `rowind` and `rowcoef` arrays of the start of the elements for each column. This array contains one entry for each column, plus one more if `collen` is `None`. If `collen` is `None`, the last entry of `start` contains the position in the `rowind` and `rowcoef` arrays at which an extra column would start, if it were present. This value is also the length of the `rowind` and `rowcoef` arrays. May be `None` if the problem contains no coefficients.

`collen: list[int]` Integer array containing the number of nonzero elements in each column. May be `None` if not required. This array is not required if the non-zero coefficients in the `rowind` and `rowcoef` arrays are contiguous, and the `start` array has an additional entry as described above.

`rowind: list[constraint]` Integer array containing the row indices for the nonzero elements in each column. May be `None` if the problem contains no coefficients. If the indices are input contiguously, with the columns in ascending order, the length of the `rowind` array is found by adding the `start` entry for the last column to the `collen` entry for the last column, or if `collen` is `None`, by the additional entry at the end of `start`.

`rowcoef: list[float]` Double array containing the nonzero element values; length as for `rowind`. May be `None` if the problem contains no coefficients.

`lb: list[float]` Double array containing the lower bounds on the columns. Use `-xpress.infinity` to represent a lower bound of minus infinity. If this is `None` then all lower bounds are 0 (zero).

`ub: list[float]` Double array containing the upper bounds on the columns. Use `xpress.infinity` to represent an upper bound of plus infinity. If this is `None` then all upper bounds are infinite.

Related controls

Integer

EXTRACOLS	Number of extra columns to be allowed for.
EXTRAELEMS	Number of extra matrix elements to be allowed for.
EXTRAROWS	Number of extra rows to be allowed for.
KEEPNROWS	Status for nonbinding rows.

SCALING	Type of scaling.

Double

MATRIXTOL	Tolerance on matrix elements.

Further information

1. The double constants `xpress.infinity` and `-xpress.infinity` are defined in the Optimizer library header file.
2. For a range constraint, the value in the `rhs` array specifies the upper bound on the constraint, while the value in the `rng` array specifies the range on the constraint. So a range constraint *j* is interpreted as:

$$\mathrm{rhs}_j - |\mathrm{rng}_j| \leqslant \sum_i a_{ij} x_i \leqslant \mathrm{rhs}_j$$

3. Passing zero for all integer arguments and `None` for all array arguments will load an empty problem.
4. Argument `rowind` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.loadMIP`, `problem.loadMIQP`, `problem.loadQP`, `problem.readProb`.

problem.loadQP

Purpose

Used to load a quadratic problem into the Optimizer data structure. Such a problem may have quadratic terms in its objective function, although not in its constraints.

Topic areas

Problem Input, Quadratic, Optimizer

Synopsis

```
problem.loadQP(probname="", rowtype=None, rhs=None, rng=None, objcoef=None,
       start=None, collen=None, rowind=None, rowcoef=None, lb=None, ub=None,
       objqcol1=None, objqcol2=None, objqcoef=None)
```

Arguments

`probname: str` A string of up to `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual) characters containing a name for the problem. If `probname` is `None`, the problem name will be an empty string.

`rowtype: list[str]` Character array containing the row types:

- `L` indicates a $\leqslant$ constraint;
- `E` indicates an = constraint;
- `G` indicates a $\geqslant$ constraint;
- `R` indicates a range constraint;
- `N` indicates a nonbinding constraint.

May be `None` if the problem contains no rows.

`rhs: list[float]` Double array containing the right hand side coefficients of the rows. The right hand side value for a range row gives the upper bound on the row. May be `None` if the problem contains no rows.

`rng: list[float]` Double array containing the range values for range rows. Values for all other rows will be ignored. May be `None` if there are no ranged constraints. The lower bound on a range row is the right hand side value minus the range value. The sign of the range value is ignored - the absolute value is used in all cases.

`objcoef: list[float]` Double array containing the objective function coefficients. This can be `None` to set all objective coefficients to 0 (zero).

`start: list[int]` Integer array containing the offsets in the `rowind` and `rowcoef` arrays of the start of the elements for each column. This array contains one entry for each column, plus one more if `collen` is `None`. If `collen` is `None`, the last entry of `start` contains the position in the `rowind` and `rowcoef` arrays at which an extra column would start, if it were present. This value is also the length of the `rowind` and `rowcoef` arrays. May be `None` if the problem contains no coefficients.

`collen: list[int]` Integer array containing the number of nonzero elements in each column. May be `None` if not required. This array is not required if the non-zero coefficients in the `rowind` and `rowcoef` arrays are contiguous, and the `start` array has an additional entry as described above.

`rowind: list[constraint]` Integer array containing the row indices for the nonzero elements in each column. May be `None` if the problem contains no coefficients. If the indices are input contiguously, with the columns in ascending order, the length of the `rowind` array is found by adding the `start` entry for the last column to the `collen` entry for the last column, or if `collen` is `None`, by the additional entry at the end of `start`.

`rowcoef: list[float]` Double array containing the nonzero element values; length as for `rowind`. May be `None` if the problem contains no coefficients.

`lb: list[float]` Double array containing the lower bounds on the columns. Use `-xpress.infinity` to represent a lower bound of minus infinity. If this is `None` then all lower bounds are 0 (zero).

`ub: list[float]` Double array containing the upper bounds on the columns. Use `xpress.infinity` to represent an upper bound of plus infinity. If this is `None` then all upper bounds are infinite.

`objqcol1: list[var]` Integer array containing the column index of the first variable in each quadratic term. May be `None` if all objective coefficients are zero.

`objqcol2: list[var]` Integer array containing the column index of the second variable in each quadratic term. May be `None` if all objective coefficients are zero.

`objqcoef: list[float]` Double array containing the quadratic coefficients. May be `None` if all objective

coefficients are zero.

Related controls

Integer

EXTRACOLS	Number of extra columns to be allowed for.
EXTRAELEMS	Number of extra matrix elements to be allowed for.
EXTRAROWS	Number of extra rows to be allowed for.
KEEPNROWS	Status for nonbinding rows.
SCALING	Type of scaling.

Double

MATRIXTOL	Tolerance on matrix elements.

Further information

1. The objective function is of the form `c'x+ 0.5 x'Qx`. Note that only the upper or lower triangular part of the Q matrix is specified.
2. The double constants `xpress.infinity` and `-xpress.infinity` are defined in the Optimizer library header file.
3. Passing zero for all integer arguments and `None` for all array arguments will load an empty problem.
4. Argument `rowind` may contain either `xpress.constraint` objects or integer indices.
5. Arguments `objqcol1, objqcol2` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.loadMIP`, `problem.loadLP`, `problem.loadMIQP`, `problem.readProb`.

problem.loadMIQP

Purpose

Used to load a MIQP problem, hence a MIP with quadratic objective coefficients, into the Optimizer data structures. Integer, binary, partial integer, semi-continuous and semi-continuous integer variables can be defined, together with sets of type 1 and 2. The reference row values for the set members are passed as an array rather than specifying a reference row.

Topic areas

Problem Input, Quadratic, Optimizer

Synopsis

```
problem.loadMIQP(probname="", rowtype=None, rhs=None, rng=None, objcoef=None,
       start=None, collen=None, rowind=None, rowcoef=None, lb=None, ub=None,
       objqcol1=None, objqcol2=None, objqcoef=None, coltype=None, entind=None,
       limit=None, settype=None, setstart=None, setind=None, refval=None)
```

Arguments

`probname: str` A string of up to `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual) characters containing a name for the problem. If `probname` is `None`, the problem name will be an empty string.

`rowtype: list[str]` Character array containing the row type:

- `L` indicates a $\leqslant$ constraint;
- `E` indicates an = constraint;
- `G` indicates a $\geqslant$ constraint;
- `R` indicates a range constraint;
- `N` indicates a nonbinding constraint.

May be `None` if the problem contains no rows.

`rhs: list[float]` Double array containing the right hand side coefficients of the rows. The right hand side value for a range row gives the upper bound on the row. May be `None` if the problem contains no rows.

`rng: list[float]` Double array containing the range values for range rows. The values in the range array will only be read for `R` type rows. The entries for other type rows will be ignored. May be `None` if not required. The lower bound on a range row is the right hand side value minus the range value. The sign of the range value is ignored - the absolute value is used in all cases.

`objcoef: list[float]` Double array containing the objective function coefficients. This can be `None` to set all objective coefficients to 0 (zero).

`start: list[int]` Integer array containing the offsets in the `rowind` and `rowcoef` arrays of the start of the elements for each column. This array contains one entry for each column, plus one more if `collen` is `None`. If `collen` is `None`, the last entry of `start` contains the position in the `rowind` and `rowcoef` arrays at which an extra column would start, if it were present. This value is also the length of the `rowind` and `rowcoef` arrays. May be `None` if the problem contains no coefficients.

`collen: list[int]` Integer array containing the number of nonzero elements in each column. May be `None` if not required. This array is not required if the non-zero coefficients in the `rowind` and `rowcoef` arrays are contiguous, and the `start` array has an additional entry as described above.

`rowind: list[constraint]` Integer array containing the row indices for the nonzero elements in each column. May be `None` if the problem contains no coefficients. If the indices are input contiguously, with the columns in ascending order, the length of the `rowind` array is found by adding the `start` entry for the last column to the `collen` entry for the last column, or if `collen` is `None`, by the additional entry at the end of `start`.

`rowcoef: list[float]` Double array containing the nonzero element values; length as for `rowind`. May be `None` if the problem contains no coefficients.

`lb: list[float]` Double array containing the lower bounds on the columns. Use `-xpress.infinity` to represent a lower bound of minus infinity. If this is `None` then all lower bounds are 0 (zero).

`ub: list[float]` Double array containing the upper bounds on the columns. Use `xpress.infinity` to represent an upper bound of plus infinity. If this is `None` then all upper bounds are infinite.

`objqcol1: list[var]` Integer array containing the column index of the first variable in each quadratic term. May be `None` if all objective coefficients are zero.

`objqcol2: list[var]` Integer array containing the column index of the second variable in each quadratic term. May be `None` if all objective coefficients are zero.

`objqcoef: list[float]` Double array containing the quadratic coefficients. May be `None` if all objective coefficients are zero.

`coltype: list[str]` Character array containing the entity types:

`B`	binary variables;
`I`	integer variables;
`P`	partial integer variables;
`S`	semi-continuous variables;
`R`	semi-continuous integers.

May be `None` if all variables are continuous.

`entind: list[var]` Integer array containing the column indices of the MIP entities. May be `None` if the problem contains no entities.

`limit: list[float]` Double array containing the integer limits for the partial integer variables and lower bounds for semi-continuous and semi-continuous integer variables (any entries in the positions corresponding to binary and integer variables will be ignored). May be `None` if not required.

`settype: list[str]` Character array containing:

`1`	SOS1 type sets;
`2`	SOS2 type sets.

May be `None` if not required.

`setstart: list[int]` Integer array containing the offsets in the `setind` and `refval` arrays indicating the start of each set. This array has one entry for each set plus one extra entry containing the offset where an additional set would start, if it were present. May be `None` if not required.

`setind: list[var]` Integer array containing the columns in each set. May be `None` if not required.

`refval: list[float]` Double array containing the reference row entries for each member of the sets. These define the order for SOS2 constraints and may be used in branching for both types. May be `None` if not required.

Related controls

Integer

`EXTRACOLS`	Number of extra columns to be allowed for.
`EXTRAELEMS`	Number of extra matrix elements to be allowed for.
`EXTRAMIPENTS`	Number of extra MIP entities to be allowed for.
`EXTRAROWS`	Number of extra rows to be allowed for.
`KEEPNROWS`	Status for nonbinding rows.
`SCALING`	Type of scaling.

Double

`MATRIXTOL`	Tolerance on matrix elements.
`SOSREFTOL`	Minimum gap between reference row entries.

Further information

1. The objective function is of the form `c'x+ 0.5 x'Qx`. Note that only the upper or lower triangular part of the `Q` matrix is specified.
2. The double constants `xpress.infinity` and `-xpress.infinity` are defined in the Optimizer library header file.
3. Passing zero for all integer arguments and `None` for all array arguments will load an empty problem.
4. Argument `rowind` may contain either `xpress.constraint` objects or integer indices.
5. Arguments `objqcol1, objqcol2, entind, setind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.loadMIP`, `problem.loadLP`, `problem.loadQP`, `problem.readProb`.

problem.fixMIPEntities

Purpose

Fixes all the MIP entities to the values of the last found MIP solution. This is useful for finding the reduced costs for the continuous variables after the integer variables have been fixed to their optimal values.

Topic areas

Problem Modification, MIP Entities, Optimizer

Synopsis

```
problem.fixMIPEntities(options)
```

Argument

`options: int` Options how to fix the MIP entities.

Bit	Meaning
`0`	If all MIP entities should be rounded to the nearest discrete value in the solution before being fixed.
`1`	If piecewise linear and general constraints as well as partial integers should be kept in the problem with only the non-convex decisions (i.e. which part of a non-convex piecewise linear function, which variable attains a maximum or the integer part of the partial integer) fixed. Otherwise all variables appearing in piecewise linear or general constraints and all partial integers will be fixed.

Further information

1. Because of tolerances, it is possible for e.g. a binary variable to be slightly fractional in the MIP solution, where it might have the value `0.999999` instead of being at exactly `1.0`. With `ifround = 0`, such a binary will be fixed at `0.999999`, but with `ifround = 1`, it will be fixed at `1.0`.
2. This command is useful for inspecting the reduced costs of the continuous variables in a matrix after the MIP entities have been fixed. Sensitivity analysis can also be performed on the continuous variables in a MIP problem using `problem.rhsSA`, `problem.objSA` or `problem.bndSA` after calling `problem.fixMIPEntities`.
3. For nonlinear problems, one can set the initial point to the solution returned by the MIP search (via `problem.nlpCurrentIV`) and then call `problem.fixMIPEntities`and reoptimize the problem using a local solver (`problem.nlpOptimize`). to obtain an approximation of the dual values

Related topics

`problem.mipOptimize`.

problem.loadModelCuts

Purpose

Specifies that a set of rows in the matrix will be treated as model cuts.

Topic areas

Data Input, Cuts, Optimizer

Synopsis

```
problem.loadModelCuts(rowind)
```

Argument

`rowind: list[constraint]` An array of row indices to be treated as cuts.

Error value

`268` Cannot perform operation on presolved matrix.

Further information

1. During presolve the model cuts are removed from the matrix and added to an internal cut pool. During the tree search, the Optimizer will regularly check this cut pool for any violated model cuts and add those that cuts off a node LP solution.
2. The model cuts must be ”true” model cuts, in the sense that they are redundant at the optimal MIP solution. The Optimizer does not guarantee to add all violated model cuts, so they must not be required to define the optimal MIP solution.
3. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
4. Argument `rowind` may contain either `xpress.constraint` objects or integer indices.

Related topics

the Optimizer Reference Manual.

problem.loadDelayedRows

Purpose

Specifies that a set of rows in the matrix will be treated as delayed rows during a tree search. These are rows that must be satisfied for any integer solution, but will not be loaded into the active set of constraints until required.

Topic areas

Problem Input, Cuts, Optimizer

Synopsis

```
problem.loadDelayedRows(rowind)
```

Argument

`rowind: list[constraint]` An array of row indices to treat as delayed rows.

Further information

1. Delayed rows must be set up before solving the problem. Any delayed rows will be removed from the matrix after presolve and added to a special pool. A delayed row will be added back into the active matrix only when such a row is violated by an integer solution found by the Optimizer.
2. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
3. Argument `rowind` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.loadModelCuts`.

problem.loadDirs

Purpose

Loads directives into the matrix.

Topic areas

Data Input, Optimizer

Synopsis

```
problem.loadDirs(colind, priority=None, dir=None, uppseudo=None, downpseudo=None)
```

Arguments

`colind: list[var]` Integer array containing the column numbers. A negative value indicates a set number (the first set being `-1`, the second `-2`, and so on).

`priority: list[int]` Integer array containing the priorities for the columns or sets. Priorities must be between 0 and 1000, where columns/sets with smallest priority will be branched on first. May be `None` if not required.

`dir: list[str]` Character array specifying the branching direction for each column or set:

- `U` the entity is to be forced up;
- `D` the entity is to be forced down;
- `N` not specified.

May be `None` if not required.

`uppseudo: list[float]` Double array containing the up pseudo costs for the columns or sets. May be `None` if not required.

`downpseudo: list[float]` Double array containing the down pseudo costs for the columns or sets. May be `None` if not required.

Further information

1. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
2. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.getDirs`, `problem.loadPresolveDirs`, `problem.readDirs`.

problem.loadBranchDirs

Purpose

Loads directives into the current problem to specify which MIP entities the Optimizer should continue to branch on when a node solution is integer feasible.

Topic areas

Data Input, Branching, Optimizer

Synopsis

```
problem.loadBranchDirs(colind, dir=None)
```

Arguments

`colind: list[var]` Integer array containing the column numbers. A negative value indicates a set number (the first set being -1, the second -2, and so on).

`dir: list[int]` Integer array containing either 0 or 1 for the entities given in `colind`. Entities for which dir is set to 1 will be branched on until fixed before an integer feasible solution is returned. If `dir` is `None`, the branching directive will be set for all entities in `colind`.

Further information

1. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
2. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.loadDirs`, `problem.readDirs`, the Optimizer Reference Manual.

problem.loadPresolveDirs

Purpose

Loads directives into the presolved matrix.

Topic areas

Data Input, Branching, Optimizer

Synopsis

```
problem.loadPresolveDirs(colind, priority=None, dir=None, uppseudo=None,
       downpseudo=None)
```

Arguments

`colind: list[var]` Integer array containing the column numbers. A negative value indicates a set number (`-1` being the first set, `-2` the second, and so on).

`priority: list[int]` Integer array containing the priorities for the columns or sets. May be `None` if not required.

`dir: list[str]` Character array specifying the branching direction for each column or set:

- `U` the entity is to be forced up;
- `D` the entity is to be forced down;
- `N` not specified.

May be `None` if not required.

`uppseudo: list[float]` Double array containing the up pseudo costs for the columns or sets. May be `None` if not required.

`downpseudo: list[float]` Double array containing the down pseudo costs for the columns or sets. May be `None` if not required.

Further information

Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.getDirs`, `problem.loadDirs`.

problem.loadMIP

Purpose

Used to load a MIP problem into the Optimizer data structures. Integer, binary, partial integer, semi-continuous and semi-continuous integer variables can be defined, together with sets of type 1 and 2. The reference row values for the set members are passed as an array rather than specifying a reference row.

Topic areas

Problem Input, MIP Entities, Optimizer

Synopsis

```
problem.loadMIP(probname="", rowtype=None, rhs=None, rng=None, objcoef=None,
       start=None, collen=None, rowind=None, rowcoef=None, lb=None, ub=None,
       coltype=None, entind=None, limit=None, settype=None, setstart=None,
       setind=None, refval=None)
```

Arguments

`probname: str` A string of up to `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual) characters containing a name for the problem. If `probname` is `None`, the problem name will be an empty string.

`rowtype: list[str]` Character array containing the row types:

- `L` indicates a $\leqslant$ constraint;
- `E` indicates an = constraint;
- `G` indicates a $\geqslant$ constraint;
- `R` indicates a range constraint;
- `N` indicates a nonbinding constraint.

May be `None` if the problem contains no rows.

`rhs: list[float]` Double array containing the right hand side coefficients of the rows. The right hand side value for a range row gives the upper bound on the row. May be `None` if the problem contains no rows.

`rng: list[float]` Double array containing the range values for range rows. Values for all other rows will be ignored. May be `None` if not required. The lower bound on a range row is the right hand side value minus the range value. The sign of the range value is ignored - the absolute value is used in all cases.

`objcoef: list[float]` Double array containing the objective function coefficients. This can be `None` to set all objective coefficients to 0 (zero).

`start: list[int]` Integer array containing the offsets in the `rowind` and `rowcoef` arrays of the start of the elements for each column. This array contains one entry for each column, plus one more if `collen` is `None`. If `collen` is `None`, the last entry of `start` contains the position in the `rowind` and `rowcoef` arrays at which an extra column would start, if it were present. This value is also the length of the `rowind` and `rowcoef` arrays. May be `None` if the problem contains no coefficients.

`collen: list[int]` Integer array containing the number of nonzero elements in each column. May be `None` if not required. This array is not required if the non-zero coefficients in the `rowind` and `rowcoef` arrays are contiguous, and the `start` array has an additional entry as described above.

`rowind: list[constraint]` Integer array containing the row indices for the nonzero elements in each column. May be `None` if the problem contains no coefficients. If the indices are input contiguously, with the columns in ascending order, the length of the `rowind` array is found by adding the `start` entry for the last column to the `collen` entry for the last column, or if `collen` is `None`, by the additional entry at the end of `start`.

`rowcoef: list[float]` Double array containing the nonzero element values; length as for `rowind`. May be `None` if the problem contains no coefficients.

`lb: list[float]` Double array containing the lower bounds on the columns. Use `-xpress.infinity` to represent a lower bound of minus infinity. If this is `None` then all lower bounds are 0 (zero).

`ub: list[float]` Double array containing the upper bounds on the columns. Use `xpress.infinity` to represent an upper bound of plus infinity. If this is `None` then all upper bounds are infinite.

`coltype: list[str]` Character array containing the entity types:
- `B` binary variables;
- `I` integer variables;
- `P` partial integer variables;
- `S` semi-continuous variables;
- `R` semi-continuous integer variables.

May be `None` if all variables are continuous.

`entind: list[var]` Integer array containing the column indices of the MIP entities. May be `None` if the problem contains no entities.

`limit: list[float]` Double array containing the integer limits for the partial integer variables and lower bounds for semi-continuous and semi-continuous integer variables (any entries in the positions corresponding to binary and integer variables will be ignored). May be `None` if not required.

`settype: list[str]` Character array containing the set types:
- `1` SOS1 type sets;
- `2` SOS2 type sets.

May be `None` if not required.

`setstart: list[int]` Integer array containing the offsets in the `setind` and `refval` arrays indicating the start of each set. This array has one entry for each set plus one extra entry containing the offset where an additional set would start, if it were present. May be `None` if not required.

`setind: list[var]` Integer array containing the columns in each set. May be `None` if not required.

`refval: list[float]` Double array containing the reference row entries for each member of the sets. These define the order for SOS2 constraints and may be used in branching for both types. May be `None` if not required.

Related controls

Integer

`EXTRACOLS`	Number of extra columns to be allowed for.
`EXTRAELEMS`	Number of extra matrix elements to be allowed for.
`EXTRAMIPENTS`	Number of extra MIP entities to be allowed for.
`EXTRAROWS`	Number of extra rows to be allowed for.
`KEEPNROWS`	Status for nonbinding rows.
`SCALING`	Type of scaling.

Double

`MATRIXTOL`	Tolerance on matrix elements.
`SOSREFTOL`	Minimum gap between reference row entries.

Further information

1. The double constants `xpress.infinity` and `-xpress.infinity` are defined in the Optimizer library header file.
2. Semi-continuous lower bounds are taken from the `limit` array. If this is `None` then they are given a default value of `1.0`. If a semi-continuous variable has a positive lower bound then this will be used as the semi-continuous lower bound and the lower bound on the variable will be set to zero.
3. Passing zero for all integer arguments and `None` for all array arguments will load an empty problem.
4. Argument `rowind` may contain either `xpress.constraint` objects or integer indices.
5. Arguments `entind, setind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.loadLP`, `problem.loadMIQP`, `problem.loadQP`, `problem.readProb`.

problem.addNames

Purpose

When a model is loaded, the rows, columns, sets, piecewise linear and general constraints of the model may not have names associated with them. This may not be important as the rows, columns, sets, piecewise linear and general constraints can be referred to by their sequence numbers. However, if you wish row, column, set, piecewise linear and general constraint names to appear in the ASCII solutions files, the names for a range of rows/columns/... can be added with `problem.addNames`.

Topic areas

Problem Creation, Optimizer

Synopsis

```
problem.addNames(type, names, first=0, last=0)
```

Arguments

`type: int` `xpress.Namespaces.ROW (=1)` for row names;
`xpress.Namespaces.COLUMN (=2)` for column names;
`xpress.Namespaces.SET (=3)` for set names;
`xpress.Namespaces.PWLCONS (=4)` for piecewise linear constraint names;
`xpress.Namespaces.GENCONS (=5)` for general constraint names;
`xpress.Namespaces.OBJECTIVE (=6)` for objective names.

`names: list[str]` Character buffer containing the string names. The length of an individual name (including the terminating `NUL`) must not exceed 1MB.

`first: int` Start of the range of rows, columns, sets, piecewise linear constraints, general constraints or objectives.

`last: int` End of the range of rows, columns, sets, piecewise linear constraints, general constraints or objectives.

Further information

This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.

Related topics

`problem.addCols`, `problem.addRows`, `problem.getNameList`.

problem.scale

Purpose

Re-scales the current matrix.

Topic areas

Numerics, Optimizer

Synopsis

```
problem.scale(rowscale, colscale)
```

Arguments

`rowscale: list[int]` Integer array of size `ROWS` (p.**??** Xpress Optimizer Reference Manual) containing the powers of `2` with which to scale the rows, or `None` if not required.

`colscale: list[int]` Integer array of size `COLS` (p.**??** Xpress Optimizer Reference Manual) containing the powers of `2` with which to scale the columns, or `None` if not required.

Related controls

Integer	
`SCALING`	Type of scaling.

Further information

1. This function currently does not apply to general nonlinear problems.
2. If `rowscale` and `colscale` are both non-`None` then they will be used to scale the matrix. Otherwise the matrix will be scaled according to the control `SCALING` (p.**??** Xpress Optimizer Reference Manual). This routine may be useful when the current matrix has been modified by calls to routines such as `problem.chgMCoef` and `problem.addRows`.
3. Any existing scaling will be replaced by the scaling specified by `rowscale` and `colscale`.
4. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.

Related topics

`problem.readProb`.

problem.readDirs

Purpose

Reads a directives file to help direct the tree search.

Topic areas

File IO, Branching, Optimizer

Synopsis

```
problem.readDirs(filename=None)
```

Argument

`filename: str` A string of up to `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual) characters containing the file name from which the directives are to be read. If omitted (or `None`), the default *problem_name* is used with a `.dir` extension.

Related controls

Double

`PSEUDOCOST` Default pseudo cost in node degradation estimation.

Further information

1. Directives can be given relating to priorities, forced branching directions, pseudo costs and model cuts. There is a priority value associated with each MIP entity. The *lower* the number, the *more* likely the entity is to be selected for branching; the *higher*, the *less* likely. By default, all MIP entities have a priority value of `500` which can be altered with a priority entry in the directives file. In general, it is advantageous for the entity's priority to reflect its relative importance in the model. Priority entries with values in excess of `1000` are illegal and are ignored. A full description of the directives file format may be found in the Optimizer Reference Manual.
2. By default, `problem.mipOptimize` will explore the branch expected to yield the best integer solution from each node, irrespective of whether this forces the MIP entity up or down. This can be overridden with an `UP` or `DN` entry in the directives file, which forces `problem.mipOptimize` to branch up first or down first on the specified entity.
3. Pseudo-costs are estimates of the unit cost of forcing an entity up or down. By default `problem.mipOptimize` uses dual information to calculate estimates of the unit up and down costs and these are added to the default pseudo costs which are set to the `PSEUDOCOST` (p.**??** Xpress Optimizer Reference Manual) control. The default pseudo costs can be overridden by a `PU` or `PD` entry in the directives file.
4. If model cuts are used, then the specified constraints are removed from the matrix and added to the Optimizer cut pool, and only put back in the matrix when they are violated by an LP solution at one of the nodes in the tree search.
5. If creating a directives file by hand, wild cards can be used to specify several vectors at once, for example `PR x1* 2` will give all MIP entities whose names start with `x1` a priority of `2`.
6. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.

Related topics

`problem.loadDirs`, the Optimizer Reference Manual.

problem.writeDirs

Purpose

Writes the tree search directives from the current problem to a directives file.

Topic areas

File IO, Branching, Optimizer

Synopsis

```
problem.writeDirs(filename=None)
```

Argument

`filename: str` A string of up to `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual) characters containing the file name to which the directives should be written. If omitted (or None), the default *problem_name* is used with a `.dir` extension.

Further information

If the problem has been presolved, only the directives for columns in the presolved problem will be written to file.

Related topics

`problem.loadDirs`, the Optimizer Reference Manual.

problem.setIndicators

Purpose

Specifies that a set of rows in the matrix will be treated as indicator constraints during a tree search. An indicator constraint is made of a `condition` and a `constraint`. The `condition` is of the type "bin = value", where `bin` is a binary variable and `value` is either 0 or 1. The `constraint` is any matrix row (may be linear, quadratic or general nonlinear). During tree search, a row configured as an indicator constraint is enforced only when condition holds, that is only if the indicator variable `bin` has the specified value. Note that every row may only get assigned a single indicator variable and term. If a row needs to be activated by multiple different terms, the row needs to be duplicated so that each term can be assigned to a distinct row. If the indicator variable should be changed, the old term needs to be deleted first (by calling `problem.delIndicators` or by calling this function with a comps argument of 0) before assigning a new one.

Topic areas

Problem Creation, Optimizer

Synopsis

```
problem.setIndicators(rowind, colind, complement)
```

Arguments

`rowind: list[constraint]` Integer array containing the indices of the rows that define the constraint part for the indicator constraints.

`colind: list[var]` Integer array containing the column indices of the indicator variables.

`complement: list[int]` Integer array with the complement flags:

- `0` not an indicator constraint (in this case the corresponding entry in the `colind` array is ignored);
- `1` for indicator constraints with condition "`bin = 1`";
- `-1` for indicator constraints with condition "`bin = 0`".

Further information

1. Indicator rows must be set up before solving the problem. Any indicator row will be removed from the matrix after presolve and added to a special pool. An indicator row will be added back into the active matrix only when its associated condition holds. An indicator variable can be used in multiple indicator rows and can also appear in normal rows and in the objective function.
2. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
3. Argument `rowind` may contain either `xpress.constraint` objects or integer indices.
4. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.getIndicators`, `problem.delIndicators`.

problem.addPwlCons

Purpose

Adds one or more piecewise linear constraints to the problem. Each piecewise linear constraint `y = f(x)` consists of an (input) column x, a (different) resultant (output column) y and a piecewise linear function f. The piecewise linear function f is described by at least two breakpoints, which are given as combinations of x- and y-values. Discontinuous piecewise linear functions are supported, in this case both the left and right limit at a given point need to be entered as breakpoints. To differentiate between left and right limit, the breakpoints need to be given as a list with non-decreasing x-values.

Topic areas

Problem Creation, Piecewise Linear and General Constraints, Optimizer

Synopsis

```
problem.addPwlCons(colind, resultant, start, xval, yval)
```

Arguments

`colind: list[var]` Integer array containing the indices of the input variables x of the piecewise linear functions.

`resultant: list[var]` Integer array containing the indices of the output variables y of the piecewise linear functions.

`start: list[int]` Integer array containing the start index of each piecewise linear constraint in the `xval` and `yval` arrays.

`xval: list[float]` Double array containing the x-values of the breakpoints.

`yval: list[float]` Double array containing the y-values of the breakpoints.

Related controls

Integer

`PWLDUALREDUCTIONS` Controls whether dual reductions may be applied to reduce the number of added variables and constraints.

Further information

1. Piecewise linear constraints must be set up before solving the problem. They are converted to additional linear constraints, continuous variables and SOS2 constraints, with the exact formulation and number of added entities depending on the convexity of the piecewise linear function and some presolving steps that are applied.
2. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
3. Arguments `colind, resultant` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.getPwlCons`, `problem.delPwlCons`.

problem.getPwlCons

Purpose

Returns the piecewise linear constraints `y = f(x)` in a given range.

Topic areas

Problem Information, Piecewise Linear and General Constraints, Optimizer

Synopsis

```
colind, resultant, start, xval, yval = problem.getPwlCons(first=0, last=PWLCONS-1)
```

Arguments

`first: int` First piecewise linear constraint in the range.

`last: int` Last piecewise linear constraint in the range.

Return value

`colind: list[int]` Integer array containing the indices of the input variables `x`.

`resultant: list[int]` Integer array containing the indices of the output variables `y`.

`start: list[int]` Integer array containing the start indices of the different constraints in the breakpoint arrays. The `x`-values of the breakpoints of piecewise linear constraint `i < last` will be given in `xval[start[i]]` to `xval[start[i+1]]`.

`xval: list[float]` Double array containing the `x`-values of the breakpoints.

`yval: list[float]` Double array containing the `y`-values of the breakpoints.

Further information

This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.

Related topics

`problem.addPwlCons`, `problem.delPwlCons`.

problem.addGenCons

Purpose

Adds one or more general constraints to the problem. Each general constraint `y = f(x1, ..., xn, c1, ..., cn)` consists of one or more (input) columns xi, zero or more constant values ci and a resultant (output column) y, different from all xi. General constraints include `maximum` and `minimum` (arbitrary number of input columns of any type and arbitrary number of input values, at least one total), `and` and `or` (at least one binary input column, no constant values, binary resultant) and `absolute value` (exactly one input column of arbitrary type, no constant values).

Topic areas

Problem Creation, Piecewise Linear and General Constraints, Optimizer

Synopsis

```
problem.addGenCons(contype, resultant, colstart, colind, valstart, val)
```

Arguments

`contype: list[GenConsType]` Integer array containing the types of the general constraints:

`xpress.GenConsType.MAX (0)`	indicates a `maximum` constraint;
`xpress.GenConsType.MIN (1)`	indicates a `minimum` constraint;
`xpress.GenConsType.AND (2)`	indicates an `and` constraint.
`xpress.GenConsType.OR (3)`	indicates an `or` constraint;
`xpress.GenConsType.ABS (4)`	indicates an `absolute value` constraint.

`resultant: list[var]` Integer array containing the indices of the output variables of the general constraints.

`colstart: list[int]` Integer array containing the start index of each general constraint in the `colind` array.

`colind: list[var]` Integer array containing the input variables in all general constraints.

`valstart: list[int]` Integer array containing the start index of each general constraint in the `val` array (may be `None`).

`val: list[float]` Double array containing the constant values in all general constraints (may be `None`).

Related controls

Integer

`GENCONSDUALREDUCTIONS` Controls whether dual reductions may be applied to reduce the number of added variables and constraints.

Further information

1. General constraints must be set up before solving the problem. They are converted to additional binary variables, indicator and linear constraints with the exact formulation and number of added entities depending on the performed presolving.
2. Note that using non-binary variables in `and`/`or` constraints or adding constant values to them or `absolute value` constraints will give an error at solve time.
3. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
4. Arguments `resultant, colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.getGenCons`, `problem.delGenCons`.

problem.getGenCons

Purpose

Returns the general constraints `y = f(x1, ..., xn, c1, ..., cm)` in a given range.

Topic areas

Problem Information, Piecewise Linear and General Constraints, Optimizer

Synopsis

```
contype, resultant, colstart, colind, valstart, val = problem.getGenCons(first=0,
       last=GENCONS-1)
```

Arguments

`first: int` First general constraint in the range.

`last: int` Last general constraint in the range.

Return value

`contype: list[GenConsType]` An integer array of length `last-first+1` containing the types of the general constraints:

- `xpress.GenConsType.MAX (0)` indicates a `maximum` constraint;
- `xpress.GenConsType.MIN (1)` indicates a `minimum` constraint;
- `xpress.GenConsType.AND (2)` indicates an `and` constraint.
- `xpress.GenConsType.OR (3)` indicates an `or` constraint;
- `xpress.GenConsType.ABS (4)` indicates an `absolute value` constraint.

`resultant: list[int]` Integer array containing the indices of the output variables `y`.

`colstart: list[int]` Integer array of length `last-first+2` containing the start index of each general constraint in the `colind` array.

`colind: list[int]` Integer array containing the indices of the input variables `xi`.

`valstart: list[int]` Integer array of length `last-first+2` containing the start index of each general constraint in the `val` array.

`val: list[float]` Integer array containing the constant values `ci`.

Further information

This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.

Related topics

`problem.addGenCons`, `problem.delGenCons`.

problem.delPwlCons

Purpose

Delete piecewise linear constraints from a problem.

Topic areas

Problem Creation, Piecewise Linear and General Constraints, Optimizer

Synopsis

```
problem.delPwlCons(pwlind)
```

Argument

`pwlind: list[int]` An integer array containing the piecewise linear constraints to delete.

Further information

1. After piecewise linear constraints have been deleted from a problem, the indices of the remaining constraints are reduced so that the piecewise linear constraints are always numbered from `0` to `PWLCONS-1` where `PWLCONS` (p.**??** Xpress Optimizer Reference Manual) is the problem attribute containing the number of non-deleted piecewise linear constraints in the problem.
2. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.

Related topics

`problem.addPwlCons`, `problem.getPwlCons`.

problem.delGenCons

Purpose

Delete general constraints from a problem.

Topic areas

Problem Creation, Piecewise Linear and General Constraints, Optimizer

Synopsis

```
problem.delGenCons(conind)
```

Argument

`conind: list[int]` An integer array containing the general constraints to delete.

Further information

1. After general constraints have been deleted from a problem, the indices of the remaining constraints are reduced down so that the general constraints are always numbered from `0` to `GENCONS-1` where `GENCONS` (p.**??** Xpress Optimizer Reference Manual) is the problem attribute containing the number of non-deleted general constraints in the problem.
2. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.

Related topics

`problem.addGenCons`, `problem.getGenCons`.

problem.dumpControls

Purpose

Displays the list of controls and their current value for those controls that have been set to a non default value.

Topic areas

Controls and Attributes, Optimizer

Synopsis

```
problem.dumpControls()
```

problem.getIndicators

Purpose

Returns the indicator constraint condition (indicator variable and complement flag) associated to the rows in a given range.

Topic areas

Problem Information, Optimizer

Synopsis

```
colind, complement = problem.getIndicators(first=0, last=ROWS-1)
```

Arguments

`first: int` First row in the range.

`last: int` Last row in the range (inclusive).

Return value

`colind: list[int]` Integer array of length `last-first+1` containing the column indices of the indicator variables.

`complement: list[int]` Integer array of length `last-first+1` containing the indicator complement flags:

`0` not an indicator constraint (in this case the corresponding entry in the `colind` array is ignored);

`1` for indicator constraints with condition "`bin = 1`";

`-1` for indicator constraints with condition "`bin = 0`".

Further information

1. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
2. Arguments `first, last` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.setIndicators`, `problem.delIndicators`.

problem.delIndicators

Purpose

Delete indicator constraints. This turns the specified rows into normal rows (not controlled by indicator variables).

Topic areas

Problem Creation, MIP Entities, Optimizer

Synopsis

```
problem.delIndicators(first=0, last=ROWS-1)
```

Arguments

`first: int` First row in the range.

`last: int` Last row in the range (inclusive).

Further information

1. This function has no effect on rows that are not indicator constraints.
2. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
3. Arguments `first, last` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.getIndicators`, `problem.setIndicators`.

problem.lpOptimize

Purpose

This function begins a search for the optimal continuous (LP) solution. The direction of optimization is given by `OBJSENSE` (p.**??** Xpress Optimizer Reference Manual). The status of the problem when the function completes can be checked using `LPSTATUS` (p.**??** Xpress Optimizer Reference Manual). Any MIP entities in the problem will be ignored.

Topic areas

Solution Process, LP, Optimizer

Synopsis

```
problem.lpOptimize(flags="")
```

Argument

`flags: str` Flags to pass to `problem.lpOptimize`. The default is `""` or `None`, in which case the algorithm used is determined by the `DEFAULTALG` (p.**??** Xpress Optimizer Reference Manual) control. If the argument includes:

- `b` the problem will be solved using the Newton barrier method, or the Hybrid gradient method if `BARALG` (p.**??** Xpress Optimizer Reference Manual) is set to 4;
- `p` the problem will be solved using the primal simplex algorithm;
- `d` the problem will be solved using the dual simplex algorithm;
- `n` the network part of the problem will be identified and solved using the network simplex algorithm;

Further information

1. The algorithm used to optimize is determined by the `DEFAULTALG` (p.**??** Xpress Optimizer Reference Manual) control if no flags are provided. By default, the concurrent algorithm is used for linear problems (except if KEEPBASIS is turned on), and the barrier algorithm is used for quadratic problems.
2. The `d` and `p` flags can be used with the `n` flag to complete the solution of the problem with either the dual or primal algorithms once the network algorithm has solved the network part of the problem.
3. The `b` flag cannot be used with the `n` flag.

Related topics

`problem.mipOptimize`, the Optimizer Reference Manual.

problem.mipOptimize

Purpose

This function begins a tree search for the optimal MIP solution. The direction of optimization is given by OBJSENSE (p.?? Xpress Optimizer Reference Manual). The status of the problem when the function completes can be checked using MIPSTATUS (p.?? Xpress Optimizer Reference Manual).

Topic areas

Solution Process, MIP Entities, Optimizer

Synopsis

```
problem.mipOptimize(flags="")
```

Argument

`flags: str` Flags to pass to `problem.mipOptimize`, which specifies how to solve the initial continuous problem where the MIP entities are relaxed. The default is `""` or `None`, in which case the choice of the LP algorithm is left to the solver. If the argument includes:

- `b` the initial continuous relaxation will be solved using the Newton barrier method (or the hybrid gradient method if BARALG (p.?? Xpress Optimizer Reference Manual) is set to 4);
- `p` the initial continuous relaxation will be solved using the primal simplex algorithm;
- `d` the initial continuous relaxation will be solved using the dual simplex algorithm;
- `n` the network part of the initial continuous relaxation will be identified and solved using the network simplex algorithm;
- `l` *(deprecated)* stop after having solved the initial continous relaxation. This flag is deprecated, use the MIPSTOPSTAGE (p.?? Xpress Optimizer Reference Manual) control instead.

Further information

1. If the `l` flag is used, the Optimizer will stop immediately after solving the initial continuous relaxation. The status of the continuous solve can be checked with LPSTATUS (p.?? Xpress Optimizer Reference Manual) and standard LP results are available, such as the objective value (LPOBJVAL (p.?? Xpress Optimizer Reference Manual)) and solution (use `problem.getLpSol`), depending on LPSTATUS (p.?? Xpress Optimizer Reference Manual).
2. If multiple of the `b`, `p`, `d` and `n` flags are provided and sufficient threads are available, a concurrent between those algorithms will be run.
3. It is possible for the Optimizer to find integer solutions before solving the initial continuous relaxation, either through heuristics or by having the user load an initial integer solution. This can potentially result in the tree search finishing before solving the continuous relaxation to optimality.
4. If the function returns without having completed the search for an optimal solution, the search can be resumed from where it stopped by calling `problem.mipOptimize` again.
5. The algorithm used to reoptimize the continuous relaxations during the tree search is given by DEFAULTALG (p.?? Xpress Optimizer Reference Manual). The default is to use the dual simplex algorithm.

Related topics

`problem.lpOptimize`, the Optimizer Reference Manual.

problem.optimize

Purpose

This function begins a search for the optimal solution of the problem. The direction of optimization is given by `OBJSENSE` (p.**??** Xpress Optimizer Reference Manual).

Topic areas

Solution Process, Optimizer

Synopsis

```
solvestatus, solstatus = problem.optimize(flags="")
```

Argument

`flags: str` Flags to pass to `problem.optimize`. The default is `""` or `None`. If the argument includes:

- `s` solve the problem to local optimality;
- `x` solve the problem to global optimality;
- `l` *(deprecated)* if a branch and bound search is necessary to solve the problem, stop after solving the root node. This flag is deprecated, use the `MIPSTOPSTAGE` (p.**??** Xpress Optimizer Reference Manual) control instead.

Return value

`solvestatus: SolveStatus` The solve status after termination. Takes the same values as `SOLVESTATUS` (p.**??** Xpress Optimizer Reference Manual)

`solstatus: SolStatus` The solution status after termination. Takes the same values as `SOLSTATUS` (p.**??** Xpress Optimizer Reference Manual)

Further information

1. A problem is identified as a MIP if it contains any of the following: MIP entities, piecewise linear constraints or general constraints, or any of the callbacks set by `problem.addOptnodeCallback`, `problem.addPreIntsolCallback` or `problem.addChangeBranchObjectCallback`.
2. `problem.optimize` automatically selects the optimization method based on the problem type. If the problem contains nonlinear formulas or a non-convex quadratic constraint or objective function, it is equivalent to calling `problem.nlpOptimize`, except that the `g` flag will be applied by default. Otherwise, for a problem identified as a MIP, it is equivalent to calling `problem.mipOptimize` and for all other problems it is equivalent to calling `problem.lpOptimize`. To determine which method was selected check the `OPTIMIZETYPEUSED` (p.**??** Xpress Optimizer Reference Manual) attribute. If the value is `xpress.OptimizeType.LOCAL`, then you can check the `LOCALSOLVERSELECTED` (p.**??** Xpress NonLinear Reference Manual) attribute to determine which local solver was selected.
3. Passing the `s` flag is equivalent to setting the `NLPSOLVER` (p.**??** Xpress NonLinear Reference Manual) control to `xpress.constants.NLPSOLVER_LOCAL`. Passing the `x` flag is equivalent to setting the `NLPSOLVER` (p.**??** Xpress NonLinear Reference Manual) control to `xpress.constants.NLPSOLVER_GLOBAL`. If neither flag is passed, and if the `NLPSOLVER` (p.**??** Xpress NonLinear Reference Manual) control is `xpress.constants.NLPSOLVER_AUTOMATIC`, then this decision will be made based on problem attributes, including convexity and the presence of user functions, and which features are authorized by the license file.
4. Any additional flags not listed above will be treated in the same way as for `problem.lpOptimize`, `problem.mipOptimize` and `problem.nlpOptimize`, depending on the type of optimization performed. The `DEFAULTALG` (p.**??** Xpress Optimizer Reference Manual) control will also behave in the same way as for these functions.
5. If there is a solve in progress, `problem.optimize` will always try to continue that solve (similar to `problem.mipOptimize` but unlike `problem.nlpOptimize`, which would only do so if the `-c` flag was given).
6. The method used to solve the problem is stored in the `OPTIMIZETYPEUSED` (p.**??** Xpress Optimizer Reference Manual) attribute.
7. Regarding the `l` flag, a branch and bound search is required when a problem is identified as a MIP, or when the global solver is selected. This includes a problem with non-linear formulas that can be reformulated as a MIP (for example, if the formulas contain only piecewise linear functions, min/max functions or convex quadratic constraints). In these cases, passing the `l` flag will cause the Optimizer to stop after solving the initial LP relaxation.

Related topics

`problem.lpOptimize`, `problem.mipOptimize`, `problem.nlpOptimize` (`NLPOPTIMIZE`), the Optimizer Reference Manual.

problem.readSlxSol

Purpose

Reads an ASCII solution file (`.slx`) created by the `problem.writeSlxSol` function.

Topic areas

File IO, Solution, Optimizer

Synopsis

```
problem.readSlxSol(filename="", flags="")
```

Arguments

`filename: str` A string of up to `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual) characters containing the file name to which the solution is to be read. If omitted, the default *problem_name* is used with a `.slx` extension.

`flags: str` Flags to pass to `problem.readSlxSol`:

- `l` read the solution as an LP solution in case of a MIP problem;
- `m` read the solution as a solution for the MIP problem;
- `a` read multiple MIP solutions from the `.slx` file and add them to the MIP problem;
- `v` use the provided filename verbatim, without appending the `.slx` extension;
- `z` read a compressed input file.

Further information

1. When `problem.readSlxSol` is called before a MIP solve, the loaded solutions will not be checked before calling `problem.mipOptimize`. By default, only the last MIP solution read from the `.slx` file will be stored. Use the `a` flag to store all MIP solutions read from the file.
2. When using the `a` flag, read solutions will be queued similarly to the user of the `problem.addMipSol` function. Each name string given by the `NAME` field in the `.slx` file will be associated with the corresponding solution. Any registered `usersolnotify` callback will be fired when the solution has been checked, and will include the read name string as one of its arguments.
3. Note that when solving with FICO Xpress Global (including non-convex quadratic problems), the `a` flag always needs to be provided.
4. Refer to the Optimizer Reference Manual for a description of the ASCII Solution (.slx) File format.

Related topics

`problem.readBinSol`, `problem.writeSlxSol`, `problem.writeBinSol` `problem.readBinSol`, `problem.addMipSol`, `problem.addUserSolNotifyCallback`.

problem.readBasis

Purpose

Instructs the Optimizer to read in a previously saved basis from a file.

Topic areas

File IO, LP, Optimizer

Synopsis

```
problem.readBasis(filename="", flags="")
```

Arguments

`filename: str` A string of up to `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual) characters containing the file name from which the basis is to be read. If omitted, the default *problem_name* is used with a `.bss` extension.

`flags: str` Flags to pass to `problem.readBasis`:

- `n` input basis file containing basic solution values;
- `t` input a compact advanced form of the basis;
- `v` use the provided filename verbatim, without appending the `.bss` extension;
- `z` read a compressed input file.

Further information

1. The only check done when reading compact basis is that the number of rows and columns in the basis agrees with the current number of rows and columns.
2. `problem.readBasis` will read the basis for the original problem even if the matrix has been presolved. The Optimizer will read the basis, checking that it is valid, and will display error messages if it detects inconsistencies.
3. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.

Related topics

`problem.loadBasis`, `problem.writeBasis`.

problem.readBinSol

Purpose

Reads a solution from a binary solution file.

Topic areas

File IO, Solution, Optimizer

Synopsis

```
problem.readBinSol(filename="", flags="")
```

Arguments

`filename: str` A string of up to `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual) characters containing the file name from which the solution is to be read. If omitted, the default *problem_name* is used with a `.sol` extension.

`flags: str` Flags to pass to `problem.readBinSol`:

- `m` load the solution as a solution for the MIP;
- `x` load the solution as a solution for the LP;
- `v` use the provided filename verbatim, without appending the `.sol` extension;
- `z` read a compressed input file.

Further information

This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.

Related topics

`problem.getSolution`, `problem.writeBinSol`, `problem.writeSol`, `problem.writePrtSol`.

problem.getInfeas

Purpose

Returns a list of infeasible primal and dual variables.

Topic areas

Solution, LP, Optimizer

Synopsis

```
x, slack, duals, djs = problem.getInfeas()
```

Return value

`x: list[int]` Integer array containing the indices of the primal infeasible variables.

`slack: list[int]` Integer array containing the indices of the primal infeasible rows.

`duals: list[int]` Integer array containing the indices of the dual infeasible rows.

`djs: list[int]` Integer array containing the indices of the dual infeasible variables.

Error values

91	A current problem is not available.
422	A solution is not available.

Related controls

Double

FEASTOL	Tolerance on RHS.
OPTIMALITYTOL	Reduced cost tolerance.

Further information

1. This function applies to linear and convex quadratic (including second order conic) problems only.
2. To find the infeasibilities in a previously saved solution, the solution must first be loaded into memory with the `problem.readBinSol` function.

Related topics

`problem.getScaledInfeas`, `problem.getIISData`, `problem.IISAll`, `problem.clearIIS`, `problem.firstIIS`, `problem.IISIsolations`, `problem.nextIIS`, `problem.IISStatus`, `problem.writeIIS`

problem.getScaledInfeas

Purpose

Returns a list primal and dual variables that are infeasible for the scaled original problem. If the problem is currently presolved, it is postsolved before the function returns.

Topic areas

LP, Numerics, Optimizer

Synopsis

```
x, slack, duals, djs = problem.getScaledInfeas()
```

Return value

`x: list[int]` Integer array containing the indices of the primal infeasible variables.

`slack: list[int]` Integer array containing the indices of the primal infeasible rows.

`duals: list[int]` Integer array containing the indices of the dual infeasible rows.

`djs: list[int]` Integer array containing the indices of the dual infeasible variables.

Error value

`422` A solution is not available.

Related controls

Double

`FEASTOL`	Tolerance on RHS.
`OPTIMALITYTOL`	Reduced cost tolerance.

Related controls

Integer

`SCALING`	Type of scaling

Further information

1. The returned row/column indices range from `0` to `ORIGINALROWS-1`/`ORIGINALCOLS-1`.
2. This function is the same as `problem.getInfeas`, except it checks for infeasibility before unscaling the problem. The two functions will behave differently only when scaling before presolve has been enabled, by setting bit `9` in `SCALING` (p.**??** Xpress Optimizer Reference Manual).
3. This function applies to linear and convex quadratic (including second order conic) problems only.

Related topics

`problem.getInfeas`, `problem.getIISData`, `problem.IISAll`, `problem.clearIIS`, `problem.firstIIS`, `problem.IISIsolations`, `problem.nextIIS`, `problem.IISStatus`, `problem.writeIIS`

problem.getUnbVec

Purpose

Returns the index vector which causes the primal simplex or dual simplex algorithm to determine that a matrix is primal or dual unbounded respectively.

Topic areas

Solution, LP, Optimizer

Synopsis

```
seq = problem.getUnbVec()
```

Return value

`seq: int` The vector causing the problem to be detected as being primal or dual unbounded. In the dual simplex case, the vector is the leaving row for which the dual simplex detected dual unboundedness. In the primal simplex case, the vector is the entering row `seq` (if `seq` is in the range `0` to `ROWS` (p.**??** Xpress Optimizer Reference Manual)−`1`) or column (variable) `seq−ROWS−SPAREROWS` (p.**??** Xpress Optimizer Reference Manual) (if `seq` is between `ROWS+SPAREROWS` and `ROWS+SPAREROWS+COLS` (p.**??** Xpress Optimizer Reference Manual)−`1`) for which the primal simplex detected primal unboundedness.

Error value

`91` A current problem is not available.

Further information

When solving using the dual simplex method, if the problem is primal infeasible then `problem.getUnbVec` returns the pivot row where dual unboundedness was detected. Also note that when solving using the dual simplex method, if the problem is primal unbounded then `problem.getUnbVec` returns -1 since the problem is dual infeasible and not dual unbounded.

Related topics

`problem.getInfeas`, `problem.lpOptimize`.

problem.crossoverLpSol

Purpose

Provides a basic optimal solution for a given solution of an LP problem. This function behaves like the crossover after the barrier algorithm.

Topic areas

Barrier, LP, Optimizer

Synopsis

```
status = problem.crossoverLpSol()
```

Return value

`status: int` The status. The status is one of:

`0` The crossover is successful.

`1` The crossover is not performed because the problem has no solution.

Related controls

Integer

`ALGAFTERCROSSOVER` Specifies which algorithm to use for cleaning up the solution.

`PREPROTECTDUAL` Whether or not to protect the given dual solution during presolve.

Further information

1. The crossover contains two phases: a crossover phase for finding a basic solution and a clean-up phase for finding a basic optimal solution. Setting `ALGAFTERCROSSOVER` to `0` will allow the crossover to skip the clean-up phase.
2. The given solution is expected to be feasible or nearly feasible, otherwise the crossover may take a long time to find a basic feasible solution. More importantly, the given solution is expected to have a small duality gap. A small duality gap indicates that the given solution is close to the optimal solution. If the given solution is far away from the optimal solution, the clean-up phase may need many simplex iterations to move to a basic optimal solution.

Related topics

`problem.loadLPSol`, `problem.readSlxSol`, the Optimizer Reference Manual.

problem.tune

Purpose

This function begins a tuner session for the current problem. The tuner will solve the problem multiple times while evaluating a list of control settings and promising combinations of them. When finished, the tuner will select and set the best control setting on the problem. Note that the direction of optimization is given by `OBJSENSE` (p.**??** Xpress Optimizer Reference Manual).

Topic areas

Tuner, Optimizer

Synopsis

```
problem.tune(flags="")
```

Argument

`flags: str` Flags to pass to `problem.tune`, which specify whether to tune the current problem as an LP or a MIP problem, and the algorithm for solving the LP problem or the initial LP relaxation of the MIP. The flags are optional. If the argument includes:

- `l` will tune the problem as an LP (mutually exclusive with flag `g`);
- `g` will tune the problem as a MIP (mutually exclusive with flag `l`);
- `x` will tune the problem as a Global Optimization problem with Xpress Global;
- `d` will use the dual simplex method;
- `p` will use the primal simplex method;
- `b` will use the barrier method;
- `n` will use the network simplex method.

Further information

Please refer to the Optimizer Reference Manual for a detailed guide of how to use the tuner.

Related topics

`problem.tuneProbSetFile`, `problem.tunerReadMethod`, `problem.tunerWriteMethod`.

problem.tuneProbSetFile

Purpose

This function begins a tuner session for a set of problems. The tuner will solve the problems multiple times while evaluating a list of control settings and promising combinations of them. When finished, the tuner will select and set the best control setting on the problems.

Topic areas

Tuner, Optimizer

Synopsis

```
problem.tuneProbSetFile(setfile, ifmip=-1, sense=0)
```

Arguments

`setfile: str` A plain text file which contains a list of problem filenames.

`ifmip: int`

`-1`	to automatically determine whether to solve the problem set as LP or MIP;
`0`	to force the tuner to tune the problem set as LP;
`1`	to force the tuner to tune the problem set as MIP.

`sense: int`

`0`	to automatically determine the sense of each problem;
`1`	to force the tuner to minimize each problem;
`-1`	to force the tuner to maximize each problem.

Further information

Please refer to the Optimizer Reference Manual for a detailed guide of how to use the tuner.

Related topics

`problem.tune`, `problem.tunerReadMethod`, `problem.tunerWriteMethod`.

problem.tunerWriteMethod

Purpose

This function writes the current tuner method to a given file or prints it to the console.

Topic areas

Tuner, File IO, Optimizer

Synopsis

```
problem.tunerWriteMethod(methodfile)
```

Argument

`methodfile: str` The method file name, to which the tuner will write the current tuner method. If the input is `"stdout"` or `"STDOUT"`, then the tuner will print the method to the console instead.

Further information

Please refer to the Optimizer Reference Manual for more information about the tuner method, and Appendix the Optimizer Reference Manual for the format of the tuner method file.

Related topics

`problem.tunerReadMethod`, `problem.tune`, `problem.tuneProbSetFile`.

problem.tunerReadMethod

Purpose

This function loads a user defined tuner method from the given file.

Topic areas

Tuner, File IO, Optimizer

Synopsis

```
problem.tunerReadMethod(methodfile)
```

Argument

`methodfile: str` The method file name, from which the tuner can load a user-defined tuner method.

Further information

Please refer to the Optimizer Reference Manual for more information about the tuner method, and Appendix the Optimizer Reference Manual for the format of the tuner method file.

Related topics

`problem.tunerWriteMethod`, `problem.tune`, `problem.tuneProbSetFile`.

problem.getLastBarSol

Purpose

Used to obtain the last barrier solution values following optimization that used the barrier solver.

Topic areas

Solution, Barrier, Optimizer

Synopsis

```
x, slack, duals, djs, status = problem.getLastBarSol()
```

Return value

`x: list[float]` Double array of length `ORIGINALCOLS` (p.**??** Xpress Optimizer Reference Manual) containing the values of the primal variables.

`slack: list[float]` Double array of length `ORIGINALROWS` (p.**??** Xpress Optimizer Reference Manual) containing the values of the slack variables.

`duals: list[float]` Double array of length `ORIGINALROWS` containing the values of the dual variables ($c_B^T B^{-1}$).

`djs: list[float]` Double array of length `ORIGINALCOLS` containing the reduced cost for each variable ($c^T - c_B^T B^{-1} A$).

`status: int` Status of the last barrier solve. Value matches that of `LPSTATUS` (p.**??** Xpress Optimizer Reference Manual) should the solve have been stopped immediately after the barrier.

Further information

1. If the barrier solver has not been used, `status` will return `xpress.LPStatus.UNSOLVED`.
2. The barrier solution or the solution candidate is always available if the status is not `xpress.LPStatus.UNSOLVED`.
3. The last barrier solution is available until the next solve, and is not invalidated by otherwise working with the problem.

Related topics

`problem.getSolution`, `problem.getCallbackSolution`.

problem.clearIIS

Purpose

Resets the search for Irreducible Infeasible Sets (IIS).

Topic areas

Infeasibility, Optimizer

Synopsis

```
problem.clearIIS()
```

Further information

1. The information stored internally about the IISs identified by `problem.firstIIS`, `problem.nextIIS` or `problem.IISAll` are cleared. Functions `problem.getIISData`, `problem.IISStatus`, `problem.writeIIS` and `problem.IISIsolations` cannot be called until the IIS identification procedure is started again.
2. This function is called automatically by `problem.firstIIS` and `problem.IISAll`.

Related topics

`problem.getIISData`, `problem.IISAll`, `problem.firstIIS`, `problem.IISIsolations`, `problem.nextIIS`, `problem.printIIS`, `problem.IISStatus`, `problem.writeIIS`

problem.firstIIS

Purpose

Initiates a search for an Irreducible Infeasible Set (IIS) in an infeasible problem.

Topic areas

Infeasibility, Optimizer

Synopsis

```
status = problem.firstIIS(mode)
```

Argument

`mode: int` The IIS search mode:

0	stops after finding the initial infeasible subproblem;
1	find an IIS, emphasizing simplicity of the IIS;
2	find an IIS, emphasizing a quick result.

Return value

`status: int` The status after the search:

0	success;
1	feasible problem;
2	error;
3	timeout or interruption.

Further information

1. This function applies to all problem types.
2. A model may have several infeasibilities. Repairing a single IIS may not make the model feasible. For this reason the Optimizer can find an IIS for each of the infeasibilities in a model. For the generation of several independent IISs use functions `problem.nextIIS` or `problem.IISAll`.
3. IIS sensitivity filter: after an optimal but infeasible first phase primal simplex, it is possible to identify a subproblem containing all the infeasibilities (corresponding to the given basis) to reduce the size of the IIS working problem dramatically, i.e., rows with zero duals (thus with artificials of zero reduced cost) and columns that have zero reduced costs may be deleted. Moreover, for rows and columns with nonzero costs, the sign of the cost is used to relax equality rows either to less than or greater than equal rows, and to drop either possible upper or lower bounds on columns.
4. Initial infeasible subproblem: The subproblem identified after the sensitivity filter is referred to as initial infeasible subproblem. Its size is crucial to the running time of the deletion filter and it contains all the infeasibilities of the first phase simplex, thus if the corresponding rows and bounds are removed the problem becomes feasible.
5. `problem.firstIIS` performs the initial sensitivity analysis on rows and columns to reduce the problem size, and sets up the initial infeasible subproblem. This subproblem significantly speeds up the generation of IISs, however in itself it may serve as an approximation of an IIS, since its identification typically takes only a fraction of time compared to the identification of an IIS.
6. The IIS approximation and the IISs generated so far are always available.

Related topics

`problem.getIISData`, `problem.IISAll`, `problem.clearIIS`, `problem.IISIsolations`, `problem.nextIIS`, `problem.printIIS`, `problem.IISStatus`, `problem.writeIIS`

problem.nextIIS

Purpose

Continues the search for further Irreducible Infeasible Sets (IIS), or calls `problem.firstIIS` if no IIS has been identified yet.

Topic areas

Infeasibility, Optimizer

Synopsis

```
status = problem.nextIIS()
```

Return value

`status: int` The status after the search:

0	success;
1	no more IIS could be found, or problem is feasible if no `problem.firstIIS` call preceded;
2	on error.

Further information

1. Returning multiple IISs is supported only for linear problems.
2. A model may have several infeasibilities. Repairing a single IIS may not make the model feasible. For this reason the Optimizer attempts to find an IIS for each of the infeasibilities in a model. You may call the `problem.nextIIS` function repeatedly, or use the `problem.IISAll` function to retrieve all IIS at once.
3. This function is not affected by the control `MAXIIS` (p.**??** Xpress Optimizer Reference Manual).
4. If the problem has been modified since the last call to `problem.firstIIS` or `problem.nextIIS`, the generation process has to be started from scratch.

Related topics

`problem.getIISData`, `problem.IISAll`, `problem.clearIIS`, `problem.firstIIS`, `problem.IISIsolations`, `problem.printIIS`, `problem.IISStatus`, `problem.writeIIS`

problem.printIIS

Purpose

Prints a given Irreducible Infeasible Set (IIS) in the log. If 0 is passed as the IIS number parameter, the initial infeasible subproblem is printed.

Topic areas

Infeasibility, Logging, Optimizer

Synopsis

```
problem.printIIS(iis)
```

Argument

`iis: int` The ordinal number of the IIS to be printed.

Related topics

`problem.getIISData`, `problem.IISAll`, `problem.clearIIS`, `problem.firstIIS`, `problem.IISIsolations`, `problem.nextIIS`, `problem.IISStatus`.

problem.IISStatus

Purpose

Returns statistics on the Irreducible Infeasible Sets (IIS) found so far by `problem.firstIIS`, `problem.nextIIS` or `problem.IISAll`.

Topic areas

Infeasibility, Optimizer

Synopsis

```
niis, nrows, ncols, suminfeas, numinfeas = problem.IISStatus()
```

Return value

`niis: int` The number of IISs found so far.

`nrows: list[int]` Array containing the number of rows in each IIS.

`ncols: list[int]` Array containing the number of bounds in each IIS.

`suminfeas: list[float]` Array containing the sum of infeasibilities in each IIS after the first phase simplex.

`numinfeas: list[int]` Array containing the number of infeasible variables in each IIS after the first phase simplex.

Further information

1. The arrays are `0` based, index 0 corresponding to the initial infeasible subproblem.
2. For the initial infeasible problem (IIS 0) the subproblem size is returned (which may be different from the number of bounds), while for the IISs the number of bounds is returned (usually much smaller than the number of columns in the IIS).
3. Note that the values in `suminfeas` and `numinfeas` depend on the IIS, and in turn on the basis where the simplex algorithm has stopped. Different IISs can have different error measures.
4. `niis` is set to `-1` if the search for IISs has not yet started.

Related topics

`problem.getIISData`, `problem.IISAll`, `problem.clearIIS`, `problem.firstIIS`, `problem.IISIsolations`, `problem.nextIIS`, `problem.printIIS`, `problem.writeIIS`

problem.IISAll

Purpose

Performs an automated search for independent Irreducible Infeasible Sets (IIS) in an infeasible problem.

Topic areas

Infeasibility, Optimizer

Synopsis

```
problem.IISAll()
```

Related controls

Integer

MAXIIS	Number of Irreducible Infeasible Sets to be found.

Further information

1. This function applies to all problem types, including general nonlinear problems. For anything but linear problems its functionality is identical to `problem.firstIIS`.
2. A model may have several infeasibilities. Repairing a single IIS may not make the model feasible. For this reason the Optimizer can find an IIS for each of the infeasibilities in a model. If the control `MAXIIS` (p.**??** Xpress Optimizer Reference Manual) is set to a positive integer value then the `problem.IISAll` command will stop if `MAXIIS` (p.**??** Xpress Optimizer Reference Manual) IISs have been found. By default the control `MAXIIS` (p.**??** Xpress Optimizer Reference Manual) is set to `-1`, in which case an IIS is found for each of the infeasibilities in the model.
3. The problem attribute `NUMIIS` (p.**??** Xpress Optimizer Reference Manual) allows the user to recover the number of IISs found in a particular search. Alternatively, the `problem.IISStatus` function may be used to retrieve the number of IISs found by `problem.firstIIS`, `problem.nextIIS` or `problem.IISAll` functions.

Related topics

`problem.getIISData`, `problem.clearIIS`, `problem.firstIIS`, `problem.IISIsolations`, `problem.nextIIS`, `problem.printIIS`, `problem.IISStatus`, `problem.writeIIS`

problem.writeIIS

Purpose

Writes an LP/MPS/CSV file containing a given Irreducible Infeasible Set (IIS). If 0 is passed as the IIS number parameter, the initial infeasible subproblem is written.

Topic areas

Infeasibility, File IO, Optimizer

Synopsis

```
problem.writeIIS(iis, filetype, filename=None, flags="lp")
```

Arguments

`iis: int` The ordinal number of the IIS to be written.

`filetype: int` Type of file to be created:

0	creates an lp/mps file containing the IIS as a linear programming problem;
1	creates a comma separated (csv) file containing the description and supplementary information on the given IIS.

`filename: str` The name of the file to be created. If omitted (or `None`), the default *problem_name*.

`flags: str` Flags passed to the `problem.writeProb` function.

Further information

1. Please note that there are problems on the boundary of being infeasible or not. For such problems, feasibility or infeasibility often depends on tolerances or even on scaling. This phenomenon makes it possible that after writing an IIS out as an LP file and reading it back, it may report feasibility. As a first check it is advised to consider the following options:
    1. turn presolve off (e.g. in console `presolve = 0`) since the nature of an IIS makes it necessary that during their identification the presolve is turned off.
    2. use the primal simplex method to solve the problem (e.g. in console `LPOPTIMIZE -p`).
    3. Tighten the feasibility tolerances `FEASTOL` (p.**??** Xpress Optimizer Reference Manual) or `MIPTOL` (p.**??** Xpress Optimizer Reference Manual).
2. Note that the original sense of the original objective function plays no role in an IIS.
3. Even though an attempt is made to identify the most infeasible IISs first by the `problem.firstIIS`, `problem.nextIIS` and `problem.IISAll` functions, it is also possible that an IIS becomes just infeasible in problems that are otherwise highly infeasible. In such cases, you may try to deal with the more stable IISs first, and consider to use the infeasibility breaker tool if only slight infeasibilities remain.
4. The LP or MPS files created by `problem.writeIIS` corresponding to an IIS contain no objective function, since infeasibility is independent of the objective.

Related topics

`problem.getIISData`, `problem.IISAll`, `problem.clearIIS`, `problem.firstIIS`, `problem.IISIsolations`, `problem.nextIIS`, `problem.printIIS`, `problem.IISStatus`

problem.IISIsolations

Purpose

Performs the isolation identification procedure for an Irreducible Infeasible Set (IIS). This function applies only to linear problems.

Topic areas

Infeasibility, Optimizer

Synopsis

```
problem.IISIsolations(iis)
```

Argument

`iis: int` The number of the IIS identified by either `problem.firstIIS`, `problem.nextIIS` or `problem.IISAll` in which the isolations should be identified.

Further information

1. An IIS isolation is a special constraint or bound in an IIS. Removing an IIS isolation constraint or bound will remove all infeasibilities in the IIS without increasing the infeasibilities in any row or column outside the IIS, thus in any other IISs. The IIS isolations thus indicate the likely cause of each independent infeasibility and give an indication of which constraint or bound to drop or modify. It is not always possible to find IIS isolations.
2. Generally, one should first look for rows or columns in the IIS which are both in isolation, and have a high dual multiplier relative to the others.
3. The `iis` parameter cannot be zero: the concept of isolations is meaningless for the initial infeasible subproblem.

Related topics

`problem.getIISData`, `problem.IISAll`, `problem.clearIIS`, `problem.firstIIS`, `problem.nextIIS`, `problem.printIIS`, `problem.IISStatus`, `problem.writeIIS`

problem.getIISData

Purpose

Returns information for an Irreducible Infeasible Set: size, variables and constraints (row and column vectors), and conflicting sides of the variables. For pure linear problems there is also information on duals, reduced costs and isolations.

Topic areas

Infeasibility, Optimizer

Synopsis

```
rowind, colind, contype, bndtype, duals, djs, isolationrows, isolationcols =
       problem.getIISData(iis)
```

Argument

`iis: int` The ordinal number of the IIS to get data for.

Return value

`rowind: list[int]` Indices of rows/sets/piecewise linear constraints/general constraints in the IIS.

`colind: list[int]` Indices of bounds (columns) in the IIS.

`contype: list[str]` Sense of rows in the IIS:

`L`	for less or equal row;
`G`	for greater or equal row.
`E`	for an equality row (for a non LP IIS);
`1`	for a SOS1 row;
`2`	for a SOS2 row;
`W`	for a piecewise linear constraint;
`X`	for a general constraint;
`I`	for an indicator row.

`bndtype: list[str]` Sense of bound in the IIS:

`U`	for upper bound;
`L`	for lower bound.
`F`	for fixed columns (for a non LP IIS);
`B`	for a binary column;
`I`	for an integer column;
`P`	for a partial integer columns;
`S`	for a semi-continuous column;
`R`	for a semi-continuous integer column.

`duals: list[float]` The dual multipliers associated with the rows.

`djs: list[float]` The dual multipliers (reduced costs) associated with the bounds.

`isolationrows: list[str]` The isolation status of the rows:

`-1`	if isolation information is not available for row (run iis isolations);
`0`	if row is not in isolation;
`1`	if row is in isolation.

`isolationcols: list[str]` The isolation status of the bounds:

`-1`	if isolation information is not available for column (run iis isolations);
`0`	if column is not in isolation;
`1`	if column is in isolation.

Further information

1. IISs are numbered from 1 to `NUMIIS` (p.**??** Xpress Optimizer Reference Manual). Index number 0 refers to the IIS approximation.
2. For the initial IIS approximation of linear problems (`iis` = 0) the number of rows and columns with a nonzero Lagrange multiplier (dual/reduced cost respectively) are returned. Please note that, in such cases, it might be necessary to call `problem.IISStatus` to retrieve the necessary size of the return arrays.
3. If there are special ordered sets, piecewise linear constraints or general constraints in the IIS, their number is included in the `rowind` array.
4. For non LP IISs, some column indices may appear more than once in the `colind` array, for example an integrality and a bound restriction for the same column.
5. Duals, reduced cost and isolation information are not available for nonlinear IIS problems, and for those the arrays are filled with zero values if they are provided.

Related topics

`problem.IISAll`, `problem.clearIIS`, `problem.firstIIS`, `problem.IISIsolations`, `problem.nextIIS`, `problem.IISStatus`, `problem.writeIIS` the Optimizer Reference Manual.

problem.loadPresolveBasis

Purpose

Loads a presolved basis from the user's areas.

Topic areas

Data Input, LP, Optimizer

Synopsis

```
problem.loadPresolveBasis(rowstat, colstat)
```

Arguments

`rowstat: list[int]` Integer array of length `ROWS` (p.**??** Xpress Optimizer Reference Manual) containing the basis status of the slack, surplus or artificial variable associated with each row. The status must be one of:

- `xpress.BasisStatus.NONBASIC_LOWER (0)` slack, surplus or artificial is non-basic at lower bound;
- `xpress.BasisStatus.BASIC (1)` slack, surplus or artificial is basic;
- `xpress.BasisStatus.NONBASIC_UPPER (2)` slack or surplus is non-basic at upper bound.

`colstat: list[int]` Integer array of length `COLS` (p.**??** Xpress Optimizer Reference Manual) containing the basis status of each of the columns in the matrix. The status must be one of:

- `xpress.BasisStatus.NONBASIC_LOWER (0)` variable is non-basic at lower bound or superbasic at zero if the variable has no lower bound;
- `xpress.BasisStatus.BASIC (1)` variable is basic;
- `xpress.BasisStatus.NONBASIC_UPPER (2)` variable is at upper bound;
- `xpress.BasisStatus.SUPERBASIC (3)` variable is super-basic.

Related topics

`problem.getBasis`, `problem.getPresolveBasis`, `problem.loadBasis`.

problem.loadSecureVecs

Purpose

Allows the user to mark rows and columns in order to prevent the presolve removing these rows and columns from the matrix.

Topic areas

Data Input, Presolve, Optimizer

Synopsis

```
problem.loadSecureVecs(rowind=None, colind=None)
```

Arguments

`rowind: list[constraint]` Integer array containing the rows to be marked. May be `None` if not required.

`colind: list[var]` Integer array containing the columns to be marked. May be `None` if not required.

Further information

1. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
2. Argument `rowind` may contain either `xpress.constraint` objects or integer indices.
3. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

the Optimizer Reference Manual.

problem.addRows

Purpose

Adds rows to the optimizer matrix.

Topic areas

Problem Creation, Optimizer

Synopsis

```
problem.addRows(rowtype, rhs, rng=None, start=None, colind=None, rowcoef=None)
```

Arguments

`rowtype: list[str]` Character array containing the row types:

- `L` indicates a $\leqslant$ row;
- `G` indicates a $\geqslant$ row;
- `E` indicates an = row.
- `R` indicates a range constraint;
- `N` indicates a nonbinding constraint.

`rhs: list[float]` Double array containing the right hand side elements.

`rng: list[float]` Double array containing the row range elements. This may be `None` if there are no ranged constraints. The values in the `rng` array will only be read for `R` type rows. The entries for other type rows will be ignored. The lower bound on a range row is the right hand side value minus the range value. The sign of the range value is ignored - the absolute value is used in all cases.

`start: list[int]` Integer array containing the offsets in the `colind` and `rowcoef` arrays of the start of the elements for each row. May be `None` if none of the rows contain nonzero coefficients.

`colind: list[var]` Integer array containing the (contiguous) column indices for the elements in each row. May be `None` if none of the rows contain nonzero coefficients.

`rowcoef: list[float]` Double array containing the (contiguous) element values. May be `None` if none of the rows contain nonzero coefficients.

Related controls

Integer

EXTRAELEMS	Number of extra matrix elements to be allowed for.
EXTRAROWS	Number of extra rows to be allowed for.

Double

MATRIXTOL	Tolerance on matrix elements.

Further information

1. Range rows are automatically converted to type `L`, with an upper bound in the slack. This must be taken into consideration, when retrieving row type, right-hand side values or range information for rows.
2. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
3. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.addCols`, `problem.addCuts`, `problem.addNames`, `problem.delRows`.

problem.delRows

Purpose

Delete rows from a matrix.

Topic areas

Problem Creation, Optimizer

Synopsis

`problem.delRows(rowind)`

Argument

`rowind: list[constraint]` An integer array containing the rows to delete.

Further information

1. After rows have been deleted from a problem, the numbers of the remaining rows are moved down so that the rows are always numbered from `0` to `ROWS-1` where `ROWS` (p.**??** Xpress Optimizer Reference Manual) is the problem attribute containing the number of non-deleted rows in the matrix.
2. If the problem has already been optimized, or an advanced basis has been loaded, and you delete a row for which the slack column is non-basic, the current basis will no longer be valid - the basis is "lost".

   If you go on to re-optimize the problem, a warning message is displayed (140) and the Optimizer automatically generates a corrected basis.
   You can avoid losing the basis by only deleting basic rows (see `problem.getBasis`), bringing a non-basic row into the basis first if necessary (see `problem.getPivots` and `problem.pivot`).
3. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
4. Argument `rowind` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.addRows`, `problem.delCols`, `problem.getBasis`, `problem.getPivots`, `problem.pivot`.

problem.addCols

Purpose

Adds columns to the optimizer matrix.

Topic areas

Problem Creation, Optimizer

Synopsis

```
problem.addCols(objcoef=None, start=None, rowind=None, rowcoef=None, lb=None,
       ub=None)
```

Arguments

`objcoef: list[float]` Double array containing the objective function coefficients of the new columns. As of version 43.00 this can be `None` to indicate that all newly created variables have an objective coefficient of 0 (zero).

`start: list[int]` Integer array containing the offsets in the `rowind` and `rowcoef` arrays of the start of the elements for each column. May be `None` if none of the columns contain nonzero coefficients.

`rowind: list[constraint]` Integer array containing the row indices for the elements in each column. May be `None` if none of the columns contain nonzero coefficients.

`rowcoef: list[float]` Double array containing the element values. May be `None` if none of the columns contain nonzero coefficients.

`lb: list[float]` Double array containing the lower bounds on the added columns. This can be `None` in which case all variables will have a lower bound of 0.

`ub: list[float]` Double array containing the upper bounds on the added columns. This can be `None` in which case all variables will have an infinite upper bound.

Related controls

Integer

EXTRACOLS	Number of extra columns to be allowed for.
EXTRAELEMS	Number of extra matrix elements to be allowed for.
EXTRAMIPENTS	Number of extra MIP entities to be allowed for.

Double

MATRIXTOL	Tolerance on matrix elements.

Further information

1. The double constants `xpress.infinity` and `-xpress.infinity` defined in the library header file can be used to represent plus and minus infinity respectively in the bound arrays.
2. If the columns are added to a MIP problem then they will be continuous variables. Use `problem.chgColType` to impose integrality conditions on such new columns.
3. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
4. Argument `rowind` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.addNames`, `problem.addRows`, `problem.delCols`, `problem.chgColType`.

problem.delCols

Purpose

Delete columns from a matrix.

Topic areas

Problem Creation, Optimizer

Synopsis

`problem.delCols(colind)`

Argument

`colind: list[var]` Integer array containing the columns to delete.

Further information

1. After columns have been deleted from a problem, the numbers of the remaining columns are moved down so that the columns are always numbered from `0` to `COLS-1` where `COLS` (p.**??** Xpress Optimizer Reference Manual) is the problem attribute containing the number of non-deleted columns in the matrix.
2. If the problem has already been optimized, or an advanced basis has been loaded, and you delete a basis column the current basis will no longer be valid - the basis is "lost".
   If you go on to re-optimize the problem, a warning message is displayed (140) and the Optimizer automatically generates a corrected basis.
   You can avoid losing the basis by only deleting non-basic columns (see `problem.getBasis`), taking a basic column out of the basis first if necessary (see `problem.getPivots` and `problem.pivot`).
3. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
4. If a deleted column is referenced by a formula then it will be replaced by the constant 0 in that formula. If the formula is a PWL formula then instead an error is raised because the input column for a PWL must be a variable and cannot be a constant zero.
5. If a deleted column is referenced by a set then it will be removed from the set. Empty sets will not be deleted by this function.
6. If a column is referenced by a piecewise linear constraint then an attempt to delete it will raise an error.
7. If a column is referenced by a general constraint then an attempt to delete it will raise an error.
8. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.addCols`, `problem.delRows`.

problem.chgColType

Purpose

Used to change the type of a specified set of columns in the matrix.

Topic areas

Problem Modification, MIP Entities, Optimizer

Synopsis

```
problem.chgColType(colind, coltype)
```

Arguments

`colind: list[var]` Integer array containing the indices of the columns.

`coltype: list[str]` Character array giving the new column types:

`C`	indicates a continuous column;
`B`	indicates a binary column;
`I`	indicates an integer column.
`S`	indicates a semi–continuous column. The semi–continuous lower bound will be set to `1.0`.
`R`	indicates a semi–integer column. The semi–integer lower bound will be set to `1.0`.
`P`	indicates a partial integer column. The partial integer limit will be set to `1.0`.

Further information

1. The column types can only be changed before the tree search is started.
2. Calling `problem.chgColType` to change any variable into a binary variable causes the bounds previously defined for the variable to be deleted and replaced by bounds of 0 and 1.
3. Calling `problem.chgColType` to change a continuous variable into an integer variable cause its lower bound to be rounded up to the nearest integer value and its upper bound to be rounded down to the nearest integer value.
4. After changing the type of a column to semi-continuous, semi-integer or partial integer, call `problem.chgGlbLimit` to set the limit value.
5. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
6. For performance reasons, avoid turning MIP entities into continuous columns by calling `problem.chgColType` many times in the presence of SOS constraints. If an application has this requirement, it is recommended to first collect all such columns beforehand and then change their type in a single call to this function.
7. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.addCols`, `problem.chgRowType`, `problem.delCols`, `problem.getColType`, `problem.chgGlbLimit`.

problem.loadBasis

Purpose

Loads a basis from the user's areas.

Topic areas

Data Input, LP, Optimizer

Synopsis

```
problem.loadBasis(rowstat, colstat)
```

Arguments

`rowstat: list[int]` Integer array of length `ORIGINALROWS` (p.**??** Xpress Optimizer Reference Manual) containing the basis status of the slack, surplus or artificial variable associated with each row. The status must be one of:

- `xpress.BasisStatus.NONBASIC_LOWER (0)` slack, surplus or artificial is non-basic at lower bound;
- `xpress.BasisStatus.BASIC (1)` slack, surplus or artificial is basic;
- `xpress.BasisStatus.NONBASIC_UPPER (2)` slack or surplus is non-basic at upper bound.
- `xpress.BasisStatus.SUPERBASIC (3)` slack or surplus is super-basic.

`colstat: list[int]` Integer array of length `ORIGINALCOLS` (p.**??** Xpress Optimizer Reference Manual) containing the basis status of each of the columns in the constraint matrix. The status must be one of:

- `xpress.BasisStatus.NONBASIC_LOWER (0)` variable is non-basic at lower bound or superbasic at zero if the variable has no lower bound;
- `xpress.BasisStatus.BASIC (1)` variable is basic;
- `xpress.BasisStatus.NONBASIC_UPPER (2)` variable is at upper bound;
- `xpress.BasisStatus.SUPERBASIC (3)` variable is super-basic.

Further information

1. The function can only be called if the problem is in original state. If the problem is currently in presolved state (indicated by attribute `PRESOLVESTATE` (p.**??** Xpress Optimizer Reference Manual)) then use function `problem.loadPresolveBasis` instead.
2. If the problem has been altered since saving an advanced basis, you may want to alter the basis as follows before loading it:
   - Make new variables non-basic at their lower bound (`colstat[icol]=0`), unless a variable has an infinite lower bound and a finite upper bound, in which case make the variable non-basic at its upper bound (`colstat[icol]=2`);
   - Make new constraints basic (`rowstat[jrow]=1`);
   - Try not to delete basic variables, or non-basic constraints.

Related topics

`problem.getBasis`, `problem.getPresolveBasis`, `problem.loadPresolveBasis`.

problem.postSolve

Purpose

Postsolve the current matrix when it is in a presolved state.

Topic areas

Solution Process, Presolve, Optimizer

Synopsis

```
problem.postSolve()
```

Further information

1. This function currently does not apply to general nonlinear problems, for general nonlinear please use `problem.nlpPostsolveProb`.
2. A problem is left in a presolved state whenever a LP or MIP optimization does not complete. In these cases `problem.postSolve` can be called to get the problem back into its original state.

Related topics

`problem.lpOptimize`, `problem.mipOptimize`

problem.delSets

Purpose

Delete sets from a problem.

Topic areas

Problem Creation, MIP Entities, Optimizer

Synopsis

```
problem.delSets(setind)
```

Argument

`setind: list[sos]` An integer array containing the sets to delete.

Further information

1. After sets have been deleted from a problem, the numbers of the remaining sets are moved down so that the sets are always numbered from `0` to `SETS-1` where `SETS` (p.**??** Xpress Optimizer Reference Manual) is the problem attribute containing the number of non-deleted sets in the problem.
2. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
3. Argument `setind` may contain either `xpress.sos` objects or integer indices.

Related topics

`problem.addSets`.

problem.addSets

Purpose

Allows sets to be added to the problem after passing it to the Optimizer using the input routines.

Topic areas

Problem Creation, MIP Entities, Optimizer

Synopsis

```
problem.addSets(settype, start, colind, refval)
```

Arguments

`settype: list[str]` Character array containing the set types:
- `1` indicates a SOS1;
- `2` indicates a SOS2;

`start: list[int]` Integer array containing the offsets in the `colind` and `refval` arrays of the start of the elements for each set.

`colind: list[var]` Integer array containing the (contiguous) column indices for the elements in each set.

`refval: list[float]` Double array containing the (contiguous) reference values. These define the order for SOS2 constraints and may be used in branching for both types. Within a set, no two elements can have the same reference values and reference values must differ by at least `SOSREFTOL` (p.**??** Xpress Optimizer Reference Manual) (relative difference).

Further information

1. Reference values must be sufficiently distinct (see `SOSREFTOL` (p.**??** Xpress Optimizer Reference Manual)).
2. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
3. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.delSets`.

problem.strongBranch

Purpose

Performs strong branching iterations on all specified bound changes. For each candidate bound change, `problem.strongBranch` performs dual simplex iterations starting from the current optimal solution of the base LP, and returns both the status and objective value reached after these iterations.

Topic areas

Branching, Optimizer

Synopsis

```
objval, status = problem.strongBranch(colind, bndtype, bndval, iterlim)
```

Arguments

`colind: list[var]` Integer array containing the indices of the columns on which the bounds will change.

`bndtype: list[str]` Character array indicating the type of bound to change:

- `U` indicates change the upper bound;
- `L` indicates change the lower bound;
- `B` indicates change both bounds, i.e. fix the column.

`bndval: list[float]` Double array giving the new bound values.

`iterlim: int` Maximum number of LP iterations to perform for each bound change.

Return value

`objval: list[float]` Objective value of each LP after performing the strong branching iterations.

`status: list[int]` Status of each LP after performing the strong branching iterations, as detailed for the `LPSTATUS` (p.**??** Xpress Optimizer Reference Manual) attribute.

Further information

1. This function currently does not apply to general nonlinear problems.
2. Prior to calling problem.strongBranch, the current LP problem must have been solved to optimality and an optimal basis must be available.
3. Argument `colind` may contain either `xpress.var` objects or integer indices.

problem.strongBranchCB

Purpose

Performs strong branching iterations on all specified bound changes. For each candidate bound change, `problem.strongBranchCB` performs dual simplex iterations starting from the current optimal solution of the base LP, and returns both the status and objective value reached after these iterations.

Topic areas

Branching, Optimizer

Synopsis

```
objval, status = problem.strongBranchCB(colind, bndtype, bndval, iterlim, callback,
       data)
strongbranchsolve(cbprob, cbdata, bndidx)
```

Arguments

`colind: list[var]` Integer array containing the indices of the columns on which the bounds will change.

`bndtype: list[str]` Character array indicating the type of bound to change:

`U`	indicates change the upper bound;
`L`	indicates change the lower bound;
`B`	indicates change both bounds, i.e. fix the column.

`bndval: list[float]` Double array giving the new bound values.

`iterlim: int` Maximum number of LP iterations to perform for each bound change.

`callback` Function to be called after each strong branch has been reoptimized. This function returns an integer. Use `0` to indicate that everything went fine. Use a return value different from `0` to signal an error. This will terminate the function.

`data: Any` User context to be provided for `callback`.

`cbprob: problem` The problem passed to the callback function, `callback`.

`cbdata: Any` The user-defined data passed as `data`.

`bndidx: int` The index of the bound for which `callback` is called.

Return value

`objval: list[float]` Array of objective values of each LP after performing the strong branching iterations.

`status: list[int]` Array of statuses of each LP after performing the strong branching iterations, as detailed for the `LPSTATUS` (p.**??** Xpress Optimizer Reference Manual) attribute.

Further information

1. This function currently does not apply to general nonlinear problems.
2. Prior to calling `problem.strongBranchCB`, the current LP problem must have been solved to optimality and an optimal basis must be available.
3. `problem.strongBranchCB` is an extension to `problem.strongBranch`. If identical input arguments are provided both will return identical results, the difference being that for the case of `problem.strongBranchCB` the `callback` function is called at the end of each LP reoptimization.
4. For each branch optimized, the LP can be interrogated: the LP status of the branch is available through checking `LPSTATUS` (p.**??** Xpress Optimizer Reference Manual), and the objective function value is available through `LPOBJVAL` (p.**??** Xpress Optimizer Reference Manual). It is possible to access the full current LP solution by using `problem.getCallbackSolution`.
5. The return value of callback function `callback` is currently ignored.
6. Argument `colind` may contain either `xpress.var` objects or integer indices.

problem.estimateRowDualRanges

Purpose

Performs a dual side range sensitivity analysis, i.e. calculates estimates for the possible ranges for dual values.

Topic areas

Sensitivity Analysis, LP, Optimizer

Synopsis

```
mindual, maxdual = problem.estimateRowDualRanges(rowind, iterlim)
```

Arguments

`rowind: list[constraint]` Row indices to analyze.

`iterlim: int` Effort limit expressed as simplex iterations per row.

Return value

`mindual: list[float]` Estimated lower bounds on the possible dual ranges.

`maxdual: list[float]` Estimated upper bounds on the possible dual ranges.

Further information

1. This function applies to linear and convex quadratic (including second order conic) problems only.
2. This function may provide better results for individual row dual ranges when called for a larger number of rows, as it collect infomation for all rows while working on individual rows.
3. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
4. Argument `rowind` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.lpOptimize`, `problem.strongBranch`

problem.setMessageStatus

Purpose

Manages suppression of messages.

Topic areas

Misc, Optimizer

Synopsis

```
problem.setMessageStatus(msgcode, status)
```

Arguments

`msgcode: int` The id number of the message. Refer to the Optimizer Reference Manual for a list of possible message numbers.

`status: int` Non-zero if the message is not suppressed; `0` otherwise.

Related topics

`problem.getMessageStatus`.

problem.getMessageStatus

Purpose

Retrieves the current suppression status of a message.

Topic areas

Misc, Optimizer

Synopsis

```
status = problem.getMessageStatus(msgcode)
```

Argument

`msgcode: int` The id number of the message. Refer to the Optimizer Reference Manual for a list of possible message numbers.

Return value

`status: int` Non-zero if the message is not suppressed; `0` otherwise.

Related topics

`problem.setMessageStatus`.

problem.chgObjSense

Purpose

Changes the problem's objective function sense to minimize or maximize.

Topic areas

Problem Modification, Optimizer

Synopsis

```
problem.chgObjSense(objsense)
```

Argument

`objsense: ObjSense` `xpress.ObjSense.MINIMIZE` to change into a minimization, or `xpress.ObjSense.MAXIMIZE` to change into a maximization problem.

Related topics

`problem.lpOptimize`, `problem.mipOptimize`.

problem.chgGlbLimit

Purpose

Used to change semi-continuous or semi-integer lower bounds, or upper limits on partial integers.

Topic areas

Problem Modification, MIP Entities, Optimizer

Synopsis

```
problem.chgGlbLimit(colind, limit)
```

Arguments

`colind: list[var]` Integer array containing the indices of the semi-continuous, semi-integer or partial integer columns that should have their limits changed.

`limit: list[float]` Double array giving the new limit values.

Further information

1. The new limits are not allowed to be negative.
2. Partial integer limits can be at most 2^{28}.
3. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
4. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.chgColType`, `problem.getMIPEntities`.

problem.saveAs

Purpose

Saves the current data structures, i.e. matrices, control settings and problem attribute settings to file and terminates the run so that optimization can be resumed later.

Topic areas

File IO, Save Restore, Optimizer

Synopsis

```
problem.saveAs(filename="")
```

Argument

`filename: str` The name of the file (without .svf) to save to.

Further information

1. The data structures are written to the file *problem_name*`.svf`. Optimization may recommence from the same point when the data structures are restored by a call to `problem.restore`. Note that the `.svf` files created are particular to the release of the Optimizer used to create them. They can only be read using the same release Optimizer as used to create them.
2. The function `problem.saveAs` is equivalent to `problem.save` with the exception of allowing to adjust the name of the file created. The name of the file must not be greater than `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual).

Related topics

`problem.restore`.

problem.restore

Purpose

Restores the Optimizer's data structures from a file created by `problem.saveAs`. Optimization may then recommence from the point at which the file was created.

Topic areas

File IO, Save Restore, Optimizer

Synopsis

```
problem.restore(probname="", flags="")
```

Arguments

`probname: str` A string of up to `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual) characters containing the problem name.

`flags: str` Additional flags

`h`	Do not restore hardware information from the file;
`v`	use the provided filename verbatim, without appending the `.svf` extension.

Further information

This routine restores the data structures from the file *problem_name*`.svf` that was created by a previous execution of `problem.saveAs`. Note that `.svf` files are particular to the release of the Optimizer used to create them. They can only be read using the same major release of the Optimizer that was used to create them.

Related topics

`problem.saveAs`.

problem.pivot

Purpose

Performs a simplex pivot by bringing variable `enter` into the basis and removing `leave`.

Topic areas

Linear Algebra, Simplex, Optimizer

Synopsis

```
problem.pivot(enter, leave)
```

Arguments

`enter: var | constraint` Index of row or column to enter basis.

`leave: var | constraint` Index of row or column to leave basis.

Error values

`425` `enter` is invalid (out of range or already basic).

`426` `leave` is invalid (out of range or not eligible, e.g. nonbasic, zero pivot, etc.).

Related controls

Double

`PIVOTTOL` Pivot tolerance.

`RELPIVOTTOL` Relative pivot tolerance.

Further information

1. Row indices are in the range `0` to `ROWS` (p.**??** Xpress Optimizer Reference Manual)`-1`, whilst columns are in the range `ROWS+SPAREROWS` (p.**??** Xpress Optimizer Reference Manual) to `ROWS+SPAREROWS+COLS` (p.**??** Xpress Optimizer Reference Manual)`-1`.
2. Arguments `enter, leave` may contain either `xpress.var` or `xpress.constraint` objects or integer indices.

Related topics

`problem.getPivotOrder`, `problem.getPivots`.

problem.loadLPSol

Purpose

Loads an LP solution for the problem into the Optimizer.

Topic areas

Solution, LP, Data Input, Optimizer

Synopsis

```
status = problem.loadLPSol(x=None, slack=None, duals=None, djs=None)
```

Arguments

`x: list[float]` Optional: Double array of length `COLS` (p.**??** Xpress Optimizer Reference Manual) (for the original problem and not the presolve problem) containing the values of the variables. May be `None`, see further information.

`slack: list[float]` Optional: double array of length `ROWS` (p.**??** Xpress Optimizer Reference Manual) containing the values of slack variables. May be `None`, see further information.

`duals: list[float]` Optional: double array of length `ROWS` (p.**??** Xpress Optimizer Reference Manual) containing the values of dual variables. May be `None`, see further information.

`djs: list[float]` Optional: double array of length `COLS` (p.**??** Xpress Optimizer Reference Manual) containing the values of reduced costs. May be `None`, see further information.

Return value

`status: int` The status. The status is one of:

`0` Solution is loaded.

`1` Solution is not loaded because the problem is in presolved status.

Further information

1. If neither variable values nor dual values are provided, the current solution in memory will be unloaded.
2. When variables `x` is `None`, the variables will be set to their bounds.
3. When slack variables `slack` is `None`, it will be computed from variables `x`. If slacks are provided, variables cannot be omitted.
4. When dual variables `duals` is `None`, both dual variables and reduced costs will be set to zero.
5. When reduced costs `djs` is `None`, it will be computed from dual variables `duals`. If reduced costs are provided, dual variables cannot be omitted.

Related topics

`problem.getSolution`, `problem.crossoverLpSol`.

problem.repairWeightedInfeas

Purpose

By relaxing a set of selected constraints and bounds of an infeasible problem, it attempts to identify a 'solution' that violates the selected set of constraints and bounds minimally, while satisfying all other constraints and bounds. Among such solution candidates, it selects one that is optimal regarding the original objective function.

Topic areas

Infeasibility, Optimizer

Synopsis

```
status = problem.repairWeightedInfeas(lepref, gepref, lbpref, ubpref, phase2, delta,
       flags)
```

Arguments

`lepref: list[float]` Array of size `ROWS` containing the preferences for relaxing the less or equal side of row.

`gepref: list[float]` Array of size `ROWS` containing the preferences for relaxing the greater or equal side of a row.

`lbpref: list[float]` Array of size `COLS` containing the preferences for relaxing lower bounds.

`ubpref: list[float]` Array of size `COLS` containing preferences for relaxing upper bounds.

`phase2: str` Controls the second phase of optimization:

- `o` use the objective sense of the original problem (default);
- `x` maximize the relaxed problem using the original objective;
- `f` skip optimization regarding the original objective;
- `n` minimize the relaxed problem using the original objective;
- `i` if the relaxation is infeasible, generate an irreducible infeasible subset for the analysis of the problem;
- `a` if the relaxation is infeasible, generate all irreducible infeasible subsets for the analysis of the problem.

`delta: float` The relaxation multiplier in the second phase -1.

`flags: str` Specifies flags to be passed to problem.optimize.

Return value

`status: int` The status after the relaxation:

- `0` relaxed optimum found;
- `1` relaxed problem is infeasible;
- `2` relaxed problem is unbounded;
- `3` solution of the relaxed problem regarding the original objective is nonoptimal;
- `4` error (when return code is nonzero);
- `5` numerical instability;
- `6` analysis of an infeasible relaxation was performed, but the relaxation is feasible.

Further information

1. This function applies to all problem types.
2. A row or bound is relaxed by introducing a new nonnegative variable that will contain the infeasibility of the row or bound. Suppose for example that row $a^T x = b$ is relaxed from below. Then a new variable ('infeasibility breaker') $s \geqslant 0$ is added to the row, which becomes $a^T x + s = b$. Observe that $a^T x$ may now take smaller values than b. To minimize such violations, the weighted sum of these new variables is minimized.
3. A preference of 0 results in the row or bound not being relaxed. The higher the preference, the more willing the modeller is to relax a given row or bound.
4. The weight of each infeasibility breaker in the objective minimizing the violations is 1/p, where `p` is the preference associated with the infeasibility breaker. Thus the higher the preference is, the lower a penalty is associated with the infeasibility breaker while minimizing the violations.
5. If a feasible solution is identified for the relaxed problem, with a sum of violations `p`, then the sum of violations is restricted to be no greater than `(1+delta)p`, and the problem is optimized with respect to the original objective function. A nonzero delta increases the freedom of the original problem. The weighted sum of violations will be available in `PENALTYVALUE` (p.**??** Xpress Optimizer Reference Manual).
6. Note that on some problems, slight modifications of delta may affect the value of the original objective drastically.
7. The default value for delta in the console is 0.001.
8. Note that because of their special associated modeling properties, binary and semi-continuous variables are not relaxed.
9. If `pflags` is set such that each penalty is the reciprocal of the preference, the following rules are applied while introducing the auxiliary variables:

Pref. array	Affects	Relaxation	Cost if pref.>0	Cost if pref.<0
`lepref`	`= rows`	$a^T x$ `- aux_var = b`	`1/lrp*aux_var`	`1/lrp*aux_var`^{2}
`lepref`	`<= rows`	$a^T x$ `- aux_var <= b`	`1/lrp*aux_var`	`1/lrp*aux_var`^{2}
`gepref`	`= rows`	$a^T x$ `+ aux_var = b`	`1/grp*aux_var`	`1/grp*aux_var`^{2}
`gepref`	`>= rows`	$a^T x$ `+ aux_var >= b`	`1/grp*aux_var`	`1/grp*aux_var`^{2}
`ubpref`	`upper bounds`	x_i `- aux_var <= u`	`1/ubp*aux_var`	`1/ubp*aux_var`^{2}
`lbpref`	`lower bounds`	x_i `+ aux_var >= l`	`1/lbp*aux_var`	`1/lbp*aux_var`^{2}

10. If an irreducible infeasible set (IIS) has been identified, the generated IIS(s) are accessible through the IIS retrieval functions, see `NUMIIS` (p.**??** Xpress Optimizer Reference Manual) and `problem.getIISData`.

Related topics

`problem.repairInfeas`, `problem.repairWeightedInfeasBounds`, `problem.optimize`, the Optimizer Reference Manual.

problem.repairWeightedInfeasBounds

Purpose

An extended version of `problem.repairWeightedInfeas` that allows for bounding the level of relaxation allowed.

Topic areas

Infeasibility, Optimizer

Synopsis

```
status = problem.repairWeightedInfeasBounds(lepref, gepref, lbpref, ubpref, lerelax,
       gerelax, lbrelax, ubrelax, phase2, delta, flags)
```

Arguments

`lepref: list[float]` Array of size `ROWS` containing the preferences for relaxing the less or equal side of row. For the console use `-lrp` value.

`gepref: list[float]` Array of size `ROWS` containing the preferences for relaxing the greater or equal side of a row. For the console use `-grp` value.

`lbpref: list[float]` Array of size `COLS` containing the preferences for relaxing lower bounds. For the console use `-lbp` value.

`ubpref: list[float]` Array of size `COLS` containing preferences for relaxing upper bounds. For the console use `-ubp` value.

`lerelax: list[float]` Array of size `ROWS` containing the upper bounds on the amount the less or equal side of a row can be relaxed. For the console use `-lrb` value.

`gerelax: list[float]` Array of size `ROWS` containing the upper bounds on the amount the greater or equal side of a row can be relaxed. For the console use `-grb` value.

`lbrelax: list[float]` Array of size `COLS` containing the upper bounds on the amount the lower bounds can be relaxed. For the console use `-lbb` value.

`ubrelax: list[float]` Array of size `COLS` containing the upper bounds on the amount the upper bounds can be relaxed. For the console use `-ubb` value.

`phase2: str` Controls the second phase of optimization:

- `o` use the objective sense of the original problem (default);
- `x` maximize the relaxed problem using the original objective;
- `f` skip optimization regarding the original objective;
- `n` minimize the relaxed problem using the original objective;
- `i` if the relaxation is infeasible, generate an irreducible infeasible subset for the analysis of the problem;
- `a` if the relaxation is infeasible, generate all irreducible infeasible subsets for the analysis of the problem.

`delta: float` The relaxation multiplier in the second phase -1.

`flags: str` Specifies flags to be passed to problem.optimize.

Return value

`status: int` The status after the relaxation:

- `0` relaxed optimum found;
- `1` relaxed problem is infeasible;
- `2` relaxed problem is unbounded;
- `3` solution of the relaxed problem regarding the original objective is nonoptimal;
- `4` error (when return code is nonzero);
- `5` numerical instability;
- `6` analysis of an infeasible relaxation was performed, but the relaxation is feasible.

Further information

1. This function applies to all problem types.
2. A row or bound is relaxed by introducing a new nonnegative variable that will contain the infeasibility of the row or bound. Suppose for example that row $a^T x = b$ is relaxed from below. Then a new variable ('infeasibility breaker') $s \geqslant 0$ is added to the row, which becomes $a^T x + s = b$. Observe that $a^T x$ may now take smaller values than b. To minimize such violations, the weighted sum of these new variables is minimized.
3. A preference of 0 results in the row or bound not being relaxed. The higher the preference, the more willing the modeller is to relax a given row or bound.
4. A negative preference indicates that a quadratic penalty cost should be applied. This can specified on a per constraint side or bound basis.
5. If a feasible solution is identified for the relaxed problem, with a sum of violations `p`, then the sum of violations is restricted to be no greater than `(1+delta)p`, and the problem is optimized with respect to the original objective function. A nonzero delta increases the freedom of the original problem. The weighted sum of violations will be available in `PENALTYVALUE` (p.**??** Xpress Optimizer Reference Manual).
6. Note that on some problems, slight modifications of delta may affect the value of the original objective drastically.
7. The default value for delta in the console is 0.001.
8. Note that because of their special associated modeling properties, binary and semi-continuous variables are not relaxed.
9. Given any row `j` with preferences `lrp=lepref[j]` and `grp=gepref[j]`, or variable `i` with bound preferences `ubp=ubpref[i]` and `lbp=lbpref[i]`, the following rules are applied while introducing the auxiliary variables:

Preference	Affects	Relaxation	Cost if pref.>0	Cost if pref.<0
`lrp`	`= rows`	$a^T x$ `- aux_var = b`	`1/lrp*aux_var`	`1/lrp*aux_var`^{2}
`lrp`	`<= rows`	$a^T x$ `- aux_var <= b`	`1/lrp*aux_var`	`1/lrp*aux_var`^{2}
`grp`	`= rows`	$a^T x$ `+ aux_var = b`	`1/grp*aux_var`	`1/grp*aux_var`^{2}
`grp`	`>= rows`	$a^T x$ `+ aux_var >= b`	`1/grp*aux_var`	`1/grp*aux_var`^{2}
`ubp`	`upper bounds`	x_i `- aux_var <= u`	`1/ubp*aux_var`	`1/ubp*aux_var`^{2}
`lbp`	`lower bounds`	x_i `+ aux_var >= l`	`1/lbp*aux_var`	`1/lbp*aux_var`^{2}

10. Only positive bounds are applied; a zero or negative bound is ignored and the amount of relaxation allowed for the corresponding row or bound is not limited. The effect of a zero bound on a row or bound would be equivalent with not relaxing it, and can be achieved by setting its preference array value to zero instead, or not including it in the preference arrays.
11. If an irreducible infeasible set (IIS) has been identified, the generated IIS(s) are accessible through the IIS retrieval functions, see `NUMIIS` (p.**??** Xpress Optimizer Reference Manual) and `problem.getIISData`.

Related topics

`problem.repairInfeas`, `problem.optimize`, the Optimizer Reference Manual.

problem.repairInfeas

Purpose

Provides a simplified interface for `problem.repairWeightedInfeas`.

Topic areas

Infeasibility, Optimizer

Synopsis

```
status = problem.repairInfeas(penalty, phase2, flags, lepref, gepref, lbpref,
       ubpref, delta)
```

Arguments

`penalty: str` The type of penalties created from the preferences:
- `c` each penalty is the reciprocal of the preference (default);
- `s` the penalties are placed in the scaled problem.

`phase2: str` Controls the second phase of optimization:
- `o` use the objective sense of the original problem (default);
- `x` maximize the relaxed problem using the original objective;
- `f` skip optimization regarding the original objective;
- `n` minimize the relaxed problem using the original objective;
- `i` if the relaxation is infeasible, generate an irreducible infeasible subset for the analysis of the problem;
- `a` if the relaxation is infeasible, generate all irreducible infeasible subsets for the analysis of the problem.

`flags: str` Specifies flags to be passed to problem.optimize.

`lepref: float` Preference for relaxing the less or equal side of row.

`gepref: float` Preference for relaxing the greater or equal side of a row.

`lbpref: float` Preferences for relaxing lower bounds.

`ubpref: float` Preferences for relaxing upper bounds.

`delta: float` The relaxation multiplier in the second phase -1. For console use `-d` value. A positive value means a relative relaxation by multiplying the first phase objective with (`delta-1`), while a negative value means an absolute relaxation, by adding `abs(delta)` to the first phase objective.

Return value

`status: int` The status after the relaxation:
- `0` relaxed optimum found;
- `1` relaxed problem is infeasible;
- `2` relaxed problem is unbounded;
- `3` solution of the relaxed problem regarding the original objective is nonoptimal;
- `4` error (when return code is nonzero);
- `5` numerical instability;
- `6` analysis of an infeasible relaxation was performed, but the relaxation is feasible.

Related controls

Integer

`DEFAULTALG` Forced algorithm selection (default for repairinfeas is primal).

Further information

1. This function applies to all problem types.
2. A row or bound is relaxed by introducing a new nonnegative variable that will contain the infeasibility of the row or bound. Suppose for example that row $a^T x = b$ is relaxed from below. Then a new variable (infeasibility breaker) $s \geqslant 0$ is added to the row, which becomes $a^T x + s = b$. Observe that $a^T x$ may now take smaller values than b. To minimize such violations, the weighted sum of these new variables is minimized.
3. A preference of 0 results in the row or bound not being relaxed.
4. A negative preference indicates that a quadratic penalty cost should be applied. This can specified on a per constraint side or bound basis.
5. Note that the set of preferences are scaling independent.
6. If a feasible solution is identified for the relaxed problem, with a sum of violations p, then the sum of violations is restricted to be no greater than `(1+delta)`p, and the problem is optimized with respect to the original objective function. A nonzero delta increases the freedom of the original problem. The weighted sum of violations will be available in `PENALTYVALUE` (p.**??** Xpress Optimizer Reference Manual).
7. Note that on some problems, slight modifications of delta may affect the value of the original objective drastically.
8. The default value for delta in the console is 0.001.
9. Note that because of their special associated modeling properties, binary and semi-continuous variables are not relaxed.
10. The default algorithm for the first phase is the simplex algorithm, since the primal problem can be efficiently warm started in case of the extended problem. These may be altered by setting the value of control `DEFAULTALG` (p.**??** Xpress Optimizer Reference Manual).
11. If `penalty` is set such that each penalty is the reciprocal of the preference, the following rules are applied while introducing the auxiliary variables:

Preference	Affects	Relaxation	Cost if pref.>0	Cost if pref.<0
`lepref`	`= rows`	$a^T x$ `- aux_var = b`	`1/lepref*aux_var`	`1/lepref*aux_var`^{2}
`lepref`	`<= rows`	$a^T x$ `- aux_var <= b`	`1/lepref*aux_var`	`1/lepref*aux_var`^{2}
`gepref`	`= rows`	$a^T x$ `+ aux_var = b`	`1/gepref*aux_var`	`1/gepref*aux_var`^{2}
`gepref`	`>= rows`	$a^T x$ `+ aux_var >= b`	`1/gepref*aux_var`	`1/gepref*aux_var`^{2}
`ubpref`	`upper bounds`	x_i `- aux_var <= u`	`1/ubpref*aux_var`	`1/ubpref*aux_var`^{2}
`lbpref`	`lower bounds`	x_i `+ aux_var >= l`	`1/lbpref*aux_var`	`1/lbpref*aux_var`^{2}

12. If an irreducible infeasible set (IIS) has been identified, the generated IIS(s) are accessible through the IIS retrieval functions, see `NUMIIS` (p.**??** Xpress Optimizer Reference Manual) and `problem.getIISData`.

Related topics

`problem.repairWeightedInfeas`, `problem.optimize`, the Optimizer Reference Manual.

problem.basisStability

Purpose

Calculates various measures for the stability of the current basis, including the basis condition number.

Topic areas

Linear Algebra, LP, Optimizer

Synopsis

```
value = problem.basisStability(type, norm, scaled)
```

Arguments

`type: int`	`0`	Condition number of the basis.
	`1`	Stability measure for the solution relative to the current basis.
	`2`	Stability measure for the duals relative to the current basis.
	`3`	Stability measure for the right hand side relative to the current basis.
	`4`	Stability measure for the basic part of the objective relative to the current basis.
`norm: int`	`0`	Use the infinity norm.
	`1`	Use the 1 norm.
	`2`	Use the Euclidian norm for vectors, and the Frobenius norm for matrices.
`scaled: int`	If the stability values are to be calculated in the scaled, or the unscaled matrix.	

Return value

`value: float` The calculated value.

Further information

1. The condition number (`type` = `0`) of an invertible matrix is the norm of the matrix multiplied with the norm of its inverse. This number is an indication of how accurate the solution can be calculated and how sensitive it is to small changes in the data. The larger the condition number is, the less accurate the solution is likely to become.
2. The stability measures (`type` = `1...4`) are using the original matrix and the basis to recalculate the various vectors related to the solution and the duals. The returned stability measure is the norm of the difference of the recalculated vector to the original one.

problem.objSA

Purpose

Returns upper and lower sensitivity ranges for specified objective function coefficients. If the objective coefficients are varied within these ranges the current basis remains optimal and the reduced costs remain valid.

Topic areas

Sensitivity Analysis, LP, Optimizer

Synopsis

```
lower, upper = problem.objSA(colind)
```

Argument

`colind: list[var]` Integer array containing the indices of the columns whose objective function coefficients sensitivity ranges are required.

Return value

`lower: list[float]` Double array containing the objective function lower range values.

`upper: list[float]` Double array containing the objective function upper range values.

Further information

1. This function does not currently apply to general nonlinear problems.
2. `problem.objSA` can only be called when an optimal solution to the current LP has been found. It cannot be used when the problem is MIP presolved.
3. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.rhsSA`, `problem.bndSA`.

problem.bndSA

Purpose

Returns upper and lower sensitivity ranges for specified variables' lower and upper bounds. If the bounds are varied within these ranges the current basis remains optimal and feasible.

Topic areas

Sensitivity Analysis, LP, Optimizer

Synopsis

```
lblower, lbupper, ublower, ubupper = problem.bndSA(colind)
```

Argument

`colind: list[var]` Integer array containing the indices of the columns whose bounds' ranges are required.

Return value

`lblower: list[float]` Double array containing the variable lower bound lower range values.

`lbupper: list[float]` Double array containing the variable lower bound upper range values.

`ublower: list[float]` Double array containing the variable upper bound lower range values.

`ubupper: list[float]` Double array containing the variable upper bound upper range values.

Further information

1. `problem.bndSA` can only be called when an optimal solution to the current LP has been found. It cannot be used when the problem is MIP presolved.
2. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.rhsSA`, `problem.objSA`.

problem.rhsSA

Purpose

Returns upper and lower sensitivity ranges for specified right hand side (RHS) function coefficients. If the RHS coefficients are varied within these ranges the current basis remains optimal and the reduced costs remain valid.

Topic areas

Sensitivity Analysis, LP, Optimizer

Synopsis

```
lower, upper = problem.rhsSA(rowind)
```

Argument

`rowind: list[constraint]` Integer array containing the indices of the rows whose RHS coefficients sensitivity ranges are required.

Return value

`lower: list[float]` Double array containing the RHS lower range values.

`upper: list[float]` Double array containing the RHS upper range values.

Further information

1. This function currently does not apply to general nonlinear problems.
2. `problem.rhsSA` can only be called when an optimal solution to the current LP has been found. It cannot be used when the problem is MIP presolved.
3. Argument `rowind` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.objSA`, `problem.bndSA`.

problem.addQMatrix

Purpose

Adds a new quadratic matrix into a row defined by triplets.

Topic areas

Problem Creation, Quadratic, Optimizer

Synopsis

```
problem.addQMatrix(row, rowqcol1, rowqcol2, rowqcoef)
```

Arguments

`row: constraint` Index of the row where the quadratic matrix is to be added.

`rowqcol1: list[var]` First index in the triplets.

`rowqcol2: list[var]` Second index in the triplets.

`rowqcoef: list[float]` Coefficients in the triplets.

Further information

1. The triplets should be filled to define the upper-triangular part of the quadratic expression. This means that to add `[x^2 + 6 xy]` the `rowqcoef` array shall contain the coefficients 1 and 3, respectively.
2. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
3. Argument `row` may contain either `xpress.constraint` objects or integer indices.
4. Arguments `rowqcol1, rowqcol2` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.loadQCQP`, `problem.getQRowCoeff`, `problem.chgQRowCoeff`, `problem.getQRowQMatrix`, `problem.getQRowQMatrixTriplets`, `problem.getQRows`, `problem.chgQObj`, `problem.chgMQObj`, `problem.getQObj`.

problem.delQMatrix

Purpose

Deletes the quadratic part of a row or of the objective function.

Topic areas

Problem Creation, Quadratic, Optimizer

Synopsis

```
problem.delQMatrix(row)
```

Argument

`row: constraint` Index of row from which the quadratic part is to be deleted.

Further information

1. If a row index of `-1` is used, the function deletes the quadratic coefficients from the objective function.
2. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
3. Argument `row` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.addRows`, `problem.delCols`, `problem.delRows`.

problem.loadQCQP

Purpose

Used to load a quadratic problem with quadratic side constraints into the Optimizer data structure. Such a problem may have quadratic terms in its objective function as well as in its constraints.

Topic areas

Problem Input, Quadratic, Optimizer

Synopsis

```
problem.loadQCQP(probname="", rowtype=None, rhs=None, rng=None, objcoef=None,
       start=None, collen=None, rowind=None, rowcoef=None, lb=None, ub=None,
       objqcol1=None, objqcol2=None, objqcoef=None, qrowind=None, nrowqcoefs=None,
       rowqcol1=None, rowqcol2=None, rowqcoef=None)
```

Arguments

`probname: str` A string of up to `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual) characters containing a name for the problem. If `probname` is `None`, the problem name will be an empty string.

`rowtype: list[str]` Character array containing the row types:

`L`	indicates a <= constraint (use this one for quadratic constraints as well);
`E`	indicates an = constraint;
`G`	indicates a >= constraint;
`R`	indicates a range constraint;
`N`	indicates a nonbinding constraint.

May be `None` if the problem contains no rows.

`rhs: list[float]` Double array containing the right hand side coefficients of the rows. The right hand side value for a range row gives the upper bound on the row. May be `None` if the problem contains no rows.

`rng: list[float]` Double array containing the range values for range rows. Values for all other rows will be ignored. May be `None` if there are no ranged constraints. The lower bound on a range row is the right hand side value minus the range value. The sign of the range value is ignored - the absolute value is used in all cases.

`objcoef: list[float]` Double array containing the objective function coefficients. This can be `None` to set all objective coefficients to 0 (zero).

`start: list[int]` Integer array containing the offsets in the `rowind` and `rowcoef` arrays of the start of the elements for each column. This array contains one entry for each column, plus one more if `collen` is `None`. If `collen` is `None`, the last entry of `start` contains the position in the `rowind` and `rowcoef` arrays at which an extra column would start, if it were present. This value is also the length of the `rowind` and `rowcoef` arrays. May be `None` if the problem contains no coefficients.

`collen: list[int]` Integer array containing the number of nonzero elements in each column. May be `None` if not required. This array is not required if the non-zero coefficients in the `rowind` and `rowcoef` arrays are contiguous, and the `start` array has an additional entry as described above.

`rowind: list[constraint]` Integer array containing the row indices for the nonzero elements in each column. May be `None` if the problem contains no coefficients. If the indices are input contiguously, with the columns in ascending order, the length of the `rowind` array is found by adding the `start` entry for the last column to the `collen` entry for the last column, or if `collen` is `None`, by the additional entry at the end of `start`.

`rowcoef: list[float]` Double array containing the nonzero element values; length as for `rowind`. May be `None` if the problem contains no coefficients.

`lb: list[float]` Double array containing the lower bounds on the columns. Use `–xpress.infinity` to represent a lower bound of minus infinity. If this is `None` then all lower bounds are 0 (zero).

`ub: list[float]` Double array containing the upper bounds on the columns. Use `xpress.infinity` to represent an upper bound of plus infinity. If this is `None` then all upper bounds are infinite.

`objqcol1: list[var]` Integer array containing the column index of the first variable in each quadratic term. May be `None` if all objective coefficients are zero.

`objqcol2: list[var]` Integer array containing the column index of the second variable in each quadratic term. May be `None` if all objective coefficients are zero.

`objqcoef: list[float]` Double array containing the quadratic coefficients. May be `None` if all objective coefficients are zero.

`qrowind: list[constraint]` Integer array containing the indices of rows with quadratic matrices in them. Note that the rows are expected to be defined in `rowtype` as type `L`. May be `None` if the problem contains no quadratic constraints.

`nrowqcoefs: list[int]` Integer array containing the number of nonzeros in each quadratic constraint matrix. May be `None` if the problem contains no quadratic constraints.

`rowqcol1: list[var]` Integer array of size `nqcelem`, where `nqcelem` equals the sum of the elements in `nrowqcoefs` (i.e. the total number of quadratic matrix elements in all the constraints). It contains the first column indices of the quadratic matrices. Indices for the first matrix are listed from `0` to `nrowqcoefs[0]-1`, for the second matrix from `nrowqcoefs[0]` to `nrowqcoefs[0]+nrowqcoefs[1]-1`, etc. May be `None` if the problem contains no quadratic constraints.

`rowqcol2: list[var]` Integer array of size `nqcelem`, containing the second index for the quadratic constraint matrices. May be `None` if the problem contains no quadratic constraints.

`rowqcoef: list[float]` Integer array of size `nqcelem`, containing the coefficients for the quadratic constraint matrices. May be `None` if the problem contains no quadratic constraints.

Related controls

Integer

EXTRACOLS	Number of extra columns to be allowed for.
EXTRAELEMS	Number of extra matrix elements to be allowed for.
EXTRAMIPENTS	Number of extra MIP entities to be allowed for.
EXTRAROWS	Number of extra rows to be allowed for.
KEEPNROWS	Status for nonbinding rows.
SCALING	Type of scaling.

Double

MATRIXTOL	Tolerance on matrix elements.

Further information

1. The objective function is of the form $c^T x + 0.5\ x^T Q x$. Note that only the upper or lower triangular part of the `Q` matrix is specified, both for the objective and constraints.
2. The double constants `xpress.infinity` and `-xpress.infinity` are defined in the Optimizer library header file.
3. Passing zero for all integer arguments and `None` for all array arguments will load an empty problem.
4. Arguments `rowind, qrowind` may contain either `xpress.constraint` objects or integer indices.
5. Arguments `objqcol1, objqcol2, rowqcol1, rowqcol2` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.loadMIP`, `problem.loadLP`, `problem.loadMIQP`, `problem.loadQP`, `problem.readProb`.

problem.loadMIQCQP

Purpose

Used to load a mixed integer quadratic problem with quadratic constraints into the Optimizer data structure. Such a problem may have quadratic terms in its objective function as well as in its constraints. Integer, binary, partial integer, semi-continuous and semi-continuous integer variables can be defined, together with sets of type 1 and 2. The reference row values for the set members are passed as an array rather than specifying a reference row.

Topic areas

Problem Input, Quadratic, Optimizer

Synopsis

```
problem.loadMIQCQP(probname="", rowtype=None, rhs=None, rng=None, objcoef=None,
       start=None, collen=None, rowind=None, rowcoef=None, lb=None, ub=None,
       objqcol1=None, objqcol2=None, objqcoef=None, qrowind=None, nrowqcoefs=None,
       rowqcol1=None, rowqcol2=None, rowqcoef=None, coltype=None, entind=None,
       limit=None, settype=None, setstart=None, setind=None, refval=None)
```

Arguments

`probname: str` A string of up to `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual) characters containing a name for the problem. If `probname` is `None`, the problem name will be an empty string.

`rowtype: list[str]` Character array containing the row types:

`L`	indicates a <= constraint (use this one for quadratic constraints as well);
`E`	indicates an = constraint;
`G`	indicates a >= constraint;
`R`	indicates a range constraint;
`N`	indicates a nonbinding constraint.

May be `None` if the problem contains no rows.

`rhs: list[float]` Double array containing the right hand side coefficients of the rows. The right hand side value for a range row gives the upper bound on the row. May be `None` if the problem contains no rows.

`rng: list[float]` Double array containing the range values for range rows. Values for all other rows will be ignored. May be `None` if there are no ranged constraints. The lower bound on a range row is the right hand side value minus the range value. The sign of the range value is ignored - the absolute value is used in all cases.

`objcoef: list[float]` Double array containing the objective function coefficients. This can be `None` to set all objective coefficients to 0 (zero).

`start: list[int]` Integer array containing the offsets in the `rowind` and `rowcoef` arrays of the start of the elements for each column. This array contains one entry for each column, plus one more if `collen` is `None`. If `collen` is `None`, the last entry of `start` contains the position in the `rowind` and `rowcoef` arrays at which an extra column would start, if it were present. This value is also the length of the `rowind` and `rowcoef` arrays. May be `None` if the problem contains no coefficients.

`collen: list[int]` Integer array containing the number of nonzero elements in each column. May be `None` if not required. This array is not required if the non-zero coefficients in the `rowind` and `rowcoef` arrays are contiguous, and the `start` array has an additional entry as described above.

`rowind: list[constraint]` Integer array containing the row indices for the nonzero elements in each column. May be `None` if the problem contains no coefficients. If the indices are input contiguously, with the columns in ascending order, the length of the `rowind` array is found by adding the `start` entry for the last column to the `collen` entry for the last column, or if `collen` is `None`, by the additional entry at the end of `start`.

`rowcoef: list[float]` Double array containing the nonzero element values; length as for `rowind`. May be `None` if the problem contains no coefficients.

`lb: list[float]` Double array containing the lower bounds on the columns. Use `-xpress.infinity` to represent a lower bound of minus infinity. If this is `None` then all lower bounds are 0 (zero).

`ub: list[float]` Double array containing the upper bounds on the columns. Use `xpress.infinity` to represent an upper bound of plus infinity. If this is `None` then all upper bounds are infinite.

`objqcol1: list[var]` Integer array containing the column index of the first variable in each quadratic term.

May be None if all objective coefficients are zero.

objqcol2: list[var] Integer array containing the column index of the second variable in each quadratic term. May be None if all objective coefficients are zero.

objqcoef: list[float] Double array containing the quadratic coefficients. May be None if all objective coefficients are zero.

qrowind: list[constraint] Integer array containing the indices of rows with quadratic matrices in them. Note that the rows are expected to be defined in rowtype as type L. May be None if the problem contains no quadratic constraints.

nrowqcoefs: list[int] Integer array containing the number of nonzeros in each quadratic constraint matrix. May be None if the problem contains no quadratic constraints.

rowqcol1: list[var] Integer array of size nqcelem, where nqcelem equals the sum of the elements in nrowqcoefs (i.e. the total number of quadratic matrix elements in all the constraints). It contains the first column indices of the quadratic matrices. Indices for the first matrix are listed from 0 to nrowqcoefs[0]−1, for the second matrix from nrowqcoefs[0] to nrowqcoefs[0]+nrowqcoefs[1]−1, etc. May be None if the problem contains no quadratic constraints.

rowqcol2: list[var] Integer array of size nqcelem, containing the second index for the quadratic constraint matrices. May be None if the problem contains no quadratic constraints.

rowqcoef: list[float] Integer array of size nqcelem, containing the coefficients for the quadratic constraint matrices. May be None if the problem contains no quadratic constraints.

coltype: list[str] Character array containing the entity types:

B	binary variables;
I	integer variables;
P	partial integer variables;
S	semi-continuous variables;
R	semi-continuous integer variables.

May be None if all variables are continuous.

entind: list[var] Integer array containing the column indices of the MIP entities. May be None if the problem contains no entities.

limit: list[float] Double array containing the integer limits for the partial integer variables and lower bounds for semi-continuous and semi-continuous integer variables (any entries in the positions corresponding to binary and integer variables will be ignored). May be None if not required.

settype: list[str] Character array containing the set types:

1	SOS1 type sets;
2	SOS2 type sets.

May be None if not required.

setstart: list[int] Integer array containing the offsets in the setind and refval arrays indicating the start of each set. This array has one entry for each set plus one extra entry containing the offset where an additional set would start, if it were present. May be None if not required.

setind: list[var] Integer array containing the columns in each set. May be None if not required.

refval: list[float] Double array containing the reference row entries for each member of the sets. These define the order for SOS2 constraints and may be used in branching for both types. May be None if not required.

Related controls

Integer

EXTRACOLS	Number of extra columns to be allowed for.
EXTRAELEMS	Number of extra matrix elements to be allowed for.
EXTRAMIPENTS	Number of extra MIP entities to be allowed for.
EXTRAROWS	Number of extra rows to be allowed for.
KEEPNROWS	Status for nonbinding rows.
SCALING	Type of scaling.

Double

MATRIXTOL	Tolerance on matrix elements.

Further information

1. The objective function is of the form $c^T x + 0.5\ x^T Q x$. Note that only the upper or lower triangular part of the `Q` matrix is specified, both for the objective and constraints.
2. The double constants `xpress.infinity` and `-xpress.infinity` are defined in the Optimizer library header file.
3. The double constants `xpress.infinity` and `-xpress.infinity` are defined in the Optimizer library header file.
4. Semi-continuous lower bounds are taken from the limit array. If this is `None` then they are given a default value of `1.0`. If a semi-continuous variable has a positive lower bound then this will be used as the semi-continuous lower bound and the lower bound on the variable will be set to zero.
5. Passing zero for all integer arguments and `None` for all array arguments will load an empty problem.
6. Arguments `rowind, qrowind` may contain either `xpress.constraint` objects or integer indices.
7. Arguments `objqcol1, objqcol2, rowqcol1, rowqcol2, entind, setind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.loadMIP`, `problem.loadLP`, `problem.loadQCQP`, `problem.loadMIQP`, `problem.loadQP`, `problem.readProb`.

problem.addObj

Purpose

Appends an objective function with the given coefficients to a multi-objective problem. The weight and priority of the objective are set to the given values.

Topic areas

Problem Creation, Multiobjective, Optimizer

Synopsis

```
problem.addObj(colind, objcoef, priority=0, weight=1)
```

Arguments

`colind: list[var]` Integer array containing the indices of the columns whose objective coefficients will change. An index of `-1` indicates that the fixed part of the objective function on the right hand side should change.

`objcoef: list[float]` Double array giving the new objective function coefficients.

`priority: int` An integer defining the relative priority for the new objective (only relevant for multi-objective problems). Higher values mean higher priority. During optimization, objectives with the same priority are combined together in a weighted sum.

`weight: float` A double defining the weight for the new objective (only relevant for multi-objective problems). If the weight is negative, the sense of this objective is reversed.

Further information

1. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
2. If all objectives have equal priority but different weights, a blended (weighted-sum) approach is applied. For cases where each objective has a different priority but the same weight, a lexicographic approach is applied. The highest priority objective is optimized first. In the second and subsequent iterations, all objectives from previous iterations are fixed to their optimal values within tolerances. If both different priorities and different weights are passed, a hybrid approach is applied: Xpress will solve the problem once for each distinct objective priority that is defined, optimizing in each iteration a linear combination of the objective functions with the same priority.
3. The optimization sense of the first objective defines the default sense for all objectives. To revert the sense of an objective in relation to the first objective, assign it a negative `weight`. For example, if the first objective is set to be maximized, further objectives to be minimized can be declared with `weight=-1`.
4. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.chgObjN`, `problem.getObjN`, `problem.delObj`, `problem.chgObj`.

problem.chgObjN

Purpose

Modifies one or more coefficients of an objective function in a multi-objective problem. If the objective already exists, any coefficients not present in the `colind` and `objcoef` arrays will unchanged. If the objective does not exist, it will be added to the problem.

Topic areas

Problem Modification, Multiobjective, Optimizer

Synopsis

```
problem.chgObjN(objidx, colind, objcoef)
```

Arguments

`objidx: int` Index of the objective function to add or modify.

`colind: list[var]` Integer array containing the indices of the columns whose objective coefficients will change. An index of `-1` indicates that the fixed part of the objective function on the right hand side should change.

`objcoef: list[float]` Double array giving the new objective function coefficients.

Further information

1. When `objidx=0`, this function is equivalent to `problem.chgObj`.
2. Any objectives with `idx < objidx` that do not already exist will be added to the problem with all zero coefficients.
3. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.
4. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.addObj`, `problem.getObjN`, `problem.delObj`, `problem.chgObj`.

problem.delObj

Purpose

Removes an objective function from the problem. Any objectives with `index > objidx` will be shifted down.

Topic areas

Problem Creation, Optimizer

Synopsis

```
problem.delObj(objidx)
```

Argument

`objidx: int` Index of the objective to remove.

Further information

1. If the problem has only one objective, `problem.delObj` clears the objective, but `OBJECTIVES` remains set to `1`.
2. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.

Related topics

`problem.chgObjN`, `problem.addObj`, `problem.getObjN`.

problem.copyControls

Purpose

Copies controls defined for one problem to another.

Topic areas

Controls and Attributes, Optimizer

Synopsis

```
problem.copyControls(src)
```

Argument

`src: problem` The problem from which the controls are copied.

Related topics

`problem.copycallbacks`

problem.interrupt

Purpose

Interrupts the Optimizer algorithms.

Topic areas

Solution Process, Optimizer

Synopsis

```
problem.interrupt(reason=xpress.enums.StopType.USER)
```

Argument

`reason: StopType` The reason for stopping. Possible reasons are:

- `xpress.StopType.NONE` do not stop;
- `xpress.StopType.TIMELIMIT` time limit hit;
- `xpress.StopType.WORKLIMIT` work limit hit;
- `xpress.StopType.CTRLC` control C hit;
- `xpress.StopType.NODELIMIT` node limit hit;
- `xpress.StopType.ITERLIMIT` iteration limit hit;
- `xpress.StopType.MIPGAP` MIP gap is sufficiently small;
- `xpress.StopType.SOLLIMIT` solution limit hit;
- `xpress.StopType.USER` user interrupt;
- `xpress.StopType.NEXTOBJECTIVE` stop the current solve, but continue with solving the next objective (see below);
- `>= 1000` user defined value.

Further information

1. The `problem.interrupt` command can be called from any callback.
2. The range of values below 1000 is reserved for future extension. The behavior of the function is undefined if `reason` is smaller than 1000 and not one of the values listed above.
3. When solving a multi-objective problem, calling `problem.interrupt` with `NEXTOBJECTIVE` instructs the Optimizer to accept the current solution and proceed with solving the next objective. For example, this can be used in the `checktime` callback to implement a different time limit for each objective. If no solution has been found, no objective fixing will take place. The `SOLVESTATUS` (p.**??** Xpress Optimizer Reference Manual) attribute reflects the status of the final solve: in particular, if the final objective was solved to optimality then `SOLVESTATUS` will be `COMPLETED`, even if earlier objectives were interrupted. If a single-objective solve, or the final solve of a multi-objective problem, is interrupted with status `NEXTOBJECTIVE`, then `SOLVESTATUS` will be `INTERRUPTED` and `STOPSTATUS` will be `NEXTOBJECTIVE`.

Related topics

None.

problem.getProbName

Purpose

Returns the current problem name.

Topic areas

Problem Information, Optimizer

Synopsis

```
name = problem.getProbName()
```

Return value

`name: str` A buffer of `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual)+1 bytes to contain the current problem name.

Related topics

`problem.setProbName`, `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual).

problem.getControlInfo

Purpose

Accesses the id number and the type information of a control given its name. A control name may be for example XPRS_PRESOLVE. Names are case-insensitive and may or may not have the XPRS_ prefix. The id number is the constant used to identify the control in the C API. The type information returned will be one of the below integer constants.

The function will return an id number of 0 and a type value of xpress.ParameterType.NOTDEFINED if the name is not recognized as a control name. Note that this will occur if the name is an attribute name and not a control name.

Topic areas

Controls and Attributes, Optimizer

Synopsis

```
id, type = problem.getControlInfo(name)
```

Argument

`name: str` The name of the control to be queried. Names are case-insensitive and may or may not have the XPRS_ prefix. A full list of all controls may be found in the Optimizer Reference Manual, or from the list in the xprs.h header file.

Return value

`id: int` The id number.

`type: ParameterType` The type information. The returned value will be one of the following constants from xprs.h:

xpress.ParameterType.NOTDEFINED	The name was not recognized.
xpress.ParameterType.INT	32-bit integer.
xpress.ParameterType.INT64	64-bit integer.
xpress.ParameterType.DOUBLE	Double precision floating point.
xpress.ParameterType.STRING	String.

Related topics

problem.getAttribInfo.

problem.getAttribInfo

Purpose

Accesses the id number and the type information of an attribute given its name. An attribute name may be for example `XPRS_ROWS`. Names are case-insensitive and may or may not have the `XPRS_` prefix. The id number is the constant used to identify the attribute in the C API. The type information returned will be one of the below integer constants.

The function will return an id number of 0 and a type value of `xpress.ParameterType.NOTDEFINED` if the name is not recognized as an attribute name. Note that this will occur if the name is a control name and not an attribute name.

Topic areas

Controls and Attributes, Optimizer

Synopsis

```
id, type = problem.getAttribInfo(name)
```

Argument

`name: str` The name of the attribute to be queried. Names are case-insensitive and may or may not have the `XPRS_` prefix. A full list of all attributes may be found in the Optimizer Reference Manual, or from the list in the `xprs.h` header file.

Return value

`id: int` The id number.

`type: ParameterType` The type id. The value will be one of the following constants from `xprs.h`:

xpress.ParameterType.NOTDEFINED	The name was not recognized.
xpress.ParameterType.INT	32-bit integer.
xpress.ParameterType.INT64	64-bit integer.
xpress.ParameterType.DOUBLE	Double precision floating point.
xpress.ParameterType.STRING	String.

Related topics

`problem.getControlInfo`.

problem.getIndex

Purpose

Returns the index for a specified row or column name.

Topic areas

Problem Information, Optimizer

Synopsis

```
index = problem.getIndex(type, name=None)
```

Arguments

`type: int` `xpress.Namespaces.ROW` (=1) if row index is required;
`xpress.Namespaces.COLUMN` (=2) if column index is required;
`xpress.Namespaces.SET` (=3) if set index is required;
`xpress.Namespaces.PWLCONS` (=4) if piecewise linear constraint index is required;
`xpress.Namespaces.GENCONS` (=5) if general constraint index is required;
`xpress.Namespaces.OBJECTIVE` (=6) if objective index is required;
`xpress.Namespaces.USERFUNC` (=7) if user function index is required;
`xpress.Namespaces.INTERNALFUNC` (=8) if an internal function index is required.

`name: str` None terminated string.

Return value

`index: int` The row or column index number. A value of `-1` will be returned if the row or column does not exist.

Further information

1. Names are case sensitive, and the names of internal functions are upper case, e.g., `COS`.
2. To get the index of a variable, constraint or sos object, use `obj.index`.

Related topics

`problem.addNames`.

problem.getQObj

Purpose

Returns a single quadratic objective function coefficient corresponding to the variable pair `(objqcol1, objqcol2)` of the Hessian matrix.

Topic areas

Problem Information, Quadratic, Optimizer

Synopsis

```
objqcoef = problem.getQObj(objqcol1, objqcol2)
```

Arguments

`objqcol1: var` Column index for the first variable in the quadratic term.

`objqcol2: var` Column index for the second variable in the quadratic term.

Return value

`objqcoef: float` The objective function coefficient.

Further information

1. `objqcoef` is the coefficient in the quadratic Hessian matrix. For example, if the objective function has the term $[3x_1x_2 + 3x_2x_1]/2$ the value retrieved by `problem.getQObj` is `3.0` and if the objective function has the term $[6x_1^2]/2$ the value retrieved by `problem.getQObj` is `6.0`.
2. Arguments `objqcol1, objqcol2` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.getMQObj`, `problem.chgQObj`, `problem.chgMQObj`.

problem.getDirs

Purpose

Used to return the directives that have been loaded into a matrix. Priorities, forced branching directions and pseudo costs can be returned. If called after presolve, `problem.getDirs` will get the directives for the presolved problem.

Topic areas

Problem Information, Branching, Optimizer

Synopsis

```
indices, prios, branchdirs, uppseudo, downpseudo = problem.getDirs()
```

Return value

`indices: list[int]` Integer array containing the column numbers (`0`, `1`, `2`,...) or negative values corresponding to special ordered sets (the first set numbered `-1`, the second numbered `-2`,...).

`prios: list[int]` Integer array containing the priorities for the columns and sets, where columns/sets with smallest priority will be branched on first.

`branchdirs: list[str]` Character array containing the branching direction for each column or set:

- `U` the entity is to be forced up;
- `D` the entity is to be forced down;
- `N` not specified.

`uppseudo: list[float]` Double array containing the up pseudo costs for the columns and sets.

`downpseudo: list[float]` Double array containing the down pseudo costs for the columns and sets.

Related topics

`problem.loadDirs`, `problem.loadPresolveDirs`.

problem.getScale

Purpose

Returns the the current scaling of the matrix.

Topic areas

Numerics, Optimizer

Synopsis

```
rowscale, colscale = problem.getScale()
```

Return value

`rowscale: list[int]` Integer array of size `ROWS` (p.**??** Xpress Optimizer Reference Manual) that will contain the powers of `2` with which the rows are currently scaled.

`colscale: list[int]` Integer array of size `COLS` (p.**??** Xpress Optimizer Reference Manual) that will contain the powers of `2` with which the columns are currently scaled.

Related topics

`problem.scale`.

problem.getPivotOrder

Purpose

Returns the pivot order of the basic variables.

Topic areas

Simplex, LP, Optimizer

Synopsis

```
pivotorder = problem.getPivotOrder()
```

Return value

`pivotorder: list[int]` Integer array of length `ROWS` (p.**??** Xpress Optimizer Reference Manual) containing the pivot order.

Further information

Row indices are in the range `0` to `ROWS` (p.**??** Xpress Optimizer Reference Manual)`-1`, whilst columns are in the range `ROWS+SPAREROWS` (p.**??** Xpress Optimizer Reference Manual) to `ROWS+SPAREROWS+COLS` (p.**??** Xpress Optimizer Reference Manual)`-1`.

Related topics

`problem.getPivots`, `problem.pivot`.

problem.getPresolveMap

Purpose

Returns the mapping of the row and column numbers from the presolve problem back to the original problem.

Topic areas

Problem Information, Presolve, Optimizer

Synopsis

```
rowmap, colmap = problem.getPresolveMap()
```

Return value

`rowmap: list[int]` Integer array of length `ROWS` (p.**??** Xpress Optimizer Reference Manual) containing the row maps.

`colmap: list[int]` Integer array of length `COLS` (p.**??** Xpress Optimizer Reference Manual) containing the column maps.

Further information

1. This function applies to linear and convex quadratic (including second order conic) problems only.
2. The presolved problem can contain rows or columns that do not map to anything in the original problem. An example of this are cuts created during the MIP solve and temporarily added to the presolved problem. It is also possible that the presolver will introduce new rows or columns. For any row or column that does not have a mapping to a row or column in the original problem, the corresponding entry in the returned `rowmap` and `colmap` arrays will be `-1`.

Related topics

the Optimizer Reference Manual.

problem.BTran

Purpose

Post-multiplies a (row) vector provided by the user by the inverse of the current basis.

Topic areas

Linear Algebra, LP, Optimizer

Synopsis

```
vec = problem.BTran(vec)
```

Argument

`vec: list[float]` Double array of length `ROWS` (p.**??** Xpress Optimizer Reference Manual) containing the values by which the basis inverse is to be multiplied.

Return value

`vec: list[float]` The result of the multiplication

Related controls

Double

`ETATOL` Tolerance on eta elements.

Further information

If the matrix is in a presolved state, `problem.BTran` works with the basis for the presolved problem.

Related topics

`problem.FTran`.

problem.FTran

Purpose

Pre-multiplies a (column) vector provided by the user by the inverse of the current matrix.

Topic areas

Linear Algebra, LP, Optimizer

Synopsis

```
vec = problem.FTran(vec)
```

Argument

`vec: list[float]` Double array of length `ROWS` (p.**??** Xpress Optimizer Reference Manual) containing the values which are to be multiplied by the basis inverse.

Return value

`vec: list[float]` The result of the multiplication

Related controls

Double

`ETATOL` Tolerance on eta elements.

Further information

If the matrix is in a presolved state, the function will work with the basis for the presolved problem.

Related topics

`problem.BTran`.

problem.sparseBTran

Purpose

Post-multiplies a (row) vector provided by the user by the inverse of the current matrix. Sparse version of `problem.BTran`.

Topic areas

Linear Algebra, Simplex, Optimizer

Synopsis

```
val, ind = problem.sparseBTran(val, ind)
```

Arguments

`val: list[float]` Array containing the non-zero values which are to be post-multiplied by the basis.

`ind: list[int]` Array containing the indices for the elements in `val`.

Return value

`val: list[float]` Array containing the non-zero values resulting from the multiplication.

`ind: list[int]` Array containing the indices for the elements returned in `val`.

Related controls

Double

`ETATOL` Tolerance on eta elements.

Further information

If the matrix is in a presolved state, the function will work with the basis for the presolved problem.

Related topics

`problem.BTran`, `problem.FTran`, `problem.sparseFTran`.

problem.sparseFTran

Purpose

Pre-multiplies a (column) vector provided by the user by the inverse of the current matrix. Sparse version of `problem.FTran`.

Topic areas

Linear Algebra, Simplex, Optimizer

Synopsis

```
val, ind = problem.sparseFTran(val, ind)
```

Arguments

`val: list[float]` Array containing the non-zero values which are to be multiplied by the basis.

`ind: list[int]` Array containing the indices for the elements in `val`.

Return value

`val: list[float]` Array containing the non-zero values resulting from the multiplication.

`ind: list[int]` Array containing the indices for the elements returned in `val`.

Related controls

Double

`ETATOL` Tolerance on eta elements.

Further information

If the matrix is in a presolved state, the function will work with the basis for the presolved problem.

Related topics

`problem.FTran`, `problem.BTran`, `problem.sparseBTran`.

problem.getObj

Purpose

Returns the objective function coefficients for the columns in a given range.

Topic areas

Problem Information, Optimizer

Synopsis

```
objcoef = problem.getObj(first=0, last=COLS-1)
```

Arguments

`first: int` First column in the range.

`last: int` Last column in the range.

Return value

`objcoef: list[float]` Double array of length `last-first+1` containing the objective function coefficients.

Further information

Arguments `first, last` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.chgObj`.

problem.getObjN

Purpose

For a given objective function, returns the objective coefficients for the columns in a given range.

Topic areas

Problem Information, Multiobjective, Optimizer

Synopsis

```
objcoef = problem.getObjN(objidx, first=0, last=COLS-1)
```

Arguments

`objidx: int` Index of the objective function whose coefficients to return.

`first: int` First column in the range.

`last: int` Last column in the range.

Return value

`objcoef: list[float]` Double array of length `last-first+1` containing the objective function coefficients.

Further information

1. This function always returns the objective coefficients from the original problem: it will raise an error if the problem is presolved.
2. The coefficients of the main objective (`objidx=0`) can also be retrieved using `problem.getObj`.
3. Arguments `first, last` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.chgObjN`, `problem.addObj`, `problem.delObj`, `problem.getObj`.

problem.getRHS

Purpose

Returns the right hand side elements for the rows in a given range.

Topic areas

Problem Information, Optimizer

Synopsis

```
rhs = problem.getRHS(first=0, last=ROWS-1)
```

Arguments

`first: int` First row in the range.

`last: int` Last row in the range.

Return value

`rhs: list[float]` Double array of length `last-first+1` containing the right hand side elements.

Further information

1. For range rows, `rhs` is populated with the upper limit of the row range.
2. Arguments `first, last` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.chgRHS`, `problem.chgRHSRange`, `problem.getRHSRange`.

problem.getRHSRange

Purpose

Returns the right hand side range values for the rows in a given range.

Topic areas

Problem Information, Optimizer

Synopsis

```
rng = problem.getRHSRange(first=0, last=ROWS-1)
```

Arguments

`first: int` First row in the range.

`last: int` Last row in the range.

Return value

`rng: list[float]` Double array of length `last-first+1` containing the right hand side range values.

Further information

1. Range values are defined as the upper limit of the row range minus the lower limit, which is always non-negative. For equality rows, the range is `0`. For inequality rows and nonbinding rows, the range is `xpress.infinity`.
2. Arguments `first, last` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.chgRHS`, `problem.chgRHSRange`, `problem.getRHS`.

problem.getLB

Purpose

Returns the lower bounds for the columns in a given range.

Topic areas

Problem Information, Optimizer

Synopsis

```
lb = problem.getLB(first=0, last=COLS-1)
```

Arguments

`first: int` First column in the range.

`last: int` Last column in the range.

Return value

`lb: list[float]` Double array of length `last-first+1` containing the lower bounds.

Further information

1. Values greater than or equal to `xpress.infinity` should be interpreted as infinite; values less than or equal to `-xpress.infinity` should be interpreted as infinite and negative.
2. Arguments `first, last` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.chgBounds`, `problem.getUB`.

problem.getUB

Purpose

Returns the upper bounds for the columns in a given range.

Topic areas

Problem Information, Optimizer

Synopsis

```
ub = problem.getUB(first=0, last=COLS-1)
```

Arguments

`first: int` First column in the range.

`last: int` Last column in the range.

Return value

`ub: list[float]` Double array of length `last-first+1` containing the upper bounds.

Further information

1. Values greater than or equal to `xpress.infinity` should be interpreted as infinite; values less than or equal to `-xpress.infinity` should be interpreted as infinite and negative.
2. Arguments `first, last` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.chgBounds`, `problem.getLB`.

problem.getCols

Purpose

Returns the nonzeros in the constraint matrix for the columns in a given range.

Topic areas

Problem Information, Optimizer

Synopsis

```
start, rowind, rowcoef = problem.getCols(first=0, last=COLS-1)
```

Arguments

`first: int` First column in the range.

`last: int` Last column in the range.

Return value

`start: list[int]` Integer array containing the indices indicating the starting offsets in the `rowind` and `rowcoef` arrays for each requested column. Column `i` starts at position `start[i]` in the `rowind` and `rowcoef` arrays, and has `start[i+1]-start[i]` elements in it.

`rowind: list[int]` Integer array containing the row indices of the nonzero coefficents for each column.

`rowcoef: list[float]` Double array containing the nonzero coefficient values.

Further information

Arguments `first, last` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.getRows`.

problem.getRows

Purpose

Returns the nonzeros in the constraint matrix for the rows in a given range.

Topic areas

Problem Information, Optimizer

Synopsis

```
start, colind, colcoef = problem.getRows(first=0, last=ROWS-1)
```

Arguments

`first: int` First row in the range.

`last: int` Last row in the range.

Return value

`start: list[int]` Integer array containing the indices indicating the starting offsets in the `colind` and `colcoef` arrays for each requested row. Row `i` starts at position `start[i]` in the `colind` and `colcoef` arrays, and has `start[i+1]-start[i]` elements in it.

`colind: list[int]` Integer array containing the column indices of the nonzero elements for each row.

`colcoef: list[float]` Double array containing the nonzero element values.

Further information

Arguments `first, last` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.getCols`, `problem.getRowType`.

problem.getMIPEntities

Purpose

Retrieves integr and entity information about a problem. It must be called before `problem.mipOptimize` if the presolve option is used.

Topic areas

Problem Information, MIP Entities, Optimizer

Synopsis

```
coltype, colind, limit, settype, start, setcols, refval = problem.getMIPEntities()
```

Return value

`coltype: list[str]` Character array of length `MIPENTS` containing the entity types. The types will be one of:

- `B` binary variables;
- `I` integer variables;
- `P` partial integer variables;
- `S` semi-continuous variables;
- `R` semi-continuous integer variables.

`colind: list[int]` Integer array of length `MIPENTS` containing the column indices of the MIP entities.

`limit: list[float]` Double array of length `MIPENTS` containing the limits for the partial integer variables and lower bounds for the semi-continuous and semi-continuous integer variables (any entries in the positions corresponding to binary and integer variables will be meaningless).

`settype: list[str]` Character array of length `SETS` containing the set types. The set types will be one of:

- `1` SOS1 type sets;
- `2` SOS2 type sets.

`start: list[int]` Integer array containing the offsets into the `setcols` and `refval` arrays indicating the start of each set. This array must be of length `SETS+1`, the last element will contain the offset where set `SETS+1` would start, and equals the length of the `setcols` and `refval` arrays, `SETMEMBERS` (p.**??** Xpress Optimizer Reference Manual).

`setcols: list[int]` Integer array of length `SETMEMBERS` containing the columns in each set.

`refval: list[float]` Double array of length `SETMEMBERS` containing the reference row entries for each member of the sets. These define the order for SOS2 constraints and may be used in branching for both types.

Related topics

`problem.loadMIP`, `problem.loadMIQP`.

problem.getRowFlags

Purpose

Retrieve if a range of rows have been set up as special rows.

Topic areas

Problem Information, Optimizer

Synopsis

```
flags = problem.getRowFlags(first=0, last=ROWS-1)
```

Arguments

`first: int`	First row index to be checked
`last: int`	Last row index to be checked

Return value

`flags: list[int]` Integer array of length `last-first+1` containing type of information (see below)

Further information

1. The `flags` array returns a bitvector for each row defining all the information that is currently attached to that row:

`xpress.RowFlag.QUADRATIC`	The row has quadratic coefficients.
`xpress.RowFlag.DELAYED`	The row is marked as a delayed row.
`xpress.RowFlag.MODELCUT`	The row is marked as a model cut.
`xpress.RowFlag.INDICATOR`	The row is used as an indicator.
`xpress.RowFlag.NONLINEAR`	The row has nonlinear coefficients.

2. Arguments `first, last` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.clearRowFlags`

problem.clearRowFlags

Purpose

Clears extra information attached to a range of rows.

Topic areas

Problem Modification, Optimizer

Synopsis

```
problem.clearRowFlags(flags, first=0, last=ROWS-1)
```

Arguments

`flags: list[int]` Integer array of length `last-first+1` including type of extra information to remove (see below)

`first: int` First row index to be checked

`last: int` Last row index to be checked

Further information

1. The `flags` array consists of a bitvector for each row defining types of information to remove:

`xpress.RowFlag.QUADRATIC`	Remove all quadratic coefficients.
`xpress.RowFlag.DELAYED`	The row will not be a delayed row.
`xpress.RowFlag.MODELCUT`	The row will not be a model cut.
`xpress.RowFlag.INDICATOR`	Remove indicators associated to the row.
`xpress.RowFlag.NONLINEAR`	Remove any nonlinear coefficients.

2. Arguments `first, last` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.getRowFlags`

problem.getCoef

Purpose

Returns a single coefficient in the constraint matrix.

Topic areas

Problem Information, Optimizer

Synopsis

```
coef = problem.getCoef(row, col)
```

Arguments

`row: constraint` Row of the constraint matrix.

`col: var` Column of the constraint matrix.

Return value

`coef: float` The coefficient.

Further information

1. It is quite inefficient to get several coefficients with the `problem.getCoef` function. It is better to use `problem.getCols` or `problem.getRows`.
2. Argument `row` may contain either `xpress.constraint` objects or integer indices.
3. Argument `col` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.getCols`, `problem.getRows`.

problem.getMQObj

Purpose

Returns the nonzeros in the quadratic objective coefficients matrix for the columns in a given range. To achieve maximum efficiency, `problem.getMQObj` returns the lower triangular part of this matrix only.

Topic areas

Problem Information, Quadratic, Optimizer

Synopsis

```
start, colind, objqcoef = problem.getMQObj(first=0, last=COLS-1)
```

Arguments

`first: int` First column in the range.

`last: int` Last column in the range.

Return value

`start: list[int]` Integer array of length `last-first+2` containing indices indicating the starting offsets in the `colind` and `objqcoef` arrays for each requested column. Column `i` starts at position `start[i]` in the `colind` and `objqcoef` arrays, and has `start[i+1]-start[i]` elements in it.

`colind: list[int]` Integer array containing the column indices of the nonzero elements in the lower triangular part of `Q`.

`objqcoef: list[float]` Double array containing the nonzero element values.

Further information

1. The objective function is of the form $c^T x + 0.5 x^T Q x$. Note that only the upper or lower triangular part of the `Q` matrix is returned.
2. Arguments `first, last` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.chgMQObj`, `problem.chgQObj`, `problem.getQObj`.

problem.writeBasis

Purpose

Writes the current basis to a file for later input into the Optimizer.

Topic areas

File IO, LP, Optimizer

Synopsis

```
problem.writeBasis(filename="", flags="")
```

Arguments

`filename: str` A string of up to `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual) characters containing the file name from which the basis is to be written. If omitted, the default *problem_name* is used with a `.bss` extension.

`flags: str` Flags to pass to `problem.writeBasis`:

- `i` output the internal presolved basis;
- `t` output a compact advanced form of the basis;
- `n` output basis file containing current solution values;
- `h` output values in single precision;
- `p` output values in full precision (obsolete as this is now default behavior);
- `v` use the provided filename verbatim, without appending the `.bss` extension;
- `z` compress the output file.

Further information

1. The `t` flag is only useful for later input to a similar problem using the `t` flag with `problem.readBasis`.
2. If the Newton barrier or hybrid gradient algorithms have been used for optimization then crossover must have been performed before there is a valid basis. This basis can then be used for restarting the simplex (primal or dual) or the hybrid gradient algorithms.
3. `problem.writeBasis` will output the basis for the original problem even if the matrix has been presolved.

Related topics

`problem.getBasis`, `problem.readBasis`.

problem.writeSol

Purpose

Writes the current solution to a CSV format ASCII file, *problem_name*`.asc` (and `.hdr`).

Topic areas

File IO, Solution, Optimizer

Synopsis

```
problem.writeSol(filename="", flags="")
```

Arguments

`filename: str` A string of up to `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual) characters containing the file name to which the solution is to be written. If omitted, the default *problem_name* will be used. The extensions `.hdr` and `.asc` will be appended.

`flags: str` Flags to control which optional fields are output:

`s`	sequence number;
`n`	name;
`t`	type;
`b`	basis status;
`a`	activity;
`c`	cost (columns), slack (rows);
`l`	lower bound;
`u`	upper bound;
`d`	dj (column; reduced costs), dual value (rows; shadow prices);
`r`	right hand side (rows).

If no flags are specified, all fields are output.

Additional flags:

`p`	outputs in full precision;
`q`	only outputs vectors with nonzero optimum value;
`x`	output the current LP solution instead of the MIP solution;
`z`	compress the output file.

Related controls

Double

`OUTPUTTOL` Tolerance on print values.

String

`OUTPUTMASK` Mask to restrict the row and column names output to file.

Further information

1. This function currently does not apply to general nonlinear problems.
2. The command produces two readable files: `filename.hdr` (the solution header file) and `filename.asc` (the CSV format solution file). The header file contains summary information, all in one line. The ASCII file contains one line of information for each row and column in the problem. Any fields appearing in the `.asc` file will be in the order the flags are described above. The order that the flags are specified by the user is irrelevant.
3. Additionally, the mask control `OUTPUTMASK` (p.**??** Xpress Optimizer Reference Manual) may be used to control which names are reported to the ASCII file. Only vectors whose names match `OUTPUTMASK` are output. `OUTPUTMASK` is set by default to "`????????`", so that all vectors are output.

Related topics

`problem.getSolution`, `problem.writePrtSol`.

problem.writeBinSol

Purpose

Writes the current MIP or LP solution to a binary solution file for later input into the Optimizer.

Topic areas

File IO, Solution, Optimizer

Synopsis

```
problem.writeBinSol(filename="", flags="")
```

Arguments

`filename: str` A string of up to `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual) characters containing the file name to which the solution is to be written. If omitted, the default *problem_name* is used with a `.sol` extension.

`flags: str` Flags to pass to `problem.writeBinSol`:

- `m` output the MIP solution;
- `x` output the LP solution;
- `v` use the provided filename verbatim, without appending the `.sol` extension;
- `z` compress the output file.

Further information

This function currently does not apply to general nonlinear problems.

Related topics

`problem.getSolution`, `problem.readBinSol`, `problem.writeSol`, `problem.writePrtSol`.

problem.writePrtSol

Purpose

Writes the current solution to a fixed format ASCII file, *problem_name* `.prt`.

Topic areas

File IO, Solution, Optimizer

Synopsis

```
problem.writePrtSol(filename="", flags="")
```

Arguments

`filename: str` A string of up to `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual) characters containing the file name to which the solution is to be written. If omitted, the default *problem_name* will be used. The extension `.prt` will be appended.

`flags: str` Flags for `problem.writePrtSol` are:

- `x` write the LP solution instead of the current MIP solution;
- `v` use the provided filename verbatim, without appending the `.prt` extension;
- `z` write a compressed output file;
- `s` include classical sensitivity analysis.

Related controls

Integer

`MAXPAGELINES` Number of lines between page breaks.

Double

`OUTPUTTOL` Tolerance on print values.

Further information

1. This function currently does not apply to general nonlinear problems.
2. The fixed width ASCII format created by this command is not as readily useful as that produced by `problem.writeSol`. The main purpose of `problem.writePrtSol` is to create a file that can be sent directly to a printer. The format of this fixed format ASCII file is described in Appendix the Optimizer Reference Manual.
3. To create a prt file for a previously saved solution, the solution must first be loaded with the `problem.readBinSol` function.

Related topics

`problem.getSolution`, `problem.readBinSol` `problem.writeBinSol`, `problem.writeSol`, the Optimizer Reference Manual.

problem.writeSlxSol

Purpose

Creates an ASCII solution file (`.slx`) using a similar format to MPS files. These files can be read back into the Optimizer using the `problem.readSlxSol` function.

Topic areas

File IO, Solution, Optimizer

Synopsis

```
problem.writeSlxSol(filename="", flags="")
```

Arguments

`filename: str` A string of up to `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual) characters containing the file name to which the solution is to be written. If omitted, the default *problem_name* is used with a `.slx` extension.

`flags: str` Flags to pass to `problem.writeSlxSol`:

- `l` write the LP solution in case of a MIP problem;
- `m` write the MIP solution;
- `p` use full precision for numerical values (obsolete as this is now default behavior);
- `s` including slack variables;
- `d` LP solution only: including dual variables;
- `r` LP solution only: including reduced cost;
- `v` use the provided filename verbatim, without appending the `.slx` extension;
- `z` compress the output file.

Related topics

`problem.readSlxSol`, `problem.writePrtSol`, `problem.writeBinSol`, `problem.readBinSol`.

problem.getPrimalRay

Purpose

Retrieves a primal ray (primal unbounded direction) for the current problem, if the problem is found to be unbounded.

Topic areas

Solution, LP, Optimizer

Synopsis

```
ray = problem.getPrimalRay()
```

Return value

`ray: Optional[list[float]]` Double array of length `COLS` (p.**??** Xpress Optimizer Reference Manual) containing the ray, or `None` if the Optimizer is not able to return a primal ray.

Further information

1. This function applies to linear and convex quadratic (including second order conic) problems only.
2. It is possible to retrieve a primal ray only when, after solving an LP problem, the final status (`LPSTATUS` (p.**??** Xpress Optimizer Reference Manual)) is xpress.LPStatus.UNBOUNDED.
3. Primal rays are not post-solved. If the problem is in a presolved state, the primal ray that is returned will be for the presolved problem. If the problem was solved with presolve on and has been restored to the original state (the default behavior), this function will not be able to return a ray. To ensure that a primal ray can be obtained, it is recommended to solve a problem with presolve turned off (`PRESOLVE` (p.**??** Xpress Optimizer Reference Manual) = 0).

Related topics

`problem.getDualRay`.

problem.getDualRay

Purpose

Retrieves a dual ray (dual unbounded direction) for the current problem, if the problem is found to be infeasible.

Topic areas

Solution, LP, Optimizer

Synopsis

```
ray = problem.getDualRay()
```

Return value

`ray: Optional[list[float]]` Double array of length `ROWS` (p.**??** Xpress Optimizer Reference Manual) containing the ray, or `None` if the Optimizer is not able to return a dual ray.

Further information

1. It is possible to retrieve a dual ray only when, after solving an LP problem, the final status (`LPSTATUS` (p.**??** Xpress Optimizer Reference Manual)) is `xpress.LPStatus.INFEAS`.
2. Dual rays are not post-solved. If the problem is in a presolved state, the dual ray that is returned will be for the presolved problem. If the problem was solved with presolve on and has been restored to the original state (the default behavior), this function will not be able to return a ray. To ensure that a dual ray can be obtained, it is recommended to solve a problem with presolve turned off (`PRESOLVE` (p.**??** Xpress Optimizer Reference Manual) = 0).

Related topics

`problem.getPrimalRay`.

problem.loadMipSol

Purpose

Loads a starting MIP solution for the problem into the Optimizer.

Topic areas

Solution, MIP Entities, Data Input, Optimizer

Synopsis

```
status = problem.loadMipSol(x)
```

Argument

`x: list[float]` Double array of length `COLS` (p.**??** Xpress Optimizer Reference Manual) (for the original problem and not the presolve problem) containing the values of the variables.

Return value

`status: int` The status. The status is one of:

`-1`	Solution rejected because an error occurred;
`0`	Solution accepted.

Further information

1. When a solution is loaded, the solution is placed in temporary storage until the MIP solve is started. Only after the MIP solve has commenced and any presolve has been applied, will the loaded solution be checked and possibly accepted as a new incumbent integer solution. There are no checks performed on the solution before the MIP solve and the returned status in `problem.loadMipSol` will always be `0` for accepted.
2. Loaded solution values will automatically be adjusted to fit within the current problem bounds.
3. It is recommended to use `problem.addMipSol` instead of `problem.loadMipSol`. `problem.addMipSol` can be called both before a solve, to load a starting solution, and during a MIP solve, to load new solutions within callbacks. `problem.addMipSol` also allows for loading of infeasible or partial solutions and comes with a callback to check the status of loaded solutions.
4. This function cannot be called while the current problem is being solved or when it is in an interrupted state. Use `problem.postSolve` to restore the problem to a non-solving state.

Related topics

`problem.addMipSol`, `problem.getSolution`.

problem.getBasis

Purpose

Returns the current basis into the user's data arrays.

Topic areas

Linear Algebra, LP, Optimizer

Synopsis

```
rowstat, colstat = problem.getBasis()
```

Return value

`rowstat: list[int]` Integer array of length `ORIGINALROWS` (p.**??** Xpress Optimizer Reference Manual) containing the basis status of the slack, surplus or artificial variable associated with each row. The status will be one of:

`xpress.BasisStatus.NONBASIC_LOWER (0)` slack, surplus or artificial is non-basic at lower bound;

`xpress.BasisStatus.BASIC (1)` slack, surplus or artificial is basic;

`xpress.BasisStatus.NONBASIC_UPPER (2)` slack or surplus is non-basic at upper bound.

`xpress.BasisStatus.SUPERBASIC (3)` slack or surplus is super-basic.

`colstat: list[int]` Integer array of length `ORIGINALCOLS` (p.**??** Xpress Optimizer Reference Manual) containing the basis status of the columns in the constraint matrix. The status will be one of:

`xpress.BasisStatus.NONBASIC_LOWER (0)` variable is non-basic at lower bound, or superbasic at zero if the variable has no lower bound;

`xpress.BasisStatus.BASIC (1)` variable is basic;

`xpress.BasisStatus.NONBASIC_UPPER (2)` variable is non-basic at upper bound;

`xpress.BasisStatus.SUPERBASIC (3)` variable is super-basic.

Further information

The basis information is always for the original (unpresolved) model.

Related topics

`problem.getPresolveBasis`, `problem.loadBasis`, `problem.loadPresolveBasis`.

problem.addManagedCuts

Purpose

Adds cuts to the Optimizer's internal cut pool from within the *cutround* callback set by `problem.addCutRoundCallback`. The cuts will be added to an internal pool of cuts managed by the Optimizer. The Optimizer will use internal priorities to dynamically load violated cuts from this pool into branch-and-bound node problems and remove inactive cuts. Cuts can be either *local* or *global*. Cuts flagged as *local* are assumed to be valid only for the the current node of the branch-and-bound search or any of its descendants. Global cuts are assumed to be valid for the whole problem and might be used on any node of the branch-and-bound search tree. The cuts should be formulated in the original space of variables and will automatically be presolved.

Topic areas

Callback, Cuts, Optimizer

Synopsis

```
problem.addManagedCuts(globalvalid, rowtype, rhs, start, colind, cutcoef)
```

Arguments

`globalvalid: int` Nonzero if the cuts should be assumed to be valid for the whole problem. If zero, cuts will be assumed to be valid only for the current node and its descendants.

`rowtype: list[str]` Character array containing the row types:

- `L` indicates a $\leqslant$ row;
- `G` indicates a $\geqslant$ row;
- `E` indicates an = row.

`rhs: list[float]` Double array containing the right hand side elements for the cuts.

`start: list[int]` Integer array containing offset into the `colind` and `cutcoef` arrays indicating the start of each cut. This array has one entry for each cut, plus one more entry indicating where an additional cut would start, if it were present.

`colind: list[var]` Integer array containing the column indices in the cuts.

`cutcoef: list[float]` Double array containing the matrix values for the cuts.

Further information

1. This function applies to linear and convex quadratic (including second order conic) problems only.
2. The columns and elements of the cuts must be stored contiguously in the `colind` and `cutcoef` arrays passed to `problem.addManagedCuts`. The starting point of each cut must be stored in the `start` array. To determine the length of the final cut, the `start` array must have one more element than the number of cuts, with the last element containing the position in `colind` and `cutcoef` where an extra cut would start, if it were present. This last entry denotes the number of nonzeros in the added cuts.
3. Cuts will not be loaded into the problem if `CUTSTRATEGY` (p.**??** Xpress Optimizer Reference Manual) is set to zero.
4. Cuts added by `problem.addManagedCuts` cannot be retrieved by `problem.getCutList` or `problem.getCPCutList`.
5. This function can only be called from a callback function set by `problem.addCutRoundCallback`.
6. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.addCutRoundCallback`, `problem.addCuts`, `CUTSTRATEGY` (p.**??** Xpress Optimizer Reference Manual), the Optimizer Reference Manual.

problem.addCuts

Purpose

Adds cuts directly to the matrix at the current node. The cuts will automatically be added to the cut pool. Cuts added to a node will automatically be inherited on any descendant node, unless explicitly deleted with a call to `problem.delCuts`.

Topic areas

Cuts, Optimizer

Synopsis

```
problem.addCuts(cuttype, rowtype, rhs, start, colind, cutcoef)
```

Arguments

`cuttype: list[int]` Integer array containing the user assigned cut types. The cut types can be any integer chosen by the user, and are used to identify the cuts in other cut manager routines using user supplied parameters. The cut type can be interpreted as an integer or a bitmap - see `problem.delCuts`.

`rowtype: list[str]` Character array containing the row types:

- `L` indicates a $\leqslant$ row;
- `G` indicates a $\geqslant$ row;
- `E` indicates an = row.

`rhs: list[float]` Double array containing the right hand side elements for the cuts.

`start: list[int]` Integer array containing offset into the `colind` and `cutcoef` arrays indicating the start of each cut. This array has one entry for each cut, plus one more entry indicating where an additional cut would start, if it were present.

`colind: list[var]` Integer array containing the column indices in the cuts.

`cutcoef: list[float]` Double array containing the matrix values for the cuts.

Further information

1. The columns and elements of the cuts must be stored contiguously in the `colind` and `cutcoef` arrays passed to `problem.addCuts`. The starting point of each cut must be stored in the `start` array. To determine the length of the final cut, the `start` array must have one more element than the number of cuts, with the last element containing the position in `colind` and `cutcoef` where an extra cut would start, if it were present. This last entry denotes the number of nonzeros in the added cuts.
2. The cuts added to the matrix are always added at the end of the matrix and the number of rows is always set to the original number rows plust the number of cuts added. If `n` cuts have been added, then the rows `0`,...,`ROWS` (p.**??** Xpress Optimizer Reference Manual)`-n-1` are the original rows, whilst the rows `ROWS-n`,...,`ROWS-1` are the added cuts. The number of cuts can be found by consulting the `CUTS` (p.**??** Xpress Optimizer Reference Manual) problem attribute.
3. This function should be called only from within callback functions set by either `problem.addNodeLPSolvedCallback`, `problem.addOptnodeCallback`, `problem.addPreIntsolCallback` or `problem.addPrenodeCallback`.
4. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.addRows`, `problem.delCPCuts`, `problem.delCuts`, `problem.getCPCutList`, `problem.getCutList`, `problem.loadCuts`, `problem.storeCuts`, the Optimizer Reference Manual.

problem.delCuts

Purpose

Deletes cuts from the matrix at the current node. Cuts from the parent node which have been automatically restored may be deleted as well as cuts added to the current node using `problem.addCuts` or `problem.loadCuts`. The cuts to be deleted can be specified in a number of ways. If a cut is ruled out by any one of the criteria it will not be deleted.

Topic areas

Cuts, Optimizer

Synopsis

```
problem.delCuts(basis, cuttype=0, interp=-1, delta=-xpress.infinity, cutind=None)
```

Arguments

`basis: int` Ensures the basis will be valid if set to `1`. If set to `0`, cuts with non-basic slacks may be deleted.

`cuttype: int` User defined type of the cut to be deleted.

`interp: int` Way in which the cut `cuttype` is interpreted:

- `-1` match all cut types;
- `1` treat cut types as numbers;
- `2` treat cut types as bit-vectors (compare the Optimizer Reference Manual) - delete if any bit matches any bit set in `cuttype`;
- `3` treat cut types as bit-vectors (compare the Optimizer Reference Manual) - delete if all bits match those set in `cuttype`.

`delta: float` Only delete cuts with an absolute slack value greater than `delta`. To delete all the cuts, this argument should be set to `-xpress.infinity`.

`cutind: list[poolcut]` Array containing the cuts which are to be deleted. This array may be `None` to delete all cuts.

Further information

1. It is usually best to drop only those cuts with basic slacks, otherwise the basis will no longer be valid and it may take many iterations to recover an optimal basis. If the `basis` parameter is set to `1`, this will ensure that cuts with non-basic slacks will not be deleted even if the other parameters specify that these cuts should be deleted. It is highly recommended that the `basis` parameter is always set to `1`.
2. The cuts to be deleted can also be specified by the size of the slack variable for the cut. Only those cuts with a slack value greater than the `delta` parameter will be deleted.
3. A list of indices of the cuts to be deleted can also be provided. The list of active cuts at a node can be obtained with the `problem.getCutList` command.
4. This function should be called only from within callback functions set by either `problem.addOptnodeCallback`, `problem.addPrenodeCallback`, `problem.addPreIntsolCallback` or `problem.addNodeLPSolvedCallback`.

Related topics

`problem.addCuts`, `problem.delCPCuts`, `problem.getCutList`, `problem.loadCuts`, the Optimizer Reference Manual.

problem.delCPCuts

Purpose

During the branch and bound search, cuts are stored in the cut pool to be applied at descendant nodes. These cuts may be removed from a given node using `problem.delCuts`, but if this is to be applied in a large number of cases, it may be preferable to remove the cut completely from the cut pool. This is achieved using `problem.delCPCuts`.

Topic areas

Cuts, Optimizer

Synopsis

```
problem.delCPCuts(cuttype=0, interp=-1, cutind=None)
```

Arguments

`cuttype: int` User defined cut type to match against.

`interp: int` Way in which the cut `cuttype` is interpreted:

- `-1` match all cut types;
- `1` treat cut types as numbers;
- `2` treat cut types as bit-vectors (compare the Optimizer Reference Manual) - delete if any bit matches any bit set in `cuttype`;
- `3` treat cut types as bit-vectors (compare the Optimizer Reference Manual) - delete if all bits match those set in `cuttype`.

`cutind: list[poolcut]` Array containing the cuts which are to be deleted. This array may be `None` to delete all cuts.

Related topics

`problem.addCuts`, `problem.delCuts`, `problem.loadCuts`, the Optimizer Reference Manual.

problem.getCutList

Purpose

Retrieves a list of cut objects for the cuts active at the current node.

Topic areas

Cuts, Optimizer

Synopsis

```
cutind = problem.getCutList(cuttype=0, interp=-1)
```

Arguments

`cuttype: int` User defined type of the cuts to be returned. A value of `-1` indicates return all active cuts.

`interp: int` Way in which the cut type is interpreted:

`-1`	get all cuts;
`1`	treat cut types as numbers;
`2`	treat cut types as bit-vectors (compare the Optimizer Reference Manual) - get cut if any bit matches any bit set in `cuttype`;
`3`	treat cut types as bit-vectors (compare the Optimizer Reference Manual) - get cut if all bits match those set in `cuttype`.

Return value

`cutind: list[poolcut]` Array containing the cut objects.

Related topics

`problem.getCPCutList`, `problem.getCPCuts`, the Optimizer Reference Manual.

problem.getCPCutList

Purpose

Returns a list of cut indices from the cut pool.

Topic areas

Cuts, Optimizer

Synopsis

```
cutind, viol = problem.getCPCutList(cuttype=0, interp=-1, delta=-xpress.infinity)
```

Arguments

`cuttype: int` The user defined type of the cuts to be returned.

`interp: int` Way in which the cut type is interpreted:

`-1`	get all cuts;
`1`	treat cut types as numbers;
`2`	treat cut types as bit-vectors (compare the Optimizer Reference Manual) - get cut if any bit matches any bit set in `cuttype`;
`3`	treat cut types as bit-vectors (compare the Optimizer Reference Manual) - get cut if all bits match those set in `cuttype`.

`delta: float` Only those cuts with a signed violation greater than delta will be returned.

Return value

`cutind: list[poolcut]` Array containing the cut objects.

`viol: list[float]` Double array containing the values of the signed violations of the cuts.

Further information

1. The violated cuts can be obtained by setting the `delta` parameter to the size of the (signed) violation required. If unviolated cuts are required as well, `delta` may be set to `-xpress.infinity` which is defined in the library header file.
2. In case of a cut of type 'L', the violation equals the negative of the slack associated with the row of the cut. In case of a cut of type 'G', the violation equals the slack associated with the row of the cut. For cuts of type 'E', the violation equals the absolute value of the slack.
3. Please note that the violations returned are absolute violations, while feasibility is checked by the Optimizer in the scaled problem.

Related topics

`problem.delCPCuts`, `problem.getCPCuts`, `problem.getCutList`, `problem.loadCuts`, `problem.getCutMap`, `problem.getCutSlack`, the Optimizer Reference Manual.

problem.getCPCuts

Purpose

Returns cuts from the cut pool.

Topic areas

Cuts, Optimizer

Synopsis

```
cuttype, rowtype, start, colind, cutcoef, rhs = problem.getCPCuts(rowind)
```

Argument

`rowind: list[poolcut]` Array containing the cut objects.

Return value

`cuttype: list[int]` Integer array containing the cut types.

`rowtype: list[str]` Character array containing the sense of the cuts (`L`, `G`, or `E`).

`start: list[int]` Integer array containing the offsets into the `colind` and `cutcoef` arrays. The last element indicates where an additional cut would start, if it were present.

`colind: list[int]` Integer array containing the column indices of the cuts.

`cutcoef: list[float]` Double array containing the matrix values.

`rhs: list[float]` Double array containing the right hand side elements for the cuts.

Related topics

`problem.getCPCutList`, `problem.getCutList`, the Optimizer Reference Manual.

problem.loadCuts

Purpose

Loads cuts from the cut pool into the matrix. Without calling `problem.loadCuts` the cuts will remain in the cut pool but will not be active at the node. Cuts loaded at a node remain active at all descendant nodes unless they are deleted using `problem.delCuts`.

Topic areas

Data Input, Cuts, Optimizer

Synopsis

```
problem.loadCuts(cutind, cuttype=0, interp=-1)
```

Arguments

`cutind: list[poolcut]` Array containing the cuts to be loaded into the matrix. These are objects returned by either `problem.storeCuts` or `problem.getCPCutList`.

`cuttype: int` Cut type.

`interp: int` The way in which the cut type is interpreted:

`-1`	load all cuts;
`1`	treat cut types as numbers;
`2`	treat cut types as bit-vectors (compare the Optimizer Reference Manual) - load cut if any bit matches any bit set in `cuttype`;
`3`	treat cut types as bit-vectors (compare the Optimizer Reference Manual) - `0` load cut if all bits match those set in `cuttype`.

Further information

This function should be called only from within callback functions set by either `problem.addOptnodeCallback`, `problem.addPrenodeCallback`, `problem.addNodeLPSolvedCallback` or `problem.addPreIntsolCallback`.

Related topics

`problem.addCuts`, `problem.delCPCuts`, `problem.delCuts`, `problem.getCPCutList`, the Optimizer Reference Manual.

problem.storeCuts

Purpose

Stores cuts into the cut pool, but does not apply them to the current node. These cuts must be explicitly loaded into the matrix using `problem.loadCuts` before they become active.

Topic areas

Cuts, Optimizer

Synopsis

```
cutind = problem.storeCuts(nodups, cuttype, rowtype, rhs, start, colind, cutcoef)
```

Arguments

`nodups: int` 0 do not exclude duplicates from the cut pool;
1 duplicates are to be excluded from the cut pool;
2 duplicates are to be excluded from the cut pool, ignoring cut type.

`cuttype: list[int]` Integer array containing the cut types. The cut types can be any integer and are used to identify the cuts.

`rowtype: list[str]` Character array containing the row types:
`L` indicates a $\leqslant$ row;
`E` indicates an = row;
`G` indicates a $\geqslant$ row.

`rhs: list[float]` Double array containing the right hand side elements for the cuts.

`start: list[int]` Integer array containing offsets into the `colind` and `dmtval` arrays indicating the start of each cut. This array has an entry for each cut, plus one more entry indicating where an additional cut would start, if it were present.

`colind: list[int]` Integer array containing the column indices in the cuts.

`cutcoef: list[float]` Double array containing the matrix values for the cuts.

Return value

`cutind: list[poolcut]` Array containing the cut objects.

Related controls

Double

`MATRIXTOL` Tolerance on matrix elements.

Further information

1. `problem.storeCuts` can be used to eliminate duplicate cuts. If the `nodups` parameter is set to `1`, the cut pool will be checked for duplicate cuts with a cut type identical to the cuts being added. If a duplicate cut is found the new cut will only be added if its right hand side value makes the cut stronger. If the cut in the pool is weaker than the added cut it will be removed unless it has been applied to an active node of the tree. If `nodups` is set to `2` the same test is carried out on all cuts, ignoring the cut type.
2. `problem.storeCuts` returns a list of the cuts added to the cut pool in the `cutind` array. If the cut is not added to the cut pool because a stronger cut exits a None will be returned. The `cutind` array can be passed directly to `problem.loadCuts` to load the most recently stored cuts into the matrix.
3. The columns and elements of the cuts must be stored contiguously in the `colind` and `dmtval` arrays passed to `problem.storeCuts`. The starting point of each cut must be stored in the `start` array. To determine the length of the final cut the `start` array must have one more element than the number of cuts, containing the point where an extra cut would start, if it were present.

Related topics

`problem.loadCuts` the Optimizer Reference Manual.

problem.presolveRow

Purpose

Presolves a row formulated in terms of the original variables such that it can be added to a presolved matrix.

Topic areas

Misc, Presolve, Optimizer

Synopsis

```
colind, rowcoef, rhs, status = problem.presolveRow(rowtype, origcolind, origrowcoef,
       origrhs)
```

Arguments

`rowtype: str` The type of the row:
- `L` indicates a $\leqslant$ row;
- `G` indicates a $\geqslant$ row.
- `E` indicates a $\geqslant$ row.

`origcolind: list[var]` Integer array containing the column indices of the row to presolve.

`origrowcoef: list[float]` Double array containing the non-zero coefficients of the row to presolve.

`origrhs: float` The right-hand side constant of the row to presolve.

Return value

`colind: list[int]` Integer array containing the column indices of the presolved row.
Allocate this to `COLS` (p.**??** Xpress Optimizer Reference Manual) elements and pass `COLS` (p.**??** Xpress Optimizer Reference Manual) as `maxcoef>` to make sure it is large enough to hold the presolved row.

`rowcoef: list[float]` Double array containing the coefficients of the presolved row.
Allocate this to `COLS` (p.**??** Xpress Optimizer Reference Manual) elements and pass `COLS` (p.**??** Xpress Optimizer Reference Manual) as `maxcoef>` to make sure it is large enough to hold the presolved row.

`rhs: float` The presolved right-hand side.

`status: int` Status of the presolved row:
- `-5` Failed to presolve the row due to presolve operations making the row nonlinear;
- `-4` Failed to presolve the equality row due to presolve operations requiring relaxation of the row;
- `-3` Failed to presolve the row due to presolve dual reductions;
- `-2` Failed to presolve the row due to presolve duplicate column reductions;
- `-1` Failed to presolve the row due to an error. Check the Optimizer error code for the cause;
- `0` The row was successfully presolved;
- `1` The row was presolved, but may be relaxed.

Related controls

Integer

PRESOLVE	Turns presolve on or off.
PRESOLVEOPS	Selects the presolve operations.

Further information

1. There are certain presolve operations that can prevent a row from being presolved exactly. If the row contains a coefficient for a column that was eliminated due to duplicate column reductions or singleton column reductions, the row might have to be relaxed to remain valid for the presolved problem. The relaxation will be done automatically by the `problem.presolveRow` function, but a return status of `+1` will be returned. If it is not possible to relax the row, a status of `-2` will be returned instead. Likewise, it is possible that certain dual reductions prevents the row from being presolved. In such a case a status of `-3` will be returned instead. Also transforming a convex quadratic objective transfer row to a convex quadratic objective and nonlinear eliminations will lead to a row not being presolved and return a status of `-5` instead.
2. If `problem.presolveRow` will be used for presolving hard constraints, such as e.g. branching bounds or delayed constraints, then dual reductions should be disabled, by setting `MIPDUALREDUCTIONS` (p.**??** Xpress Optimizer Reference Manual) to 0.
3. If the user knows in advance which columns will have non-zero coefficients in rows that will be presolved, it is possible to protect these individual columns through the `problem.loadSecureVecs` function. This way the Optimizer is left free to apply all possible reductions to the remaining columns.
4. Argument `origcolind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.addCuts`, `problem.loadSecureVecs`, `problem.storeCuts`.

problem.presolveSol

Purpose

Presolves a primal solution formulated in the input space into the corresponding solution formulated in the presolved space. The problem itself is unchanged.

Topic areas

Solution, Presolve, Optimizer

Synopsis

```
prex = problem.presolveSol(origx)
```

Argument

`origx: list[float]` Double array of length `INPUTCOLS` (p.**??** Xpress Optimizer Reference Manual) with the values of the primal variables in the input space. Cannot be `None`.

Return value

`prex: list[float]` Double array of length `COLS` (p.**??** Xpress Optimizer Reference Manual) containing the values of the primal variables.

Further information

This function is useful for advanced users implementing heuristics directly in the presolved space. Once a new solution is obtained, it needs to be mapped to the original space in order to be passed back into the solver.

Related topics

`problem.getSolution`, `problem.postSolveSol`, `problem.getCallbackSolution`, `problem.getCallbackPresolveSolution`.

problem.postSolveSol

Purpose

Postsolves a primal solution formulated in the presolved space into the corresponding solution formulated in the input space. The problem itself is unchanged.

Topic areas

Solution, Presolve, Optimizer

Synopsis

```
origx = problem.postSolveSol(prex)
```

Argument

`prex: list[float]` Double array of length `COLS` (p.**??** Xpress Optimizer Reference Manual) with the values of the primal variables in the presolved space. Cannot be `None`.

Return value

`origx: list[float]` Double array of length `INPUTCOLS` (p.**??** Xpress Optimizer Reference Manual) containing the values of the primal variables.

Further information

This function is useful for advanced users implementing heuristics directly in the presolved space. Once a new solution is obtained, it needs to be mapped to the original space in order to be passed back into the solver.

Related topics

`problem.getSolution`, `problem.presolveSol`, `problem.getCallbackSolution`, `problem.getCallbackPresolveSolution`.

problem.getPivots

Purpose

Returns a list of potential leaving variables if a specified variable enters the basis.

Topic areas

Simplex, LP, Optimizer

Synopsis

```
outlist, x, objval = problem.getPivots(enter)
```

Argument

`enter: var | constraint` Index of the specified row or column to enter basis.

Return value

`outlist: list[int]` Integer array containing list of potential leaving variables.

`x: list[float]` Double array of length `ROWS` (p.**??** Xpress Optimizer Reference Manual)+`SPAREROWS` (p.**??** Xpress Optimizer Reference Manual)+`COLS` (p.**??** Xpress Optimizer Reference Manual) containing the values of all the variables that would result if `enter` entered the basis.

`objval: float` The objective function value that would result if `enter` entered the basis.

Error value

`425` Indicates `enter` is invalid (out of range or already basic).

Further information

1. If the variable `enter` enters the basis and the problem is degenerate then several basic variables are candidates for leaving the basis. If variable `enter` were to be pivoted in, then because the problem is degenerate, the resulting values of the objective function and all the variables do not depend on which of the candidates from `outlist` is chosen to leave the basis.

   The value of the objective is returned in `objval` and the values of the variables into `x`.

2. Row indices are in the range `0` to `ROWS` (p.**??** Xpress Optimizer Reference Manual)`-1`, whilst columns are in the range `ROWS+SPAREROWS` (p.**??** Xpress Optimizer Reference Manual) to `ROWS+SPAREROWS+COLS` (p.**??** Xpress Optimizer Reference Manual)`-1`.

3. Argument `enter` may contain either `xpress.var` or `xpress.constraint` objects or integer indices.

Related topics

`problem.getPivotOrder`, `problem.pivot`.

problem.writeProb

Purpose

Writes the current problem to an MPS or LP file.

Topic areas

File IO, Optimizer

Synopsis

```
problem.writeProb(filename="", flags="")
```

Arguments

`filename: str` A string of up to `MAXPROBNAMELENGTH` (p.**??** Xpress Optimizer Reference Manual) characters to contain the file name to which the problem is to be written. If omitted, the default *problem_name* is used with a `.mps` extension, unless the `l` flag is used in which case the extension is `.lp`.

`flags: str` Flags, which can be one or more of the following:

- `o` one element per line;
- `n` output the scaled problem;
- `s` scrambled vector names;
- `l` output in LP format;
- `p` output values in full precision (obsolete as this is now default behavior);
- `t` omit the Xpress header in LP or MPS format;
- `v` use the provided filename verbatim, without appending the `.mps` or `.lp` extension;
- `z` compress the output file.

Further information

1. If `problem.loadLP`, `problem.loadMIP`, `problem.loadMIQP` or `problem.loadQP` is used to obtain a matrix then there is no association between the objective function and the `N` rows in the matrix and so a separate `N` row (called `__OBJ___`) is created when you do an `problem.writeProb`. Also if you do an `problem.readProb` and then change either the objective row or the `N` row in the matrix corresponding to the objective row, you lose the association between the two and the `__OBJ___` row is created when you do an `problem.writeProb`. To remove the objective row from the matrix when doing an `problem.readProb`, set `KEEPNROWS` (p.**??** Xpress Optimizer Reference Manual) to `-1` before `problem.readProb`.
2. **Warning:** If `problem.readProb` is used to input a problem, then the input file will be overwritten by `problem.writeProb` if a new filename is not specified.
3. The Optimizer can write compressed matrix files in the following formats, as long as the command-line tool necessary to compress the file can be located in the path: `bzip2`, `xz`, `lzma`, `Z`, `zip`, `tar`, `tgz`. The Optimizer has built-in support for the `gz` format: no external tools are necessary to write gzipped matrix files.

Related topics

`problem.readProb`.

problem.calcSlacks

Purpose

Calculates the row slack values for a given solution.

Topic areas

Solution, Optimizer

Synopsis

```
slacks = problem.calcSlacks(solution)
```

Argument

`solution: list[float]` Double array of length `ORIGINALCOLS` that holds the solution to calculate the slacks for.

Return value

`slacks: list[float]` Double array of length `ORIGINALROWS` containing the calculated row slacks.

Further information

1. This function applies to linear and convex quadratic (including second order conic) problems only. For nonlinear, please use `problem.nlpCalcSlacks`.
2. The calculations are always carried out in the original problem, even if the problem is currently presolved.
3. If using the function during a solve (e.g. from a callback), use `ORIGINALCOLS` (p.**??** Xpress Optimizer Reference Manual) and `ORIGINALROWS` (p.**??** Xpress Optimizer Reference Manual) to retrieve the non-presolved dimensions of the problem.

Related topics

`problem.calcReducedCosts`, `problem.calcSolInfo`, `problem.calcObjective`, `problem.nlpCalcSlacks`.

problem.calcReducedCosts

Purpose

Calculates the reduced cost values for a given (row) dual solution.

Topic areas

Linear Algebra, Solution, Optimizer

Synopsis

```
djs = problem.calcReducedCosts(duals, solution=None)
```

Arguments

`duals: list[float]` Double array of length `ORIGINALROWS` that holds the dual solution to calculate the reduced costs for.

`solution: list[float]` Double array of length `ORIGINALCOLS` that holds the primal solution. This is necessary for quadratic problems, otherwise may be `None`.

Return value

`djs: list[float]` Double array of length `ORIGINALCOLS` containing the calculated reduced costs.

Further information

1. This function applies to linear and convex quadratic (including second order conic) problems only.
2. The calculations are always carried out in the original problem, even if the problem is currently presolved.
3. If using the function during a solve (e.g. from a callback), use `ORIGINALCOLS` (p.**??** Xpress Optimizer Reference Manual) and `ORIGINALROWS` (p.**??** Xpress Optimizer Reference Manual) to retrieve the non-presolved dimensions of the problem.

Related topics

`problem.calcSlacks`, `problem.calcSolInfo`, `problem.calcObjective`.

problem.calcObjective

Purpose

Calculates the objective value of a given solution.

Topic areas

Solution, Optimizer

Synopsis

```
objval = problem.calcObjective(solution)
```

Argument

`solution: list[float]` Double array of length COLS that holds the solution.

Return value

`objval: float` The calculated objective value.

Further information

The calculations are always carried out in the original problem, even if the problem is currently presolved.

Related topics

`problem.calcSlacks`, `problem.calcSolInfo`, `problem.calcReducedCosts`.

problem.calcObjN

Purpose

Calculates the objective value of the given objective function in a multi-objective problem.

Topic areas

Solution, Multiobjective, Optimizer

Synopsis

```
objval = problem.calcObjN(objidx, solution=None)
```

Arguments

`objidx: int` Index of the objective function to calculate.

`solution: list[float]` Double array of length `COLS` that holds the solution. May be `None` to use the current solution.

Return value

`objval: float` The calculated objective value.

Related topics

`problem.calcObjective`.

problem.calcSolInfo

Purpose

Calculates the required property of a solution, like maximum infeasibility of a given primal and dual solution.

Topic areas

Solution, Optimizer

Synopsis

```
value = problem.calcSolInfo(property, solution=None, duals=None)
```

Arguments

`property: int` Defined the property to be calculated.

- `xpress.SolInfo.ABSPRIMALINFEAS` the calculated maximum absolute primal infeasibility is returned.
- `xpress.SolInfo.RELPRIMALINFEAS` the calculated maximum relative primal infeasibility is returned.
- `xpress.SolInfo.ABSDUALINFEAS` the calculated maximum absolute dual infeasibility is returned.
- `xpress.SolInfo.RELDUALINFEAS` the calculated maximum relative dual infeasibility is returned.
- `xpress.SolInfo.MAXMIPFRACTIONAL` the calculated maximum absolute MIP fractionality or SOS infeasibility.
- `xpress.SolInfo.ABSMIPINFEAS` the calculated maximum absolute MIP infeasibility (including delayed rows, indicators, general and piecewise linear constraints) is returned.
- `xpress.SolInfo.RELMIPINFEAS` the calculated maximum relative MIP infeasibility (including delayed rows, indicators, general and piecewise linear constraints) is returned.

`solution: list[float]` Double array of length `ORIGINALCOLS` that holds the solution. May be `None` when asking for dual infeasibility.

`duals: list[float]` Double array of length `ORIGINALROWS` that holds the dual solution. May be `None` when asking for primal/MIP infeasibility.

Return value

`value: float` The calculated value.

Further information

1. This function applies to linear and convex quadratic (including second order conic) problems only.
2. The calculations are always carried out in the original problem, even if the problem is currently presolved.

Related topics

`problem.calcSlacks`, `problem.calcObjective`, `problem.calcReducedCosts`.

problem.getRowType

Purpose

Returns the row types for the rows in a given range.

Topic areas

Problem Information, Optimizer

Synopsis

```
rowtype = problem.getRowType(first=0, last=ROWS-1)
```

Arguments

`first: int` First row in the range.

`last: int` Last row in the range.

Return value

`rowtype: list[str]` Character array of length `last-first+1` characters containing the row types:

- `N` indicates a free constraint;
- `L` indicates a $\leqslant$ constraint;
- `E` indicates an = constraint;
- `G` indicates a $\geqslant$ constraint;
- `R` indicates a range constraint.

Further information

Arguments `first, last` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.chgRowType`, `problem.getRows`.

problem.getPresolveBasis

Purpose

Returns the current basis from memory into the user's data areas. If the problem is presolved, the presolved basis will be returned. Otherwise the original basis will be returned.

Topic areas

Simplex, LP, Optimizer

Synopsis

```
rowstat, colstat = problem.getPresolveBasis()
```

Return value

`rowstat: list[int]` Integer array of length `ROWS` (p.**??** Xpress Optimizer Reference Manual) containing the basis status of the stack, surplus or artificial variable associated with each row. The status will be one of:

`xpress.BasisStatus.NONBASIC_LOWER (0)` slack, surplus or artificial is non-basic at lower bound;

`xpress.BasisStatus.BASIC (1)` slack, surplus or artificial is basic;

`xpress.BasisStatus.NONBASIC_UPPER (2)` slack or surplus is non-basic at upper bound.

`colstat: list[int]` Integer array of length `COLS` (p.**??** Xpress Optimizer Reference Manual) containing the basis status of the columns in the constraint matrix. The status will be one of:

`xpress.BasisStatus.NONBASIC_LOWER (0)` variable is non-basic at lower bound, or superbasic at zero if the variable has no lower bound;

`xpress.BasisStatus.BASIC (1)` variable is basic;

`xpress.BasisStatus.NONBASIC_UPPER (2)` variable is at upper bound;

`xpress.BasisStatus.SUPERBASIC (3)` variable is super-basic.

Related topics

`problem.getBasis`, `problem.loadBasis`, `problem.loadPresolveBasis`.

problem.getColType

Purpose

Returns the column types for the columns in a given range.

Topic areas

Problem Information, MIP Entities, Optimizer

Synopsis

```
coltype = problem.getColType(first=0, last=COLS-1)
```

Arguments

`first: int` First column in the range.

`last: int` Last column in the range.

Return value

`coltype: list[str]` Character array of length `last-first+1` containing the column types:

- `C` indicates a continuous variable;
- `I` indicates an integer variable;
- `B` indicates a binary variable;
- `S` indicates a semi-continuous variable;
- `R` indicates a semi-continuous integer variable;
- `P` indicates a partial integer variable.

Further information

Arguments `first, last` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.chgColType`, `problem.getRowType`.

problem.getQRowCoeff

Purpose

Returns a single quadratic constraint coefficient corresponding to the variable pair (`rowqcol1`, `rowqcol2`) of the Hessian of a given constraint.

Topic areas

Problem Information, Quadratic, Optimizer

Synopsis

```
rowqcoef = problem.getQRowCoeff(row, rowqcol1, rowqcol2)
```

Arguments

`row: constraint` The quadratic row where the coefficient is to be looked up.

`rowqcol1: var` Column index for the first variable in the quadratic term.

`rowqcol2: var` Column index for the second variable in the quadratic term.

Return value

`rowqcoef: float` The objective function coefficient.

Further information

1. The coefficient returned corresponds to the Hessian of the constraint. That means the for constraint `x + [x^2 + 6 xy] <= 10` `problem.getQRowCoeff` would return `1` as the coefficient of `x^2` and `3` as the coefficient of `xy`.
2. Argument `row` may contain either `xpress.constraint` objects or integer indices.
3. Arguments `rowqcol1, rowqcol2` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.loadQCQP`, `problem.addQMatrix`, `problem.chgQRowCoeff`, `problem.getQRowQMatrix`, `problem.getQRowQMatrixTriplets`, `problem.getQRows`, `problem.chgQObj`, `problem.chgMQObj`, `problem.getQObj`.

problem.getQRowQMatrix

Purpose

Returns the nonzeros in a quadratic constraint coefficients matrix for the columns in a given range. To achieve maximum efficiency, `problem.getQRowQMatrix` returns the lower triangular part of this matrix only.

Topic areas

Problem Information, Quadratic, Optimizer

Synopsis

```
start, colind, rowqcoef = problem.getQRowQMatrix(row, first=0, last=COLS-1)
```

Arguments

`row: constraint` Index of the row for which the quadratic coefficients are to be returned.

`first: int` First column in the range.

`last: int` Last column in the range.

Return value

`start: list[int]` Integer array containing indices indicating the starting offsets in the `colind` and `rowqcoef` arrays for each requested column. It must be length of at least `last-first+2`. Column `i` starts at position `start[i]` in the `colind` and `rowqcoef` arrays, and has `start[i+1]-start[i]` elements in it.

`colind: list[int]` Integer array containing the column indices of the nonzero elements in the lower triangular part of Q.

`rowqcoef: list[float]` Double array containing the nonzero element values.

Further information

1. Argument `row` may contain either `xpress.constraint` objects or integer indices.
2. Arguments `first, last` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.loadQCQP`, `problem.getQRowCoeff`, `problem.addQMatrix`, `problem.chgQRowCoeff`, `problem.getQRowQMatrixTriplets`, `problem.getQRows`, `problem.chgQObj`, `problem.chgMQObj`, `problem.getQObj`.

problem.getQRowQMatrixTriplets

Purpose

Returns the nonzeros in a quadratic constraint coefficients matrix as triplets (index pairs with coefficients). To achieve maximum efficiency, `problem.getQRowQMatrixTriplets` returns the lower triangular part of this matrix only.

Topic areas

Problem Information, Quadratic, Optimizer

Synopsis

```
rowqcol1, rowqcol2, rowqcoef = problem.getQRowQMatrixTriplets(row)
```

Argument

`row: constraint` Index of the row for which the quadratic coefficients are to be returned.

Return value

`rowqcol1: list[int]` First index in the triplets.

`rowqcol2: list[int]` Second index in the triplets.

`rowqcoef: list[float]` Coefficients in the triplets.

Further information

1. If a row index of `-1` is used, the function returns the quadratic coefficients for the objective function.
2. Argument `row` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.loadQCQP`, `problem.getQRowCoeff`, `problem.addQMatrix`, `problem.chgQRowCoeff`, `problem.getQRowQMatrix`, `problem.getQRows`, `problem.chgQObj`, `problem.chgMQObj`, `problem.getQObj`.

problem.chgQRowCoeff

Purpose

Changes a single quadratic coefficient in a row.

Topic areas

Problem Modification, Quadratic, Optimizer

Synopsis

```
problem.chgQRowCoeff(row, rowqcol1, rowqcol2, rowqcoef)
```

Arguments

`row: constraint` Index of the row where the quadratic matrix is to be changed.

`rowqcol1: var` First index of the coefficient to be changed.

`rowqcol2: var` Second index of the coefficient to be changed.

`rowqcoef: float` The new coefficient.

Further information

1. This function may be used to add new nonzero coefficients, or even to define the whole quadratic expression with it. Doing that, however, is significantly less efficient than adding the whole expression with `problem.addQMatrix`.
2. Argument `row` may contain either `xpress.constraint` objects or integer indices.
3. Arguments `rowqcol1, rowqcol2` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.loadQCQP`, `problem.getQRowCoeff`, `problem.addQMatrix`, `problem.chgQRowCoeff`, `problem.getQRowQMatrix`, `problem.getQRowQMatrixTriplets`, `problem.getQRows`, `problem.chgQObj`, `problem.chgMQObj`, `problem.getQObj`.

problem.getQRows

Purpose

Returns the list indices of the rows that have quadratic coefficients.

Topic areas

Problem Information, Quadratic, Optimizer

Synopsis

```
rowind = problem.getQRows()
```

Return value

`rowind: list[int]` Array of length `QCONSTRAINTS` containing the indices of rows with quadratic coefficients in them.

Related topics

`problem.loadQCQP`, `problem.getQRowCoeff`, `problem.addQMatrix`, `problem.chgQRowCoeff`, `problem.getQRowQMatrix`, `problem.getQRowQMatrixTriplets`, `problem.chgQObj`, `problem.chgMQObj`, `problem.getQObj`.

problem.chgBounds

Purpose

Used to change the bounds on columns in the matrix.

Topic areas

Problem Modification, Optimizer

Synopsis

```
problem.chgBounds(colind, bndtype, bndval)
```

Arguments

`colind: list[var]` Integer array containing the indices of the columns on which the bounds will change.

`bndtype: list[str]` Character array indicating the type of bound to change:

`U`	indicates change the upper bound;
`L`	indicates change the lower bound;
`B`	indicates change both bounds, i.e. fix the column.

`bndval: list[float]` Double array giving the new bound values.

Further information

1. A column index may appear twice in the `colind` array so it is possible to change both the upper and lower bounds on a variable in one go.
2. `problem.chgBounds` can be used from within callbacks to modify the local node bounds of a branch-and-bound search. In this case, the solver expects column indices to match the presolved problem. This function can be called from within the following callbacks: `problem.addPrenodeCallback`, `problem.addNodeLPSolvedCallback`, `problem.addOptnodeCallback`.
3. The double constants `xpress.infinity` and `-xpress.infinity` defined in the library header file can be used to represent plus and minus infinity respectively in the bound (`bndval`) array.
4. If the upper bound on a binary variable is changed to be greater than 1 or the lower bound is changed to be less than 0 then the variable will become an integer variable.
5. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.getLB`, `problem.getUB`,

problem.triggerRestart

Purpose

Triggers a restart of the MIP search process.

Topic areas

Solution Process, Presolve, Optimizer

Synopsis

```
status = problem.triggerRestart()
```

Return value

`status: int` Memory to return whether the restart request was accepted (0) or denied (1).

Related controls

Integer

MIPRESTART	Control strategy for in-tree restarts.
ROOTPRESOLVE	Control strategy for root restarts.

Further information

This function requests a restart of the MIP search process. The request can be rejected if MIP restarts are not supported, e.g., in case the model has quadratic or nonlinear constraints, or if restarts are disabled by the user via the MIPRESTART (p.**??** Xpress Optimizer Reference Manual) control.

Related topics

problem.mipOptimize

problem.getNameList

Purpose

Returns the names for the rows, columns, sets, piecewise linear constraints, general constraints or objectives in a given range.

Topic areas

Names Manager, Optimizer

Synopsis

```
names = problem.getNameList(type, first=0, last=0)
```

Arguments

`type: int` `xpress.Namespaces.ROW` `(=1)` if row names are required;
`xpress.Namespaces.COLUMN` `(=2)` if column names are required;
`xpress.Namespaces.SET` `(=3)` if set names are required;
`xpress.Namespaces.PWLCONS` `(=4)` if piecewise linear constraint names are required;
`xpress.Namespaces.GENCONS` `(=5)` if general constraint names are required;
`xpress.Namespaces.OBJECTIVE` `(=6)` if objective function names are required.

`first: int` First row, column, set, piecewise linear or general constraint in the range.

`last: int` Last row, column, set, piecewise linear or general constraint in the range.

Return value

`names: list[str]` The names.

Further information

The names will be returned as a list of strings.

Related topics

`problem.addNames`.

problem.addMipSol

Purpose

Adds a new feasible, infeasible or partial MIP solution for the problem to the Optimizer.

Topic areas

Solution, Optimizer

Synopsis

```
problem.addMipSol(solval, colind=None, name="")
```

Arguments

`solval: list[float]` Double array containing solution values.

`colind: list[var]` Optional integer array containing column indices from `0` to `INPUTCOLS` (p.**??** Xpress Optimizer Reference Manual)`-1`, for the solution values provided in `solval`. Duplicate entries are not allowed. May be `None` when `solval` provides a complete solution vector, in which case `solval` should have `INPUTCOLS` (p.**??** Xpress Optimizer Reference Manual) elements.

`name: str` An optional name to associate with the solution. Can be `None`.

Related controls

Integer

`USERSOLHEURISTIC` Controls the local search heuristic for an infeasible or partial solution.

Further information

1. The function returns immediately after passing the solution to the Optimizer. The solution is placed in a pool until the Optimizer is able to analyze the solution during a MIP solve.
2. If the provided solution is found to be infeasible, a limited local search heuristic will be run in an attempt to find a close feasible integer solution.
3. If a partial solution is provided, integer columns will be fixed to any provided values and a limited local search will be run in an attempt to find integer feasible values for the remaining unspecified columns. Values provided for continuous column in partial solutions are currently ignored.
4. The `problem.addUserSolNotifyCallback` callback function can be used to discover the outcome of a loaded solution. The optional name provided as `name` will be returned in the callback function.
5. If one or more solutions are loaded during the `problem.addOptnodeCallback` callback, the Optimizer will process all loaded solutions and fire the callback again. This will be repeated as long as new solutions are loaded during the callback. You can check the `CALLBACKCOUNT_OPTNODE` (p.**??** Xpress Optimizer Reference Manual) attribute if you only want to run once.
6. The provided solution must be in the input space.
7. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.addUserSolNotifyCallback`, `problem.addOptnodeCallback`.

problem.getCutSlack

Purpose

Used to calculate the slack value of a cut with respect to the current LP relaxation solution. The slack is calculated from the cut itself, and might be requested for any cut (even if it is not currently loaded into the problem).

Topic areas

Cuts, Optimizer

Synopsis

```
slack = problem.getCutSlack(cutind)
```

Argument

`cutind: poolcut` Cut object for which the slack is to be calculated.

Return value

`slack: float` The slack.

Related topics

`problem.getCPCutList`, `problem.delCPCuts`, `problem.getCutList`, `problem.loadCuts`, `problem.getCutMap`, `problem.getCPCuts`, the Optimizer Reference Manual.

problem.getCutMap

Purpose

Used to return in which rows a list of cuts are currently loaded into the Optimizer. This is useful for example to retrieve the duals associated with active cuts.

Topic areas

Cuts, Optimizer

Synopsis

```
cutmap = problem.getCutMap(cutind)
```

Argument

`cutind: list[poolcut]` Array containing the cut objects for which the row index is requested.

Return value

`cutmap: list[int]` Integer array containing the row indices.

Further information

For cuts currently not loaded into the problem, a row index of `-1` is returned.

Related topics

`problem.getCPCutList`, `problem.delCPCuts`, `problem.getCutList`, `problem.loadCuts`, `problem.getCutSlack`, `problem.getCPCuts`, the Optimizer Reference Manual.

problem.getPresolveSol

Purpose

Returns the solution for the presolved problem from memory.

Topic areas

Solution, Presolve, Optimizer

Synopsis

```
x, slack, duals, djs = problem.getPresolveSol()
```

Return value

`x: list[float]` Double array of length `COLS` (p.**??** Xpress Optimizer Reference Manual) containing the values of the primal variables.

`slack: list[float]` Double array of length `ROWS` (p.**??** Xpress Optimizer Reference Manual) containing the values of the slack variables.

`duals: list[float]` Double array of length `ROWS` containing the values of the dual variables.

`djs: list[float]` Double array of length `COLS` containing the reduced cost for each variable.

Further information

1. In callbacks, it is recommended to use `problem.getCallbackPresolveSolution` instead of `problem.getPresolveSol`.
2. This function applies to linear and convex quadratic (including second order conic) problems only.
3. If the problem has not been presolved, the solution in memory will be returned.
4. The solution to the original problem should be returned using the related function `problem.getLpSol`.
5. If called during a callback during MIP tree search the solution of the current node will be returned.
6. When an integer solution is found during a tree search, it is always set up as a solution to the current node; therefore the integer solution is available as the current node solution and can be retrieved with `problem.getLpSol` and `problem.getPresolveSol`.

Related topics

`problem.getSolution`, `problem.getCallbackSolution`, `problem.getLpSol`, the Optimizer Reference Manual, `problem.postSolveSol`.

problem.getLpSol

Purpose

Used to obtain the LP solution values following optimization.

Topic areas

Solution, LP, Optimizer

Synopsis

```
x, slack, duals, djs = problem.getLpSol()
```

Return value

`x: list[float]` Double array of length `ORIGINALCOLS` (p.**??** Xpress Optimizer Reference Manual) containing the values of the primal variables.

`slack: list[float]` Double array of length `ORIGINALROWS` (p.**??** Xpress Optimizer Reference Manual) containing the values of the slack variables.

`duals: list[float]` Double array of length `ORIGINALROWS` containing the values of the dual variables ($c_B^T B^{-1}$).

`djs: list[float]` Double array of length `ORIGINALCOLS` containing the reduced cost for each variable ($c^T - c_B^T B^{-1} A$).

Further information

1. After solving an LP, it is recommended to use `problem.getSolution` to retrieve the solution.
2. When an integer solution is found during a tree search, it is always set up as a solution to the current node; therefore the integer solution is available as the current node. It is recommended to retrieved the solution with `problem.getCallbackSolution` and `problem.getCallbackPresolveSolution`.
3. The function always returns data in the original space, even if the problem is currently in presolved state. Use `problem.getPresolveSol` if you need the solution in terms of the presolved model.
4. If the matrix is modified after calling `problem.lpOptimize`, then the solution will no longer be available.
5. If the problem has been presolved, then `problem.getLpSol` returns the solution to the original problem. The only way to obtain the presolved solution is to call the related function, `problem.getPresolveSol`.

Related topics

`problem.getSolution`, `problem.getCallbackSolution`, `problem.getPresolveSol`, `problem.writePrtSol`, `problem.writeSol`, `problem.postSolveSol`.

problem.getCallbackPresolveSolution

Purpose

Returns the solution to the presolved problem associated with the current callback.

Topic areas

Solution, Callback, Optimizer

Synopsis

```
x = problem.getCallbackPresolveSolution(first=0, last=COLS-1)
```

Arguments

`first: int` First column in the solution to return.

`last: int` Last column in the solution to return.

Return value

`x: Optional[list[float]]` Double array of length `last-first+1` containing the values of the primal variables.

Further information

1. Column indices should be in the range `0` to `COLS` (p.**??** Xpress Optimizer Reference Manual)`-1`.
2. See `problem.getCallbackSolution` for a list of callbacks where this function can be called.
3. Arguments `first, last` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.getCallbackPresolveSlacks`, `problem.getCallbackPresolveDuals`, `problem.getCallbackPresolveRedCosts`, `problem.getCallbackSolution`, `problem.getSolution`.

problem.getCallbackPresolveSlacks

Purpose

Returns the slack values from the solution to the presolved problem associated with the current callback.

Topic areas

Solution, Callback, Optimizer

Synopsis

```
slacks = problem.getCallbackPresolveSlacks(first=0, last=ROWS-1)
```

Arguments

`first: int` First row whose slack value to return.

`last: int` Last row whose slack value to return.

Return value

`slacks: Optional[list[float]]` Double array of length `last-first+1` containing the values of the slack variables.

Further information

1. Row indices should be in the range `0` to `ROWS` (p.**??** Xpress Optimizer Reference Manual)-`1`.
2. See `problem.getCallbackSolution` for a list of callbacks where this function can be called.
3. Arguments `first, last` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.getCallbackPresolveSolution`, `problem.getCallbackPresolveDuals`, `problem.getCallbackPresolveRedCosts`, `problem.getCallbackSlacks`, `problem.getSlacks`.

problem.getCallbackPresolveDuals

Purpose

Returns the dual values from the solution to the presolved problem associated with the current callback.

Topic areas

Solution, Callback, Optimizer

Synopsis

```
duals = problem.getCallbackPresolveDuals(first=0, last=ROWS-1)
```

Arguments

`first: int` First row whose dual value to return.

`last: int` Last row whose dual value to return.

Return value

`duals: Optional[list[float]]` Double array of length `last-first+1` containing the values of the dual variables.

Further information

1. Row indices should be in the range `0` to `ROWS` (p.**??** Xpress Optimizer Reference Manual)`-1`.
2. See `problem.getCallbackSolution` for a list of callbacks where this function can be called. Note that a dual solution is not available in all contexts where a primal solution is available.
3. Arguments `first, last` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.getCallbackPresolveSolution`, `problem.getCallbackPresolveSlacks`, `problem.getCallbackPresolveRedCosts`, `problem.getCallbackDuals`, `problem.getDuals`.

problem.getCallbackPresolveRedCosts

Purpose

Returns the reduced costs from the solution to the presolved problem associated with the current callback.

Topic areas

Solution, Callback, Optimizer

Synopsis

```
djs = problem.getCallbackPresolveRedCosts(first=0, last=COLS-1)
```

Arguments

`first: int` First column whose reduced cost to return.

`last: int` Last column whose reduced cost to return.

Return value

`djs: Optional[list[float]]` Double array of length `last-first+1` containing the reduced costs of the variables.

Further information

1. Column indices should be in the range `0` to `COLS` (p.**??** Xpress Optimizer Reference Manual)`-1`.
2. See `problem.getCallbackSolution` for a list of callbacks where this function can be called. Note that reduced costs are not available in all contexts where a primal solution is available.
3. Arguments `first, last` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.getCallbackPresolveSolution`, `problem.getCallbackPresolveSlacks`, `problem.getCallbackPresolveDuals`, `problem.getCallbackRedCosts`, `problem.getRedCosts`.

problem.chgObj

Purpose

Used to change the objective function coefficients.

Topic areas

Problem Modification, Optimizer

Synopsis

```
problem.chgObj(colind, objcoef)
```

Arguments

`colind: list[var]` Integer array containing the indices of the columns whose objective coefficients will change. An index of `-1` indicates that the fixed part of the objective function on the right hand side should change.

`objcoef: list[float]` Double array giving the new objective function coefficients.

Further information

1. The value of the fixed part of the objective function can be obtained using the `OBJRHS` (p.**??** Xpress Optimizer Reference Manual) problem attribute.
2. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.chgCoef`, `problem.chgMCoef`, `problem.chgMQObj`, `problem.chgQObj`, `problem.getObj`.

problem.chgCoef

Purpose

Used to change a single coefficient in the matrix. If the coefficient does not already exist, a new coefficient will be added to the matrix. If many coefficients are being added to a row of the matrix, it may be more efficient to delete the old row of the matrix and add a new row.

Topic areas

Problem Modification, Optimizer

Synopsis

```
problem.chgCoef(row, col, coef)
```

Arguments

`row: constraint` Row index for the coefficient.

`col: var` Column index for the coefficient.

`coef: float` New value for the coefficient. If `coef` is zero, any existing coefficient will be deleted.

Related controls

Double

`MATRIXTOL` Tolerance on matrix elements.

Further information

1. `problem.chgMCoef` is more efficient than multiple calls to `problem.chgCoef` and should be used in its place in such circumstances.
2. Argument `row` may contain either `xpress.constraint` objects or integer indices.
3. Argument `col` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.addCols`, `problem.addRows`, `problem.chgMCoef`, `problem.chgMQObj`, `problem.chgObj`, `problem.chgQObj`, `problem.chgRHS`, `problem.getCols`, `problem.getRows`.

problem.chgMCoef

Purpose

Used to change multiple coefficients in the matrix. If any coefficient does not already exist, it will be added to the matrix. If many coefficients are being added to a row of the matrix, it may be more efficient to delete the old row of the matrix and add a new one.

Topic areas

Problem Modification, Optimizer

Synopsis

```
problem.chgMCoef(rowind, colind, rowcoef)
```

Arguments

`rowind: list[constraint]` Integer array containing the row indices of the coefficients to be changed.

`colind: list[var]` Integer array containing the column indices of the coefficients to be changed.

`rowcoef: list[float]` Double array containing the new coefficient values. If an element of `rowcoef` is zero, the coefficient will be deleted.

Related controls

Double

`MATRIXTOL` Tolerance on matrix elements.

Further information

1. `problem.chgMCoef` is more efficient than repeated calls to `problem.chgCoef` and should be used in its place if many coefficients are to be changed.
2. Argument `rowind` may contain either `xpress.constraint` objects or integer indices.
3. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.chgCoef`, `problem.chgMQObj`, `problem.chgObj`, `problem.chgQObj`, `problem.chgRHS`, `problem.getCols`, `problem.getRHS`.

problem.chgMQObj

Purpose

Used to change multiple quadratic coefficients in the objective function. If any of the coefficients does not exist already, new coefficients will be added to the objective function.

Topic areas

Problem Modification, Quadratic, Optimizer

Synopsis

```
problem.chgMQObj(objqcol1, objqcol2, objqcoef)
```

Arguments

`objqcol1: list[var]` Integer array containing the column index of the first variable in each quadratic term.

`objqcol2: list[var]` Integer array containing the column index of the second variable in each quadratic term.

`objqcoef: list[float]` New values for the coefficients. If an entry in `objqcoef` is `0`, the corresponding entry will be deleted. These are the coefficients of the quadratic Hessian matrix.

Further information

1. If `objqcol1[t]` is not equal to `objqcol2[t]`, then both the matrix elements `(objqcol1[t], objqcol2[t])` and `(objqcol2[t], objqcol1[t])` are changed to leave the Hessian symmetric.
2. The quadratic matrix coefficients are implicitly divided by two. But since the coefficients for $x_i x_j$ (`i` not equal to `j`) appear twice, only the coefficients for x_i^2 should be multiplied by two in the `objqcoef` array to account for the implicit division. See the example above.
3. `problem.chgMQObj` is more efficient than repeated calls to `problem.chgQObj` and should be used in its place when several coefficients are to be changed.
4. Arguments `objqcol1, objqcol2` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.chgCoef`, `problem.chgMCoef`, `problem.chgObj`, `problem.chgQObj`, `problem.getQObj`.

problem.chgQObj

Purpose

Used to change a single quadratic coefficient in the objective function corresponding to the variable pair `(objqcol1,objqcol2)` of the Hessian matrix.

Topic areas

Problem Modification, Quadratic, Optimizer

Synopsis

```
problem.chgQObj(objqcol1, objqcol2, objqcoef)
```

Arguments

`objqcol1: var` Column index for the first variable in the quadratic term.

`objqcol2: var` Column index for the second variable in the quadratic term.

`objqcoef: float` New value for the coefficient in the quadratic Hessian matrix. If an entry in `objqcoef` is `0`, the corresponding entry will be deleted.

Further information

1. If `objqcol1` is not equal to `objqcol2`, then both the matrix elements `(objqcol1, objqcol2)` and `(objqcol2, objqcol1)` are changed to leave the Hessian symmetric.
2. The quadratic matrix coefficients are implicitly divided by two. But since the coefficients for $x_i x_j$ (`i` not equal to `j`) appear twice, only the coefficients for x_i^2 should be multiplied by two in the `objqcoef` argument to account for the implicit division. See the example above.
3. Arguments `objqcol1, objqcol2` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.chgCoef`, `problem.chgMCoef`, `problem.chgMQObj`, `problem.chgObj`, `problem.getQObj`.

problem.chgRHS

Purpose

Used to change right–hand side values of the problem.

Topic areas

Problem Modification, Optimizer

Synopsis

```
problem.chgRHS(rowind, rhs)
```

Arguments

`rowind: list[constraint]` Integer array containing the indices of the rows on which the right hand side values will change.

`rhs: list[float]` Double array giving the right hand side values.

Further information

Argument `rowind` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.chgCoef`, `problem.chgMCoef`, `problem.chgRHSRange`, `problem.getRHS`, `problem.getRHSRange`.

problem.chgRHSRange

Purpose

Used to change the range for a row of the problem matrix.

Topic areas

Problem Modification, Optimizer

Synopsis

```
problem.chgRHSRange(rowind, rng)
```

Arguments

`rowind: list[constraint]` Integer array containing the indices of the rows on which the range elements will change.

`rng: list[float]` Double array giving the range values.

Further information

1. If the range specified on the row is r, what happens depends on the row type and value of r. It is possible to convert non-range rows using this routine.

Value of r	Row type	Effect
$r \geqslant 0$	$= b, \leqslant b$	$b - r \leqslant \sum a_j x_j \leqslant b$
$r \geqslant 0$	$\geqslant b$	$b \leqslant \sum a_j x_j \leqslant b + r$
$r < 0$	$= b, \leqslant b$	$b \leqslant \sum a_j x_j \leqslant b - r$
$r < 0$	$\geqslant b$	$b + r \leqslant \sum a_j x_j \leqslant b$

2. Argument `rowind` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.chgCoef`, `problem.chgMCoef`, `problem.chgRHS`, `problem.getRHSRange`.

problem.chgRowType

Purpose

Used to change the type of a row in the matrix.

Topic areas

Problem Modification, Optimizer

Synopsis

```
problem.chgRowType(rowind, rowtype)
```

Arguments

`rowind: list[constraint]` Integer array containing the indices of the rows.

`rowtype: list[str]` Character array giving the new row types:

`L`	indicates a $\leqslant$ row;
`E`	indicates an = row;
`G`	indicates a $\geqslant$ row;
`R`	indicates a range row;
`N`	indicates a free row.

Further information

1. A row can be changed to a range type row by first changing the row to an `R` or `L` type row and then changing the range on the row using `problem.chgRHSRange`.
2. Argument `rowind` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.addRows`, `problem.chgColType`, `problem.chgRHS`, `problem.chgRHSRange`, `problem.delRows`, `problem.getRowType`.

problem.msAddJob

Purpose

Adds a multistart job to the multistart pool

Topic areas

Multistart, Data Input, Nonlinear

Synopsis

```
problem.msAddJob(description=None, colind=None, initial=None, control=None,
       data=None)
```

Arguments

`description: str` Text description of the job. Used for messaging, may be None if not required.

`colind: list[var]` Indices of the variables for which to set an initial value. May be None if no initial values are provided.

`initial: list[float]` Initial values for the variables for which to set an initial value. May be None if no initial values are provided.

`control: dict` Python dictionary with control strings as keys and numbers as values. Note that only numerical controls are allowed.

`data: Any` Job-specific user context object to be passed to the multistart callbacks.

Further information

1. Adds a mutistart job, applying the specified initial point and option combinations on top of the base problem, i.e. the options and initial values specified to the function is applied on top of the existing settings.
2. See `MSMAXBOUNDRANGE` (p.**??** Xpress NonLinear Reference Manual) for controlling the range in which initial values are generated.
3. This function allows for loading empty template jobs, that can then be identified using the data variable.
4. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.msAddPreset`, `problem.msAddCustomPreset`, `problem.msClear`

problem.msAddPreset

Purpose

Loads a preset of jobs into the multistart job pool.

Topic areas

Multistart, Data Input, Nonlinear

Synopsis

```
problem.msAddPreset(preset, maxjobs, description=None, data=None)
```

Arguments

`preset: int` Which preset to load.

`maxjobs: int` Maximum number of jobs to be added to the multistart pool.

`description: str` Text description of the preset. Used for messaging, may be None if not required.

`data: Any` Job-specific user context object to be passed to the multistart callbacks.

Further information

1. The following presets are defined:

   `xpress.constants.MSSET_INITIALVALUES` generate maxjobs number of random base points.

   `xpress.constants.MSSET_SOLVERS` load all solvers.

   `xpress.constants.MSSET_SLP_BASIC` load the most typical SLP tuning settings. A maximum of maxjobs jobs are loaded.

   `xpress.constants.MSSET_SLP_EXTENDED` load a comprehensive set of SLP tuning settings. A maximum of maxjobs jobs are loaded.

   `xpress.constants.MSSET_KNITRO_BASIC` load the most typical Knitro tuning settings. A maximum of maxjobs jobs are loaded.

   `xpress.constants.MSSET_KNITRO_EXTENDED` load a comprehensive set of Knitro tuning settings. A maximum of maxjobs jobs are loaded.

   `xpress.constants.MSSET_INITIALFILTERED` generate maxjobs number of random base points, filtered by a merit function centred on initial feasibility.

2. See `MSMAXBOUNDRANGE` (p.**??** Xpress NonLinear Reference Manual) for controlling the range in which initial values are generated.

Related topics

`problem.msAddJob`, `problem.msAddCustomPreset`, `problem.msClear`

problem.msAddCustomPreset

Purpose

A combined version of problem.msAddJob and problem.msAddPreset. The preset described is loaded, topped up with the specific settings supplied

Topic areas

Multistart, Data Input, Nonlinear

Synopsis

```
problem.msAddCustomPreset(preset, maxjobs, description=None, colind=None,
       initial=None, control=None, data=None)
```

Arguments

`preset: int` Which preset to load.

`maxjobs: int` Maximum number of jobs to be added to the multistart pool.

`description: str` Text description of the job. Used for messaging, may be None if not required.

`colind: list[var]` Indices of the variables for which to set an initial value. May be None if no initial values are provided.

`initial: list[float]` Initial values for the variables for which to set an initial value. May be None if no initial values are provided.

`control: dict` Python dictionary with control strings as keys and numbers as values. Note that only numerical controls are allowed.

`data: Any` Job-specific user context object to be passed to the multistart callbacks.

Further information

1. This function allows for repeatedly calling the same multistart preset (e.g. initial values) using different basic controls.
2. The following presets are defined:

   `xpress.constants.MSSET_INITIALVALUES` generate maxjobs number of random base points.

   `xpress.constants.MSSET_SOLVERS` load all solvers.

   `xpress.constants.MSSET_SLP_BASIC` load the most typical SLP tuning settings. A maximum of maxjobs jobs are loaded.

   `xpress.constants.MSSET_SLP_EXTENDED` load a comprehensive set of SLP tuning settings. A maximum of maxjobs jobs are loaded.

   `xpress.constants.MSSET_KNITRO_BASIC` load the most typical Knitro tuning settings. A maximum of maxjobs jobs are loaded.

   `xpress.constants.MSSET_KNITRO_EXTENDED` load a comprehensive set of Knitro tuning settings. A maximum of maxjobs jobs are loaded.

   `xpress.constants.MSSET_INITIALFILTERED` generate maxjobs number of random base points, filtered by a merit function centred on initial feasibility.
3. See `MSMAXBOUNDRANGE` (p.**??** Xpress NonLinear Reference Manual) for controlling the range in which initial values are generated.
4. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.msAddPreset`, `problem.msAddJob`, `problem.msClear`

problem.nlpPrintEvalInfo

Purpose

Print a summary of any evaluation errors that may have occurred during solving a problem

Topic areas

Logging, Nonlinear

Synopsis

```
problem.nlpPrintEvalInfo()
```

Related topics

```
problem.addNlpCoefEvalErrorCallback
```

problem.nlpValidate

Purpose

Validate the feasibility of constraints in a converged solution

Topic areas

Solution, Nonlinear

Synopsis

```
problem.nlpValidate()
```

Further information

`problem.nlpValidate` checks the feasibility of a converged solution against relative and absolute tolerances for each constraint. The left hand side and the right hand side of the constraint are calculated using the converged solution values. If the calculated values imply that the constraint is infeasible, then the difference (D) is tested against the absolute and relative validation tolerances.

If $D < XSLP_VALIDATIONTOL_A$

then the constraint is within the absolute validation tolerance. The total positive (TPos) and negative contributions (TNeg) to the left hand side are also calculated.

If $D < MAX(ABS(TPos), ABS(TNeg)) * XSLP_VALIDATIONTOL_R$

then the constraint is within the relative validation tolerance. For each constraint which is outside both the absolute and relative validation tolerances, validation factors are calculated which are the factors by which the infeasibility exceeds the corresponding validation tolerance; the smallest factor is printed in the validation report.

The validation index `NLPVALIDATIONINDEX_A` (p.**??** Xpress NonLinear Reference Manual) is the largest absolute validation factor multiplied by the absolute validation tolerance; the validation index `NLPVALIDATIONINDEX_R` (p.**??** Xpress NonLinear Reference Manual) is the largest relative validation factor multiplied by the relative validation tolerance.

Related topics

`NLPVALIDATIONINDEX_A` (p.**??** Xpress NonLinear Reference Manual), `NLPVALIDATIONINDEX_R` (p.**??** Xpress NonLinear Reference Manual), `NLPVALIDATIONTOL_A` (p.**??** Xpress NonLinear Reference Manual), `NLPVALIDATIONTOL_R` (p.**??** Xpress NonLinear Reference Manual)

problem.nlpOptimize

Purpose

Maximize or minimize an SLP problem

Topic areas

Solution Process, Nonlinear

Synopsis

```
problem.nlpOptimize(flags="")
```

Argument

`flags: str` Flags to pass to `problem.nlpOptimize`. The default is `""` or `None`, in which case the solve stops after solving the root relaxation (or a continuous problem to completion) and it restarts the solve without continuing.

`g`	Perform a branch and bound search if necessary to solve the problem;
`c`	continue a previously interrupted solve.

All other flags are passed to the Optimizer: see `problem.lpOptimize`.

Related controls

Integer

NLPLOG	Determines the amount of iteration logging information produced.
NLPPRESOLVE	Bit map determining the type of nonlinear presolve used before the SLP optimization starts.
SLPALGORITHM	Bit map determining the SLP algorithm(s) used in the optimization.
SLPAUGMENTATION	Bit map determining the type of augmentation used to create the linearization.
SLPCASCADE	Bit map determining the type of cascading (recalculation of SLP variable values) used during the SLP optimization.

Further information

1. If `problem.slpConstruct` has not already been called, it will be called first, using the augmentation defined by the control variable `SLPAUGMENTATION` (p.**??** Xpress NonLinear Reference Manual).
2. If determining rows are provided, then cascading will be invoked in accordance with the setting of the control variable `SLPCASCADE` (p.**??** Xpress NonLinear Reference Manual).

Related topics

`problem.slpConstruct`.

problem.nlpCurrentIV

Purpose

Transfer the current solution to initial values

Topic areas

Data Input, Nonlinear

Synopsis

```
problem.nlpCurrentIV()
```

Further information

Provides a way to set the current iterates solution as initial values, make changes to parameters or to the underlying nonlinear problem and then rerun the SLP optimization process.

Related topics

`problem.slpReInitialize`, `problem.slpUnConstruct`

problem.nlpValidateRow

Purpose

Prints an extensive analysis on a given constraint of the SLP problem

Topic areas

Solution, Nonlinear

Synopsis

```
problem.nlpValidateRow(row)
```

Argument

`row: constraint` The index of the row to be analyzed

Further information

1. The analysis will include the readable format of the original constraint and the augmented constraint. For infeasible constraints, the absolute and relative infeasibility is calculated. Variables in the constraints are listed including their value in the solution of the last linearization, the internal value (e.g. cascaded), reduced cost, step bound and convergence status. Scaling analysis is also provided.
2. Argument `row` may contain either `xpress.constraint` objects or integer indices.

problem.nlpValidateKKT

Purpose

Validates the first order optimality conditions also known as the Karush-Kuhn-Tucker (KKT) conditions versus the currect solution

Topic areas

Solution, Nonlinear

Synopsis

```
problem.nlpValidateKKT(mode, respectbasis, updatemult, violtarget)
```

Arguments

`mode: int` The calculation mode can be:

0	recalculate the reduced costs at the current solution using the current dual solution.
1	minimize the sum of KKT violations by adjusting the dual solution.
2	perform both.

`respectbasis: int` The following ways are defined to assess if a constraint is active:

0	evaluate the recalculated slack activity versus `SLPECFTOL_R` (p.**??** Xpress NonLinear Reference Manual).
1	use the basis status of the slack in the linearized problem if available.
2	use both.

`updatemult: int` The calculated values can be:

0	only used to calculate the `NLPVALIDATIONINDEX_K` (p.**??** Xpress NonLinear Reference Manual) measure.
1	used to update the current dual solution and reduced costs.

`violtarget: float` When calculating the best KKT multipliers, it is possible to enforce an even distribution of reduced costs violations by enforcing a bound on them.

Further information

The bounds enforced by violtarget are automatically relaxed if the desired accuracy cannot be achieved.

problem.msClear

Purpose

Removes all scheduled jobs from the multistart job pool

Topic areas

Multistart, Nonlinear

Synopsis

```
problem.msClear()
```

Related topics

`problem.msAddJob`, `problem.msAddPreset`, `problem.msAddCustomPreset`

problem.nlpEvaluateFormula

Purpose

Evaluate a formula using the current values of the variables

Topic areas

Solution, Nonlinear

Synopsis

```
value = problem.nlpEvaluateFormula(parsed, type, values)
```

Arguments

`parsed: int` integer indicating whether the formula of the item is in internal unparsed format (`parsed`=0) or parsed (reverse Polish) format (`parsed`=1).

`type: list[int]` Integer array of token types for the formula.

`values: list[float]` Double array of values corresponding to `type`.

Return value

`value: float` A double precision value to receive the result of the calculation.

Further information

1. The formula in `type` and `values` must be terminated by an `xpress.constants.TOK_EOF` token.
2. The formula cannot include ”complicated” functions, such as user functions which return more than one value

Related topics

`problem.slpEvaluateCoef`

problem.nlpValidateVector

Purpose

Validate the feasibility of constraints for a given solution

Topic areas

Solution, Nonlinear

Synopsis

```
suminf, sumscaledinf, objval = problem.nlpValidateVector(solution)
```

Argument

`solution: list[float]` A vector of length `COLS` containing the solution vector to be checked.

Return value

`suminf: float` The sum of infeasibility.

`sumscaledinf: float` The sum of scaled (relative) infeasibility.

`objval: float` The net objective.

Further information

`problem.nlpValidateVector` works the same way as `problem.nlpValidate`, and will update `NLPVALIDATIONINDEX_A` (p.**??** Xpress NonLinear Reference Manual) and `NLPVALIDATIONINDEX_R` (p.**??** Xpress NonLinear Reference Manual).

Related topics

`NLPVALIDATIONINDEX_A` (p.**??** Xpress NonLinear Reference Manual), `NLPVALIDATIONINDEX_R` (p.**??** Xpress NonLinear Reference Manual), `NLPVALIDATIONTOL_A` (p.**??** Xpress NonLinear Reference Manual), `NLPVALIDATIONTOL_R` (p.**??** Xpress NonLinear Reference Manual)

problem.nlpAddFormulas

Purpose

Add non-linear formulas to the SLP problem.

Topic areas

Problem Modification, Nonlinear

Synopsis

```
problem.nlpAddFormulas(rowind, formulastart, parsed, type, value)
```

Arguments

`rowind: list[constraint]` Integer array holding index of row for the coefficient.

`formulastart: list[int]` Integer array holding the start position in the arrays `type` and `value` of the formula for the coefficients. The last element should be set to the next position after the end of the last formula.

`parsed: int` Integer indicating whether the token arrays are formatted as internal unparsed (`parsed`=0) or internal parsed reverse Polish (`parsed`=1).

`type: list[int]` Array of token types providing the formula for each coefficient.

`value: list[float]` Array of values corresponding to the types in `type`.

Further information

1. The formula is made up of a list of tokens in `type` and `value` starting at `formulastart[j]`. The tokens follow the rules for parsed or unparsed formulae as indicated by the setting of `parsed`. The formula must be terminated with an `xpress.constants.TOK_EOF` token. If several rows share the same nonlinear expression, they can have the same value in `formulastart`. For possible token types and values see the NonLinear reference manual.
2. The `add` functions load additional items into the SLP problem. The corresponding `load` functions delete any existing items first.
3. The behaviour for existing formulas is additive: the formula defined in the parameters are added to any existing nonlinear expressions in the row. However, due to performance considerations, such duplications should be avoided when possible.
4. Argument `rowind` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.nlpGetFormulaStr`, `problem.nlpAddFormulas`, `problem.nlpChgFormulaStr`, `problem.nlpChgFormula`, `problem.nlpLoadFormulas`, `problem.nlpGetFormulaRows`, `problem.nlpGetFormula`, `problem.nlpDelFormulas`

problem.nlpChgFormulaStr

Purpose

Add or replace a single matrix formula using a character string for the formula.

Topic areas

Problem Modification, Nonlinear

Synopsis

```
problem.nlpChgFormulaStr(row, formula)
```

Arguments

`row: constraint` The index of the matrix row for the coefficient.

`formula: str` Character string holding the formula with the tokens separated by spaces.

Further information

1. If the coefficient already exists as a constant or formula, it will be changed into the new coefficient. If it does not exist, it will be added to the problem.
2. This function can only be used if all the operands in the formula can be correctly identified as constants, existing columns, character variables or functions. Therefore, if a formula refers to a new column, that new item must be added to the Xpress NonLinear problem first.
3. Argument `row` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.nlpGetFormulaStr`, `problem.nlpAddFormulas`, `problem.nlpChgFormula`, `problem.nlpLoadFormulas`, `problem.nlpGetFormulaRows`, `problem.nlpGetFormula`, `problem.nlpDelFormulas`

problem.nlpChgFormula

Purpose

Add or replace a single matrix formula using a parsed or unparsed formula

Topic areas

Problem Modification, Nonlinear

Synopsis

```
problem.nlpChgFormula(row, parsed, type, value)
```

Arguments

`row: constraint` The index of the matrix row for the coefficient.

`parsed: int` Integer indicating the whether the token arrays are formatted as internal unparsed (`parsed`=0) or internal parsed reverse Polish (`parsed`=1).

`type: list[int]` Array of token types providing the description and formula for each item.

`value: list[float]` Array of values corresponding to the types in `type`.

Further information

1. If the row already has a nonlinear expression in it, it will be changed into the new formula. If it does not exist, it will be added to the problem.
2. Argument `row` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.nlpGetFormulaStr`, `problem.nlpAddFormulas`, `problem.nlpChgFormulaStr`, `problem.nlpLoadFormulas`, `problem.nlpGetFormulaRows`, `problem.nlpGetFormula`, `problem.nlpDelFormulas`

problem.nlpGetFormula

Purpose

Retrieve a single matrix formula as a formula split into tokens.

Topic areas

Problem Information, Nonlinear

Synopsis

```
type, value = problem.nlpGetFormula(row, parsed)
```

Arguments

`row: constraint` Integer holding the row index for the formula.

`parsed: int` Integer indicating whether the formula of the row is to be returned in internal unparsed format (`parsed`=0) or parsed (reverse Polish) format (`parsed`=1).

Return value

`type: list[int]` Integer array containing the token types for the formula.

`value: list[float]` Double array of values corresponding to `type`.

Further information

1. The `type` and `value` arrays are terminated by an `xpress.constants.TOK_EOF` token.
2. Argument `row` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.nlpGetFormulaStr`, `problem.nlpAddFormulas`, `problem.nlpChgFormulaStr`, `problem.nlpChgFormula`, `problem.nlpLoadFormulas`, `problem.nlpGetFormulaRows`, `problem.nlpGetFormula`, `problem.nlpDelFormulas`

problem.nlpGetFormulaRows

Purpose

Retrieve the list of positions of the nonlinear formulas in the problem

Topic areas

Problem Information, Nonlinear

Synopsis

```
rowind = problem.nlpGetFormulaRows()
```

Return value

`rowind: list[int]` Integer array used for returning the row positions of the nonlinear formulas.

Related topics

`problem.nlpGetFormulaStr`, `problem.nlpAddFormulas`, `problem.nlpChgFormulaStr`, `problem.nlpChgFormula`, `problem.nlpLoadFormulas`, `problem.nlpGetFormulaRows`, `problem.nlpGetFormula`, `problem.nlpDelFormulas`

problem.nlpLoadFormulas

Purpose

Load non-linear formulas into the SLP problem

Topic areas

Problem Information, Nonlinear

Synopsis

```
problem.nlpLoadFormulas(rowind, formulastart, parsed, type, value)
```

Arguments

`rowind: list[constraint]` Integer array holding index of row for the coefficient.

`formulastart: list[int]` Integer array holding the start position in the arrays `type` and `value` of the formula for the coefficients. The last element should be set to the next position after the end of the last formula.

`parsed: int` Integer indicating whether the token arrays are formatted as internal unparsed (`parsed`=0) or internal parsed reverse Polish (`parsed`=1).

`type: list[int]` Array of token types providing the formula for each coefficient.

`value: list[float]` Array of values corresponding to the types in `type`.

Further information

1. Formula `j` is made up of a list of tokens in `type` and `value` starting at `formulastart[j]`. The tokens follow the rules for parsed or unparsed formulae as indicated by the setting of `parsed`. The formula must be terminated with an `xpress.constants.TOK_EOF` token. If several formulas share the same nonlinear expressions, they can have the same value in `formulastart`. For possible token types and values see the NonLinear reference manual.
2. The `load` functions load items into the SLP problem. Any existing items of the same type are deleted first. The corresponding `add` functions add or replace items leaving other items of the same type unchanged.
3. Argument `rowind` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.nlpGetFormulaStr`, `problem.nlpAddFormulas`, `problem.nlpChgFormulaStr`, `problem.nlpChgFormula`, `problem.nlpLoadFormulas`, `problem.nlpGetFormulaRows`, `problem.nlpGetFormula`, `problem.nlpDelFormulas`

problem.nlpDelFormulas

Purpose

Delete nonlinear formulas from the current problem

Topic areas

Problem Modification, Nonlinear

Synopsis

```
problem.nlpDelFormulas(rowind)
```

Argument

`rowind: list[constraint]` Row indices of the SLP nonlinear formulas to delete.

Further information

Argument `rowind` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.nlpGetFormulaStr`, `problem.nlpAddFormulas`, `problem.nlpChgFormulaStr`, `problem.nlpChgFormula`, `problem.nlpLoadFormulas`, `problem.nlpGetFormulaRows`, `problem.nlpGetFormula`, `problem.nlpDelFormulas`

problem.nlpGetFormulaStr

Purpose

Retrieve a single matrix formula in a character string.

Topic areas

Problem Information, Nonlinear

Synopsis

```
formula = problem.nlpGetFormulaStr(row)
```

Argument

`row: constraint` Integer holding the row index for the formula.

Return value

`formula: str` Character buffer containing the formula in the same format as used for input from a file.

Further information

Argument `row` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.nlpAddFormulas`, `problem.nlpChgFormulaStr`, `problem.nlpChgFormula`, `problem.nlpLoadFormulas`, `problem.nlpGetFormulaRows`, `problem.nlpGetFormula`, `problem.nlpDelFormulas`

problem.nlpSetInitVal

Purpose

Set the initial value of columns

Topic areas

Data Input, Nonlinear

Synopsis

```
problem.nlpSetInitVal(colind, initial)
```

Arguments

`colind: list[var]` Array with index of the column for which the initial value is provided.

`initial: list[float]` Array with the initial value.

Further information

Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.slpSetDetRow`

problem.slpGetCoefFormula

Purpose

Retrieve a single matrix coefficient as a formula split into tokens. For a simpler version of this function see `problem.nlpGetFormula`.

Topic areas

SLP, Problem Information, Nonlinear

Synopsis

```
factor, type, value = problem.slpGetCoefFormula(row, col, parsed)
```

Arguments

`row: constraint` Integer holding the row index for the coefficient.

`col: var` Integer holding the column index for the coefficient.

`parsed: int` Integer indicating whether the formula of the item is to be returned in internal unparsed format (`parsed`=0) or parsed (reverse Polish) format (`parsed`=1).

Return value

`factor: float` A double precision variable to receive the value of the constant factor multiplying the formula in the coefficient.

`type: list[int]` Integer array containing the token types for the formula.

`value: list[float]` Double array of values corresponding to `type`.

Further information

1. The `type` and `value` arrays are terminated by an `xpress.constants.TOK_EOF` token.
2. If the requested coefficient is constant, then `factor` will be set to 1.0 and the value will be returned with token type `xpress.constants.TOK_CON`.
3. Argument `row` may contain either `xpress.constraint` objects or integer indices.
4. Argument `col` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.nlpGetFormulaStr`, `problem.nlpAddFormulas`, `problem.nlpChgFormulaStr`, `problem.nlpChgFormula`, `problem.nlpLoadFormulas`, `problem.nlpGetFormulaRows`, `problem.nlpGetFormula`, `problem.nlpDelFormulas`

problem.slpGetCoefs

Purpose

Retrieve the list of positions of the nonlinear coefficients in the problem. For a simpler version of this function see `problem.nlpGetFormulaRows`.

Topic areas

SLP, Problem Information, Nonlinear

Synopsis

```
rowind, colind = problem.slpGetCoefs()
```

Return value

`rowind: list[int]` Integer array used for returning the row positions of the coefficients.

`colind: list[int]` Integer array used for returning the column positions of the coefficients.

Related topics

`problem.slpGetCoefStr`, `problem.slpGetCoefFormula`

problem.slpLoadCoefs

Purpose

Load non-linear coefficients into the SLP problem. For a simpler version of this function see `problem.nlpLoadFormulas`.

Topic areas

SLP, Problem Information, Nonlinear

Synopsis

```
problem.slpLoadCoefs(rowind, colind, factor, formulastart, parsed, type, coef)
```

Arguments

`rowind: list[constraint]` Integer array holding index of row for the coefficient.

`colind: list[var]` Integer array holding index of column for the coefficient.

`factor: list[float]` Double array holding factor by which formula is scaled. If this is `None`, then a value of 1.0 will be used.

`formulastart: list[int]` Integer array holding the start position in the arrays `type` and `coef` of the formula for the coefficients. The last element should be set to the next position after the end of the last formula.

`parsed: int` Integer indicating whether the token arrays are formatted as internal unparsed (`parsed`=0) or internal parsed reverse Polish (`parsed`=1).

`type: list[int]` Array of token types providing the formula for each coefficient.

`coef: list[float]` Array of values corresponding to the types in `type`.

Further information

1. The j^{th} coefficient is made up of two parts: `factor` and `Formula`. `factor` is a constant multiplier, which can be provided in the `factor` array. If Xpress NonLinear can identify a constant factor in `Formula`, then it will use that as well, to minimize the size of the formula which has to be calculated. `Formula` is made up of a list of tokens in `type` and `coef` starting at `formulastart[j]`. The tokens follow the rules for parsed or unparsed formulae as indicated by the setting of `parsed`. The formula must be terminated with an `xpress.constants.TOK_EOF` token. If several coefficients share the same formula, they can have the same value in `formulastart`. For possible token types and values see the NonLinear reference manual.
2. The `load` functions load items into the SLP problem. Any existing items of the same type are deleted first. The corresponding `add` functions add or replace items leaving other items of the same type unchanged.
3. Argument `rowind` may contain either `xpress.constraint` objects or integer indices.
4. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.nlpGetFormulaStr`, `problem.nlpAddFormulas`, `problem.nlpChgFormulaStr`, `problem.nlpChgFormula`, `problem.nlpLoadFormulas`, `problem.nlpGetFormulaRows`, `problem.nlpGetFormula`, `problem.nlpDelFormulas`

problem.slpDelCoefs

Purpose

Delete coefficients from the current problem. For a simpler version of this function see `problem.nlpDelFormulas`.

Topic areas

SLP, Problem Modification, Nonlinear

Synopsis

```
problem.slpDelCoefs(rowind, colind)
```

Arguments

`rowind: list[constraint]` Row indices of the SLP coefficients to delete.

`colind: list[var]` Column indices of the SLP coefficients to delete.

Further information

1. Argument `rowind` may contain either `xpress.constraint` objects or integer indices.
2. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.nlpGetFormulaStr`, `problem.nlpAddFormulas`, `problem.nlpChgFormulaStr`, `problem.nlpChgFormula`, `problem.nlpLoadFormulas`, `problem.nlpGetFormulaRows`, `problem.nlpGetFormula`, `problem.nlpDelFormulas`

problem.slpGetCoefStr

Purpose

Retrieve a single matrix coefficient as a formula in a character string. For a simpler version of this function see `problem.nlpGetFormulaStr`.

Topic areas

SLP, Problem Information, Nonlinear

Synopsis

```
factor, formula = problem.slpGetCoefStr(row, col)
```

Arguments

`row: constraint` Integer holding the row index for the coefficient.

`col: var` Integer holding the column index for the coefficient.

Return value

`factor: float` A double precision variable to receive the value of the constant factor multiplying the formula in the coefficient.

`formula: str` Character buffer containing the formula in the same format as used for input from a file.

Further information

1. If the requested coefficient is constant, then `factor` will be set to 1.0 and the value will be formatted in `formula`.
2. Argument `row` may contain either `xpress.constraint` objects or integer indices.
3. Argument `col` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.nlpGetFormulaStr`, `problem.nlpAddFormulas`, `problem.nlpChgFormulaStr`, `problem.nlpChgFormula`, `problem.nlpLoadFormulas`, `problem.nlpGetFormulaRows`, `problem.nlpGetFormula`, `problem.nlpDelFormulas`

problem.slpSetDetRow

Purpose

Set the determining row of a variable

Topic areas

SLP, Cascading, Data Input, Nonlinear

Synopsis

```
problem.slpSetDetRow(colind, rowind)
```

Arguments

`colind: list[var]` Array with the index of the column for which the determining row is set.

`rowind: list[constraint]` Array with the index of the determining row.

Further information

1. Argument `rowind` may contain either `xpress.constraint` objects or integer indices.
2. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.nlpSetInitVal`

problem.slpAddCoefs

Purpose

Add non-linear coefficients to the SLP problem. For a simpler version of this function see `problem.nlpAddFormulas`.

Topic areas

SLP, Problem Modification, Nonlinear

Synopsis

```
problem.slpAddCoefs(rowind, colind, factor, formulastart, parsed, type, value)
```

Arguments

`rowind: list[constraint]` Integer array holding index of row for the coefficient.

`colind: list[var]` Integer array holding index of column for the coefficient.

`factor: list[float]` Double array holding factor by which the formula is scaled. If this is `None`, then a value of 1.0 will be used.

`formulastart: list[int]` Integer array holding the start position in the arrays `type` and `value` of the formula for the coefficients. The last element should be set to the next position after the end of the last formula.

`parsed: int` Integer indicating whether the token arrays are formatted as internal unparsed (`parsed`=0) or internal parsed reverse Polish (`parsed`=1).

`type: list[int]` Array of token types providing the formula for each coefficient.

`value: list[float]` Array of values corresponding to the types in `type`.

Further information

1. The j^{th} coefficient is made up of two parts: `factor` and `Formula`. `factor` is a constant multiplier, which can be provided in the `factor` array. If Xpress NonLinear can identify a constant factor in `Formula`, then it will use that as well, to minimize the size of the formula which has to be calculated. `Formula` is made up of a list of tokens in `type` and `value` starting at `formulastart[j]`. The tokens follow the rules for parsed or unparsed formulae as indicated by the setting of `parsed`. The formula must be terminated with an `xpress.constants.TOK_EOF` token. If several coefficients share the same formula, they can have the same value in `formulastart`. For possible token types and values see the NonLinear reference manual.
2. The `add` functions load additional items into the SLP problem. The corresponding `load` functions delete any existing items first.
3. The behaviour for existing coefficients is additive: the formula defined in the parameters are added to any existing formula coefficients. However, due to performance considerations, such duplications should be avoided when possible.
4. Argument `rowind` may contain either `xpress.constraint` objects or integer indices.
5. Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.nlpGetFormulaStr`, `problem.nlpAddFormulas`, `problem.nlpChgFormulaStr`, `problem.nlpChgFormula`, `problem.nlpLoadFormulas`, `problem.nlpGetFormulaRows`, `problem.nlpGetFormula`, `problem.nlpDelFormulas`

problem.slpChgCoefStr

Purpose

Add or change a single matrix coefficient using a character string for the formula. For a simpler version of this function see `problem.nlpChgFormulaStr`.

Topic areas

SLP, Problem Modification, Nonlinear

Synopsis

```
problem.slpChgCoefStr(row, col, factor, formula)
```

Arguments

`row: constraint` The index of the matrix row for the coefficient.

`col: var` The index of the matrix column for the coefficient.

`factor: float` A double precision variable holding the constant multiplier for the formula. If `factor` is `None`, a value of 1.0 will be used.

`formula: str` Character string holding the formula with the tokens separated by spaces.

Further information

1. If the coefficient already exists as a constant or formula, it will be changed into the new coefficient. If it does not exist, it will be added to the problem.
2. A coefficient is made up of two parts: `factor` and `formula`. `factor` is a constant multiplier which can be provided in the `factor` variable. If Xpress NonLinear can identify a constant factor in the formula, then it will use that as well, to minimize the size of the formula which has to be calculated.
3. This function can only be used if all the operands in the formula can be correctly identified as constants, existing columns, character variables or functions. Therefore, if a formula refers to a new column, that new item must be added to the Xpress NonLinear problem first.
4. Argument `row` may contain either `xpress.constraint` objects or integer indices.
5. Argument `col` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.nlpGetFormulaStr`, `problem.nlpAddFormulas`, `problem.nlpChgFormulaStr`, `problem.nlpChgFormula`, `problem.nlpLoadFormulas`, `problem.nlpGetFormulaRows`, `problem.nlpGetFormula`, `problem.nlpDelFormulas`

problem.slpChgCoef

Purpose

Add or change a single matrix coefficient using a parsed or unparsed formula. For a simpler version of this function see `problem.nlpChgFormula`.

Topic areas

SLP, Problem Modification, Nonlinear

Synopsis

```
problem.slpChgCoef(row, col, factor, parsed, type, value)
```

Arguments

`row: constraint` The index of the matrix row for the coefficient.

`col: var` The index of the matrix column for the coefficient.

`factor: float` A double precision variable holding the constant multiplier for the formula. If `factor` is `None`, a value of 1.0 will be used.

`parsed: int` Integer indicating the whether the token arrays are formatted as internal unparsed (`parsed`=0) or internal parsed reverse Polish (`parsed`=1).

`type: list[int]` Array of token types providing the description and formula for each item.

`value: list[float]` Array of values corresponding to the types in `type`.

Further information

1. If the coefficient already exists as a constant or formula, it will be changed into the new coefficient. If it does not exist, it will be added to the problem.
2. A coefficient is made up of two parts: `factor` and `Formula`. `factor` is a constant multiplier which can be provided in the `factor` variable. If Xpress NonLinear can identify a constant factor in the Formula, then it will use that as well, to minimize the size of the formula which has to be calculated.
3. Argument `row` may contain either `xpress.constraint` objects or integer indices.
4. Argument `col` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.nlpGetFormulaStr`, `problem.nlpAddFormulas`, `problem.nlpChgFormulaStr`, `problem.nlpChgFormula`, `problem.nlpLoadFormulas`, `problem.nlpGetFormulaRows`, `problem.nlpGetFormula`, `problem.nlpDelFormulas`

problem.slpGetColInfo

Purpose

Get current column information.

Topic areas

SLP, Solution, Nonlinear

Synopsis

```
info = problem.slpGetColInfo(type, col)
```

Arguments

`type: int` Type of information (see below)

`col: var` Index of the column whose information is to be handled

Return value

`info: Any` The information

Further information

1. The following constants are provided for column information handling:

   `xpress.constants.SLPCOLINFO_VALUE` Get the current value of the column

   `xpress.constants.SLPCOLINFO_RDJ` Get the current reduced cost of the column

   `xpress.constants.SLPCOLINFO_DELTAINDEX` Get the delta variable index associated to the column

   `xpress.constants.SLPCOLINFO_DELTA` Get the delta value (change since previous value) of the column

   `xpress.constants.SLPCOLINFO_DELTADJ` Get the delta variables reduced cost

   `xpress.constants.SLPCOLINFO_UPDATEROW` Get the index of the update (or step bound) row associated to the column

   `xpress.constants.SLPCOLINFO_SB` Get the step bound on the variable

   `xpress.constants.SLPCOLINFO_SBDUAL` Get the dual multiplier of the step bound row for the variable

   `xpress.constants.SLPCOLINFO_DETROW` The determining row for the variable (or -1 if it is not determined)

   `xpress.constants.SLPCOLINFO_CONVERGENCESTATUS` The convergence status of the variable (0 if unconverged, 1-based index of the convergencetolerance that applied otherwise)

2. The convergence status will be an item from the following list:

convergencestatus	Tolerance
0	unconverged
1	Closure tolerance (TC)
2	Absolute delta tolerance (TA)
3	Relative delta tolerance (RA)
4	Absolute coefficient tolerance (TM)
5	Relative coefficient tolerance (RM)
6	Absolute impact tolerance (TI)
7	Relative impact tolerance (RI)
8	Absolute slack tolerance (TS)
9	Relative slack tolerance (RS)
other	user convergence

3. Argument `col` may contain either `xpress.var` objects or integer indices.

problem.slpGetRowInfo

Purpose

Get current row information.

Topic areas

SLP, Solution, Nonlinear

Synopsis

```
info = problem.slpGetRowInfo(type, row)
```

Arguments

`type: int` Type of information (see below)

`row: constraint` Index of the row whose information is to be handled

Return value

`info: Any` The information

Further information

1. The following constants are provided for row information handling:

   `xpress.constants.SLPROWINFO_SLACK` Get the current slack value of the row

   `xpress.constants.SLPROWINFO_DUAL` Get the current dual multiplier of the row

   `xpress.constants.SLPROWINFO_NUMPENALTYERRORS` Get the number of times the penalty error vector has been active for the row

   `xpress.constants.SLPROWINFO_MAXPENALTYERROR` Get the maximum size of the penalty error vector activity for the row

   `xpress.constants.SLPROWINFO_TOTALPENALTYERROR` Get the total size of the penalty error vector activity for the row

   `xpress.constants.SLPROWINFO_CURRENTPENALTYERROR` Get the size of the penalty error vector activity in the current iteration for the row

   `xpress.constants.SLPROWINFO_CURRENTPENALTYFACTOR` Set the size of the penalty error factor for the current iteration for the row

   `xpress.constants.SLPROWINFO_PENALTYCOLUMNPLUS` Get the index of the positive penalty column for the row (+)

   `xpress.constants.SLPROWINFO_PENALTYCOLUMNPLUSVALUE` Get the value of the positive penalty column for the row (+)

   `xpress.constants.SLPROWINFO_PENALTYCOLUMNPLUSDJ` Get the reduced cost of the positive penalty column for the row (+)

   `xpress.constants.SLPROWINFO_PENALTYCOLUMNMINUS` Get the index of the negative penalty column for the row (-)

   `xpress.constants.SLPROWINFO_PENALTYCOLUMNMINUSVALUE` Get the value of the negative penalty column for the row (-)

   `xpress.constants.SLPROWINFO_PENALTYCOLUMNMINUSDJ` Get the reduced cost of the negative penalty column for the row (-)

2. Argument `row` may contain either `xpress.constraint` objects or integer indices.

problem.slpCascadeSol

Purpose

Re-calculate consistent values for SLP variables based on the current values of the remaining variables.

Topic areas

SLP, Solution Process, Cascading, Nonlinear

Synopsis

```
problem.slpCascadeSol()
```

Further information

1. See the section on cascading for an extended discussion of the types of cascading which can be performed.
2. `problem.slpCascadeSol` is called automatically during the SLP iteration process and so it is not normally necessary to perform an explicit cascade calculation.
3. The variables are re-calculated in accordance with the order generated by `problem.slpCascadeOrder`.

Related topics

`problem.slpCascadeOrder`, `SLPCASCADE` (p.**??** Xpress NonLinear Reference Manual), `SLPCASCADENLIMIT` (p.**??** Xpress NonLinear Reference Manual), `SLPCASCADETOL_PA` (p.**??** Xpress NonLinear Reference Manual), `SLPCASCADETOL_PR` (p.**??** Xpress NonLinear Reference Manual)

problem.slpCascadeOrder

Purpose

Establish a re-calculation sequence for SLP variables with determining rows.

Topic areas

SLP, Data Input, Nonlinear

Synopsis

```
problem.slpCascadeOrder()
```

Further information

`problem.slpCascadeOrder` is called automatically at the start of the SLP iteration process and so it is not normally necessary to perform an explicit cascade ordering.

Related topics

`problem.slpCascadeSol`

problem.slpChgRowStatus

Purpose

Change the status setting of a constraint

Topic areas

SLP, Data Input, Bit-vector, Nonlinear

Synopsis

```
problem.slpChgRowStatus(row, status)
```

Arguments

`row: constraint` The index of the matrix row to be changed.

`status: int` An integer holding a bitmap with the new status settings. If the status is to be changed, always get the current status first (use `problem.slpGetRowStatus`) and then change settings as required. The only settings likely to be changed are:

`Bit 11` Set if row must not have a penalty error vector. This is the equivalent of an enforced constraint (SLPDATA type EC).

Further information

1. If `status` is `None` the current status will remain unchanged.
2. Argument `row` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.slpGetRowStatus`

problem.slpChgRowWt

Purpose

Set or change the initial penalty error weight for a row

Topic areas

SLP, Data Input, Nonlinear

Synopsis

```
problem.slpChgRowWt(row, weight)
```

Arguments

`row: constraint` The index of the row whose weight is to be set or changed.

`weight: float` A double precision variable holding the new value of the weight. May be `None` if not required.

Further information

1. A positive value is interpreted as a multiplier of the default row weight calculated by Xpress-SLP.
2. A negative value is interpreted as a fixed value: the absolute value is used directly as the row weight.
3. The initial row weight is used only when the augmented structure is created.
4. Argument `row` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.slpGetRowWT`

problem.slpChgDeltaType

Purpose

Changes the type of the delta assigned to a nonlinear variable

Topic areas

SLP, Problem Modification, Nonlinear

Synopsis

```
problem.slpChgDeltaType(varind, deltatypes, values)
```

Arguments

`varind: list[var]` Indices of the variables to change the deltas for.

`deltatypes: list[int]` Type of the delta variable:

- `0 (xpress.constants.SLPDELTA_CONT)` Differentiable variable, default.
- `1 (xpress.constants.SLPDELTA_SEMICONT)` Variable where a minimum perturbation size given in `values` may be required before a significant change in the problem is achieved.
- `2 (xpress.constants.SLPDELTA_INTEGER)` Variable defined over the grid size given in `values`.
- `3 (xpress.constants.SLPDELTA_EXPLORE)` Variable where a meaningful step size should automatically be detected, with an upper limit given in `values`.

`values: list[float]` Grid or minimum step sizes for the variables.

Further information

1. Changing the delta type of a variables makes the variable nonlinear.
2. Argument `varind` may contain either `xpress.var` objects or integer indices.

Related topics

`SLPSEMICONTDELTAS` (p.**??** Xpress NonLinear Reference Manual), `SLPINTEGERDELTAS` (p.**??** Xpress NonLinear Reference Manual), `SLPEXPLOREDELTAS` (p.**??** Xpress NonLinear Reference Manual)

problem.slpChgCascadeNLimit

Purpose

Set a variable specific cascade iteration limit

Topic areas

SLP, Data Input, Nonlinear

Synopsis

```
problem.slpChgCascadeNLimit(col, limit)
```

Arguments

`col: var` The index of the column corresponding to the SLP variable for which the cascading limit is to be imposed.

`limit: int` The new cascading iteration limit.

Further information

1. A value set by this function will overwrite the value of `SLPCASCADENLIMIT` (p.**??** Xpress NonLinear Reference Manual) for this variable. To remove any previous value set by this function, use an iteration limit of 0.
2. Argument `col` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.slpCascadeOrder`, `SLPCASCADE` (p.**??** Xpress NonLinear Reference Manual), `SLPCASCADENLIMIT` (p.**??** Xpress NonLinear Reference Manual), `SLPCASCADETOL_PA` (p.**??** Xpress NonLinear Reference Manual), `SLPCASCADETOL_PR` (p.**??** Xpress NonLinear Reference Manual)

problem.slpConstruct

Purpose

Create the full augmented SLP matrix and data structures, ready for optimization

Topic areas

SLP, Solution Process, Nonlinear

Synopsis

```
problem.slpConstruct()
```

Further information

1. `problem.slpConstruct` adds new rows and columns to the SLP matrix and calculates initial values for the non-linear coefficients. Which rows and columns are added will depend on the setting of `SLPAUGMENTATION` (p.**??** Xpress NonLinear Reference Manual). Names for the new rows and columns are generated automatically, based on the existing names and the string controls such as `SLPDELTAFORMAT`.
2. Once `problem.slpConstruct` has been called, no new rows, columns or non-linear coefficients can be added to the problem. Any rows or columns which will be required must be added first. Non-linear coefficients must not be changed; constant matrix elements can generally be changed after `problem.slpConstruct`.
3. `problem.slpConstruct` is called automatically by the SLP optimization procedure, and so only needs to be called explicitly if changes need to be made between the augmentation and the optimization.

problem.slpGetRowStatus

Purpose

Retrieve the status setting of a constraint

Topic areas

SLP, Solution, Bit-vector, Nonlinear

Synopsis

```
status = problem.slpGetRowStatus(row)
```

Argument

`row: constraint` The index of the matrix row whose data is to be obtained.

Return value

`status: int` The status settings.

Further information

1. See the section on bitmap settings for details on the possible information in `status`.
2. Argument `row` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.slpChgRowStatus`

problem.slpGetRowWT

Purpose

Get the initial penalty error weight for a row

Topic areas

SLP, Data Information, Nonlinear

Synopsis

```
weight = problem.slpGetRowWT(row)
```

Argument

`row: constraint` The index of the row whose weight is to be retrieved.

Return value

`weight: float` A double precision variable to receive the value of the weight.

Further information

1. The initial row weight is used only when the augmented structure is created. After that, the current weighting can be accessed using `problem.slpGetRowInfo`.
2. Argument `row` may contain either `xpress.constraint` objects or integer indices.

Related topics

`problem.slpChgRowWt`, `problem.slpGetRowInfo`

problem.slpEvaluateCoef

Purpose

Evaluate a coefficient using the current values of the variables

Topic areas

Solution, Nonlinear

Synopsis

```
value = problem.slpEvaluateCoef(row, col)
```

Arguments

`row: constraint` Integer index of the row.

`col: var` Integer index of the column.

Return value

`value: float` A double precision value to receive the result of the calculation.

Further information

1. The values of the variables are obtained from the solution.
2. Argument `row` may contain either `xpress.constraint` objects or integer indices.
3. Argument `col` may contain either `xpress.var` objects or integer indices.

Related topics

`problem.nlpEvaluateFormula`

problem.slpReInitialize

Purpose

Reset the SLP problem to match a just augmented system

Topic areas

SLP, Solution Process, Nonlinear

Synopsis

```
problem.slpReInitialize()
```

Further information

Can be used to rerun the SLP optimization process with updated parameters, penalties or initial values, but unchanged augmentation.

Related topics

`problem.slpUnConstruct`, `problem.nlpCurrentIV`,

problem.slpUnConstruct

Purpose

Removes the augmentation and returns the problem to its pre-linearization state

Topic areas

SLP, Solution Process, Nonlinear

Synopsis

```
problem.slpUnConstruct()
```

Further information

Only limited changes are allowed to an augmented problem.

Related topics

`problem.slpConstruct`

problem.slpUpdateLinearization

Purpose

Updates the current linearization

Topic areas

SLP, Solution Process, Nonlinear

Synopsis

```
problem.slpUpdateLinearization()
```

Further information

1. Updates the augmented problem (the linearization) to match the current base point. The base point is the current SLP solution.
2. The linearization must be present, and this function can only be called after the problem has been augmented by `problem.slpConstruct`.

Related topics

`problem.slpConstruct`

problem.slpFixPenalties

Purpose

Fixe the values of the error vectors

Topic areas

Solution Process, Nonlinear

Synopsis

```
status = problem.slpFixPenalties()
```

Return value

`status: int` Return status after fixing the penalty variables: 0 is successful, nonzero otherwise.

Further information

1. The function fixes the values of all error vectors on their current values. It also removes their objective cost contribution.
2. The function is intended to support post optimization analysis, by removing any possible direct effect of the error vectors from the dual and reduced cost values.
3. The problem.slpFixPenalties will automatically reoptimize the linearization. However, as the XSLP convergence and infeasibility checks (regarding the original non-linear problem) will not be carried out, this function will not update the SLP solution itself. The updated values will be accessible using problem.getLpSol instead.

problem.nlpPostsolveProb

Purpose

Restores the problem to its pre-solve state

Topic areas

Presolve, Nonlinear

Synopsis

```
problem.nlpPostsolveProb()
```

Related controls

Integer

NLPPOSTSOLVE	Determines if postsolve is applied automatically.

Further information

If Xpress-SLP was used to solve the problem, postsolve will unconstruct the problem before postsolving (including any reformulation that might have been applied)

Related topics

NLPPOSTSOLVE (p.**??** Xpress NonLinear Reference Manual)

problem.nlpCalcSlacks

Purpose

Calculate the slack values for the provided solution in the non-linear problem

Topic areas

Solution, Nonlinear

Synopsis

```
slack = problem.nlpCalcSlacks(solution)
```

Argument

`solution: list[float]` The solution for which the slacks are requested.

Return value

`slack: list[float]` Array of length `ROWS` containing the slacks.

Related topics

`problem.nlpValidate`, `problem.nlpValidateRow`

problem.slpSetInitStepBounds

Purpose

Set the initial step bounds of columns

Topic areas

Data Input, Nonlinear

Synopsis

```
problem.slpSetInitStepBounds(colind, initial)
```

Arguments

`colind: list[var]` Array with index of the column for which the initial step bound is provided.

`initial: list[float]` Array with the initial step bounds.

Further information

Argument `colind` may contain either `xpress.var` objects or integer indices.

problem.addLplogCallback

Purpose

Declares a simplex log callback function which is called after every `LPLOG` iterations of the simplex algorithm. This callback function will be called in addition to any callbacks already added by problem.addLplogCallback.

Topic areas

Callback, LP

Synopsis

```
problem.addLplogCallback(lplog, data=None, priority=0)
lplog(cbprob, cbdata)
```

Arguments

`lplog` The callback function, which takes arguments `cbprob` and `cbdata`, and has an integer return value. This function is called every `LPLOG` simplex iterations including iteration `0` and the final iteration.

`data: Any` A user-defined data to be passed to the callback function, `lplog`.

`priority: int` An integer that determines the order in which multiple lplog callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function, `lplog`.

`cbdata: Any` The user-defined data passed as `data` when setting up the callback.

Related controls

Integer

`LPLOG` Frequency and type of simplex algorithm log.

Further information

If the callback function returns a nonzero value the solution process will be interrupted.

Related topics

`problem.removeLplogCallback`, `problem.addBarlogCallback`, `problem.addMiplogCallback`, `problem.addMessageCallback`.

problem.removeLplogCallback

Purpose

Removes a simplex log callback function previously added by `problem.addLplogCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Logging

Synopsis

```
problem.removeLplogCallback(lplog, data=None)
lplog(cbprob, cbdata)
```

Arguments

`lplog`	The callback function to remove. If None then all lplog callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If None, then the data value will not be checked and all lplog callbacks with the function `lplog` will be removed.

Related topics

`problem.addLplogCallback`

problem.addMiplogCallback

Purpose

Declares a MIP log callback function, called each time the MIP log is printed. This callback function will be called in addition to any callbacks already added by problem.addMiplogCallback.

Topic areas

Callback, Logging

Synopsis

```
problem.addMiplogCallback(miplog, data=None, priority=0)
miplog(cbprob, cbdata)
```

Arguments

`miplog` The callback function, which takes arguments `cbprob` and `cbdata`, and has an integer return value. This function is called whenever the MIP log is printed as determined by the `MIPLOG` control.

`data: Any` A user-defined data to be passed to the callback function, `miplog`.

`priority: int` An integer that determines the order in which multiple MIP log callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function, `miplog`.

`cbdata: Any` The user-defined data passed as `data` when setting up the callback.

Related controls

Integer

`MIPLOG` MIP print flag.

Further information

If the callback function returns a nonzero value, the tree search will be interrupted.

Related topics

`problem.removeMiplogCallback`, `problem.addBarlogCallback`, `problem.addLplogCallback`, `problem.addMessageCallback`.

problem.removeMiplogCallback

Purpose

Removes a MIP log callback function previously added by `problem.addMiplogCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Logging

Synopsis

```
problem.removeMiplogCallback(miplog, data=None)
miplog(cbprob, cbdata)
```

Arguments

`miplog`	The callback function to remove. If None then all MIP log callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If None, then the data value will not be checked and all MIP log callbacks with the function `miplog` will be removed.

Related topics

`problem.addMiplogCallback`

problem.addCutlogCallback

Purpose

Declares a cut log callback function, called each time the cut log is printed. This callback function will be called in addition to any callbacks already added by problem.addCutlogCallback.

Topic areas

Callback, Logging

Synopsis

```
problem.addCutlogCallback(cutlog, data=None, priority=0)
result = cutlog(cbprob, cbdata)
```

Arguments

`cutlog` The callback function, which takes arguments `cbprob` and `cbdata`, and has an integer return value.

`data: Any` A user-defined data to be passed to the callback function, `cutlog`.

`priority: int` An integer that determines the order in which multiple cut log callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function, `cutlog`.

`cbdata: Any` The user-defined data passed as `data` when setting up the callback.

Return value

`result: int` The result of the callback

Further information

Return a non-zero value from `cutlog` to stop cutting on the current node.

Related topics

`problem.removeCutlogCallback`.

problem.removeCutlogCallback

Purpose

Removes a cut log callback function previously added by `problem.addCutlogCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Logging

Synopsis

```
problem.removeCutlogCallback(cutlog, data=None)
result = cutlog(cbprob, cbdata)
```

Arguments

`cutlog`	The callback function to remove. If None then all cut log callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If None, then the data value will not be checked and all cut log callbacks with the function `cutlog` will be removed.

Related topics

`problem.addCutlogCallback`

problem.addBarlogCallback

Purpose

Declares a barrier log callback function, called at each iteration during the interior point algorithm. This callback function will be called in addition to any barrier log callbacks already added by problem.addBarlogCallback.

This functionality is not available for the hybrid gradient method.

Topic areas

Callback, Barrier

Synopsis

```
problem.addBarlogCallback(barlog, data=None, priority=0)
barlog(cbprob, cbdata)
```

Arguments

`barlog` The callback function itself, which takes arguments `cbprob` and `cbdata`, and has an integer return value. If the value returned by `barlog` is nonzero, the solution process will be interrupted. This function is called at every barrier iteration.

`data: Any` A user-defined data to be passed to the callback function, `barlog`.

`priority: int` An integer that determines the order in which multiple barrier log callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function, `barlog`.

`cbdata: Any` The user-defined data passed as `data` when setting up the callback.

Further information

If the callback function returns a nonzero value, the Optimizer run will be interrupted.

Related topics

`problem.removeBarlogCallback`, `problem.addMiplogCallback`, `problem.addLplogCallback`, `problem.addMessageCallback`.

problem.removeBarlogCallback

Purpose

Removes a Newton barrier log callback function previously added by `problem.addBarlogCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Barrier

Synopsis

```
problem.removeBarlogCallback(barlog, data=None)
barlog(cbprob, cbdata)
```

Arguments

`barlog`	The callback function to remove. If None then all barrier log callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If None, then the data value will not be checked and all barrier log callbacks with the function `barlog` will be removed.

Related topics

`problem.addBarlogCallback`.

problem.addOptnodeCallback

Purpose

Declares an optimal node callback function, called during the branch and bound search, after the LP relaxation has been solved for the current node, and after any internal cuts and heuristics have been applied, but before the Optimizer checks if the current node should be branched. This callback function will be called in addition to any callbacks already added by problem.addOptnodeCallback.

Topic areas

Callback, Branch and Bound Search

Synopsis

```
problem.addOptnodeCallback(optnode, data=None, priority=0)
infeasible = optnode(cbprob, cbdata)
```

Arguments

`optnode` The callback function, which takes arguments `cbprob`, `cbdata` and `infeasible`, and has no return value.

`data: Any` A user-defined data to be passed to the callback function, `optnode`.

`priority: int` An integer that determines the order in which multiple node-optimal callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function, `optnode`.

`cbdata: Any` The user-defined data passed as `data` when setting up the callback.

Return value

`infeasible: int` The feasibility status. If set to a nonzero value by the user, the current node will be declared infeasible.

Related topics

`problem.removeOptnodeCallback`, `problem.addInfnodeCallback`, `problem.addIntsolCallback`, `problem.addNodecutoffCallback`, `CALLBACKCOUNT_OPTNODE` (p.**??** Xpress Optimizer Reference Manual).

problem.removeOptnodeCallback

Purpose

Removes a node-optimal callback function previously added by `problem.addOptnodeCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Branch and Bound Search

Synopsis

```
problem.removeOptnodeCallback(optnode, data=None)
infeasible = optnode(cbprob, cbdata)
```

Arguments

`optnode`	The callback function to remove. If None then all node-optimal callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If None, then the data value will not be checked and all node-optimal callbacks with the function `optnode` will be removed.

Related topics

`problem.addOptnodeCallback`

problem.addPrenodeCallback

Purpose

Declares a preprocess node callback function, called before the LP relaxation of a node has been optimized, so the solution at the node will not be available. This callback function will be called in addition to any callbacks already added by problem.addPrenodeCallback.

Topic areas

Callback, Branch and Bound Search

Synopsis

```
problem.addPrenodeCallback(prenode, data=None, priority=0)
infeasible = prenode(cbprob, cbdata)
```

Arguments

`prenode` The callback function, which takes arguments `cbprob`, `cbdata` and `infeasible`, and has no return value. This function is called before a node is reoptimized and the node may be made infeasible by setting `infeasible` to `1`.

`data: Any` A user-defined data to be passed to the callback function, `prenode`.

`priority: int` An integer that determines the order in which multiple preprocess node callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function, `prenode`.

`cbdata: Any` The user-defined data passed as `data` when setting up the callback.

Return value

`infeasible: int` The feasibility status. If set to a nonzero value by the user, the current node will be declared infeasible by the Optimizer.

Related topics

`problem.removePrenodeCallback`, `problem.addInfnodeCallback`, `problem.addIntsolCallback`, `problem.addNodecutoffCallback`, `problem.addOptnodeCallback`.

problem.removePrenodeCallback

Purpose

Removes a preprocess node callback function previously added by `problem.addPrenodeCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Branch and Bound Search

Synopsis

```
problem.removePrenodeCallback(prenode, data=None)
infeasible = prenode(cbprob, cbdata)
```

Arguments

`prenode`	The callback function to remove. If None then all preprocess node callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If None, then the data value will not be checked and all preprocess node callbacks with the function `prenode` will be removed.

Related topics

`problem.addPrenodeCallback`

problem.addInfnodeCallback

Purpose

Declares a user infeasible node callback function, called after the current node has been found to be infeasible during the Branch and Bound search. The callback is also invoked if the current node gets cut off, i.e., if its objective is proven to exceed the current primal bound. This callback function will be called in addition to any callbacks already added by problem.addInfnodeCallback.

Topic areas

Callback, Branch and Bound Search

Synopsis

```
problem.addInfnodeCallback(infnode, data=None, priority=0)
infnode(cbprob, cbdata)
```

Arguments

`infnode` The callback function, which takes arguments `cbprob` and `cbdata`, and has no return value. This function is called after the current node has been found to be infeasible.

`data: Any` A user-defined data to be passed to the callback function, `infnode`.

`priority: int` An integer that determines the order in which multiple user infeasible node callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function, `infnode`.

`cbdata: Any` The user-defined data passed as `data` when setting up the callback.

Further information

Nodes that are cut off while *not* being the active node, are reported in the `nodecutoff` callback.

Related topics

`problem.removeInfnodeCallback`, `problem.addOptnodeCallback`, `problem.addIntsolCallback`, `problem.addNodecutoffCallback`.

problem.removeInfnodeCallback

Purpose

Removes a user infeasible node callback function previously added by `problem.addInfnodeCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Branch and Bound Search

Synopsis

```
problem.removeInfnodeCallback(infnode, data=None)
infnode(cbprob, cbdata)
```

Arguments

`infnode`	The callback function to remove. If None then all user infeasible node callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If None, then the data value will not be checked and all user infeasible node callbacks with the function `infnode` will be removed.

Related topics

`problem.addInfnodeCallback`

problem.addNodecutoffCallback

Purpose

Declares a user node cutoff callback function, called every time a node is cut off as a result of an improved integer solution being found during the branch and bound search. This callback function will be called in addition to any callbacks already added by problem.addNodecutoffCallback.

Topic areas

Callback, Branch and Bound Search

Synopsis

```
problem.addNodecutoffCallback(nodecutoff, data=None, priority=0)
nodecutoff(cbprob, cbdata, node)
```

Arguments

`nodecutoff` The callback function, which takes arguments `cbprob`, `cbdata` and `node`, and has no return value. This function is called every time a node is cut off as the result of an improved integer solution being found.

`data: Any` A user-defined data to be passed to the callback function, `nodecutoff`.

`priority: int` An integer that determines the order in which multiple node-optimal callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function, `nodecutoff`.

`cbdata: Any` The user-defined data passed as `data` when setting up the callback.

`node: int` The node id of the node that is cut off. This id cannot be queried from the `CURRENTNODE` (p.**??** Xpress Optimizer Reference Manual) attribute as for other callbacks since this callback is not invoked in the context of the node being cutoff.

Further information

1. This function allows the user to keep track of the eligible nodes. Note that the LP solution will not be available from this callback.
2. This callback is invoked only for nodes that are pruned from the active node pool. If the *current* node gets cut off then the `infnode` callback is invoked to report this.

Related topics

`problem.removeNodecutoffCallback`, `problem.addOptnodeCallback`, `problem.addInfnodeCallback`, `problem.addIntsolCallback`.

problem.removeNodecutoffCallback

Purpose

Removes a node-cutoff callback function previously added by `problem.addNodecutoffCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Branch and Bound Search

Synopsis

```
problem.removeNodecutoffCallback(nodecutoff, data=None)
nodecutoff(cbprob, cbdata, node)
```

Arguments

`nodecutoff`	The callback function to remove. If None then all node-cutoff callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If None, then the data value will not be checked and all node-cutoff callbacks with the function `nodecutoff` will be removed.

Related topics

`problem.addNodecutoffCallback`

problem.addIntsolCallback

Purpose

Declares a user integer solution callback function, called every time an integer solution is found by heuristics or during the Branch and Bound search. This callback function will be called in addition to any callbacks already added by problem.addIntsolCallback.

Topic areas

Callback, Solution

Synopsis

```
problem.addIntsolCallback(intsol, data=None, priority=0)
intsol(cbprob, cbdata)
```

Arguments

`intsol` The callback function, which takes arguments `cbprob` and `cbdata`, and has no return value. This function is called if the current node is found to have an integer feasible solution, i.e. every time an integer feasible solution is found.

`data: Any` A user-defined data to be passed to the callback function, `intsol`.

`priority: int` An integer that determines the order in which multiple integer solution callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function, `intsol`.

`cbdata: Any` The user-defined data passed as `data` when setting up the callback.

Further information

1. This callback is useful if the user wants to retrieve the integer solution when it is found.
2. To retrieve the integer solution, use `problem.getCallbackSolution`, `problem.getCallbackPresolveSolution` or `problem.getSolution`.
3. This callback is called after a new integer solution was found by the Optimizer. Use a callback set by `problem.addPreIntsolCallback` in order to be notified *before* a new integer solution is accepted by the Optimizer, which allows for the new solution to be rejected.
4. For this callback attribute `MIPINFEAS` (p.**??** Xpress Optimizer Reference Manual) is set to -1 if the solution was found by a heuristic.

Related topics

`problem.removeIntsolCallback`, `problem.addPreIntsolCallback`.

problem.removeIntsolCallback

Purpose

Removes an integer solution callback function previously added by `problem.addIntsolCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Solution

Synopsis

```
problem.removeIntsolCallback(intsol, data=None)
intsol(cbprob, cbdata)
```

Arguments

`intsol`	The callback function to remove. If None then all integer solution callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If None, then the data value will not be checked and all integer solution callbacks with the function `intsol` will be removed.

Related topics

`problem.addIntsolCallback`

problem.addPreIntsolCallback

Purpose

Declares a user integer solution callback function, called when an integer solution is found by heuristics or during the branch and bound search, but before it is accepted by the Optimizer. This callback allows the user to accept or reject a candidate MIP solution before it is accepted as the new incumbent, and in some cases allows the user to modify the node problem (see notes below). This callback function will be called in addition to any integer solution callbacks already added by problem.addPreIntsolCallback.

Topic areas

Callback, Solution

Synopsis

```
problem.addPreIntsolCallback(preintsol, data=None, priority=0)
reject, cutoff = preintsol(cbprob, cbdata, soltype, cutoff)
```

Arguments

`preintsol` The callback function, which takes arguments `cbprob`, `cbdata`, `soltype`, `reject` and `cutoff`, and has no return value. This function is called when an integer solution is found, but before the solution is accepted by the Optimizer, allowing the user to reject the solution. Note that the integer solution could be rejected by the Optimizer even if the user does not reject it. This can happen in the context of a parallel solve if, before accepting the solution, a better one is found by another task running in parallel.

`data: Any` A user-defined data to be passed to the callback function, `preintsol`.

`priority: int` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function, `preintsol`.

`cbdata: Any` The user-defined data passed as data when setting up the callback.

`soltype: int` The type of MIP solution that has been found:

- `0` The continuous relaxation solution to the current node of the tree search, which has been found to be integer feasible.
- `1` A MIP solution found by a heuristic.
- `2` A MIP solution provided by the user.

`cutoff: float` The new cutoff value that the Optimizer will use if the solution is accepted.

Return value

`reject: int` Set this to 1 if the solution should be rejected. When `soltype` is zero, this will also drop the node problem.

`cutoff: float` If `cutoff` is assigned a value, the new value will be used instead. The cutoff value will not be updated if the solution is rejected.

Related controls

Integer

MIPABSCUTOFF

Further information

1. To retrieve the integer solution, use `problem.getCallbackSolution` or `problem.getCallbackPresolveSolution`. The matching objective value should be queried through the `LPOBJVAL` (p.**??** Xpress Optimizer Reference Manual) attribute. `problem.getSolution` and `MIPOBJVAL` (p.**??** Xpress Optimizer Reference Manual) will not return the newly found solution and matching objective value, because the incumbent solution has not been updated at this point.
2. When `soltype` is zero, the current node problem can be modified in this callback, including adding cuts, changing bounds and adding branching objects. If the problem is changed, the node LP relaxation will be resolved and checked for branching candidates again. This might retrigger any `optnode` (ref. `problem.addOptnodeCallback`) callback and the node will be branched, if required. Making any changes when `soltype` is not zero will raise an error.
3. When `soltype` is zero, it is possible to add new solutions using `problem.addMipSol`. If the node problem was not modified during the callback, any added solution will be checked after the current solution has been accepted as a new incumbent. Otherwise, the added solution will be checked after the node relaxation has been resolved.
4. When `soltype` is zero and the solution is rejected by setting `reject`, the node itself will be dropped without further branching, regardless of whether the node problem is modified within the callback. Dropping the node implies dropping any cuts or solutions you added via `problem.addCuts` or `problem.addMipSol`, respectively. Adding cuts via `problem.addCuts` (in case `soltype` is zero) will implicitly reject the solution, there is no need to set `reject` in that case.
5. The cutoff value returned in `cutoff` is applied only if the solution is not rejected and the node problem (when `soltype` is zero) is not modified. To change the cutoff value when rejecting a solution, the control `MIPABSCUTOFF` should be set instead.
6. For this callback attribute `MIPINFEAS` (p.**??** Xpress Optimizer Reference Manual) is set to -1 if the solution was found by a heuristic.

Related topics

`problem.removePreIntsolCallback`, `problem.addOptnodeCallback`, `problem.addIntsolCallback`, `problem.addCuts`, `problem.addMipSol`.

problem.removePreIntsolCallback

Purpose

Removes a pre-integer solution callback function previously added by `problem.addPreIntsolCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Solution

Synopsis

```
problem.removePreIntsolCallback(preintsol, data=None)
reject, cutoff = preintsol(cbprob, cbdata, soltype, cutoff)
```

Arguments

`preintsol`	The callback function to remove. If None then all user pre-integer solution callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If None, then the data value will not be checked and all user pre-integer solution callbacks with the function `preintsol` will be removed.

Related topics

`problem.addPreIntsolCallback`

problem.addMessageCallback

Purpose

Declares an output callback function, called every time a text line relating to the given problem is output by the Optimizer. This callback function will be called in addition to any callbacks already added by `problem.addMessageCallback`. Note that Optimizer messages passed to the callback do not end with a newline character; the user callback is expected to append such required newline characters itself.

Topic areas

Callback, Logging

Synopsis

```
problem.addMessageCallback(message, data=None, priority=0)
message(cbprob, cbdata, msg, msgtype)
```

Arguments

`message` The callback function, which takes arguments `cbprob`, `cbdata`, `msg` and `msgtype`, and has no return value.

`data: Any` A user-defined data to be passed to the callback function.

`priority: int` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function.

`cbdata: Any` The user-defined data passed as `data` when setting up the callback.

`msg: str` A None terminated character array (string) containing the message, which may simply be a new line. The total number of bytes (including `NUL` terminator) will not exceed `512`. If a message needs to be truncated to meet this limit, the last four bytes in `msg` are set to "...\0".

`msgtype: int` Indicates the type of output message:

- `1` information messages;
- `2` (not used);
- `3` warning messages;
- `4` error messages.

A negative value indicates that the Optimizer is about to finish and the buffers should be flushed at this time if the output is being redirected to a file.

Related controls

Integer

`OUTPUTLOG` All messages are disabled if set to zero.

Further information

1. Screen output is automatically created by the Optimizer Console only. To produce output when using the Optimizer library, it is necessary to define this callback function and use it to print the messages to the screen (`stdout`).
2. This function offers one method of handling the messages which describe any warnings and errors that may occur during execution.
3. Querying problem data or changing controls is not possible within this callback.

Related topics

`problem.removeMessageCallback`, `problem.addBarlogCallback`, `problem.addMiplogCallback`, `problem.addLplogCallback`, `problem.setLogFile`.

problem.removeMessageCallback

Purpose

Removes a message callback function previously added by `problem.addMessageCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Logging

Synopsis

```
problem.removeMessageCallback(message, data=None)
message(cbprob, cbdata, msg, msgtype)
```

Arguments

`message`	The callback function to remove. If None then all message callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If None, then the data value will not be checked and all message callbacks with the function `message` will be removed.

Related topics

`problem.addMessageCallback`

problem.addMipThreadCallback

Purpose

Declares a MIP thread callback function, called every time a MIP worker problem is created by the parallel MIP code. This callback function will be called in addition to any callbacks already added by problem.addMipThreadCallback.

Topic areas

Callback, Parallel

Synopsis

```
problem.addMipThreadCallback(mipthread, data=None, priority=0)
mipthread(cbprob, cbdata, threadprob)
```

Arguments

`mipthread` The callback function, which takes arguments `cbprob`, `cbdata` and `threadprob`, and has no return value.

`data: Any` A user-defined data to be passed to the callback function.

`priority: int` An integer that determines the order in which multiple callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function.

`cbdata: Any` The user-defined data passed to the callback function.

`threadprob: problem` The new problem for the MIP thread

Related controls

Integer

`MIPTHREADS` Number of MIP threads to create.

Further information

This function will be called when a new MIP worker problem is created. Each worker problem receives a unique identifier that can be obtained through the `MIPTHREADID` (p.**??** Xpress Optimizer Reference Manual) attribute. Worker problems can be matched with different system threads at different points of a solve, so the system thread that is responsible for executing the callback is not necessarily the same thread used for all subsequent callbacks for the same worker problem. On the other hand, worker problems are always assigned to a single thread at a time and the same nodes are always solved on the same worker problem in repeated runs of a deterministic MIP solve. A worker problem therefore acts as a virtual thread through the node solves.

Related topics

`problem.removeMipThreadCallback`,`problem.addMipThreadDestroyCallback`, `MIPTHREADS` (p.**??** Xpress Optimizer Reference Manual), `MAXMIPTASKS` (p.**??** Xpress Optimizer Reference Manual).

problem.removeMipThreadCallback

Purpose

Removes a callback function previously added by `problem.addMipThreadCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Parallel

Synopsis

```
problem.removeMipThreadCallback(mipthread, data=None)
mipthread(cbprob, cbdata, threadprob)
```

Arguments

`mipthread`	The callback function to remove. If None then all variable branching callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If None, then the data value will not be checked and all variable branching callbacks with the function `mipthread` will be removed.

Related topics

`problem.addMipThreadCallback`

problem.addMipThreadDestroyCallback

Purpose

Declares a destroy MIP thread callback function, called every time a MIP thread is destroyed by the parallel MIP code. This callback function will be called in addition to any callbacks already added by problem.addMipThreadDestroyCallback.

Topic areas

Callback, Parallel

Synopsis

```
problem.addMipThreadDestroyCallback(destroymt, data=None, priority=0)
destroymt(cbprob, cbdata)
```

Arguments

`destroymt` The callback function, which takes arguments `cbprob` and `cbdata`, and has no return value.

`data: Any` A user-defined data to be passed to the callback function.

`priority: int` An integer that determines the order in which multiple callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The thread problem passed to the callback function.

`cbdata: Any` The user-defined data passed as `data` when setting up the callback.

Related controls

Integer

`MIPTHREADS` Number of MIP threads to create.

Further information

1. This callback is useful for freeing up any user data created in the MIP thread callback.
2. Querying problem data or changing controls is not possible within this callback.

Related topics

`problem.removeMipThreadDestroyCallback`,`problem.addMipThreadCallback`.

problem.removeMipThreadDestroyCallback

Purpose

Removes a child thread destruction callback function previously added by `problem.addMipThreadDestroyCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Parallel

Synopsis

```
problem.removeMipThreadDestroyCallback(destroymt, data=None)
destroymt(cbprob, cbdata)
```

Arguments

`destroymt`	The callback function to remove. If None then all thread destruction callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If None, then the data value will not be checked and all thread destruction callbacks with the function `destroymt` will be removed.

Related topics

`problem.addMipThreadDestroyCallback`

problem.addBarIterationCallback

Purpose

Declares a barrier iteration callback function, called after each iteration during the interior point algorithm, with the ability to access the current barrier solution/slack/duals or reduced cost values, and to ask barrier to stop. This callback function will be called in addition to any callbacks already added by problem.addBarIterationCallback.

This functionality is not available for the hybrid gradient method.

Topic areas

Callback, Barrier

Synopsis

```
problem.addBarIterationCallback(bariteration, data=None, priority=0)
action = bariteration(cbprob, cbdata)
```

Arguments

`bariteration` The callback function, which takes arguments `cbprob`, `cbdata`, and `action` serving as an integer return value. This function is called at every barrier iteration.

`data: Any` A user-defined data to be passed to the callback function, `bariteration`.

`priority: int` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function, `bariteration`.

`cbdata: Any` The user-defined data passed as `data` when setting up the callback.

Return value

`action: int` Defines a return value controlling barrier:

<0	continue with the next iteration;
=0	let barrier decide (use default stopping criteria);
1	barrier stops with status not defined;
2	barrier stops with optimal status;
3	barrier stops with dual infeasible status;
4	barrier stops wih primal infeasible status.

Further information

1. Only the functions for retrieving the solution and the attribute/control value retrieving and setting routines are expected to be called from the callback.
2. General barrier iteration values are available via the following attributes:
   - `BARPRIMALOBJ` (p.**??** Xpress Optimizer Reference Manual) - current primal objective
   - `BARDUALOBJ` (p.**??** Xpress Optimizer Reference Manual) - current dual objective
   - `BARPRIMALINF` (p.**??** Xpress Optimizer Reference Manual) - current primal infeasibility
   - `BARDUALINF` (p.**??** Xpress Optimizer Reference Manual) - current dual infeasibility
   - `BARCGAP` (p.**??** Xpress Optimizer Reference Manual) - current complementary gap
3. Please note that these values refer to the scaled and presolved problem used by barrier, and may differ from the ones calculated from the postsolved solution returned by `problem.getCallbackSolution`.

Related topics

`problem.removeBarIterationCallback`.

problem.removeBarIterationCallback

Purpose

Removes a barrier iteration callback function previously added by `problem.addBarIterationCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Barrier

Synopsis

```
problem.removeBarIterationCallback(bariteration, data=None)
action = bariteration(cbprob, cbdata)
```

Arguments

`bariteration` The callback function to remove. If None then all bariteration callback functions added with the given user-defined data value will be removed.

`data: Any` The data value that the callback was added with. If None, then the data value will not be checked and all barrier iteration callbacks with the function `bariteration` will be removed.

Related topics

`problem.addBarIterationCallback`.

problem.addPresolveCallback

Purpose

Declares a callback to be called after presolve has been performed.

Topic areas

Callback, Presolve

Synopsis

```
problem.addPresolveCallback(presolve, data=None, priority=0)
presolve(cbprob, cbdata)
```

Arguments

`presolve` The callback function itself, which takes arguments `cbprob` and `cbdata`, and has no return value. This function is called after presolve is complete.

`data: Any` A user-defined data to be passed to the callback function, `presolve`.

`priority: int` An integer that determines the order in which multiple presolve callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function.

`cbdata: Any` The user-defined data passed as `data` when setting up the callback.

Related topics

`problem.removePresolveCallback`

problem.removePresolveCallback

Purpose

Removes a presolve callback function previously added by `problem.addPresolveCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Presolve

Synopsis

```
problem.removePresolveCallback(presolve, data=None)
presolve(cbprob, cbdata)
```

Arguments

`presolve`	The callback function to remove. If None then all presolve callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If None, then the data value will not be checked and presolve callbacks with the function `presolve` will be removed.

Related topics

`problem.addPresolveCallback`.

problem.addNewnodeCallback

Purpose

Declares a callback function that will be called every time a new node is created during the branch and bound search. This callback function will be called in addition to any callbacks already added by problem.addNewnodeCallback.

Topic areas

Callback, Branch and Bound Search

Synopsis

```
problem.addNewnodeCallback(newnode, data=None, priority=0)
newnode(cbprob, cbdata, parentnode, node, branch)
```

Arguments

`newnode` The callback function, which takes arguments `myprob`, `cbdata`, `parentnode`, `node` and `branch`. This function is called every time a new node is created through branching.

`data: Any` A user-defined data to be passed to the callback function.

`priority: int` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function, `newnode`.

`cbdata: Any` The user-defined data passed as `data` when setting up the callback.

`parentnode: int` Unique identifier for the parent of the new node.

`node: int` Unique identifier assigned to the new node.

`branch: int` The sequence number of the new node amongst the child nodes of `parentnode`. For regular branches on a MIP entity this will be either `0` or `1`.

Further information

1. For regular branches on a MIP entity, `branch` will be either zero or one, depending on whether the new node corresponds to branching the MIP entity up or down.
2. When branching on a custom `xpress.branchobj`, `branch` refers to the given branch index of the data.

Related topics

`problem.removeNewnodeCallback`.

problem.removeNewnodeCallback

Purpose

Removes a new-node callback function previously added by `problem.addNewnodeCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Branch and Bound Search

Synopsis

```
problem.removeNewnodeCallback(newnode, data=None)
newnode(cbprob, cbdata, parentnode, node, branch)
```

Arguments

`newnode`	The callback function to remove. If None then all separation callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If None, then the data value will not be checked and all separation callbacks with the function `newnode` will be removed.

Related topics

`problem.addNewnodeCallback`

problem.addChangeBranchObjectCallback

Purpose

Declares a callback function that will be called after the selection of a MIP entity to branch on. This callback allows the user to inspect and replace the Optimizer's choice of how to branch the current node. This callback will also be called in the case when there are no candidates to branch on, that is, when all MIP entities are already satisfied. This callback function will be called in addition to any callbacks already added by problem.addChangeBranchObjectCallback.

Topic areas

Callback, Branching

Synopsis

```
problem.addChangeBranchObjectCallback(chgbranchobject, data=None, priority=0)
newbranch = chgbranchobject(cbprob, cbdata, branch)
```

Arguments

`chgbranchobject` The callback function, which takes arguments `cbprob`, `cbdata`, `branch` and `newbranch`. This function is called every time the Optimizer has selected a candidate entity for branching.

`data: Any` A user-defined data to be passed to the callback function, `chgbranchobject`.

`priority: int` An integer that determines the order in which multiple callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function, `chgbranchobject`.

`cbdata: Any` The user defined data passed as `data` when setting up the callback.

`branch: branchobj` The candidate branching data selected by the Optimizer. Will be `None` if no candidates exist.

Return value

`newbranch: branchobj` Optional new branching data to replace the Optimizer's selection. If `branch` or `None` is passed back, no change will be applied.

Further information

1. The branching data given by the Optimizer provides a linear description of how the Optimizer intends to branch on the selected candidate. This will often be one of standard MIP entities of the current problem, but can also be e.g. a split disjunction or a structural branch, if those features are turned on.
2. The functions `branchobj.getBranches`, `branchobj.getBounds` and `branchobj.getRows` can be used to inspect the given branching data.
3. Refer to `xpress.branchobj` on how to create a new branching data to replace the Optimizer's selection. Note that the new branching data should be created with a priority value no higher than the current data to guarantee it will be used for branching.

Related topics

`problem.removeChangeBranchObjectCallback`, `xpress.branchobj`.

problem.removeChangeBranchObjectCallback

Purpose

Removes a callback function previously added by `problem.addChangeBranchObjectCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Branching

Synopsis

```
problem.removeChangeBranchObjectCallback(chgbranchobject, data=None)
newbranch = chgbranchobject(cbprob, cbdata, branch)
```

Arguments

`chgbranchobject` The callback function to remove. If None then all branch data callback functions added with the given user-defined data value will be removed.

`data: Any` The data value that the callback was added with. If None, then the data value will not be checked and all branch data callbacks with the function `chgbranchobject` will be removed.

Related topics

`problem.addChangeBranchObjectCallback`

problem.addGapNotifyCallback

Purpose

Declares a gap notification callback, to be called when a MIP solve reaches a predefined target, set using the MIPRELGAPNOTIFY (p.**??** Xpress Optimizer Reference Manual), MIPABSGAPNOTIFY (p.**??** Xpress Optimizer Reference Manual), MIPABSGAPNOTIFYOBJ (p.**??** Xpress Optimizer Reference Manual) and/or MIPABSGAPNOTIFYBOUND (p.**??** Xpress Optimizer Reference Manual) controls.

Topic areas

Callback, Limits

Synopsis

```
problem.addGapNotifyCallback(gapnotify, data=None, priority=0)
relgapnotifytarget, absgapnotifytarget, absgapnotifyobjtarget,
       absgapnotifyboundtarget = gapnotify(cbprob, cbdata)
```

Arguments

`gapnotify` The callback function.

`data: Any` A user-defined data to be passed to the callback function, `gapnotify`.

`priority: int` An integer that determines the order in which multiple estimate callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The current problem.

`cbdata: Any` The user-defined data passed as `data` when setting up the callback.

Return value

`relgapnotifytarget: float` The value the MIPRELGAPNOTIFY (p.**??** Xpress Optimizer Reference Manual) control will be set to after this callback. May be modified within the callback in order to set a new notification target.

`absgapnotifytarget: float` The value the MIPABSGAPNOTIFY (p.**??** Xpress Optimizer Reference Manual) control will be set to after this callback. May be modified within the callback in order to set a new notification target.

`absgapnotifyobjtarget: float` The value the MIPABSGAPNOTIFYOBJ (p.**??** Xpress Optimizer Reference Manual) control will be set to after this callback. May be modified within the callback in order to set a new notification target.

`absgapnotifyboundtarget: float` The value the MIPABSGAPNOTIFYBOUND (p.**??** Xpress Optimizer Reference Manual) control will be set to after this callback. May be modified within the callback in order to set a new notification target.

Further information

The target values that caused the callback to be triggered will automatically be reset to prevent the same callback from being fired again.

Related topics

MIPRELGAPNOTIFY (p.**??** Xpress Optimizer Reference Manual), MIPABSGAPNOTIFY (p.**??** Xpress Optimizer Reference Manual), MIPABSGAPNOTIFYOBJ (p.**??** Xpress Optimizer Reference Manual), MIPABSGAPNOTIFYBOUND (p.**??** Xpress Optimizer Reference Manual), `problem.removeGapNotifyCallback`.

problem.removeGapNotifyCallback

Purpose

Removes a callback function previously added by `problem.addGapNotifyCallback`. The specified callback function will no longer be removed after it has been returned.

Topic areas

Callback, Limits

Synopsis

```
problem.removeGapNotifyCallback(gapnotify, data=None)
relgapnotifytarget, absgapnotifytarget, absgapnotifyobjtarget,
       absgapnotifyboundtarget = gapnotify(cbprob, cbdata)
```

Arguments

`gapnotify`	The callback function to remove. If `None` then all `gapnotify` callback functions added with the given user-defined object will be removed.
`data: Any`	The user-defined context object that the callback was added with. If `None` then the context object will not be checked and all the `gapnotify` callbacks with the function `gapnotify` will be removed.

Related topics

`problem.addGapNotifyCallback`.

problem.addUserSolNotifyCallback

Purpose

Declares a callback function to be called each time a solution added by `problem.addMipSol` has been processed. This callback function will be called in addition to any callbacks already added by `problem.addUserSolNotifyCallback`.

Topic areas

Callback, Solution

Synopsis

```
problem.addUserSolNotifyCallback(usersolnotify, data=None, priority=0)
usersolnotify(cbprob, cbdata, solname, status)
```

Arguments

`usersolnotify` The callback function, which takes arguments `cbprob`, `cbdata`, `solname` and `status` and has no return value.

`data: Any` A user-defined data to be passed to the callback function, `usersolnotify`.

`priority: int` An integer that determines the order in which multiple callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to `0` if not required.

`cbprob: problem` The problem passed to the callback function, `usersolnotify`.

`cbdata: Any` The user-defined data passed as data when setting up the callback.

`solname: str` The string name assigned to the solution when it was loaded into the Optimizer using `problem.addMipSol`.

`status: int` One of the following status values:

- `1` Solution is feasible. (`xpress.UserSolStatus.ACCEPTED_FEASIBLE`)
- `2` Solution is feasible after reoptimizing with fixed MIP entities. (`xpress.UserSolStatus.ACCEPTED_OPTIMIZED`)
- `3` A local search heuristic was applied and a feasible solution discovered. (`xpress.UserSolStatus.SEARCHED_SOL`)
- `4` A local search heuristic was applied but a feasible solution was not found. (`xpress.UserSolStatus.SEARCHED_NOSOL`)
- `5` Solution is infeasible and a local search could not be applied. (`xpress.UserSolStatus.REJECTED_INFEAS_NOSEARCH`)
- `6` Solution is partial and a local search could not be applied. (`xpress.UserSolStatus.REJECTED_PARTIAL_NOSEARCH`)
- `7` Failed to reoptimize the problem with MIP entities fixed to the provided solution. Likely because a time or iteration limit was reached. (`xpress.UserSolStatus.REJECTED_FAILED_OPTIMIZE`)
- `8` Solution is dropped. This can happen if the MIP problem is changed or solved to completion before the solution could be processed. (`xpress.UserSolStatus.DROPPED`)
- `9` The solution is worse than the current MIP cutoff value. (`xpress.UserSolStatus.REJECTED_CUTOFF`)

Further information

If presolve is turned on, any solution added with `problem.addMipSol` will first be presolved before it can be checked. The value returned in `status` refers to the presolved solution, which might have had values adjusted due to bound changes, fixing of variables, etc.

Related topics

`problem.removeUserSolNotifyCallback`, `problem.addMipSol`.

problem.removeUserSolNotifyCallback

Purpose

Removes a user solution notification callback previously added by `problem.addUserSolNotifyCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Solution

Synopsis

```
problem.removeUserSolNotifyCallback(usersolnotify, data=None)
usersolnotify(cbprob, cbdata, solname, status)
```

Arguments

`usersolnotify` The callback function to remove. If `None` then all user solution notification callback functions added with the given user defined data value will be removed.

`data: Any` The data value that the callback was added with. If `None`, then the data value will not be checked and all integer solution callbacks with the function `usersolnotify` will be removed.

Related topics

`problem.addUserSolNotifyCallback`.

problem.addNodeLPSolvedCallback

Purpose

Declares a callback function, called during the branch and bound search, after the LP relaxation has been solved for the current node, but before any internal cuts and heuristics have been applied. This callback function will be called in addition to any callbacks already added by problem.addNodeLPSolvedCallback.

Topic areas

Callback, Branch and Bound Search

Synopsis

```
problem.addNodeLPSolvedCallback(nodelpsolved, data=None, priority=0)
nodelpsolved(cbprob, cbdata)
```

Arguments

`nodelpsolved` The callback function, which takes arguments `cbprob` and `cbdata`, and has no return value.

`data: Any` A user-defined data to be passed to the callback function, `nodelpsolved`.

`priority: int` An integer that determines the order in which multiple node-optimal callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function, `nodelpsolved`.

`cbdata: Any` The user-defined data passed as `data` when setting up the callback.

Related topics

`problem.removeNodeLPSolvedCallback`, `problem.addOptnodeCallback`,

problem.removeNodeLPSolvedCallback

Purpose

Removes a node callback function previously added by `problem.addNodeLPSolvedCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Branch and Bound Search

Synopsis

```
problem.removeNodeLPSolvedCallback(nodelpsolved, data=None)
nodelpsolved(cbprob, cbdata)
```

Arguments

`nodelpsolved` The callback function to remove. If None then all node-optimal callback functions added with the given user-defined data value will be removed.

`data: Any` The data value that the callback was added with. If None, then the data value will not be checked and all node-optimal callbacks with the function `optnode` will be removed.

Further information

1. This callback is fired only for the first continuous relaxation solve of each node (including the root) in a MIP branch-and-bound search, and only if the continuous relaxation is feasible and solved to optimality.
2. It is allowed to tighten the local node bounds and add or remove cutting planes from within this callback, but the callback will not be fired again in case such modifications triggers a resolve of the continuous relaxation of the node.

Related topics

`problem.addNodeLPSolvedCallback`

problem.addComputeRestartCallback

Purpose

Declares a callback to be called when a solve executed in compute mode needs to be restarted.

Topic areas

Callback, Compute Interface

Synopsis

```
problem.addComputeRestartCallback(computerestart, data=None, priority=0)
computerestart(cbprob, cbdata)
```

Arguments

`computerestart` The callback function itself, which takes arguments `cbprob` and `cbdata`, and has no return value. This function is called when a solve had to be restarted in compute mode.

`data: Any` A user-defined data to be passed to the callback function, `computerestart`.

`priority: int` An integer that determines the order in which multiple computerestart callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function, `computerestart`.

`cbdata: Any` The user-defined data passed as `data` when setting up the callback.

Related topics

`problem.removeComputeRestartCallback`

problem.removeComputeRestartCallback

Purpose

Removes a computerestart callback function previously added by `problem.addComputeRestartCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Compute Interface

Synopsis

```
problem.removeComputeRestartCallback(computerestart, data=None)
computerestart(cbprob, cbdata)
```

Arguments

`computerestart` The callback function to remove. If None then all computerestart callback functions added with the given user-defined data value will be removed.

`data: Any` The data value that the callback was added with. If None, then the data value will not be checked and all computerestart callbacks with the function `computerestart` will be removed.

Related topics

`problem.addComputeRestartCallback`.

problem.addBeforeObjectiveCallback

Purpose

Declares a callback which will be called before each objective in a multi-objective problem is solved.

Topic areas

Callback, Multiobjective

Synopsis

```
problem.addBeforeObjectiveCallback(beforeobjective, data=None, priority=0)
beforeobjective(cbprob, cbdata)
```

Arguments

`beforeobjective` The callback function itself, which takes arguments `cbprob` and `cbdata`. This function is called before every solve.

`data: Any` A user-defined data to be passed to the callback function, `beforeobjective`.

`priority: int` An integer that determines the order in which multiple before-objective callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function, `beforeobjective`.

`cbdata: Any` The user-defined data passed as `data` when setting up the callback.

Further information

1. If the solve does not stop early, this callback will be invoked `OBJSTOSOLVE` (p.**??** Xpress Optimizer Reference Manual) times. Note that this can be less than than `OBJECTIVES` (p.**??** Xpress Optimizer Reference Manual) times, since all objectives with the same priority are combined into a single objective and solved at once.
2. During this callback, the main objective (as queried with `problem.getObj`) will reflect the current objective being solved, which may be a weighted combination of several objectives with the same priority level. But changes made to the problem to fix earlier objectives at their optimal value will not be visible in the problem during this callback.
3. Controls set during this callback will retain their values when solving all subsequent objectives, except for controls which set resource limits: `TIMELIMIT` (p.**??** Xpress Optimizer Reference Manual), `SOLTIMELIMIT` (p.**??** Xpress Optimizer Reference Manual), `MAXTIME` (p.**??** Xpress Optimizer Reference Manual), `MAXNODE` (p.**??** Xpress Optimizer Reference Manual), `LPITERLIMIT` (p.**??** Xpress Optimizer Reference Manual), `NLPMAXTIME` (p.**??** Xpress NonLinear Reference Manual) and `SLPITERLIMIT` (p.**??** Xpress NonLinear Reference Manual). These are recomputed before each call to `beforeobjective`, based on their original values before the solve and the resources consumed when solving all previous objectives. Changes to resource limit controls will therefore only be retained while solving the current objective.

Related topics

`problem.removeBeforeObjectiveCallback`, `problem.addAfterObjectiveCallback`.

problem.removeBeforeObjectiveCallback

Purpose

Removes a callback function previously added by `problem.addBeforeObjectiveCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Multiobjective

Synopsis

```
problem.removeBeforeObjectiveCallback(beforeobjective, data=None)
beforeobjective(cbprob, cbdata)
```

Arguments

`beforeobjective` The callback function to remove. If None then all before-objective callback functions added with the given user-defined data value will be removed.

`data: Any` The data value that the callback was added with. If None, then the data value will not be checked and all before-objective callbacks with the function `beforeobjective` will be removed.

Related topics

`problem.addBeforeObjectiveCallback`.

problem.addAfterObjectiveCallback

Purpose

Declares a callback which will be called after each objective in a multi-objective problem is solved.

Topic areas

Callback, Multiobjective

Synopsis

```
problem.addAfterObjectiveCallback(afterobjective, data=None, priority=0)
afterobjective(cbprob, cbdata)
```

Arguments

`afterobjective` The callback function itself, which takes arguments `cbprob` and `cbdata`. This function is called after every solve.

`data: Any` A user-defined data to be passed to the callback function, `afterobjective`.

`priority: int` An integer that determines the order in which multiple after-objective callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function, `afterobjective`.

`cbdata: Any` The user-defined data passed as `data` when setting up the callback.

Further information

If the solve does not stop early, this callback will be invoked `OBJSTOSOLVE` (p.**??** Xpress Optimizer Reference Manual) times. Note that this can be less than than `OBJECTIVES` (p.**??** Xpress Optimizer Reference Manual) times, since all objectives with the same priority are combined into a single objective and solved at once.

Related topics

`problem.removeAfterObjectiveCallback`, `problem.addBeforeObjectiveCallback`.

problem.removeAfterObjectiveCallback

Purpose

Removes a callback function previously added by `problem.addAfterObjectiveCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Multiobjective

Synopsis

```
problem.removeAfterObjectiveCallback(afterobjective, data=None)
afterobjective(cbprob, cbdata)
```

Arguments

`afterobjective` The callback function to remove. If None then all after-objective callback functions added with the given user-defined data value will be removed.

`data: Any` The data value that the callback was added with. If None, then the data value will not be checked and all after-objective callbacks with the function `afterobjective` will be removed.

Related topics

`problem.addAfterObjectiveCallback`.

problem.addCheckTimeCallback

Purpose

Declares a callback function which is called every time the Optimizer checks if the time limit has been reached. This callback function will be called in addition to any callbacks already added by problem.addCheckTimeCallback.

Topic areas

Callback, Limits

Synopsis

```
problem.addCheckTimeCallback(checktime, data=None, priority=0)
checktime(cbprob, cbdata)
```

Arguments

`checktime` The callback function, which takes arguments `cbprob` and `cbdata`, and has an integer return value. This function is called every time the Optimizer checks against the time limit.

`data: Any` A user-defined data to be passed to the callback function, `checktime`.

`priority: int` An integer that determines the order in which multiple checktime callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function, `checktime`.

`cbdata: Any` The user-defined data passed as `data` when setting up the callback.

Further information

1. If the callback function returns a nonzero value the solution process will be interrupted. If multiple callbacks of this type are registered, then all of them will be invoked and the solution process will be interrupted if any of them returned a nonzero value.
2. Querying problem data or changing controls is not possible within this callback.

Related topics

`problem.removeCheckTimeCallback` `MAXTIME` (p.**??** Xpress Optimizer Reference Manual) `CHECKSONMAXTIME` (p.**??** Xpress Optimizer Reference Manual) `MAXCHECKSONMAXTIME` (p.**??** Xpress Optimizer Reference Manual)

problem.removeCheckTimeCallback

Purpose

Removes a callback function previously added by `problem.addCheckTimeCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Limits

Synopsis

```
problem.removeCheckTimeCallback(checktime, data=None)
checktime(cbprob, cbdata)
```

Arguments

`checktime`	The callback function to remove. If None then all checktime callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If None, then the data value will not be checked and all checktime callbacks with the function `checktime` will be removed.

Related topics

`problem.addCheckTimeCallback`

problem.addCutRoundCallback

Purpose

Declares a callback function that is called when the Optimizer could separate cutting planes during the branch and bound search. The callback allows a user to add cuts managed by the Optimizer, through `problem.addManagedCuts`. It also allows the user some degree of control over when and how many rounds of cuts the Optimizer should apply for the local node of the branch-and-bound search. This callback function will be called in addition to any callbacks already added by problem.addCutRoundCallback.

Topic areas

Callback, Cuts

Synopsis

```
problem.addCutRoundCallback(cutround, data=None, priority=0)
action = cutround(cbprob, cbdata, ifxpresscuts)
```

Arguments

`cutround` The callback function, which takes arguments `cbprob`, `cbdata`, `ifxpresscuts` and `action` and has no return value. This function is called before each potential round of cut separation in the branch and bound search.

`data: Any` A user-defined data to be passed to the callback function, `cutround`.

`priority: int` An integer that determines the order in which multiple callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function, `cutround`.

`cbdata: Any` The user-defined data passed as `data` when setting up the callback.

`ifxpresscuts: int` An integer set to 1 if the Optimizer will apply a round of cuts after this callback. 0 otherwise.

Return value

`action: int` An integer return value that specifies the action the Optimizer should take:

`-1`	Continue unchanged. The default action.
`0`	No further rounds of cuts should be applied on this node.
`1`	The Optimizer should apply one more round of cutting, regardless of the value of `ifxpresscuts`
`2`	The Optimizer should process any changes applied during this callback and fire the callback again, but skip any Optimizer cutting.

Further information

1. Optimizer managed cuts can be added using `problem.addManagedCuts`. The Optimizer will check for any violated cuts, together with any new or violated Optimizer created cuts (if `action` is `-1` or `+1`), to add to the problem.
2. It is possible to use `problem.addCuts`, `problem.loadCuts` and `problem.delCuts` to add and remove cutting planes and `problem.chgBounds` to tighten bounds, although it is recommended to use `problem.addManagedCuts` to provide new cutting planes to the Optimizer in this callback. Separation of delayed constraints is best done using either the `optnode` or `preintsol` callbacks.
3. It is possible to add new solutions using `problem.addMipSol`. These will be checked immediately after the callback, but adding solutions will not automatically trigger another firing of this callback.
4. If cutting planes are added with `problem.addManagedCuts`, the callback will be fired again after the reoptimization if and only if a non-zero value is returned in `action`.
5. If cuts are added directly to, or deleted from, the problem, or bounds are tightened, a reoptimization of the node LP relaxation will automatically be triggered open the return from the callback function. If separation of Optimizer cuts is enabled or managed cuts were added during the callback, such cuts will only be separated based on the new solution after reoptimization. The callback will be fired again if and only if a non-zero value is returned in `action`.
6. The attribute `CUTROUNDS` (p.**??** Xpress Optimizer Reference Manual) can be used for determining how many rounds of cuts has been applied on the current node.

Related topics

`problem.removeCutRoundCallback`, `problem.addManagedCuts`, `problem.addMipSol`, `problem.addCuts`, `problem.loadCuts`, `problem.delCuts`, `problem.chgBounds`, `CUTROUNDS` (p.**??** Xpress Optimizer Reference Manual).

problem.removeCutRoundCallback

Purpose

Removes a cut round callback function previously added by `problem.addCutRoundCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Cuts

Synopsis

```
problem.removeCutRoundCallback(cutround, data=None)
action = cutround(cbprob, cbdata, ifxpresscuts)
```

Arguments

`cutround`	The callback function to remove. If None then all cut round callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If None, then the data value will not be checked and all cut manager callbacks with the function `cutround` will be removed.

Related topics

`problem.addCutRoundCallback`

problem.addSlpCascadeEndCallback

Purpose

Add a user callback to be called at the end of the cascading process, after the last variable has been cascaded

Topic areas

Callback, SLP, Cascading

Synopsis

```
problem.addSlpCascadeEndCallback(slpcascadeend, data=None, priority=0)
slpcascadeend(cbprob, cbdata)
```

Arguments

`slpcascadeend` The function to be called at the end of the cascading process. `slpcascadeend` returns an integer value. If the return value is nonzero, the SLP iterations will stop.

`data: Any` A user-defined object, which can be used for any purpose by the function. `data` is passed to `slpcascadeend` as `cbdata`.

`priority: int` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function.

`cbdata: Any` The user-defined object passed as `data` to `problem.addSlpCascadeEndCallback`.

Further information

This callback can be used at the end of the cascading, when all the solution values have been recalculated.

Related topics

`problem.slpCascadeSol`, `problem.addSlpCascadeStartCallback`, `problem.addSlpCascadeVarCallback`, `problem.addSlpCascadeVarFailCallback` `problem.removeSlpCascadeEndCallback`

problem.removeSlpCascadeEndCallback

Purpose

Removes a callback function previously added by `problem.addSlpCascadeEndCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, SLP, Cascading

Synopsis

```
problem.removeSlpCascadeEndCallback(slpcascadeend, data=None)
slpcascadeend(cbprob, cbdata)
```

Arguments

`slpcascadeend` The callback function to remove. If `None` then all slpcascadeend callback functions added with the given user-defined data value will be removed.

`data: Any` The data value that the callback was added with. If `None`, then the data value will not be checked and all slpcascadeend callbacks with the function `slpcascadeend` will be removed.

Related topics

`problem.addSlpCascadeEndCallback`

problem.addSlpCascadeStartCallback

Purpose

Add a user callback to be called at the start of the cascading process, before any variables have been cascaded

Topic areas

Callback, SLP, Cascading

Synopsis

```
problem.addSlpCascadeStartCallback(slpcascadestart, data=None, priority=0)
result = slpcascadestart(cbprob, cbdata)
```

Arguments

`slpcascadestart` The function to be called at the start of the cascading process. `slpcascadestart` returns an integer value. If the return value is nonzero, the cascading process will be omitted for the current SLP iteration, but the optimization will continue.

`data: Any` A user-defined object, which can be used for any purpose by the function. `data` is passed to `slpcascadestart` as `cbdata`.

`priority: int` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function.

`cbdata: Any` The user-defined object passed as `data` to `problem.addSlpCascadeStartCallback`.

Return value

`result: int` The result of the callback

Further information

This callback can be used at the start of the cascading, before any of the solution values have been recalculated.

Related topics

`problem.slpCascadeSol`, `problem.addSlpCascadeEndCallback`, `problem.addSlpCascadeVarCallback`, `problem.addSlpCascadeVarFailCallback` `problem.removeSlpCascadeStartCallback`

problem.removeSlpCascadeStartCallback

Purpose

Removes a callback function previously added by `problem.addSlpCascadeStartCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, SLP, Cascading

Synopsis

```
problem.removeSlpCascadeStartCallback(slpcascadestart, data=None)
result = slpcascadestart(cbprob, cbdata)
```

Arguments

`slpcascadestart` The callback function to remove. If `None` then all slpcascadestart callback functions added with the given user-defined data value will be removed.

`data: Any` The data value that the callback was added with. If `None`, then the data value will not be checked and all slpcascadestart callbacks with the function `slpcascadestart` will be removed.

Related topics

`problem.addSlpCascadeStartCallback`

problem.addSlpCascadeVarCallback

Purpose

Add a user callback to be called after each column has been cascaded

Topic areas

Callback, SLP, Cascading

Synopsis

```
problem.addSlpCascadeVarCallback(slpcascadevar, data=None, priority=0)
result = slpcascadevar(cbprob, cbdata, col)
```

Arguments

`slpcascadevar` The function to be called after each column has been cascaded. `slpcascadevar` returns an integer value. If the return value is nonzero, the cascading process will be omitted for the remaining variables during the current SLP iteration, but the optimization will continue.

`data: Any` A user-defined object, which can be used for any purpose by the function. `data` is passed to `slpcascadevar` as `cbdata`.

`priority: int` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function.

`cbdata: Any` The user-defined object passed as `data` to `problem.addSlpCascadeVarCallback`.

`col: int` The number of the column which has been cascaded.

Return value

`result: int` The result of the callback

Further information

This callback can be used after each variable has been cascaded and its new value has been calculated.

Related topics

`problem.slpCascadeSol`, `problem.addSlpCascadeEndCallback`, `problem.addSlpCascadeStartCallback`, `problem.addSlpCascadeVarFailCallback` `problem.removeSlpCascadeVarCallback`

problem.removeSlpCascadeVarCallback

Purpose

Removes a callback function previously added by `problem.addSlpCascadeVarCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, SLP, Cascading

Synopsis

```
problem.removeSlpCascadeVarCallback(slpcascadevar, data=None)
result = slpcascadevar(cbprob, cbdata, col)
```

Arguments

`slpcascadevar` The callback function to remove. If `None` then all slpcascadevar callback functions added with the given user-defined data value will be removed.

`data: Any` The data value that the callback was added with. If `None`, then the data value will not be checked and all slpcascadevar callbacks with the function `slpcascadevar` will be removed.

Related topics

`problem.addSlpCascadeVarCallback`

problem.addSlpCascadeVarFailCallback

Purpose

Add a user callback to be called after cascading a column was not successful

Topic areas

Callback, SLP, Cascading

Synopsis

```
problem.addSlpCascadeVarFailCallback(slpcascadevarfail, data=None, priority=0)
result = slpcascadevarfail(cbprob, cbdata, col)
```

Arguments

`slpcascadevarfail` The function to be called after cascading a column was not successful. `slpcascadevarfail` returns an integer value. If the return value is nonzero, the cascading process will be omitted for the remaining variables during the current SLP iteration, but the optimization will continue.

`data: Any` A user-defined object, which can be used for any purpose by the function. `data` is passed to `slpcascadevarfail` as `cbdata`.

`priority: int` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function.

`cbdata: Any` The user-defined object passed as `data` to `problem.addSlpCascadeVarFailCallback`.

`col: int` The number of the column which has been cascaded.

Return value

`result: int` The result of the callback

Related topics

`problem.slpCascadeSol`, `problem.addSlpCascadeEndCallback`, `problem.addSlpCascadeStartCallback`, `problem.addSlpCascadeVarCallback` `problem.removeSlpCascadeVarFailCallback`

problem.removeSlpCascadeVarFailCallback

Purpose

Removes a callback function previously added by `problem.addSlpCascadeVarFailCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, SLP, Cascading

Synopsis

```
problem.removeSlpCascadeVarFailCallback(slpcascadevarfail, data=None)
result = slpcascadevarfail(cbprob, cbdata, col)
```

Arguments

`slpcascadevarfail` The callback function to remove. If `None` then all slpcascadevarfail callback functions added with the given user-defined data value will be removed.

`data: Any` The data value that the callback was added with. If `None`, then the data value will not be checked and all slpcascadevarfail callbacks with the function `slpcascadevarfail` will be removed.

Related topics

`problem.addSlpCascadeVarFailCallback`

problem.addSlpConstructCallback

Purpose

Add a user callback to be called during the Xpress-SLP augmentation process

Topic areas

Callback, SLP

Synopsis

```
problem.addSlpConstructCallback(slpconstruct, data=None, priority=0)
result = slpconstruct(cbprob, cbdata)
```

Arguments

`slpconstruct` The function to be called during problem augmentation. `slpconstruct` returns an integer value. See below for an explanation of the values.

`data: Any` A user-defined object, which can be used for any purpose by the function. `data` is passed to `slpconstruct` as `cbdata`.

`priority: int` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function.

`cbdata: Any` The user-defined object passed as `data` to `problem.addSlpConstructCallback`.

Return value

`result: int` The result of the callback

Further information

1. This callback can be used during the problem augmentation, generally (although not exclusively) to change the initial values for the variables.
2. The following return codes are accepted:

`0`	Normal return: augmentation continues
`-1`	Return to recalculate matrix values
`-2`	Return to recalculate row weights and matrix entries
`other`	Error return: augmentation terminates, `problem.slpConstruct` terminates with a nonzero error code.

3. The return values -1 and -2 will cause the callback to be called a second time after the matrix has been recalculated. It is the responsibility of the callback to ensure that it does ultimately exit with a return value of zero.

Related topics

`problem.slpConstruct` `problem.removeSlpConstructCallback`

problem.removeSlpConstructCallback

Purpose

Removes a callback function previously added by `problem.addSlpConstructCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, SLP

Synopsis

```
problem.removeSlpConstructCallback(slpconstruct, data=None)
result = slpconstruct(cbprob, cbdata)
```

Arguments

`slpconstruct` The callback function to remove. If `None` then all slpconstruct callback functions added with the given user-defined data value will be removed.

`data: Any` The data value that the callback was added with. If `None`, then the data value will not be checked and all slpconstruct callbacks with the function `slpconstruct` will be removed.

Related topics

`problem.addSlpConstructCallback`

problem.addSlpIntSolCallback

Purpose

Add a user callback to be called during MISLP when an integer solution is obtained

Topic areas

Callback, MISLP

Synopsis

```
problem.addSlpIntSolCallback(slpintsol, data=None, priority=0)
slpintsol(cbprob, cbdata)
```

Arguments

`slpintsol` The function to be called when an integer solution is obtained. `slpintsol` returns an integer value. At present, the return value is ignored.

`data: Any` A user-defined object, which can be used for any purpose by the function. `data` is passed to `slpintsol` as `cbdata`.

`priority: int` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function.

`cbdata: Any` The user-defined object passed as `data` to `problem.addSlpIntSolCallback`.

Further information

1. This callback must be used during MISLP instead of the `problem.addIntsolCallback` callback which is used for MIP problems.
2. Note that for SLP-MIP-SLP (see `SLPMIPALGORITHM` (p.**??** Xpress NonLinear Reference Manual)), the slpintsol callback will only be fired once after the second SLP solve terminated, in which case the solution can also only be queried via `problem.getSolution` but not via `problem.getLpSol` (unless postsolve is disabled).

Related topics

`problem.removeSlpIntSolCallback`

problem.removeSlpIntSolCallback

Purpose

Removes a callback function previously added by `problem.addSlpIntSolCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, MISLP

Synopsis

```
problem.removeSlpIntSolCallback(slpintsol, data=None)
slpintsol(cbprob, cbdata)
```

Arguments

`slpintsol`	The callback function to remove. If `None` then all slpintsol callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If `None`, then the data value will not be checked and all slpintsol callbacks with the function `slpintsol` will be removed.

Related topics

`problem.addSlpIntSolCallback`

problem.addSlpIterEndCallback

Purpose

Add a user callback to be called at the end of each SLP iteration

Topic areas

Callback, SLP

Synopsis

```
problem.addSlpIterEndCallback(slpiterend, data=None, priority=0)
slpiterend(cbprob, cbdata)
```

Arguments

`slpiterend` The function to be called at the end of each SLP iteration. `slpiterend` returns an integer value. If the return value is nonzero, the SLP iterations will stop.

`data: Any` A user-defined object, which can be used for any purpose by the function. `data` is passed to `slpiterend` as `cbdata`.

`priority: int` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function.

`cbdata: Any` The user-defined object passed as `data` to `problem.addSlpIterEndCallback`.

Further information

This callback can be used at the end of each SLP iteration to carry out any further processing and/or stop any further SLP iterations.

Related topics

`problem.addSlpIterStartCallback`, `problem.addSlpIterVarCallback`
`problem.removeSlpIterEndCallback`

problem.removeSlpIterEndCallback

Purpose

Removes a callback function previously added by `problem.addSlpIterEndCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, SLP

Synopsis

```
problem.removeSlpIterEndCallback(slpiterend, data=None)
slpiterend(cbprob, cbdata)
```

Arguments

`slpiterend`	The callback function to remove. If `None` then all slpiterend callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If `None`, then the data value will not be checked and all slpiterend callbacks with the function `slpiterend` will be removed.

Related topics

`problem.addSlpIterEndCallback`

problem.addSlpIterStartCallback

Purpose

Add a user callback to be called at the start of each SLP iteration

Topic areas

Callback, SLP

Synopsis

```
problem.addSlpIterStartCallback(slpiterstart, data=None, priority=0)
slpiterstart(cbprob, cbdata)
```

Arguments

`slpiterstart` The function to be called at the start of each SLP iteration. `slpiterstart` returns an integer value. If the return value is nonzero, the SLP iterations will stop.

`data: Any` A user-defined object, which can be used for any purpose by the function. `data` is passed to `slpiterstart` as `cbdata`.

`priority: int` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function.

`cbdata: Any` The user-defined object passed as `data` to `problem.addSlpIterStartCallback`.

Further information

This callback can be used at the start of each SLP iteration before the optimization begins.

Related topics

`problem.addSlpIterEndCallback`, `problem.addSlpIterVarCallback`
`problem.removeSlpIterStartCallback`

problem.removeSlpIterStartCallback

Purpose

Removes a callback function previously added by `problem.addSlpIterStartCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, SLP

Synopsis

```
problem.removeSlpIterStartCallback(slpiterstart, data=None)
slpiterstart(cbprob, cbdata)
```

Arguments

`slpiterstart` The callback function to remove. If `None` then all slpiterstart callback functions added with the given user-defined data value will be removed.

`data: Any` The data value that the callback was added with. If `None`, then the data value will not be checked and all slpiterstart callbacks with the function `slpiterstart` will be removed.

Related topics

`problem.addSlpIterStartCallback`

problem.addSlpIterVarCallback

Purpose

Add a user callback to be called after each column has been tested for convergence

Topic areas

Callback, SLP, SLP-convergence

Synopsis

```
problem.addSlpIterVarCallback(slpitervar, data=None, priority=0)
result = slpitervar(cbprob, cbdata, col)
```

Arguments

`slpitervar` The function to be called after each column has been tested for convergence. `slpitervar` returns an integer value. The return value is interpreted as a convergence status. The possible values are:

`< 0`	The variable has not converged;
`0`	Keep the internal convergence status;
`1 to 10`	The column has converged on a system-defined convergence criterion (these values should not normally be returned);
`> 10`	The variable has converged on user criteria.

`data: Any` A user-defined object, which can be used for any purpose by the function. `data` is passed to `slpitervar` as `cbdata`.

`priority: int` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function.

`cbdata: Any` The user-defined object passed as `data` to `problem.addSlpIterVarCallback`.

`col: int` The number of the column which has been tested for convergence.

Return value

`result: int` The result of the callback

Further information

This callback can be used after each variable has been checked for convergence, and allows the convergence status to be reset if required.

Related topics

`problem.addSlpIterEndCallback`, `problem.addSlpIterStartCallback`
`problem.removeSlpIterVarCallback`

problem.removeSlpIterVarCallback

Purpose

Removes a callback function previously added by `problem.addSlpIterVarCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, SLP, SLP-convergence

Synopsis

```
problem.removeSlpIterVarCallback(slpitervar, data=None)
result = slpitervar(cbprob, cbdata, col)
```

Arguments

`slpitervar`	The callback function to remove. If `None` then all slpitervar callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If `None`, then the data value will not be checked and all slpitervar callbacks with the function `slpitervar` will be removed.

Related topics

`problem.addSlpIterVarCallback`

problem.addSlpDrColCallback

Purpose

Add a user callback used to override the update of variables with small determining column

Topic areas

Callback, SLP, Cascading

Synopsis

```
problem.addSlpDrColCallback(slpdrcol, data=None, priority=0)
result, value = slpdrcol(cbprob, data, col, detcol, detval, lb, ub)
```

Arguments

`slpdrcol` The function to be called during cascading for each variable with a determining column `slpdrcol` returns an integer value. If the return value is positive, it will indicate that the value has been fixed, and cascading should be omitted for the variable. A negative value indicates that a previously fixed value has been relaxed. If no action is taken, a 0 return value should be used.

`data: Any` A user-defined object, which can be used for any purpose. by the function. `data` is passed to `slpdrcol` as `data`.

`priority: int` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function.

`data: Any` A user-defined object, which can be used for any purpose. by the function. `data` is passed to `slpdrcol` as `data`.

`col: int` The index of the column for which the determining columns is checked.

`detcol: int` The index of the determining column for the column that is being updated.

`detval: float` The value of the determining column in the current SLP iteration.

`lb: float` The original lower bound of column `col`. The callback provides this value as a reference, should the bound be updated or changed during the solution process.

`ub: float` The original upper bound of column `col`. The callback provides this value as a reference, should the bound be updated or changed during the solution process.

Return value

`result: int` The result of the callback

`value: float` Used to return the new value for column `col`, should it need to be updated, in which case the callback must return a positive result to indicate that this value should be used.

Further information

If set, this callback is called as part of the cascading procedure. Please see the NonLinear reference manual for more information.

Related topics

`SLPDRCOLTOL` (p.**??** Xpress NonLinear Reference Manual), `problem.slpCascadeSol`, `problem.addSlpCascadeEndCallback`, `problem.addSlpCascadeStartCallback` `problem.removeSlpDrColCallback`

problem.removeSlpDrColCallback

Purpose

Removes a callback function previously added by `problem.addSlpDrColCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, SLP, Cascading

Synopsis

```
problem.removeSlpDrColCallback(slpdrcol, data=None)
result, value = slpdrcol(cbprob, data, col, detcol, detval, lb, ub)
```

Arguments

`slpdrcol`	The callback function to remove. If `None` then all slpdrcol callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If `None`, then the data value will not be checked and all slpdrcol callbacks with the function `slpdrcol` will be removed.

Related topics

`problem.addSlpDrColCallback`

problem.addMsJobStartCallback

Purpose

Add a user callback to be called every time a new multistart job is created, and the pre-loaded settings are applied

Topic areas

Callback, Multistart

Synopsis

```
problem.addMsJobStartCallback(msjobstart, data=None, priority=0)
status = msjobstart(cbprob, cbdata, jobdata, jobdesc)
```

Arguments

`msjobstart` The function to be called when a new multistart job is created

`data: Any` User data passed to the callback function.

`priority: int` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function.

`cbdata: Any` The user-defined object passed as `data` to `problem.addMsJobStartCallback`.

`jobdata: Any` Job specific user-defined object, as specified by the multistart job creating API functions.

`jobdesc: str` The description of the problem as specified by the multistart job creating API functions.

Return value

`status: int` User return status variable:
0 - normal return, solve the job,
1 - disregard this job and continue,
2 - Stop multistart.

Further information

All mulit-start jobs operate on an independent copy of the original problem, and any modification to the problem is allowed, including structural changes. Please note, however, that any modification will be carried over to the base problem, should a modified problem be declared the winner problem.

Related topics

`problem.addMsJobEndCallback`, `problem.addMsWinnerCallback`
`problem.removeMsJobStartCallback`

problem.removeMsJobStartCallback

Purpose

Removes a callback function previously added by `problem.addMsJobStartCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Multistart

Synopsis

```
problem.removeMsJobStartCallback(msjobstart, data=None)
status = msjobstart(cbprob, cbdata, jobdata, jobdesc)
```

Arguments

`msjobstart`	The callback function to remove. If `None` then all msjobstart callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If `None`, then the data value will not be checked and all msjobstart callbacks with the function `msjobstart` will be removed.

Related topics

`problem.addMsJobStartCallback`

problem.addMsJobEndCallback

Purpose

Add a user callback to be called every time a new multistart job finishes. Can be used to overwrite the default solution ranking function

Topic areas

Callback, Multistart

Synopsis

```
problem.addMsJobEndCallback(msjobend, data=None, priority=0)
status = msjobend(cbprob, cbdata, jobdata, jobdesc)
```

Arguments

`msjobend` The function to be called when a multistart job finishes

`data: Any` User data passed to the callback function.

`priority: int` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function.

`cbdata: Any` The user-defined object passed as `data` to `problem.addMsJobEndCallback`.

`jobdata: Any` Job specific user-defined object, as specified by the multistart job creating API functions.

`jobdesc: str` The description of the problem as specified by the multistart job creating API functions.

Return value

`status: int` User return status variable:
0 - use the default evaluation of the finished job
1 - disregard the result and continue
2 - stop the multistart search

Further information

The multistart pool is dynamic, and this callback can be used to load new multistart jobs using the normal API functions.

Related topics

`problem.addMsJobStartCallback`, `problem.addMsWinnerCallback`
`problem.removeMsJobEndCallback`

problem.removeMsJobEndCallback

Purpose

Removes a callback function previously added by `problem.addMsJobEndCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Multistart

Synopsis

```
problem.removeMsJobEndCallback(msjobend, data=None)
status = msjobend(cbprob, cbdata, jobdata, jobdesc)
```

Arguments

`msjobend`	The callback function to remove. If `None` then all msjobend callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If `None`, then the data value will not be checked and all msjobend callbacks with the function `msjobend` will be removed.

Related topics

`problem.addMsJobEndCallback`

problem.addMsWinnerCallback

Purpose

Add a user callback to be called every time a multistart winner has been declared

Topic areas

Callback, Multistart

Synopsis

```
problem.addMsWinnerCallback(mswinner, data=None, priority=0)
result = mswinner(cbprob, cbdata, jobdata, jobdesc)
```

Arguments

`mswinner` The function to be called when a multistart winner is declared. The return code is used to indicate whether callbacks should be copied over from the winner ($\neq$ `0`) or not (`0`).

`data: Any` User data passed to the callback function.

`priority: int` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function.

`cbdata: Any` The user-defined object passed as `data` to `problem.addMsWinnerCallback`.

`jobdata: Any` Job specific user-defined object, as specified by the multistart job creating API functions.

`jobdesc: str` The description of the problem as specified by the multistart job creating API functions.

Return value

`result: int` The result of the callback

Related topics

`problem.addMsJobStartCallback`, `problem.addMsJobEndCallback`
`problem.removeMsWinnerCallback`

problem.removeMsWinnerCallback

Purpose

Removes a callback function previously added by `problem.addMsWinnerCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Multistart

Synopsis

```
problem.removeMsWinnerCallback(mswinner, data=None)
result = mswinner(cbprob, cbdata, jobdata, jobdesc)
```

Arguments

`mswinner`	The callback function to remove. If `None` then all mswinner callback functions added with the given user-defined data value will be removed.
`data: Any`	The data value that the callback was added with. If `None`, then the data value will not be checked and all mswinner callbacks with the function `mswinner` will be removed.

Related topics

`problem.addMsWinnerCallback`

problem.addNlpCoefEvalErrorCallback

Purpose

Add a user callback to be called when an evaluation of a coefficient fails during the solve

Topic areas

Callback, Numerics

Synopsis

```
problem.addNlpCoefEvalErrorCallback(nlpcoefevalerror, data=None, priority=0)
result = nlpcoefevalerror(cbprob, cbdata, col, row)
```

Arguments

`nlpcoefevalerror` The function to be called when an evaluation fails.

`data: Any` A user-defined object, which can be used for any purpose by the function. `data` is passed to `nlpcoefevalerror` as `cbdata`.

`priority: int` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function.

`cbdata: Any` The user-defined object passed as `data` to `problem.addNlpCoefEvalErrorCallback`.

`col: int` The column position of the coefficient.

`row: int` The row position of the coefficient.

Return value

`result: int` The result of the callback

Further information

This callback can be used to capture when an evaluation of a coefficient fails. The callback is called only once for each coefficient.

Related topics

`problem.nlpPrintEvalInfo` `problem.removeNlpCoefEvalErrorCallback`

problem.removeNlpCoefEvalErrorCallback

Purpose

Removes a callback function previously added by `problem.addNlpCoefEvalErrorCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, Numerics

Synopsis

```
problem.removeNlpCoefEvalErrorCallback(nlpcoefevalerror, data=None)
result = nlpcoefevalerror(cbprob, cbdata, col, row)
```

Arguments

`nlpcoefevalerror` The callback function to remove. If `None` then all nlpcoefevalerror callback functions added with the given user-defined data value will be removed.

`data: Any` The data value that the callback was added with. If `None`, then the data value will not be checked and all nlpcoefevalerror callbacks with the function `nlpcoefevalerror` will be removed.

Related topics

`problem.addNlpCoefEvalErrorCallback`

problem.addSlpPreUpdateLinearizationCallback

Purpose

Add a user callback to be called before the linearization is updated

Topic areas

Callback, SLP

Synopsis

```
problem.addSlpPreUpdateLinearizationCallback(slppreupdatelinearization, data=None,
       priority=0)
ifrepeat = slppreupdatelinearization(cbprob, cbdata, ifrepeat)
```

Arguments

`slppreupdatelinearization` The function to be called before the linearization is updated. `slppreupdatelinearization` returns an integer value. If the return value is nonzero, the optimization will return an error code and the "User Return Code" error will be set.

`data: Any` A user-defined object, which can be used for any purpose by the function. `data` is passed to `slppreupdatelinearization` as `cbdata`.

`priority: int` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

`cbprob: problem` The problem passed to the callback function.

`cbdata: Any` The user-defined object passed as `data` to `problem.addSlpPreUpdateLinearizationCallback`.

`ifrepeat: int` Indicates the call number, starting at 1.

Return value

`ifrepeat: int` If returned nonzero, another call to the callback will be scheduled. If returned zero, a final call with `ifrepeat` set to `-1` will be made.

Further information

1. When the linearization is updated, all user functions are evaluated and their derivatives calculated at the current base point. In some models, it is cheaper to compute the derivatives for all user functions at the same time, thereby avoiding repeated calculations for each function. This callback is intended to be used in such cases.
2. During each SLP iteration, the callback is invoked repeatedly, with `ifrepeat` indicating the current call number (starting from 1), until the callback indicates that no further calls are needed, by setting `ifrepeat = 0`. Between each callback invocation, the solver evaluates the user functions without requesting derivatives. After the callback has set `ifrepeat = 0`, the user functions are evaluated one more time, this time requesting derivatives, and then finally the callback is called with `ifrepeat == -1`, marking the end of the linearization update. The only time derivatives will be requested outside of this sequence is during KKT validation. This can be disabled during the solve by clearing the `xpress.constants.SLPCONVERGEBIT_VALIDATION_K` bit in `SLPCONVERGENCEOPS` (p.**??** Xpress NonLinear Reference Manual), ensuring that derivatives can always be precomputed.
3. One way that this callback can be used to precompute derivatives for user functions is as follows:
    1. On each SLP iteration, the callback is first called with `ifrepeat = 1`. This is a signal that derivatives will be needed soon. The callback sets a flag to indicate that user functions should capture their input values.
    2. When the callback returns, the user functions are evaluated without requesting derivatives. Each user function captures its input values somewhere, and returns the correct function value.
    3. The callback is called again, with `ifrepeat == 2`. The callback now computes derivates for all user functions using the captured input values. The callback clears the flag so that user functions no longer capture their input values, and sets `ifrepeat = 0` to indicate that no further calls are needed.
    4. When the callback returns, the user functions are evaluated again. Derivatives are requested, and the user functions return the precomputed derivative values.
    5. The callback is invoked one more time for this iteration with `ifrepeat == -1`, marking the end of the linearization update. User functions should behave normally from this point.

Related topics

User functions

`problem.removeSlpPreUpdateLinearizationCallback`

problem.removeSlpPreUpdateLinearizationCallback

Purpose

Removes a callback function previously added by `problem.addSlpPreUpdateLinearizationCallback`. The specified callback function will no longer be called after it has been removed.

Topic areas

Callback, SLP

Synopsis

```
problem.removeSlpPreUpdateLinearizationCallback(slppreupdatelinearization,
       data=None)
ifrepeat = slppreupdatelinearization(cbprob, cbdata, ifrepeat)
```

Arguments

`slppreupdatelinearization` The callback function to remove. If `None` then all slppreupdatelinearization callback functions added with the given user-defined data value will be removed.

`data: Any` The data value that the callback was added with. If `None`, then the data value will not be checked and all slppreupdatelinearization callbacks with the function `slppreupdatelinearization` will be removed.

Related topics

`problem.addSlpPreUpdateLinearizationCallback`

problem.addConstraint

Purpose

Adds one or more constraints to the problem.

Synopsis

```
problem.addConstraint(*constraints, name=None)
```

Arguments

`constraints` Constraints or list/tuples/array of constraints.

`name` Optional name or names to assign to the constraints.

Example

```
N = 20
p = xpress.problem()
x = [p.addVariable() for i in range(N)]
c = [x[i] <= x[i+1] for i in range(N-1)]
c2 = x[0] >= x[19]
p.addConstraint(x[2] == x[4])
p.addConstraint(c, c2)
```

Further information

1. All constraint arguments can be single `xpress.constraint` objects, or lists, tuples, NumPy arrays, generators, dictionaries or other iterable collections of `xpress.constraint` objects.
2. If the `name` argument is provided, then only a single constraint or a single collection of constraints may be provided in the positional arguments. The `name` argument must either be a collection of names with the same dimensions as the provided constraint collection, or a single name prefix. In this case, the name for each constraint will be produced by combining the name prefix with the index of the constraint within the collection.

problem.addIndicator

Purpose

Adds one or more indicator constraints to the problem.

Synopsis

```
problem.addIndicator(*indicators)
```

Argument

`indicators` Tuples containing an indicator constraints, or list/tuples/array of tuples containing a binary condition and a constraint.

Example

```
p = xpress.problem()
x = p.addVariable(vartype=xpress.binary)
y = p.addVariable(lb=10, ub=20)
z = p.addVariable()
ind1 = (x==1, y+z <= 40)
p.addIndicator(ind1)
```

Further information

All arguments can be single indicator constraints or lists, tuples, or NumPy arrays created as indicator constraints. An indicator constraint is a tuple of two elements, the first being a condition (i.e. a binary variable being 0 or 1) and the second being the constraint.

problem.addObjective

Purpose

Adds one or more objective functions to the problem.

Synopsis

```
problem.addObjective(*objs, priority=None, weight=None, abstol=None, reltol=None)
```

Arguments

`objs`	Objectives to add to the problem. An error will be returned if any variable in any objective was not already added to the problem via `addVariable`.
`priority`	(optional) An integer defining the relative priority for the new objective (only relevant for multi-objective problems). Higher values mean higher priority.
`weight`	(optional) A double defining the weight for the new objective (only relevant for multi-objective problems). If the weight is negative, the sense of this objective is reversed.
`abstol`	(optional) Absolute tolerance for the new objectives (only relevant for multi-objective problems).
`reltol`	(optional) Relative tolerance for the new objectives (only relevant for multi-objective problems).

Example

The following example adds two objective functions to the problem:

```
prob.addObjective(x1 + x2)
prob.addObjective(2*x3 - x4)
```

Further information

1. In case this method is used in a multi-objective context, the objective function must be linear, as a multi-objective problem may only contain linear objective functions. The problem itself may be any kind supported by Xpress, but any nonlinear objective terms must be modeled using a transfer variable.
2. If all objectives have equal priority but different weights, a blended (weighted-sum) approach is applied. For cases where each objective has a different priority but the same weight, a lexicographic approach is applied. The highest priority objective is optimized first. In the second and subsequent iterations, all objectives from previous iterations are fixed to their optimal values within the tolerances defined by `abstol` and `reltol`. If both different priorities and different weights are passed, a hybrid approach is applied: Xpress will solve the problem once for each distinct objective priority that is defined, optimizing in each iteration a linear combination of the objective functions with the same priority.
3. The optimization sense of the first objective defines the default sense for all objectives. To revert the sense of an objective in relation to the first objective, assign it a negative `weight`. For example, if the first objective is set to be maximized, further objectives to be minimized can be declared with `weight=-1`.
4. To add multiple objectives with a different `priority` and/or a different `weight`, make individual calls to `addObjective` and pass the corresponding value(s) in each call.

Related topics

`problem.setObjective`, `problem.addObj`, `problem.chgObjN`, `problem.delObj`, `problem.chgObj`.

problem.addSOS

Purpose

Creates a special ordered set (SOS) constraint within the problem.

Synopsis

```
s = problem.addSOS(indices, weights, type=1, name='')
```

Arguments

`indices`	List of variables composing the SOS constraint.
`weights`	List of weights (one per variable). These define the order for SOS2 constraints and may be used in branching for both types.
`type`	Type of the SOS constraint. Can be 1 (default) or 2.
`name`	Name of the SOS constraint.

Example

```
N = 20
p = xpress.problem()
x = [p.addVariable() for i in range(N)]
s1 = p.addSOS(x, [i+2 for i in range(N)])
s2 = p.addSOS([x[0], x[2]], [4,6])
```

Further information

Weights must be sufficiently distinct (see the `SOSREFTOL` control in the Optimizer manual).

The `addSOS` function can also be used to add existing unlinked SOS constraints to a problem. This usage is deprecated. See `xpress.sos` for examples of using unlinked SOS constraints.

problem.addVariable

Purpose

Creates a variable within the problem.

Synopsis

```
v = problem.addVariable(name=None, lb=0, ub=xpress.infinity, threshold=1,
      vartype=xpress.continuous)
```

Arguments

`name`	a Python UTF-8 string containing the name of the variable (its ASCII version will be saved if written onto a file); a default name is assigned if the user does not specify it.	
`lb`	Lower bound (0 by default).	
`ub`	Upper bound (+infinity by default).	
`threshold`	the threshold for semi-continuous, semi-integer, and partially integer variables; it must be between its lower and its upper bound; it has no default, so if a variable is defined as `xpress.partiallyinteger` the threshold must be specified.	
`vartype`	`xpress.continuous`	for continuous variables;
	`xpress.binary`	for binary variables;
	`xpress.integer`	for integer variables;
	`xpress.semicontinuous`	for semi-continuous variables;
	`xpress.semiinteger`	for semi-integer variables;
	`xpress.partiallyinteger`	for partially integer variables.

Example

```
p = xpress.problem()
x = p.addVariable(vartype=xpress.binary)
Y = [p.addVariable() for i in range(20)]
```

Further information

The `addVariable` function can also be used to add existing unlinked variables to a problem. This usage is deprecated. See `xpress.var` for examples of using unlinked variables.

Related topics

`problem.addVariables`

problem.addVariables

Purpose

Creates a dictionary or array of variables and adds them to the problem. Similar to the creation of a single variable with `problem.addVariable`, `addVariables` allows for using one or more index sets, specified as sets, lists, `range` objects, or any iterable object. Specifying a number `k` as an argument is equivalent to `range(k)` but can be used to create a multi-dimensional `xpress.ndarray` of variables, and allows for more efficient creation. The result is otherwise a Python dictionary of variables, whose keys are tuple of indices. A collection of variables `x` that is created with `addVariables` can be indexed, for instance, as `x[i,j]` where `i` and `j` are indices in the lists provided.

Synopsis

```
x = problem.addVariables(*indices, name="x", lb=0, ub=xpress.infinity, threshold=1,
      vartype=xpress.continuous)
```

Arguments

`indices`	One or more lists, sets, ranges, or iterable objects to be combined; in alternative, one can specify one or more numbers `k` to signify the range `0..k-1`. Using only numbers as argument will yield an `xpress.ndarray` with the dimensions as specified by the arguments themselves.
`name`	Prefix to be added to the name of each variable; see notes for more information.
`lb`	Lower bound for all variables.
`ub`	Upper bound for all variables.
`threshold`	Threshold for all variables; only used if the variables are partially integer.
`vartype`	Type of all variables, similar to the definition of single variables.

Example

The following creates a dictionary containing 6 variables whose indices vary in the set `{(0,'a'),(0,'b'),(0,'c'),(1,'a'),(1,'b'),(1,'c')}`:

```
x = problem.addVariables([0,1],['a','b','c'])
```

The following creates a dictionary containing 6 variables whose indices vary in the set `{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)}`:

```
x = problem.addVariables(2,3)
```

The code below creates a dictionary containing 5 integer variables with names `'y(a)', 'y(b)', 'y(c)', 'y(d)', 'y(e)'` and creates a constraint to bound their sum:

```
x = problem.addVariables(['a','b','c','d','e'],
name='y', vartype=xpress.integer)
con1 = xpress.Sum(x) <= 4
```

The code below creates a dictionary whose keys range from 0 to 4:

```
x = problem.addVariables(range(5),
name='y', vartype=xpress.integer)
con1 = xpress.Sum(x) <= 4
```

The following example creates an `xpress.ndarray` of dimensions `3,7,4` without assigning names to the variables:

```
x = problem.addVariables(3,7,4, name="", lb=-1, ub=1)
```

Note that specifying anything other than a number yields a dictionary rather than an `xpress.ndarray`. Finally, the following creates a variable indexed by the set defined right before:

```
S = set()
S.add('john')
S.add('cleese')
x = problem.addVariables(S, name='y', vartype=xpress.integer)
```

Further information

1. The name of each variable is created by concatenating its indices together. If the `name` argument is given as a non-empty string, this will be prepended to the name of each variable. If the `name` argument is given as an empty string, no names will be assigned to the variables. This option can be used to create large arrays of variables more quickly, since it will not be necessary to calculate a name for each variable.
2. All lists must contain non-repeated elements to avoid having variables with equal names. If a list in the argument is, for instance, `['a','b','a']`, an error is returned.

Related topics

`problem.addVariable`

problem.copy

Purpose

Obtains a copy of a problem.

Synopsis

```
p = problem.copy()
```

Example

```
p = xpress.problem()
x = [p.addVariable() for _ in range(10)]
p.addConstraint(xpress.Sum(x) <= 10)
p2 = p.copy()
x2 = p.getVariable()
assert(len(x) == len(x2))
```

Further information

All aspects of the problem are copied, including control values and callbacks.

Related topics

`problem.copycallbacks`.

problem.copycallbacks

Purpose

Copies callback functions defined for one problem to another.

Synopsis

```
problem.copycallbacks(src)
```

Argument

`src` The problem from which the callbacks are copied.

Example

The following sets up a message callback function `callback` for problem `prob1` and then copies this to the problem `prob2`.

```
prob1 = xp.problem()
prob1.addcbmessage(callback, None, 0)
prob2 = xp.problem()
prob2.copycallbacks(prob1)
```

Related topics

`problem.copyControls`, `problem.copy`.

problem.delConstraint

Purpose

Delete one or more constraints from the problem.

Synopsis

```
problem.delConstraint(constraints)
```

Example

```
N = 20
p = xpress.problem()
x = [p.addVariable() for i in range(N)]
p.addConstraint(x[i] >= x[i+1] for i in range(N-1))
p.delConstraint(2) # deletes x[2] >= x[3]
```

Further information

1. The argument can be a single constraint or a list, tuple or NumPy array of constraints. Instead of constraint objects, indices can also be used (from 0 to `ROWS`-1). The index of a constraint can be obtained with `problem.getIndex`.
2. Indicator constraints are indexed as constraints, hence they can also be deleted with this function.

problem.delSOS

Purpose

Delete one or more SOSs from the problem.

Synopsis

```
problem.delSOS(sets)
```

Example

```
N = 20
p = xpress.problem()
x = [p.addVariable() for i in range(N)]
s = p.addSOS(x, i+1 for i in range(N))
p.delSOS(s)
```

Further information

The arguments can be a single SOS or a list, tuple or NumPy array of SOSs. Instead of SOS objects, indices can also be used (from 0 to `SETS`-1). The index SOS can be obtained with `problem.getIndex`.

problem.delVariable

Purpose

Delete one or more variables from the problem.

Synopsis

```
problem.delVariable(variables)
```

Example

```
N = 20
p = xpress.problem()
x = [p.addVariable() for i in range(N)]
p.addConstraint(x[i] >= x[i+1] for i in range(N-1))

# deletes x[2], x[3], i.e., third and fourth variable
p.delVariable(x[2:4])
```

Further information

The argument can be single variables or a list, tuple or NumPy array of variables. Instead of variable objects, indices can also be used (from 0 to `COLS`-1). The index of a variable can be obtained with `problem.getIndex`.

problem.getAttrib

Purpose

Retrieves one or more attributes of a problem.

Synopsis

```
a = problem.getAttrib(*attribs, solveidx=None)
```

Example

```
p = xpress.problem()
p.read("example.lp")
print(p.getAttrib('cols'), "columns and ",
      p.getAttrib('rows'), "rows")
prob_attrib = p.getAttrib()
attr_subset = p.getAttrib(['cols', 'rows'])
```

Further information

1. This function can be passed either a single attribute name, whose value will be returned, or a list of attribute names, in which case the return value is a dictionary associating each key in the list with its value. If no argument is provided, a dictionary containing all attributes of the problem will be returned. Attributes can also be specified by id. In that case the keys for those attributes in a returned dictionary will be their ids.
2. If `solveidx` is provided, the function returns the attribute values from the specified multi-objective solve.

problem.getCallbackDuals

Purpose

Obtains the dual values from the solution associated with the current callback.

Synopsis

```
duals = problem.getCallbackDuals(*constraints)
```

Argument

`constraints` (optional) constraint objects whose duals will be returned. If none is provided, a list of duals for all constraints in the problem will be returned.

Return value

`duals` A list of dual values if `*constraints` contains more than one constraint object or is not provided, a single dual value otherwise.

Further information

See `problem.getCallbackSolution` for a list of callbacks where this function can be called. Note that a dual solution is not available in all contexts where a primal solution is available.

Related topics

`problem.getCallbackSolution`, `problem.getCallbackSlacks`, `problem.getCallbackRedCosts`, `problem.getDuals`, `constraint.getDual`.

problem.getCallbackRedCosts

Purpose

Obtains the reduced costs from the solution associated with the current callback.

Synopsis

```
djs = problem.getCallbackRedCosts(*variables)
```

Argument

`variables` (optional) variable objects whose reduced costs will be returned. If none is provided, a list of reduced costs for all variables in the problem will be returned.

Return value

`djs` A list of reduced costs if `*variables` contains more than one variable object or is not provided, a single reduced cost otherwise.

Further information

See `problem.getCallbackSolution` for a list of callbacks where this function can be called. Note that reduced costs are not available in all contexts where a primal solution is available.

Related topics

`problem.getCallbackSolution`, `problem.getCallbackSlacks`, `problem.getCallbackDuals`, `problem.getRedCosts`, `var.getRedCost`.

problem.getCallbackSlacks

Purpose

Obtains the slack values from the solution associated with the current callback.

Synopsis

```
slacks = problem.getCallbackSlacks(*constraints)
```

Argument

`constraints` (optional) constraint objects whose slack values will be returned. If none is provided, a list of slack values for all constraints in the problem will be returned.

Return value

`slacks` A list of slack values if `*constraints` contains more than one constraint object or is not provided, a single slack value otherwise.

Further information

See `problem.getCallbackSolution` for a list of callbacks where this function can be called.

Related topics

`problem.getCallbackSolution`, `problem.getCallbackDuals`, `problem.getCallbackRedCosts`, `problem.getSlacks`, `constraint.getSlack`.

problem.getCallbackSolution

Purpose

Obtains the solution associated with the current callback.

Synopsis

```
sol = problem.getCallbackSolution(*variables)
```

Argument

`variables` (optional) variable objects whose solution values will be returned. If none is provided, a list of solution values for all variables in the problem will be returned.

Return value

`sol` A list of solution values if `*variables` contains more than one variable object or is not provided, a single solution value otherwise.

Further information

This function may only be called within the following callbacks:

Callback	Solution type returned
bariteration	Barrier iteration solution
chgbranchobject	Node relaxation solution
intsol	New incumbent solution
nodelpsolved	Node relaxation solution
optnode	Node relaxation solution
preintsol	Candidate incumbent solution
slpcascadeend	SLP cascading solution
slpcascadestart	SLP iteration solution
slpiterend	SLP iteration solution
slpiterstart	SLP iteration solution
slpitervar	SLP iteration solution
slppreupdatelinearization	SLP iteration solution

Related topics

`problem.getCallbackSlacks`, `problem.getCallbackDuals`, `problem.getCallbackRedCosts`, `problem.getSolution`, `var.getSolution`.

problem.getConstraint

Purpose

Returns one or more constraint of a problem corresponding to one or more indices passed as arguments. These constraints are returned as Python objects and can be used to access and manipulate the problem.

Synopsis

```
r = problem.getConstraint(index, first, last)
```

Arguments

`first`	(optional) The first index of the constraints to be returned. It must be between 0 and `ROWS - 1`.
`last`	(optional) The last index of the constraints to be returned. It must be between 0 and `ROWS - 1`.
`index`	(optional) Either an integer or a list of integers (not necessarily sorted) with the index/indices of all constraints to be returned, all between 0 and `ROWS - 1`

Further information

All arguments are optional. If neither of them is provided, the return value is a list with all constraints of the problem. Otherwise, either `first` and `last` or just `index` can be passed.

Related topics

`problem.getVariable`, `problem.getSOS`.

problem.getControl

Purpose

Retrieves one or more controls of a problem. Can also be used to retrieve objective controls.

Synopsis

```
c = problem.getControl(*controls, objidx=None)
```

Arguments

`controls` Names or numeric ids of the controls whose values to retrieve. If the `objidx` argument is provided, the control must be one of the following objective controls:

- `priority` the priority of the objective
- `weight` the weight of the objective
- `reltol` the relative tolerance of the objective
- `abstol` the absolute tolerance of the objective
- `rhs` the constant part of the objective

`objidx` (optional) Index of the objective whose control to retrieve.

Example

```
p = xpress.problem()
[...]
print("tolerance for feasibility and optimality: ",
  p.getControl('feastol'), p.getControl('miprelstop'))
all_ctrls = p.getControl()
ctrl_subset = p.getControl(['presolve', 'miprelstop', 'feastol'])
```

Further information

This function can be passed either a single control name, whose value will be returned, or a list of control names, in which case the return value is a dictionary associating each key in the list with its value. If no argument is provided, a dictionary containing all controls of the problem will be returned. Controls can also be specified by id. In that case the keys for those controls in a returned dictionary will be their ids.

Related topics

`problem.setControl`.

problem.getDuals

Purpose

Obtains the dual values associated with the incumbent solution during or after optimization with `problem.optimize`, `problem.mipOptimize`, `problem.lpOptimize` or `problem.nlpOptimize`.

Synopsis

```
duals = problem.getDuals(*constraints)
```

Argument

`constraints` (optional) constraint objects whose duals will be returned. If none is provided, a list of duals for all constraints in the problem will be returned.

Return value

`duals` A list of dual values if `*constraints` contains more than one constraint object or is not provided, a single dual value otherwise.

Example

```
import xpress as xp
import numpy as np
p = xp.problem()
x = p.addVariables(10)
A = np.random.random((5,10))
b = np.random.random(5)
constr = xp.Dot(A,x) >= b
p.addConstraint(constr)
p.setObjective(xp.Sum(x))
p.optimize()
print("Duals of last two constraints:", p.getDuals(constr[-2:]))
```

Related topics

`problem.getSolution`, `problem.getSlacks`, `problem.getRedCosts`, `problem.getCallbackDuals`, `problem.getCallbackPresolveDuals`, `constraint.getDual`.

problem.getRedCosts

Purpose

Obtains the reduced costs associated with the incumbent solution during or after optimization with `problem.optimize`, `problem.mipOptimize`, `problem.lpOptimize` or `problem.nlpOptimize`.

Synopsis

```
djs = problem.getRedCosts(*variables)
```

Argument

`variables` (optional) variable objects whose reduced costs will be returned. If none is provided, a list of reduced costs of all variables in the problem will be returned.

Return value

`djs` A list of reduced cost values if `*variables` contains more than one variable object or is not provided, a single reduced cost value otherwise.

Example

```
import xpress as xp
import numpy as np
p = xp.problem()
x = p.addVariables(10, name='y')  # creates 10 variables named 'y(0)', 'y(1)', etc.
A = np.random.random((5,10))
b = np.random.random(5)
constr = xp.Dot(A,x) >= b
p.addConstraint(constr)
p.setObjective(xp.Sum(x))
p.optimize()
print("Reduced costs of first two variables:", p.getRedCosts(x[:2]))
print("Reduced costs of last two variables:", p.getRedCosts('y(8)', 'y(9)'))
```

Related topics

`problem.optimize`, `problem.getSolution`, `problem.getDuals`, `problem.getSlacks`, `problem.getCallbackRedCosts`, `problem.getCallbackPresolveRedCosts`, `var.getRedCost`.

problem.getSlacks

Purpose

Obtains the slack values associated with the incumbent solution during or after optimization with `problem.optimize`, `problem.mipOptimize`, `problem.lpOptimize` or `problem.nlpOptimize`.

Synopsis

```
slacks = problem.getSlacks(*constraints)
```

Argument

`constraints` (optional) constraint objects whose slacks will be returned. If none is provided, a list of slacks for all constraints in the problem will be returned.

Return value

`slacks` A list of slack values if `*constraints` contains more than one constraint object or is not provided, a single slack value otherwise.

Example

```
import xpress as xp
import numpy as np
p = xp.problem()
x = p.addVariables(10)
A = np.random.random((5,10))
b = np.random.random(5)
constr = xp.Dot(A,x) >= b
p.addConstraint(constr)
p.setObjective(xp.Sum(x))
p.optimize()
print("slack of 2nd and 3rd constraint:", p.getSlacks(constr[1], constr[2]))
print("slack of first three constraints:", p.getSlacks(constr[:3]))
```

Related topics

`problem.optimize`, `problem.getSolution`, `problem.getDuals`, `problem.getRedCosts`, `problem.getCallbackSlacks`, `problem.getCallbackPresolveSlacks`, `constraint.getSlack`.

problem.getSolution

Purpose

Obtains the incumbent solution during or after optimization with `problem.optimize`, `problem.mipOptimize`, `problem.lpOptimize` or `problem.nlpOptimize`.

Synopsis

```
x = problem.getSolution(args=None, flatten=False)
```

Arguments

`args`	(optional) specify indices, names, or objects whose solution value is requested. If `None`, it is assumed that all indices of the problem's variables are requested. Starting with version 8.8, `args` can contain expressions, both linear and nonlinear, and dictionaries thereof, in order to allow for more flexible evaluation of functions of the problem solution
`flatten`	(optional) allows for backward compatibility with previous versions of the Xpress Python interface. Regardless of whether the passed object is a (nested) list, tuples, the returned value is a flattened list containing all requested values.

Example

Below are a few possible uses of the function. Note that one can specify variable names, variable indices, or variable objects, and embed them in lists, dictionaries, NumPy arrays, and tuples.

```
x = p.addVariables(name='X')
[...]
a = p.getSolution())               # Returns a list with an optimal solution
b = p.getSolution(x[0]))           # Only returns the value of x[0]
c = p.getSolution(0)               # As above
d = p.getSolution(x)               # Gets the values of all variables in the vector x
e = p.getSolution(range(4))        # Gets the value of x[0], x[1], x[2], x[3], i.e.,
                                   # the first four variables of the problem
f = p.getSolution('X(0)')          # Gets the value of x[0] by its name
g = p.getSolution({                # Returns a dictionary containing:
  1: x,                            #   1: value of all variables
  2: 1,                            #   2: value of x[1]
  3: 'X(2)'                        #   3: value of x[2]
})
h = p.getSolution([                # Returns a list of lists: [
  [x[0], x[1]],                    #   [value of x[0], value of x[1]],
  [x[2], x[3]]                     #   [value of x[1], value of x[2]]
])                                 # ]
h = p.getSolution([                # Returns a flat list containing the values of
  [x[0], x[1]],                    # x[0], x[1], x[2], x[3]
  [x[2], x[3]]
], flatten=True)
i = p.getSolution(x[0] + 3*x[1])   # Gets the value of an expression under the
                                   # current solution
j = p.getSolution(np.array(x))     # Gets a NumPy array with the solution of x]
```

Further information

For efficiency reasons it is preferable that one call to `getSolution` is made, as the whole vector is obtained at each call and only the desired portion is returned.

The function `xpress.evaluate` is more flexible in that it allows more argument types. Apart from the case where the `args` argument contains indices and names of the variables, `getSolution` is equivalent to a call to `xpress.evaluate`.

Related topics

`xpress.evaluate`, `problem.getDuals`, `problem.getSlacks`, `problem.getRedCosts`, `problem.getCallbackSolution`, `problem.getCallbackPresolveSolution`, `var.getSolution`.

problem.getSOS

Purpose

Returns one or more SOSs of a problem corresponding to one or more indices passed as arguments. These SOSs are returned as Python objects and can be used to access and manipulate the problem.

Synopsis

```
x = problem.getSOS(index, first, last)
```

Arguments

`first`	(optional) The first index of the SOSs to be returned.
`last`	(optional) The last index of the SOSs to be returned.
`index`	(optional) Either an integer or a list of integers (not necessarily sorted) with the index/indices of all SOSs to be returned.

Further information

All arguments are optional. If neither of them is provided, the return value is a list with all SOSs of the problem. Otherwise, either `first` and `last` or just `index` can be passed.

Related topics

`problem.getVariable`, `problem.getConstraint`.

problem.getVariable

Purpose

Returns one or more variables of a problem corresponding to one or more indices passed as arguments. These variables are returned as Python objects and can be used to access and manipulate the problem.

Synopsis

```
x = problem.getVariable(index, first, last)
```

Arguments

`index`	(optional) Either an integer or a list of integers (not necessarily sorted) with the index/indices of all variables to be returned, all between 0 and `COLS - 1`
`first`	(optional) The first index of the variables to be returned. It must be between 0 and `COLS - 1`.
`last`	(optional) The last index of the variables to be returned. It must be between 0 and `COLS - 1`.

Further information

All arguments are optional. If neither of them is provided, the return value is a list with all variables of the problem. Otherwise, either `first` and `last` or just `index` can be passed.

Related topics

`problem.getConstraint`, `problem.getSOS`.

problem.reset

Purpose

Clears all information regarding an optimization problem and returns it to the same status as it would be after creation (i.e., after the instruction `p = xpress.problem()`).

Synopsis

```
problem.reset()
```

Example

```
p = xpress.problem()
p.read("problem0", "l")
p.optimize()
x0 = p.getSolution()
p.reset()
p.read("problem1", "")
p.optimize()
x1 = p.getSolution()
```

Related topics

`problem.readProb`.

problem.save

Purpose

Alias of `problem.saveAs`.

problem.setControl

Purpose

Sets one or more controls of a problem or objective.

Synopsis

```
problem.setControl(control, value, objidx=None)
problem.setControl(dict, objidx=None)
```

Arguments

`control` Name or numeric id of the control whose value to change. If the `objidx` argument is provided, the control can be either a solver control or one of the following objective controls:

- `priority` the priority of the objective
- `weight` the weight of the objective
- `reltol` the relative tolerance of the objective
- `abstol` the absolute tolerance of the objective
- `rhs` the constant part of the objective

`value` Value to which to set the control.

`dict` A dictionary mapping names or numeric ids of controls to values to which to set them.

`objidx` (optional) Index of the objective whose control to modify.

Example

```
p = xpress.problem()
p.setControl('miprelstop', 1e-4)
p.setControl({'feastol': 1e-4, 'presolve': 0})
```

Further information

1. As mentioned in the previous chapter, there is an alternative way to set and retrieve controls. It works by querying the data structure `controls` of each problem or, if one wants to set a control to be used by all problems defined subsequently, the global control object `xpress.controls`.
2. This function can be used in two ways depending on whether one wants to set one or more controls. In the first case, the arguments form a pair (string, value) where the first element is the lower-case name of a control (see the Xpress Optimizer reference manual for a complete list of controls). In the second case, the argument is a Python *dictionary* whose keys are control name string and whose values are the value of the control. Instead of control names it is also possible to use their numeric ids.

Related topics

`problem.getControl`.

problem.setObjective

Purpose

Sets the objective function of the problem.

Synopsis

```
problem.setObjective(expr, sense=None, objidx=0, priority=None, weight=None,
       abstol=None, reltol=None)
```

Arguments

`expr`	An expression involving variables which have been added to the problem prior to this call. An error will be returned if any variable in the objective was not already added to the problem via `addVariable`.
`sense`	(optional) Either `xpress.ObjSense.MINIMIZE` or `xpress.ObjSense.MAXIMIZE` (by default the sense will be left unchanged).
`objidx`	(optional) Index of the objective to modify.
`priority`	(optional) An integer defining the relative priority for the new objective (only relevant for multi-objective problems). Higher values mean higher priority.
`weight`	(optional) A double defining the weight for the new objective (only relevant for multi-objective problems). If the weight is negative, the sense of this objective is reversed.
`abstol`	(optional) Absolute tolerance for the new objective (only relevant for multi-objective problems).
`reltol`	(optional) Relative tolerance for the new objective (only relevant for multi-objective problems).

Example

The following example sets the objective function of the problem to $[2x_1^2 + 3x_1x_2 + 5x_2^2 + 4x_1 + 4]$:

```
p = xpress.problem()
x1 = p.addVariable()
x2 = p.addVariable()
p.setObjective(2*x1**2 + 3*x1*x2 + 5*x2**2 + 4*x1 + 4)
```

Further information

1. Multiple calls to `setObjective` are allowed, and each replaces the old objective function with a new one.
2. In case this method is used in a multi-objective context, the objective function must be linear, as a multi-objective problem may only contain linear objective functions. The problem itself may be any kind supported by Xpress, but any nonlinear objective terms must be modeled using a transfer variable.
3. To add multiple objectives to a problem, use the `problem.addObjective` function.

Related topics

`problem.addVariable`, `problem.addObjective`, `problem.addObj`, `problem.chgObjN`, `problem.delObj`.

problem.getOutputEnabled

Purpose

Returns True if Optimizer messages will be written to the Python output stream, False otherwise.

Synopsis

```
enabled = problem.getOutputEnabled()
```

Related topics

`problem.setOutputEnabled`, `xpress.getOutputEnabled`, `xpress.setOutputEnabled`.

problem.setOutputEnabled

Purpose

Enables or disables writing Optimizer messages to the Python output stream.

Synopsis

```
problem.setOutputEnabled(enabled)
```

Argument

`enabled` True if Optimizer messages should be written to the Python output stream for this problem, False otherwise.

Further information

By default, Optimizer messages are printed to `stdout`. This behaviour can be disabled for a given problem by calling `problem.setOutputEnabled` with argument `False`. This does not affect user message callbacks added with `problem.addMessageCallback`.

Related topics

`problem.getOutputEnabled`, `xpress.setOutputEnabled`, `problem.addMessageCallback`, `problem.removeMessageCallback`.

8.10 Xpress `variable` object methods

`var.getRedCost`	Returns the reduced cost for this variable from the incumbent solution during or after optimization of a continuous problem with `problem.optimize`, `problem.lpOptimize` or `problem.nlpOptimize`. [Linear Algebra, LP, Optimizer, Solution] p. 507
`var.getSolution`	Returns the primal value of this variable from the incumbent solution during or after optimization with `problem.optimize`, `problem.mipOptimize`, `problem.lpOptimize` or `problem.nlpOptimize`. [Optimizer, Solution] p. 506

var.getSolution

Purpose

Returns the primal value of this variable from the incumbent solution during or after optimization with `problem.optimize`, `problem.mipOptimize`, `problem.lpOptimize` or `problem.nlpOptimize`.

Topic areas

Solution, Optimizer

Synopsis

```
x = var.getSolution()
```

Return value

`x: list[float]` The solution value of the variable.

Further information

1. The function always returns data in the input space, even if the problem is currently in presolved state.
2. During a deterministic MIP solve, the incumbent solution is only available once it has been deterministically communicated to the current MIP worker problem.
3. During an SLP solve, the incumbent solution is not necessarily feasible. It is generally the best solution seen so far in terms of feasibility and the objective function, depending on the values of the `SLPFILTER` and `NLPMERITLAMBDA` controls.

Related topics

`problem.optimize`, `constraint.getDual`, `constraint.getSlack`, `var.getRedCost`.

var.getRedCost

Purpose

Returns the reduced cost for this variable from the incumbent solution during or after optimization of a continuous problem with `problem.optimize`, `problem.lpOptimize` or `problem.nlpOptimize`.

Topic areas

Linear Algebra, Solution, LP, Optimizer

Synopsis

```
dj = var.getRedCost()
```

Return value

`dj: list[float]` Reduced cost of the variable.

Further information

The function always returns data in the input space, even if the problem is currently in presolved state.

Related topics

`problem.optimize`, `constraint.getDual`, `constraint.getSlack`, `var.getSolution`.

8.11 Xpress `constraint` object methods

`constraint.getBody`	Returns an expression representing the left-hand side of the constraint.	p. 509
`constraint.getDual`	Returns the dual value for this constraint from the incumbent solution during or after optimization of a continuous problem with `problem.optimize`, `problem.lpOptimize` or `problem.nlpOptimize`. [LP, Optimizer, Solution]	p. 511
`constraint.getSlack`	Returns the slack value for this constraint from the incumbent solution during or after optimization with `problem.optimize`, `problem.mipOptimize`, `problem.lpOptimize` or `problem.nlpOptimize`. [LP, Optimizer, Solution]	p. 510

constraint.getBody

Purpose

Returns an expression representing the left-hand side of the constraint.

Synopsis

```
expr = constraint.getBody()
```

Example

The following example creates a new constraint with the same left-hand side expression as an existing constraint:

```
lhs = con1.getBody()
con2 = con1.getBody() >= 1
```

constraint.getSlack

Purpose

Returns the slack value for this constraint from the incumbent solution during or after optimization with `problem.optimize`, `problem.mipOptimize`, `problem.lpOptimize` or `problem.nlpOptimize`.

Topic areas

Solution, LP, Optimizer

Synopsis

```
slack = constraint.getSlack()
```

Return value

`slack: list[float]` The slack value of the constraint.

Further information

During a deterministic MIP solve, the incumbent solution is only available once it has been deterministically communicated to the current MIP worker problem.

Related topics

`problem.optimize`, `constraint.getDual`, `var.getSolution`, `var.getRedCost`.

constraint.getDual

Purpose

Returns the dual value for this constraint from the incumbent solution during or after optimization of a continuous problem with `problem.optimize`, `problem.lpOptimize` or `problem.nlpOptimize`.

Topic areas

Solution, LP, Optimizer

Synopsis

```
dual = constraint.getDual()
```

Return value

`dual: list[float]` The dual value of the constraint.

Further information

The function always returns data in the input space, even if the problem is currently in presolved state.

Related topics

`problem.optimize`, `var.getSolution`, `constraint.getSlack`, `var.getRedCost`.

8.12 Xpress `branchobj` object methods

The following pages present the methods of the `branchobj` class, i.e., the methods used when creating and manipulating branching objects. Their invocation can be as follows:

```
import xpress as xp
b = xp.branchobj()
b.addbranches(3)
```


`branchobj.addBounds` Adds new bounds to a branch of a user branching object. [Branching Object] p. 518

`branchobj.addBranches` Adds new, empty branches to a user defined branching object. [Branching Object] p. 514

`branchobj.addCuts` Adds stored user cuts as new constraints to a branch of a user branching object. [Branching Object] p. 522

`branchobj.addRows` Adds new constraints to a branch of a user branching object. [Branching Object] p. 520

`branchobj.getBounds` Returns the bounds for a branch of a user branching object. [Branching Object] p. 519

`branchobj.getBranches` Returns the number of branches of a branching object. [Branching Object] p. 515

`branchobj.getID` Returns the unique identifier assigned to a branching object. [Branching Object] p. 523

`branchobj.getRows` Returns the constraints for a branch of a user branching object. [Branching Object] p. 521

`branchobj.setPreferredBranch` Specifies which of the child nodes corresponding to the branches of the object should be explored first. [Branching Object] p. 517

`branchobj.setPriority` Sets the priority value of a user branching object. [Branching Object] p. 516

`branchobj.store` Adds a new user branching object to the Optimizer's list of candidates for branching. [Branching Object] p. 513

`branchobj.validate` Verifies that a given branching object is valid for branching on the current branch-and-bound node of a MIP solve. [Branching Object] p. 524

branchobj.store

Purpose

Adds a new user branching object to the Optimizer's list of candidates for branching. This function is available only through the callback functions set by `problem.addOptnodeCallback` or `problem.addPreIntsolCallback` (if the `soltype` is 0).

Topic area

Branching Object

Synopsis

```
status = branchobj.store()
```

Return value

`status: int` The returned status from checking the provided branching object:

- `0` The object was accepted successfully.
- `1` Failed to presolve the object due to dual reductions in presolve.
- `2` Failed to presolve the object due to duplicate column reductions in presolve.
- `3` The object contains an empty branch.
- `4` Failed to presolve the object due to presolve operations requiring relaxation of a row.
- `5` Failed to presolve the object due to it becoming nonlinear due to nonlinear eliminations.

The object was not added to the candidate list if a non zero status is returned.

Further information

1. To ensure that a user branching object expressed in terms of the original matrix columns can be applied to the presolved problem, it might be necessary to turn off certain presolve operations.
2. If any of the original matrix columns referred to in the object are unbounded, dual reductions might prevent the corresponding bound or constraint from being presolved. To avoid this, dual reductions should be turned off in presolve, by clearing bit 3 of the integer control `PRESOLVEOPS` (p.**??** Xpress Optimizer Reference Manual).
3. If one or more of the original matrix columns of the object are duplicates in the original matrix, but not in the branching object, it might not be possible to presolve the object due to duplicate column eliminations in presolve. To avoid this, duplicate column eliminations should be turned off in presolve, by clearing bit 5 of `PRESOLVEOPS` (p.**??** Xpress Optimizer Reference Manual).
4. As an alternative to turning off the above mentioned presolve features, it is possible to protect individual columns of a the problem from being modified by presolve. Use the `problem.loadSecureVecs` function to mark any columns that might be branched on using branching objects.
5. The branching candidate is only for the current node. Once Xpress branched on that node, all branching candidates are deleted.
6. The branching candidates stored by this function are evaluated together with Xpress's internal branching candidates. A branching object's priority can be used to give preference to objects, see `branchobj.setPriority`.

Related topics

`xpress.branchobj`, `branchobj.validate`.

branchobj.addBranches

Purpose

Adds new, empty branches to a user defined branching object.

Topic area

Branching Object

Synopsis

```
branchobj.addBranches(nbranches)
```

Argument

`nbranches: int` Number of new branches to create.

Related topics

`xpress.branchobj`.

branchobj.getBranches

Purpose

Returns the number of branches of a branching object.

Topic area

Branching Object

Synopsis

```
nbranches = branchobj.getBranches()
```

Return value

`nbranches: int` The number of branches.

Related topics

`xpress.branchobj`, `branchobj.addBranches`.

branchobj.setPriority

Purpose

Sets the priority value of a user branching object.

Topic area

Branching Object

Synopsis

```
branchobj.setPriority(priority)
```

Argument

`priority: int` The new priority value to assign to the branching object, which must be a number from 0 to 1000. User branching objects are created with a default priority value of 500.

Further information

1. A candidate branching object with lowest priority number will always be selected for branching before an object with a higher number.
2. Priority values must be an integer from 0 to 1000. User branching objects and MIP entities are by default assigned a priority value of 500. Special branching objects, such as those arising from structural branches or split disjunctions are assigned a priority value of 400.

Related topics

`xpress.branchobj`, the Optimizer Reference Manual.

branchobj.setPreferredBranch

Purpose

Specifies which of the child nodes corresponding to the branches of the object should be explored first.

Topic area

Branching Object

Synopsis

```
branchobj.setPreferredBranch(branch)
```

Argument

`branch: int` The number of the branch to mark as preferred.

Related topics

`xpress.branchobj`.

branchobj.addBounds

Purpose

Adds new bounds to a branch of a user branching object.

Topic area

Branching Object

Synopsis

```
branchobj.addBounds(branch, bndtype, colind, bndval)
```

Arguments

`branch: int` The number of the branch to add the new bounds for. This branch must already have been created using `branchobj.addBranches`. Branches are indexed starting from zero.

`bndtype: list[str]` Character array indicating the type of bounds to add:

- `L` Lower bound.
- `U` Upper bound.

`colind: list[var]` Integer array containing the column indices for the new bounds.

`bndval: list[float]` Double array giving the bound values.

Further information

Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`xpress.branchobj`.

branchobj.getBounds

Purpose

Returns the bounds for a branch of a user branching object.

Topic area

Branching Object

Synopsis

```
bndtype, colind, bndval = branchobj.getBounds(branch)
```

Argument

`branch: int` The number of the branch to get the bounds for.

Return value

`bndtype: list[str]` Character array containing the types of bounds:

- `L` Lower bound.
- `U` Upper bound.

`colind: list[int]` Integer array containing the column indices.

`bndval: list[float]` Double array containing the bound values.

Further information

Note that this function returns the bounds in the presolved space.

Related topics

`xpress.branchobj`, `branchobj.addBounds`.

branchobj.addRows

Purpose

Adds new constraints to a branch of a user branching object.

Topic area

Branching Object

Synopsis

```
branchobj.addRows(branch, rowtype, rhs, start, colind, rowcoef)
```

Arguments

`branch: int` The number of the branch to add the new constraints for. This branch must already have been created using `branchobj.addBranches`. Branches are indexed starting from zero.

`rowtype: list[str]` Character array indicating the type of constraints to add:

- `L` Less than type.
- `G` Greater than type.
- `E` Equality type.

`rhs: list[float]` Double array containing the right hand side values.

`start: list[int]` Integer array containing the offsets of the `colind` and `rowcoef` arrays of the start of the non zero coefficients in the new constraints.

`colind: list[var]` Integer array containing the column indices for the non zero coefficients.

`rowcoef: list[float]` Double array containing the non zero coefficient values.

Further information

Argument `colind` may contain either `xpress.var` objects or integer indices.

Related topics

`xpress.branchobj`.

branchobj.getRows

Purpose

Returns the constraints for a branch of a user branching object.

Topic area

Branching Object

Synopsis

```
rowtype, rhs, start, colind, rowcoef = branchobj.getRows(branch)
```

Argument

`branch: int` The number of the branch to get the constraints from.

Return value

`rowtype: list[str]` Character array containing the types of the rows:

- `L` Less than type.
- `G` Greater than type.
- `E` Equality type.

`rhs: list[float]` Double array containing the right hand side values.

`start: list[int]` Integer array containing the offsets of the `colind` and `rowcoef` arrays of the start of the non zero coefficients in the returned constraints.

`colind: list[int]` Integer array containing the column indices for the non zero coefficients.

`rowcoef: list[float]` Double array containing the non zero coefficient values.

Further information

Note that this function returns the rows in the presolved space.

Related topics

`xpress.branchobj`, `branchobj.addRows`.

branchobj.addCuts

Purpose

Adds stored user cuts as new constraints to a branch of a user branching object.

Topic area

Branching Object

Synopsis

```
branchobj.addCuts(branch, cutind)
```

Arguments

`branch: int` The number of the branch to add the cuts for. This branch must already have been created using `branchobj.addBranches`. Branches are indexed starting from zero.

`cutind: list[poolcut]` Array containing the user cut objects that should be added to the branch.

Related topics

`xpress.branchobj`, `branchobj.addRows`.

branchobj.getID

Purpose

Returns the unique identifier assigned to a branching object.

Topic area

Branching Object

Synopsis

```
id = branchobj.getID()
```

Return value

`id: int` The identifier.

Further information

1. Branching objects associated with existing column entities (binaries, integers, semi–continuous and partial integers), are given an identifier from 1 to `MIPENTS` (p.**??** Xpress Optimizer Reference Manual).
2. Branching objects associated with existing Special Ordered Sets, are given an identifier from `MIPENTS` (p.**??** Xpress Optimizer Reference Manual)+`1` to `MIPENTS` (p.**??** Xpress Optimizer Reference Manual)+`SETS` (p.**??** Xpress Optimizer Reference Manual).
3. User created branching objects will always have a negative identifier.

Related topics

`xpress.branchobj`.

branchobj.validate

Purpose

Verifies that a given branching object is valid for branching on the current branch-and-bound node of a MIP solve. The function will check that all branches are non-empty, and if required, verify that the branching object can be presolved.

Topic area

Branching Object

Synopsis

```
status = branchobj.validate()
```

Return value

`status: int` The returned status from checking the provided branching object:

0	The object is acceptable.
1	Failed to presolve the object due to dual reductions in presolve.
2	Failed to presolve the object due to duplicate column reductions in presolve.
3	The object contains an empty branch.
4	Failed to presolve the object due to presolve operations requiring relaxation of a row.
5	Failed to presolve the object due to it becoming nonlinear due to nonlinear eliminations.

Related topics

`xpress.branchobj`.

8.13 Xpress constants and enumerations

Many controls, attributes and function arguments have a fixed set of possible values. These values are available in the `xpress` package as enumerations, or in the `xpress.constants` package as constants.

`xpress.AllowCompute` Enumeration used in `xpress.setComputeAllowed`, `xpress.getComputeAllowed`. p. 583

`xpress.AlwaysNeverAutomatic` Enumeration used in `problem.controls.CROSSOVER`, `problem.controls.DUALIZE`, `problem.controls.ROOTPRESOLVE`, `problem.controls.BRANCHSTRUCTURAL`, `problem.controls.QUADRATICUNSHIFT`. p. 537

`xpress.BacktrackAlg` Enumeration used in `problem.controls.BACKTRACK`, `problem.controls.BACKTRACKTIE`. p. 539

`xpress.BARHGOps` Enumeration used in `problem.controls.BARHGOPS`. p. 591

`xpress.BarHGPrecision` Enumeration used in `problem.controls.BARHGPRECISION`. p. 594

`xpress.BarOrder` Enumeration used in `problem.controls.BARORDER`. p. 534

`xpress.BarPresolveOps` Enumeration used in `problem.controls.BARPRESOLVEOPS`. p. 562

`xpress.BasisStatus` p. 592

`xpress.BranchChoice` Enumeration used in `problem.controls.BRANCHCHOICE`. p. 540

`xpress.CholeskyAlgorithm` Enumeration used in `problem.controls.CHOLESKYALG`. p. 541

`xpress.Clamping` p. 580

`xpress.ComputeLog` Enumeration used in `problem.controls.COMPUTELOG`. p. 584

`xpress.constants` Constants used in Xpress NonLinear controls, attributes and function arguments. p. 596

`xpress.CrossoverDynamicReduction` Enumeration used in `problem.controls.CROSSOVERDRP`. p. 542

`xpress.CutSelect` Enumeration used in `problem.controls.CUTSELECT`. p. 558

`xpress.CutStrategy` Enumeration used in `problem.controls.CUTSTRATEGY`. p. 555

`xpress.DefaultAlg` Enumeration used in `problem.controls.DEFAULTALG`. p. 535

`xpress.DualGradient` Enumeration used in `problem.controls.DUALGRADIENT`. p. 543

`xpress.DualizeOps` p. 560

`xpress.DualStrategy` Enumeration used in `problem.controls.DUALSTRATEGY`. p. 544

`xpress.FeasibilityPump` Enumeration used in `problem.controls.FEASIBILITYPUMP`. p. 545

`xpress.GenConsType` p. 579

`xpress.Globalboundingbox` p. 586

`xpress.GlobalLSHEURStrategy` p. 590

`xpress.HeuristicSearchSelect` Enumeration used in `problem.controls.HEURSEARCHROOTSELECT`, `problem.controls.HEURSEARCHTREESELECT`. p. 546

`xpress.IISOps` Enumeration used in `problem.controls.IISOPS`. p. 588

`xpress.IISSolStatus` Enumeration used in `problem.attributes.IISSOLSTATUS`. p. 531

`xpress.LPFlags` Enumeration used in `problem.controls.LPFLAGS`. p. 578

`xpress.LPStatus` Enumeration used in `problem.attributes.LPSTATUS`. p. 529

`xpress.MipPresolve` Enumeration used in `problem.controls.MIPPRESOLVE`. p. 552

`xpress.MipRestart` Enumeration used in `problem.controls.MIPRESTART`. p. 563

`xpress.MIPStatus` Enumeration used in `problem.attributes.MIPSTATUS`. p. 530

`xpress.MIPStopStage` Enumeration used in `problem.controls.MIPSTOPSTAGE`. p. 595

`xpress.MultiObjOps` Enumeration used in `problem.controls.MULTIOBJOPS`. p. 587

`xpress.Namespaces` p. 585

`xpress.NodeSelectionCriteria` Enumeration used in `problem.controls.NODESELECTION`. p. 547

xpress.ObjControl		p. 582
xpress.ObjSense	Enumeration used in problem.chgObjSense.	p. 569
xpress.OnOff	Enumeration used in problem.controls.AUTOPERTURB, problem.controls.IFCHECKCONVEXITY.	p. 538
xpress.OptimizeType	Enumeration used in problem.attributes.OPTIMIZETYPEUSED.	p. 533
xpress.OutputDetail	Enumeration used in problem.controls.OUTPUTLOG.	p. 548
xpress.ParameterType	Enumeration used in problem.getControlInfo, problem.getAttribInfo.	p. 570
xpress.PreProbing	Enumeration used in problem.controls.PREPROBING.	p. 549
xpress.Presolve	Enumeration used in problem.controls.PRESOLVE.	p. 553
xpress.PresolveCoefElim	Enumeration used in problem.controls.PRECOEFELIM.	p. 564
xpress.PresolveDomColumn	Enumeration used in problem.controls.PREDOMCOL.	p. 566
xpress.PresolveDomRow	Enumeration used in problem.controls.PREDOMROW.	p. 565
xpress.PresolveOperations	Enumeration used in problem.controls.PRESOLVEOPS.	p. 550
xpress.PresolveState	Enumeration used in problem.attributes.PRESOLVESTATE.	p. 551
xpress.Pricing	Enumeration used in problem.controls.PRICINGALG.	p. 554
xpress.PrimalUnshift	Enumeration used in problem.controls.PRIMALUNSHIFT.	p. 567
xpress.QConvexity		p. 571
xpress.RefineOps		p. 559
xpress.RepairIndefiniteQuadratic	Enumeration used in problem.controls.REPAIRINDEFINITEQ.	p. 568
xpress.RowFlag		p. 581
xpress.Scaling	Enumeration used in problem.controls.SCALING.	p. 557
xpress.SDPCutStrategy	Enumeration used in problem.controls.SDPCUTSTRATEGY.	p. 593
xpress.SolAvailable		p. 532
xpress.SolInfo		p. 572
xpress.SolStatus	Enumeration used in problem.attributes.SOLSTATUS.	p. 527
xpress.SolveStatus	Enumeration used in problem.attributes.SOLVESTATUS.	p. 528
xpress.StopType	Enumeration used in problem.interrupt.	p. 536
xpress.TreeDiagnostics	Enumeration used in problem.controls.TREEDIAGNOSTICS.	p. 561
xpress.TunerHistory	Enumeration used in problem.controls.TUNERHISTORY.	p. 576
xpress.TunerMethod	Enumeration used in problem.controls.TUNERMETHOD.	p. 574
xpress.TunerMode	Enumeration used in problem.controls.TUNERMODE.	p. 573
xpress.TunerRootAlg		p. 577
xpress.TunerTarget	Enumeration used in problem.controls.TUNERTARGET.	p. 575
xpress.UserSolStatus		p. 589
xpress.VariableSelection	Enumeration used in problem.controls.VARSELECTION.	p. 556

xpress.SolStatus

Description

Enumeration used in `problem.attributes.SOLSTATUS`.

Constants

`NOTFOUND` 0

`OPTIMAL` 1

`FEASIBLE` 2

`INFEASIBLE` 3

`UNBOUNDED` 4

xpress.SolveStatus

Description

Enumeration used in `problem.attributes.SOLVESTATUS`.

Constants

`UNSTARTED` 0

`STOPPED` 1

`FAILED` 2

`COMPLETED` 3

xpress.LPStatus

Description

Enumeration used in `problem.attributes.LPSTATUS`.

Constants

```
UNSTARTED  0
OPTIMAL  1
INFEAS  2
CUTOFF  3
UNFINISHED  4
UNBOUNDED  5
CUTOFF_IN_DUAL  6
UNSOLVED  7
NONCONVEX  8
```

xpress.MIPStatus

Description

Enumeration used in `problem.attributes.MIPSTATUS`.

Constants

```
NOT_LOADED 0
LP_NOT_OPTIMAL 1
LP_OPTIMAL 2
NO_SOL_FOUND 3
SOLUTION 4
INFEAS 5
OPTIMAL 6
UNBOUNDED 7
```

xpress.IISSolStatus

Description

Enumeration used in `problem.attributes.IISSOLSTATUS`.

Constants

`UNSTARTED` 0

`FEASIBLE` 1

`COMPLETED` 2

`UNFINISHED` 3

xpress.SolAvailable

Constants

```
NOTFOUND 0
OPTIMAL 1
FEASIBLE 2
```

xpress.OptimizeType

Description

Enumeration used in `problem.attributes.OPTIMIZETYPEUSED`.

Constants

`NONE`	-1
`LP`	0
`MIP`	1
`LOCAL`	2
`GLOBAL`	3

xpress.BarOrder

Description

Enumeration used in `problem.controls.BARORDER`.

Constants

`DEFAULT` 0

`MIN_DEGREE` 1

`MIN_LOCAL_FILL` 2

`NESTED_DISSECTION` 3

xpress.DefaultAlg

Description

Enumeration used in `problem.controls.DEFAULTALG`.

Constants

```
DEFAULT  1
DUAL     2
PRIMAL   3
BARRIER  4
NETWORK  5
```

xpress.StopType

Description

Enumeration used in `problem.interrupt`.

Constants

`NONE` 0
`TIMELIMIT` 1
`CTRLC` 2
`NODELIMIT` 3
`ITERLIMIT` 4
`MIPGAP` 5
`SOLLIMIT` 6
`GENERICERROR` 7
`MEMORYERROR` 8
`USER` 9
`SOLVECOMPLETE` 10
`LICENSELOST` 11
`NUMERICALERROR` 13
`WORKLIMIT` 14
`NEXTOBJECTIVE` 15
`MIPSTAGE` 16

xpress.AlwaysNeverAutomatic

Description

Enumeration used in `problem.controls.CROSSOVER`, `problem.controls.DUALIZE`, `problem.controls.ROOTPRESOLVE`, `problem.controls.BRANCHSTRUCTURAL`, `problem.controls.QUADRATICUNSHIFT`.

Constants

`AUTOMATIC` -1

`NEVER` 0

`ALWAYS` 1

xpress.OnOff

Description

Enumeration used in `problem.controls.AUTOPERTURB`, `problem.controls.IFCHECKCONVEXITY`.

Constants

`OFF`	0
`ON`	1

xpress.BacktrackAlg

Description

Enumeration used in `problem.controls.BACKTRACK`, `problem.controls.BACKTRACKTIE`.

Constants

`AUTOMATIC` -1

`BEST_ESTIMATE` 2

`BEST_BOUND` 3

`DEEPEST_NODE` 4

`HIGHEST_NODE` 5

`EARLIEST_NODE` 6

`LATEST_NODE` 7

`RANDOM` 8

`MIN_INFEAS` 9

`BEST_ESTIMATE_MIN_INFEAS` 10

`DEEPEST_BEST_ESTIMATE` 11

`DEEPEST_BEST_BOUND` 12

xpress.BranchChoice

Description

Enumeration used in `problem.controls.BRANCHCHOICE`.

Constants

`MIN_EST_FIRST` 0

`MAX_EST_FIRST` 1

`INC_SOL_FIRST` 2

`ROOTNODE_SOL_FARTHER` 3

xpress.CholeskyAlgorithm

Description

Enumeration used in `problem.controls.CHOLESKYALG`.

Constants

`PULL_CHOLESKY`	0
`PUSH_CHOLESKY`	1

xpress.CrossoverDynamicReduction

Description

Enumeration used in `problem.controls.CROSSOVERDRP`.

Constants

`BEFORE_CROSSOVER` 1

`INSIDE_CROSSOVER` 2

`AGGRESSIVE_BEFORE_CROSSOVER` 4

xpress.DualGradient

Description

Enumeration used in `problem.controls.DUALGRADIENT`.

Constants

`AUTOMATIC` -1

`DEVEX` 0

`STEEPESTEDGE` 1

`DIRECTSTEEPESTEDGE` 2

`SPARSEDEVEX` 3

xpress.DualStrategy

Description

Enumeration used in `problem.controls.DUALSTRATEGY`.

Constants

`REMOVE_INFEAS_WITH_PRIMAL` 0

`REMOVE_INFEAS_WITH_DUAL` 1

xpress.FeasibilityPump

Description

Enumeration used in `problem.controls.FEASIBILITYPUMP`.

Constants

`AUTOMATIC` -1

`NEVER` 0

`ALWAYS` 1

`LASTRESORT` 2

xpress.HeuristicSearchSelect

Description

Enumeration used in `problem.controls.HEURSEARCHROOTSELECT`, `problem.controls.HEURSEARCHTREESELECT`.

Constants

`LOCAL_SEARCH_LARGE_NEIGHBOURHOOD` 0

`LOCAL_SEARCH_NODE_NEIGHBOURHOOD` 1

`LOCAL_SEARCH_SOLUTION_NEIGHBOURHOOD` 2

`LOCAL_SEARCH_COMBINE` 3

`LOCAL_SEARCH_ZERO_OBJECTIVE` 5

`LOCAL_SEARCH_AUXILIARY_OBJECTIVE` 6

xpress.NodeSelectionCriteria

Description

Enumeration used in `problem.controls.NODESELECTION`.

Constants

`LOCAL_FIRST` 1

`BEST_FIRST` 2

`LOCAL_DEPTH_FIRST` 3

`BEST_FIRST_THEN_LOCAL_FIRST` 4

`DEPTH_FIRST` 5

xpress.OutputDetail

Description

Enumeration used in `problem.controls.OUTPUTLOG`.

Constants

```
NO_OUTPUT  0
FULL_OUTPUT  1
ERRORS_AND_WARNINGS  3
ERRORS  4
```

xpress.PreProbing

Description

Enumeration used in `problem.controls.PREPROBING`.

Constants

`AUTOMATIC` -1

`DISABLED` 0

`LIGHT` 1

`FULL` 2

`FULL_AND_REPEAT` 3

xpress.PresolveOperations

Description

Enumeration used in `problem.controls.PRESOLVEOPS`.

Constants

`SINGLETONCOLUMNREMOVAL` 1
`SINGLETONROWREMOVAL` 2
`FORCINGROWREMOVAL` 4
`DUALREDUCTIONS` 8
`REDUNDANTROWREMOVAL` 16
`DUPLICATECOLUMNREMOVAL` 32
`DUPLICATEROWREMOVAL` 64
`STRONGDUALREDUCTIONS` 128
`VARIABLEELIMINATIONS` 256
`NOIPREDUCTIONS` 512
`NOGLOBALDOMAINCHANGE` 1024
`NOADVANCEDIPREDUCTIONS` 2048
`NOINTEGERELIMINATIONS` 4096
`NOSOLUTIONENUMERATION` 8192
`LINEARLYDEPENDANTROWREMOVAL` 16384
`NOINTEGERVARIABLEANDSOSDETECTION` 32768
`NOIMPLIEDBOUNDS` 65536
`NOCLIQUEPRESOLVE` 131072
`NOMOD2REDUCTIONS` 262144
`NODUALREDONGLOBALS` 536870912

xpress.PresolveState

Description

Enumeration used in `problem.attributes.PRESOLVESTATE`.

Constants

`PROBLEMLOADED` 1

`PROBLEMLPPRESOLVED` 2

`PROBLEMMIPPRESOLVED` 4

`SOLUTIONVALID` 128

xpress.MipPresolve

Description

Enumeration used in `problem.controls.MIPPRESOLVE`.

Constants

`REDUCED_COST_FIXING` 1

`LOGIC_PREPROCESSING` 2

`ALLOW_CHANGE_BOUNDS` 8

`DUAL_REDUCTIONS` 16

`GLOBAL_COEFFICIENT_TIGHTENING` 32

`OBJECTIVE_BASED_REDUCTIONS` 64

`ALLOW_TREE_RESTART` 128

`SYMMETRY_REDUCTIONS` 256

xpress.Presolve

Description

Enumeration used in `problem.controls.PRESOLVE`.

Constants

`NOPRIMALINFEASIBILITY` -1

`NONE` 0

`DEFAULT` 1

`KEEPREDUNDANTBOUNDS` 2

`REMOVEREDUNDANTBOUNDS` 3

xpress.Pricing

Description

Enumeration used in `problem.controls.PRICINGALG`.

Constants

`PARTIAL` -1

`DEFAULT` 0

`DEVEX` 1

`STEEPEST_EDGE` 2

`STEEPEST_EDGE_UNIT_WEIGHTS` 3

xpress.CutStrategy

Description

Enumeration used in `problem.controls.CUTSTRATEGY`.

Constants

```
DEFAULT  -1
NONE     0
CONSERVATIVE  1
MODERATE  2
AGGRESSIVE  3
```

xpress.VariableSelection

Description

Enumeration used in `problem.controls.VARSELECTION`.

Constants

`AUTOMATIC` -1

`MIN_UPDOWN_PSEUDO_COSTS` 1

`SUM_UPDOWN_PSEUDO_COSTS` 2

`MAX_UPDOWN_PSEUDO_COSTS_PLUS_TWICE_MIN` 3

`MAX_UPDOWN_PSEUDO_COSTS` 4

`DOWN_PSEUDO_COST` 5

`UP_PSEUDO_COST` 6

`WEIGHTED_COMBINATION_UPDOWN_PSEUDO_COST` 7

`PRODUCT_UPDOWN_PSEUDO_COST` 8

xpress.Scaling

Description

Enumeration used in `problem.controls.SCALING`.

Constants

`ROW_SCALING` 1
`COLUMN_SCALING` 2
`ROW_SCALING_AGAIN` 4
`MAXIMUM` 8
`CURTIS_REID` 16
`BY_MAX_ELEM_NOT_GEO_MEAN` 32
`BIGM` 64
`SIMPLEX_OBJECTIVE_SCALING` 128
`IGNORE_QUADRATIC_ROW_PART` 256
`BEFORE_PRESOLVE` 512
`NO_SCALING_ROWS_UP` 1024
`NO_SCALING_COLUMNS_DOWN` 2048
`DISABLE_GLOBAL_OBJECTIVE_SCALING` 4096
`RHS_SCALING` 8192
`NO_AGGRESSIVE_Q_SCALING` 16384
`SLACK_SCALING` 32768
`RUIZ` 65536
`DOGLEG` 131072
`BEFORE_AND_AFTER_PRESOLVE` 262144

xpress.CutSelect

Description

Enumeration used in `problem.controls.CUTSELECT`.

Constants

`CLIQUE` 1855

`MIR` 1887

`COVER` 1951

`FLOWPATH` 3871

`IMPLICATION` 5919

`LIFT_AND_PROJECT` 10015

`DISABLE_CUT_ROWS` 18207

`GUB_COVER` 34591

`DEFAULT` -1

xpress.RefineOps

Constants

```
LPOPTIMAL 1
MIPSOLUTION 2
MIPNODELP 8
LPPRESOLVE 16
ITERATIVEREFINER 32
REFINERPRECISION 64
REFINERUSEPRIMAL 128
REFINERUSEDUAL 256
MIPFIXGLOBALS 512
MIPFIXGLOBALSTARGET 1024
```

xpress.DualizeOps

Constants

```
SWITCHALGORITHM 1
```

xpress.TreeDiagnostics

Description

Enumeration used in `problem.controls.TREEDIAGNOSTICS`.

Constants

`MEMORY_USAGE_SUMMARIES` 1

`MEMORY_SAVED_REPORTS` 2

`MEMORY_SAVING_LOG` 4

xpress.BarPresolveOps

Description

Enumeration used in `problem.controls.BARPRESOLVEOPS`.

Constants

`STANDARD_PRESOLVE` 0

`EXTRA_BARRIER_PRESOLVE` 1

xpress.MipRestart

Description

Enumeration used in `problem.controls.MIPRESTART`.

Constants

```
DEFAULT  -1
OFF       0
MODERATE  1
AGGRESSIVE  2
```

xpress.PresolveCoefElim

Description

Enumeration used in `problem.controls.PRECOEFELIM`.

Constants

`DISABLED` 0

`AGGRESSIVE` 1

`CAUTIOUS` 2

xpress.PresolveDomRow

Description

Enumeration used in `problem.controls.PREDOMROW`.

Constants

`AUTOMATIC` -1

`DISABLED` 0

`CAUTIOUS` 1

`MEDIUM` 2

`AGGRESSIVE` 3

xpress.PresolveDomColumn

Description

Enumeration used in `problem.controls.PREDOMCOL`.

Constants

`AUTOMATIC` -1

`DISABLED` 0

`CAUTIOUS` 1

`AGGRESSIVE` 2

`GENERAL` 3

xpress.PrimalUnshift

Description

Enumeration used in `problem.controls.PRIMALUNSHIFT`.

Constants

`ALLOW_DUAL_UNSHIFT` 0

`NO_DUAL_UNSHIFT` 1

xpress.RepairIndefiniteQuadratic

Description

Enumeration used in `problem.controls.REPAIRINDEFINITEQ`.

Constants

`REPAIR_IF_POSSIBLE` 0

`NO_REPAIR` 1

xpress.ObjSense

Description

Enumeration used in `problem.chgObjSense`.

Constants

`MINIMIZE` 1

`MAXIMIZE` -1

xpress.ParameterType

Description

Enumeration used in `problem.getControlInfo`, `problem.getAttribInfo`.

Constants

`NOTDEFINED`	0
`INT`	1
`INT64`	2
`DOUBLE`	3
`STRING`	4

xpress.QConvexity

Constants

```
UNKNOWN  -1
NONCONVEX  0
CONVEX  1
REPAIRABLE  2
CONVEXCONE  3
CONECONVERTABLE  4
```

xpress.SolInfo

Constants

```
ABSPRIMALINFEAS 0
RELPRIMALINFEAS 1
ABSDUALINFEAS 2
RELDUALINFEAS 3
MAXMIPFRACTIONAL 4
ABSMIPINFEAS 5
RELMIPINFEAS 6
```

xpress.TunerMode

Description

Enumeration used in `problem.controls.TUNERMODE`.

Constants

`AUTOMATIC`	-1
`OFF`	0
`ON`	1

xpress.TunerMethod

Description

Enumeration used in `problem.controls.TUNERMETHOD`.

Constants

`AUTOMATIC` -1

`LPQUICK` 0

`MIPQUICK` 1

`MIPCOMPREHENSIVE` 2

`MIPROOTFOCUS` 3

`MIPTREEFOCUS` 4

`MIPSIMPLE` 5

`SLPQUICK` 6

`MISLPQUICK` 7

`MIPHEURISTICS` 8

`GLOBALQUICK` 9

`LPNUMERICS` 10

xpress.TunerTarget

Description

Enumeration used in `problem.controls.TUNERTARGET`.

Constants

`AUTOMATIC` -1

`TIMEGAP` 0

`TIMEBOUND` 1

`TIMEOBJVAL` 2

`INTEGRAL` 3

`SLPTIME` 4

`SLPOBJVAL` 5

`SLPVALIDATION` 6

`GAP` 7

`BOUND` 8

`OBJVAL` 9

`PRIMALINTEGRAL` 10

xpress.TunerHistory

Description

Enumeration used in `problem.controls.TUNERHISTORY`.

Constants

`IGNORE` 0

`APPEND` 1

`REUSE` 2

xpress.TunerRootAlg

Constants

```
DUAL   1
PRIMAL 2
BARRIER 4
NETWORK 8
```

xpress.LPFlags

Description

Enumeration used in `problem.controls.LPFLAGS`.

Constants

```
DUAL   1
PRIMAL 2
BARRIER 4
NETWORK 8
```

xpress.GenConsType

Constants

MAX	0
MIN	1
AND	2
OR	3
ABS	4

xpress.Clamping

Constants

```
PRIMAL 1
DUAL   2
SLACKS 4
RDJ    8
```

xpress.RowFlag

Constants

```
QUADRATIC 1
DELAYED 2
MODELCUT 4
INDICATOR 8
NONLINEAR 16
```

xpress.ObjControl

Constants

```
PRIORITY  20001
WEIGHT    20002
ABSTOL    20003
RELTOL    20004
RHS       20005
```

xpress.AllowCompute

Description

Enumeration used in `xpress.setComputeAllowed`, `xpress.getComputeAllowed`.

Constants

`ALWAYS` 1

`NEVER` 0

`DEFAULT` -1

xpress.ComputeLog

Description

Enumeration used in `problem.controls.COMPUTELOG`.

Constants

```
NEVER  0
REALTIME  1
ONCOMPLETION  2
ONERROR  3
```

xpress.Namespaces

Constants

```
ROW    1
COLUMN 2
SET    3
PWLCONS 4
GENCONS 5
OBJECTIVE 6
USERFUNC 7
INTERNALFUNC 8
USERFUNCNOCASE 9
INTERNALFUNCNOCASE 10
```

xpress.Globalboundingbox

Constants

```
NOT_APPLIED  0
ORIGINAL  1
AUXILIARY  2
```

xpress.MultiObjOps

Description

Enumeration used in `problem.controls.MULTIOBJOPS`.

Constants

`ENABLED` 1

`PRESOLVE` 2

`RCFIXING` 4

xpress.IISOps

Description

Enumeration used in `problem.controls.IISOPS`.

Constants

`BINARY` 1
`ZEROLOWER` 2
`FIXEDVAR` 4
`BOUND` 8
`GENINTEGRALITY` 16
`INTEGRALITY` 17
`VARIABLE` 25
`EQUALITY` 32
`GENERAL` 64
`PWL` 128
`SET` 256
`INDICATOR` 512
`DELAYED` 1024
`CONSTRAINT` 2048

xpress.UserSolStatus

Constants

```
NOT_CHECKED  0
ACCEPTED_FEASIBLE  1
ACCEPTED_OPTIMIZED  2
SEARCHED_SOL  3
SEARCHED_NOSOL  4
REJECTED_INFEAS_NOSEARCH  5
REJECTED_PARTIAL_NOSEARCH  6
REJECTED_FAILED_OPTIMIZE  7
DROPPED  8
REJECTED_CUTOFF  9
```

xpress.GlobalLSHEURStrategy

Constants

```
DEFAULT  -1
NONE   0
CONSERVATIVE  1
MODERATE  2
AGGRESSIVE  3
```

xpress.BARHGOps

Description

Enumeration used in `problem.controls.BARHGOPS`.

Constants

`ASYM_AVG` 1

`START_L1` 2

`START_L2` 4

`START_LINF` 8

`RESTART_L2` 16

`OMEGA_CONTRACT` 32

`OMEGA_INF` 64

`MAX_OBJSCALE` 128

`NO_OBJSCALE` 256

`HPDHG` 512

`HBASE` 1024

`UNWEIGHTED_AVERAGE` 2048

`RESTART_NOSCALE` 4096

`RESTART_OMEGA` 8192

xpress.BasisStatus

Constants

```
NONBASIC_LOWER 0
BASIC 1
NONBASIC_UPPER 2
SUPERBASIC 3
```

xpress.SDPCutStrategy

Description

Enumeration used in `problem.controls.SDPCUTSTRATEGY`.

Constants

```
DEFAULT  -1
NONE     0
CONSERVATIVE  1
MODERATE  2
AGGRESSIVE  3
```

xpress.BarHGPrecision

Description

Enumeration used in `problem.controls.BARHGPRECISION`.

Constants

`AUTOMATIC` -1

`SINGLE` 0

`DOUBLE` 1

xpress.MIPStopStage

Description

Enumeration used in `problem.controls.MIPSTOPSTAGE`.

Constants

`NONE` 0

`PRESOLVE` 1

`PREROOTHEURS` 2

`INITIALRELAXATION` 3

`ROOTNODE` 4

xpress.constants

Description

Constants used in Xpress NonLinear controls, attributes and function arguments.

Constants

```
DEL_COLON 2
DEL_COMMA 1
IFUN_ABS 17
IFUN_ARCCOS 31
IFUN_ARCSIN 30
IFUN_ARCTAN 32
IFUN_COS 28
IFUN_ERF 49
IFUN_ERFC 50
IFUN_EXP 16
IFUN_LN 15
IFUN_LOG 14
IFUN_LOG10 14
IFUN_MAX 34
IFUN_MIN 33
IFUN_PWL 35
IFUN_SIGN 46
IFUN_SIN 27
IFUN_SQRT 18
IFUN_TAN 29
KKT_CALCULATION_ACTIVITY_BASED 0
KKT_CALCULATION_ACTIVITY_BOTH 2
KKT_CALCULATION_MEASURE_BOTH 2
KKT_CALCULATION_MINIMZE_KKT_ERROR 1
KKT_CALCULATION_RECALCULATE_RDJ 0
KKT_CALCULATION_RESPECT_BASIS 1
KKT_JUST_CALCULATE 0
KKT_UPDATE_MULTIPLIERS 1
LOCALSOLVER_AUTO -1
LOCALSOLVER_KNITRO 1
LOCALSOLVER_OPTIMIZER 2
LOCALSOLVER_XSLP 0
MSSET_INITIALFILTERED 6
MSSET_INITIALVALUES 0
MSSET_KNITRO_BASIC 4
MSSET_KNITRO_EXTENDED 5
MSSET_SLP_BASIC 2
MSSET_SLP_EXTENDED 3
MSSET_SOLVERS 1
NLP1DERIVATIVE 128
NLP2DERIVATIVE 64
NLPALLCALCS 32
NLPALLDERIVATIVES 256
```

```
NLPFORMULACOEFFCOLUMNINDEX -1000
NLPINSTANCEFUNCTION 512
NLPINTERNALFUNCNAMES 8
NLPINTERNALFUNCNAMESNOCASE 10
NLPPRESOLVEDOMAIN 32
NLPPRESOLVEFIXALL 4
NLPPRESOLVEFIXZERO 2
NLPPRESOLVEINTBOUNDS 16
NLPPRESOLVELEVEL_BASIC 2
NLPPRESOLVELEVEL_FULL 4
NLPPRESOLVELEVEL_LINEAR 3
NLPPRESOLVELEVEL_LOCALIZED 1
NLPPRESOLVEOPS_ELIMINATIONS 2048
NLPPRESOLVEOPS_GENERAL 1
NLPPRESOLVEOPS_NOLINEAR 4096
NLPPRESOLVEOPS_NOSIMPLIFIER 8192
NLPPRESOLVEOPS_NO_DUAL_SIDE 1024
NLPPRESOLVEPROTECT 32768
NLPPRESOLVESETBOUNDS 8
NLPRECALC 8
NLPREFORMULATE_ABS 128
NLPREFORMULATE_ALLABS 512
NLPREFORMULATE_ALLMINMAX 1024
NLPREFORMULATE_MINMAX 256
NLPREFORMULATE_PWL 64
NLPREFORMULATE_QCQP2SLP 8
NLPREFORMULATE_QP2SLP 2
NLPREFORMULATE_QPSOLVE 32
NLPREFORMULATE_SLP2QCQP 4
NLPREFORMULATE_SLP2QP 1
NLPREFORMULATE_SOCP2SLP 16
NLPSOLSTATUS_GLOBALLYOPTIMAL_NODUALS 3
NLPSOLSTATUS_GLOBALLYOPTIMAL_WITHDUALS 4
NLPSOLSTATUS_LOCALLYOPTIMAL_WITHDUALS 2
NLPSOLSTATUS_NONE 0
NLPSOLSTATUS_SOLUTION_NODUALS 1
NLPSOLVER_AUTOMATIC -1
NLPSOLVER_GLOBAL 2
NLPSOLVER_LOCAL 1
NLPSTATUS_INFEASIBLE 4
NLPSTATUS_LOCALLY_INFEASIBLE 3
NLPSTATUS_LOCALLY_OPTIMAL 1
NLPSTATUS_NOSOLUTION 3
NLPSTATUS_OPTIMAL 2
NLPSTATUS_SOLUTION 1
NLPSTATUS_UNBOUNDED 5
NLPSTATUS_UNFINISHED 6
NLPSTATUS_UNSOLVED 7
```

```
NLPSTATUS_UNSTARTED 0
NLPTOLCALC 16
NLPUSERFUNCNAMES 7
NLPUSERFUNCNAMESNOCASE 9
OP_DIVIDE 4
OP_EXPONENT 2
OP_MINUS 6
OP_MULTIPLY 3
OP_PLUS 5
OP_UMINUS 1
SLPACTIVEPENALTY 32768
SLPACTIVESBROW 262144
SLPACTIVESTEPBOUND 131072
SLPALLERRORVECTORS 8
SLPALLROWERRORVECTORS 512
SLPAMEANWEIGHT 32
SLPANALYZE_AUTOSAVEPOOL 32
SLPANALYZE_EXTENDEDFINALSUMMARY 8
SLPANALYZE_INFEASIBLE_ITERATION 16
SLPANALYZE_RECORDCASCADE 2
SLPANALYZE_RECORDLINEARIZATION 1
SLPANALYZE_RECORDLINESEARCH 4
SLPANALYZE_SAVEFILE 256
SLPANALYZE_SAVEITERBASIS 128
SLPANALYZE_SAVELINEARIZATIONS 64
SLPBOUNDSSET 67108864
SLPCASCADECONTRACTION 536870912
SLPCASCADEDBOUNDS 16384
SLPCASCADE_ALL 1
SLPCASCADE_ALL_COEF_VAR 4
SLPCASCADE_ALL_STRUCT_VAR 16
SLPCASCADE_AUTOAPPLY 256
SLPCASCADE_COEF_VAR 2
SLPCASCADE_DRCOL_PREVOUSVALUE 64
SLPCASCADE_DRCOL_PVRANGE 128
SLPCASCADE_SECONDARY_GROUPS 32
SLPCASCADE_STRUCT_VAR 8
SLPCLAMPEXTENDEDACTIVESB 32768
SLPCLAMPEXTENDEDALL 65536
SLPCOLINFO_CONVERGENCESTATUS 11
SLPCOLINFO_DELTA 4
SLPCOLINFO_DELTADJ 5
SLPCOLINFO_DELTAINDEX 3
SLPCOLINFO_DETROW 10
SLPCOLINFO_LPVALUE 9
SLPCOLINFO_RDJ 2
SLPCOLINFO_SB 7
SLPCOLINFO_SBDUAL 8
```

```
SLPCOLINFO_UPDATEROW 6
SLPCOLINFO_VALUE 1
SLPCONVERGEBIT_ATOL 2
SLPCONVERGEBIT_CTOL 1
SLPCONVERGEBIT_EXTENDEDSCALING 1024
SLPCONVERGEBIT_ITOL 8
SLPCONVERGEBIT_MTOL 4
SLPCONVERGEBIT_NOQUADCHECK 8192
SLPCONVERGEBIT_OTOL 256
SLPCONVERGEBIT_REQUIRE_OTOL_R 32768
SLPCONVERGEBIT_STOL 16
SLPCONVERGEBIT_USER 32
SLPCONVERGEBIT_VALIDATION 2048
SLPCONVERGEBIT_VALIDATION_K 4096
SLPCONVERGEBIT_VTOL 64
SLPCONVERGEBIT_WTOL 512
SLPCONVERGEBIT_XTOL 128
SLPDELTA_CONT 0
SLPDELTA_EXPLORE 3
SLPDELTA_INTEGER 2
SLPDELTA_SEMICONT 1
SLPDYNAMICDAMPING 8
SLPEQUALITYERRORVECTORS 4
SLPESCALATEPENALTIES 256
SLPESTIMATESTEPBOUNDS 4
SLPEVENHANDEDAUGMENTATION 2
SLPFILTER_CASCADE 2
SLPFILTER_KEEPBEST 1
SLPFILTER_ZEROLINESEARCH 4
SLPFILTER_ZEROLINESEARCHTR 8
SLPFIXEDVAR 33554432
SLPGRIDANNEALING 4
SLPGRIDCYCLIC 2
SLPGRIDENUMERATE 1
SLPHASCALCIV 8
SLPHASCONVERGED 65536
SLPHASDELTA 2
SLPHASERROR 768
SLPHASIV 4
SLPHASMINUSERROR 512
SLPHASNOCOEFS 1
SLPHASNONZEROCOEF 4096
SLPHASPLUSERROR 256
SLPHASSLPELEMENT 65536
SLPHASZEROBOUND 16777216
SLPHOLDVALUES 16
SLPISDELTA 256
SLPISDETERMININGROW 1024
```

SLPISEQUALSCOLUMN 16384
SLPISERRORVECTOR 6144
SLPISINCOEFS 4194304
SLPISINGLOBAL 8388608
SLPISINTEGER 268435456
SLPISMINUSERRORVECTOR 4096
SLPISMINUSPENALTYDELTA 1024
SLPISMISCROW 64
SLPISMISCVECTOR 8192
SLPISPENALTYDELTA 1536
SLPISPENALTYROW 4
SLPISPLUSERRORVECTOR 2048
SLPISPLUSPENALTYDELTA 512
SLPISSBROW 128
SLPISSTRUCTURALCOLUMN 2097152
SLPISUPDATEROW 2
SLPMAXCOSTOPTION 2048
SLPMINIMUMAUGMENTATION 1
SLPMIPFINALFIXSLP 256
SLPMIPFINALRELAXSLP 128
SLPMIPINITIALFIXSLP 8
SLPMIPINITIALRELAXSLP 4
SLPMIPINITIALSLP 1
SLPMIPNODEFIXSLP 32
SLPMIPNODELIMITSLP 64
SLPMIPNODERELAXSLP 16
SLPMIPWITHINSLP 512
SLPNOERRORVECTORS 2048
SLPNOFORMULADOMAINIV 2048
SLPNOLPPOLISHING 8192
SLPNONZEROBOUND 1024
SLPNOPRESOLVECOEFFICIENTS 256
SLPNOPRESOLVEDELTAS 512
SLPNOSTEPBOUNDS 1
SLPNOUPDATEIFONLYIV 1024
SLPNOUSEFULDELTA 134217728
SLPPENALTYDELTAVECTORS 16
SLPQUICKCONVERGENCECHECK 128
SLPREDUNDANTROW 8192
SLPRESETDELTAZ 64
SLPRESIDUALERRORS 4096
SLPRETAINPREVIOUSVALUE 32
SLPROOTMIPDRIVEN 4096
SLPROWINFO_CURRENTPENALTYERROR 6
SLPROWINFO_CURRENTPENALTYFACTOR 7
SLPROWINFO_DUAL 2
SLPROWINFO_MAXPENALTYERROR 4
SLPROWINFO_NUMPENALTYERRORS 3

SLPROWINFO_PENALTYCOLUMNMINUS 11
SLPROWINFO_PENALTYCOLUMNMINUSDJ 13
SLPROWINFO_PENALTYCOLUMNMINUSVALUE 12
SLPROWINFO_PENALTYCOLUMNPLUS 8
SLPROWINFO_PENALTYCOLUMNPLUSDJ 10
SLPROWINFO_PENALTYCOLUMNPLUSVALUE 9
SLPROWINFO_SLACK 1
SLPROWINFO_TOTALPENALTYERROR 5
SLPSBFROMABSVALUES 128
SLPSBFROMVALUES 64
SLPSKIPIVLPHEURISTICS 4096
SLPSLPTHENMIP 1024
SLPSTATUS_ADAPTIVEITERS 33554432
SLPSTATUS_CLAMPING 16777216
SLPSTATUS_CONSECUTIVE_INFEAS 4194304
SLPSTATUS_CONVERGEDOBJOBJ 128
SLPSTATUS_CONVERGEDOBJSBX 2
SLPSTATUS_CONVERGEDOBJUCC 1
SLPSTATUS_ENFORCED 2097152
SLPSTATUS_ERROTOL 131072
SLPSTATUS_EVTOL 262144
SLPSTATUS_INTEGERINFEASIBLE 32
SLPSTATUS_KEEPBEST 8388608
SLPSTATUS_LPINFEASIBLE 4
SLPSTATUS_LPUNFINISHED 8
SLPSTATUS_MAXSLPITERATIONS 16
SLPSTATUS_MAXTIME 512
SLPSTATUS_NOVARSINACTIVE 4096
SLPSTATUS_OBJQNONCONVEX 67108864
SLPSTATUS_OTOL 8192
SLPSTATUS_POLISHED 524288
SLPSTATUS_POLISH_FAILURE 1048576
SLPSTATUS_RESIDUALPENALTIES 64
SLPSTATUS_USER 1024
SLPSTATUS_VARSLINKEDINACTIVE 2048
SLPSTATUS_VTOL 16384
SLPSTATUS_WTOL 65536
SLPSTATUS_XTOL 32768
SLPSTEPBOUNDROWS 256
SLPSTEPBOUNDSASREQUIRED 2
SLPSWITCHTOPRIMAL 512
SLPTOLSETBIT_RA 4
SLPTOLSETBIT_RI 64
SLPTOLSETBIT_RM 16
SLPTOLSETBIT_RS 256
SLPTOLSETBIT_TA 2
SLPTOLSETBIT_TC 1
SLPTOLSETBIT_TI 32

```
SLPTOLSETBIT_TM  8
SLPTOLSETBIT_TS  128
SLPTOLSET_RA  2
SLPTOLSET_RI  6
SLPTOLSET_RM  4
SLPTOLSET_RS  8
SLPTOLSET_TA  1
SLPTOLSET_TC  0
SLPTOLSET_TI  5
SLPTOLSET_TM  3
SLPTOLSET_TS  7
SLPTRACEMASK_CASCADE  8
SLPTRACEMASK_COLS  4
SLPTRACEMASK_CONVERGE  4096
SLPTRACEMASK_DUAL  64
SLPTRACEMASK_GENERALFIT  1
SLPTRACEMASK_LINESEARCH  8192
SLPTRACEMASK_REDUCEDCOST  512
SLPTRACEMASK_ROWS  2
SLPTRACEMASK_SLACK  32
SLPTRACEMASK_SLPVALUE  1024
SLPTRACEMASK_SOLUTION  256
SLPTRACEMASK_STEPBOUND  2048
SLPTRACEMASK_TYPE  16
SLPTRACEMASK_WEIGHT  128
SLPTRANSFERROW  262144
SLPUNCONVERGEDROW  16384
SLPUSEFULDELTA  134217728
TOK_COL  10
TOK_CON  1
TOK_DEL  24
TOK_EOF  0
TOK_FUN  11
TOK_IFUN  12
TOK_LB  21
TOK_OP  31
TOK_RB  22
USERFUNCTION_MAP  1
USERFUNCTION_MAPDELTA  4
USERFUNCTION_MULTIMAP  3
USERFUNCTION_MULTIMAPDELTA  6
USERFUNCTION_VECMAP  2
USERFUNCTION_VECMAPDELTA  5
```

8.14 Deprecated functionality

The following functions are deprecated and will be removed in a future release.

`branchobj.addbounds` *(Deprecated)* Adds new bounds to a branch of a user branching object. p. 913

`branchobj.addbranches` *(Deprecated)* Adds new, empty branches to a user defined branching object. p. 914

`branchobj.addcuts` *(Deprecated)* Adds stored user cuts as new constraints to a branch of a user branching object. p. 915

`branchobj.addrows` *(Deprecated)* Adds new constraints to a branch of a user branching object. p. 916

`branchobj.getbounds` *(Deprecated)* Returns the bounds for a branch of a user branching object. p. 917

`branchobj.getbranches` *(Deprecated)* Returns the number of branches of a branching object. p. 918

`branchobj.getid` *(Deprecated)* Returns the unique identifier assigned to a branching object. p. 919

`branchobj.getlasterror` *(Deprecated)* Returns the last error encountered during a call to the given branch object. p. 920

`branchobj.getrows` *(Deprecated)* Returns the constraints for a branch of a user branching object. p. 921

`branchobj.setpreferredbranch` *(Deprecated)* Specifies which of the child nodes corresponding to the branches of the object should be explored first. p. 922

`branchobj.setpriority` *(Deprecated)* Sets the priority value of a user branching object. p. 923

`problem.addcbafterobjective` *(Deprecated)* Declares a callback which will be called after each objective in a multi-objective problem is solved. p. 638

`problem.addcbbariteration` *(Deprecated)* Declares a barrier iteration callback function, called after each iteration during the interior point algorithm, with the ability to access the current barrier solution/slack/duals or reduced cost values, and to ask barrier to stop. p. 614

`problem.addcbbarlog` *(Deprecated)* Declares a barrier log callback function, called at each iteration during the interior point algorithm. p. 616

`problem.addcbbeforeobjective` *(Deprecated)* Declares a callback which will be called before each objective in a multi-objective problem is solved. p. 637

`problem.addcbchecktime` *(Deprecated)* Declares a callback function which is called every time the Optimizer checks if the time limit has been reached. p. 617

`problem.addcbchgbranchobject` *(Deprecated)* Declares a callback function that will be called every time the Optimizer has selected a MIP entity for branching. p. 618

`problem.addcbcutlog` *(Deprecated)* Declares a cut log callback function, called each time the cut log is printed. p. 619

`problem.addcbcutround` *(Deprecated)* Declares a callback function that is called when the Optimizer could separate cutting planes during the branch and bound search. p. 620

`problem.addcbdestroymt` *(Deprecated)* Declares a callback function that is called every time a MIP thread is destroyed by the parallel MIP code. p. 621

`problem.addcbgapnotify` *(Deprecated)* Declares a gap notification callback, to be called when a MIP solve reaches a predefined target, set using the `miprelgapnotify`, `mipabsgapnotify`, `mipabsgapnotifyobj`, and/or `mipabsgapnotifybound` controls. p. 622

`problem.addcbinfnode` *(Deprecated)* Declares a user infeasible node callback function, called after the current node has been found to be infeasible during the Branch and Bound search. p. 625

`problem.addcbintsol` *(Deprecated)* Declares a user integer solution callback function, called every time an integer solution is found by heuristics or during the Branch and Bound search. p. 626

`problem.addcblplog` *(Deprecated)* Declares a simplex log callback function which is called after every LPLOG iterations of the simplex algorithm. p. 627

`problem.addcbmessage` *(Deprecated)* Declares an output callback function, called every time a text line relating to the given prob is output by the Optimizer. p. 628

`problem.addcbmiplog` *(Deprecated)* Declares a MIP log callback function, called each time the MIP log is printed. p. 624

`problem.addcbmipthread` *(Deprecated)* Declares a MIP thread callback function, called every time a MIP worker problem is created by the parallel MIP code. p. 629

`problem.addcbnewnode` *(Deprecated)* Declares a callback function that will be called every time a new node is created during the branch and bound search. p. 630

`problem.addcbnodecutoff` *(Deprecated)* Declares a user node cutoff callback function, called every time a node is cut off as a result of an improved integer solution being found during the branch and bound search. p. 631

`problem.addcbnodelpsolved` *(Deprecated)* Declares a node LP solved callback function, called during the branch and bound search, after the LP relaxation has been solved for the current node, but before any internal cuts and heuristics have been applied. p. 632

`problem.addcboptnode` *(Deprecated)* Declares an optimal node callback function, called during the branch and bound search, after the LP relaxation has been solved for the current node, and after any internal cuts and heuristics have been applied, but before the Optimizer checks if the current node should be branched. p. 633

`problem.addcbpreintsol` *(Deprecated)* Declares a user integer solution callback function, called when an integer solution is found by heuristics or during the branch and bound search, but before it is accepted by the Optimizer. p. 634

`problem.addcbprenode` *(Deprecated)* Declares a preprocess node callback function, called before the LP relaxation of a node has been optimized, so the solution at the node will not be available. p. 635

`problem.addcbpresolve` *(Deprecated)* Declares a callback to be called after presolve has been performed. p. 639

`problem.addcbusersolnotify` *(Deprecated)* Declares a callback function to be called each time a solution added by problem.addmipsol has been processed. p. 636

`problem.addcoefs` *(Deprecated)* Add non-linear coefficients to the SLP problem p. 640

`problem.addcols` *(Deprecated)* Add columns to the problem after passing it to the Optimizer using the input routines. p. 642

`problem.addcuts` *(Deprecated)* Adds cuts directly to the matrix at the current node. p. 643

`problem.adddfs` *(Deprecated)* Add a set of distribution factors p. 644

`problem.addgencons` *(Deprecated)* Adds one or more general constraints to the problem. p. 645

`problem.addmipsol` *(Deprecated)* Adds a new feasible, infeasible or partial MIP solution for the problem to the Optimizer. p. 646

`problem.addnames` *(Deprecated)* Associates names with rows, columns, sets, piecewise linear constraints, general constraints or objectives. p. 647

`problem.addobj` *(Deprecated)* Appends an objective function with the given coefficients to a multi-objective problem. p. 648

`problem.addpwlcons` *(Deprecated)* Adds one or more piecewise linear constraints to the problem. p. 649

`problem.addqmatrix` *(Deprecated)* Adds a new quadratic matrix into a row defined by triplets. p. 650

`problem.addrows` *(Deprecated)* Adds rows and their coefficient to the problem. p. 651

`problem.addsetnames` *(Deprecated)* Associates names with a range of sets. p. 652

`problem.addtolsets` *(Deprecated)* Add sets of standard tolerance values to an SLP problem p. 653

`problem.addvars` *(Deprecated)* Add SLP variables defined as matrix columns to an SLP problem p. 654

`problem.basisstability` *(Deprecated)* Returns various measures for the stability of the current basis, including the basis condition number. p. 655

`problem.bndsa` *(Deprecated)* Returns upper and lower sensitivity ranges for specified variables' lower and upper bounds. p. 656

`problem.btran` *(Deprecated)* Post-multiplies a (row) vector provided by the user by the inverse of the current basis. p. 657

`problem.calcobjective` *(Deprecated)* Returns the objective value of a given solution. p. 659

`problem.calcobjn` *(Deprecated)* Returns the value of a given objective. p. 658

`problem.calcreducedcosts` *(Deprecated)* Returns the reduced cost values for a given (row) dual solution. p. 660

`problem.calcslacks` *(Deprecated)* Calculates the row slack values for a given solution. p. 661

`problem.calcsolinfo` *(Deprecated)* Returns the required property of a solution, like maximum infeasibility of a given primal and duals solution. p. 662

`problem.cascade` *(Deprecated)* Re-calculate consistent values for SLP variables based on the current values of the remaining variables p. 663

`problem.cascadeorder` *(Deprecated)* Establish a re-calculation sequence for SLP variables with determining rows. p. 664

`problem.chgbounds` *(Deprecated)* Changes the bounds on columns in the problem. p. 665

`problem.chgcascadenlimit` *(Deprecated)* Set a variable specific cascade iteration limit p. 668

`problem.chgccoef` *(Deprecated)* Add or change a single matrix coefficient using a string for the formula p. 692

`problem.chgcoef` *(Deprecated)* Changes a single coefficient in the problem. p. 666

`problem.chgcoltype` *(Deprecated)* Changes the type of a column in the problem. p. 667

`problem.chgdeltatype` *(Deprecated)* Changes the type of the delta assigned to a nonlinear variable p. 693

`problem.chgdf` *(Deprecated)* Set or change a distribution factor p. 694

`problem.chgglblimit` *(Deprecated)* Changes semi-continuous or semi-integer lower bounds, or upper limits on partial integers. p. 695

`problem.chgmcoef` *(Deprecated)* Change multiple coefficients in the problem. p. 696

`problem.chgmqobj` *(Deprecated)* Change multiple quadratic coefficients in the objective function. p. 698

`problem.chgnlcoef` *(Deprecated)* Add or change a single matrix coefficient using a parsed or unparsed formula p. 699

`problem.chgobj` *(Deprecated)* Change the objective function coefficients. p. 701

`problem.chgobjn` *(Deprecated)* Modifies one or more coefficients of an objective function in a multi-objective problem. p. 697

`problem.chgobjsense` *(Deprecated)* Changes the problem's objective function objsense to minimize or maximize. p. 702

`problem.chgqobj` *(Deprecated)* Change a single quadratic coefficient in the objective function corresponding to the variable pair `(objqcol1,objqcol2)` of the Hessian matrix. p. 703

`problem.chgqrowcoeff` *(Deprecated)* Changes a single quadratic coefficient in a row. p. 704

`problem.chgrhs` *(Deprecated)* Changes right–hand side values of the problem. p. 705

`problem.chgrhsrange` *(Deprecated)* Change the range for one or more rows of the problem. p. 706

`problem.chgrowstatus` *(Deprecated)* Change the status setting of a constraint p. 707

`problem.chgrowtype` *(Deprecated)* Changes the type of a row in the problem. p. 708

`problem.chgrowwt` *(Deprecated)* Set or change the initial penalty error weight for a row p. 709

`problem.chgtolset` *(Deprecated)* Add or change a set of convergence tolerances used for SLP variables p. 710

`problem.chgvar` *(Deprecated)* Define a column as an SLP variable or change the characteristics and values of an existing SLP variable p. 711

`problem.construct` *(Deprecated)* Create the full augmented SLP matrix and data structures, ready for optimization p. 712

`problem.copycontrols` *(Deprecated)* Copies controls defined for one problem to another. p. 713

`problem.crossoverlpsol` *(Deprecated)* Provides a basic optimal solution for a given solution of an LP problem. p. 714

`problem.delcoefs` *(Deprecated)* Delete coefficients from the current problem p. 715

`problem.delcpcuts` *(Deprecated)* Deletes cuts from the cut pool. p. 716

`problem.delcuts` *(Deprecated)* Deletes cuts from the matrix at the current node. p. 717

`problem.delgencons` *(Deprecated)* Delete general constraints from a problem. p. 718

`problem.delindicators` *(Deprecated)* Delete indicator constraints. p. 719

`problem.delobj` *(Deprecated)* Removes an objective function from a multi-objective problem. p. 721

`problem.delpwlcons` *(Deprecated)* Delete piecewise linear constraints from a problem. p. 720

`problem.delqmatrix` *(Deprecated)* Deletes the quadratic part of a row or of the objective function. p. 722

`problem.deltolsets` *(Deprecated)* Delete tolerance sets from the current problem p. 723

`problem.delvars` *(Deprecated)* Convert SLP variables to normal columns. p. 724

`problem.dumpcontrols` *(Deprecated)* Displays the list of controls and their current value for those controls that have been set to a non default value. p. 725

`problem.estimaterowdualranges` *(Deprecated)* Performs a dual side range sensitivity analysis, i.e., calculates estimates for the possible ranges for dual values. p. 726

`problem.evaluatecoef` *(Deprecated)* Evaluate a coefficient using the current values of the variables p. 727

`problem.evaluateformula` *(Deprecated)* Evaluate a formula using the current values of the variables p. 728

`problem.fixmipentities` *(Deprecated)* Fixes all the MIP entities to the values of the last found MIP solution. p. 729

`problem.fixpenalties` *(Deprecated)* Fixes the values of the error vectors p. 730

`problem.ftran` *(Deprecated)* Pre-multiplies a (column) vector provided by the user by the inverse of the current matrix. p. 731

`problem.getattribinfo` *(Deprecated)* Accesses the id number and the type information of an attribute given its name. p. 732

`problem.getbasis` *(Deprecated)* Returns the current basis into the user's data arrays. p. 733

`problem.getbasisval` *(Deprecated)* Returns the current basis status for a specific col or row. p. 734

`problem.getccoef` *(Deprecated)* Retrieve a single nonlinear matrix coefficient as a formula in a string. p. 735

`problem.getcoef` *(Deprecated)* Returns a single coefficient in the constraint matrix. p. 736

`problem.getcoefformula` *(Deprecated)* Retrieve a single nonlinear matrix coefficient as a formula split into tokens p. 737

`problem.getcoefs` *(Deprecated)* Retrieve the list of positions of the nonlinear coefficients in the problem p. 738

`problem.getcolinfo` *(Deprecated)* Get current column information. p. 739

`problem.getcols` *(Deprecated)* Returns the nonzeros in the constraint matrix for the columns in a given range. p. 740

`problem.getcoltype` *(Deprecated)* Returns the column types for the columns in a given range. p. 741

`problem.getcontrolinfo` *(Deprecated)* Accesses the id number and the type information of a control given its name. p. 742

`problem.getcpcutlist` *(Deprecated)* Returns a list of cut indices from the cut pool. p. 743

`problem.getcpcuts` *(Deprecated)* Returns cuts from the cut pool. p. 744

`problem.getcutlist` *(Deprecated)* Retrieves a list of cuts for the cuts active at the current node. p. 745

`problem.getcutmap` *(Deprecated)* Returns in which rows a list of cutind are currently loaded into the Optimizer. p. 746

`problem.getcutslack` *(Deprecated)* Used to calculate the slack value of a cut with respect to the current LP relaxation solution. p. 747

`problem.getdf` *(Deprecated)* Get a distribution factor p. 749

`problem.getdirs` *(Deprecated)* Returns the directives that have been loaded into a problem. p. 748

`problem.getDual` *(Deprecated)* p. 750

`problem.getdualray` *(Deprecated)* Retrieves a dual ray (dual unbounded direction) for the current problem, if the problem is found to be infeasible. p. 751

`problem.getgencons` *(Deprecated)* Returns the general constraints `y = f(x1, ..., xn, c1, ..., cm)` in a given range. p. 752

`problem.getiisdata` *(Deprecated)* Returns information for an Irreducible Infeasible Set: size, variables (row and column vectors) and conflicting sides of the variables, duals and reduced costs. p. 754

`problem.getIndexFromName` *(Deprecated)* Returns the index for a specified row or column name. p. 756

`problem.getindicators` *(Deprecated)* Returns the indicator constraint condition (indicator variable and complement flag) associated to the rows in a given range. p. 757

`problem.getinfeas` *(Deprecated)* Returns a list of infeasible primal and dual variables. p. 758

`problem.getlastbarsol` *(Deprecated)* Obtains the last barrier solution values following optimization that used the barrier solver. p. 759

`problem.getlasterror` *(Deprecated)* Returns the error message corresponding to the last error triggered by a library function. p. 760

`problem.getlb` *(Deprecated)* Returns the lower bounds on the columns in a given range. p. 761

`problem.getlpsol` *(Deprecated)* Obtains the LP solution values following optimization. p. 762

`problem.getlpsolval` *(Deprecated)* Used to obtain a single LP solution value following optimization. p. 763

`problem.getmessagestatus` *(Deprecated)* Returns the current suppression status of a message: nonzero if the message is not suppressed; `0` otherwise. p. 764

`problem.getmipentities` *(Deprecated)* Retrieves MIP entity information about a problem. p. 753

`problem.getmipsol` *(Deprecated)* Obtains the solution values of the last MIP solution that was found. p. 765

`problem.getmipsolval` *(Deprecated)* Used to obtain a single solution value of the last MIP solution that was found. p. 766

`problem.getmqobj` *(Deprecated)* Returns the nonzeros in the quadratic objective coefficients' matrix for the columns in a given range. p. 767

`problem.getnamelist` *(Deprecated)* Returns the names for the rows, columns, sets, piecewise linear constraints, general constraints or objectives in a given range. p. 769

`problem.getobj` *(Deprecated)* Returns the objective function coefficients for the columns in a given range. p. 770

`problem.getobjn` *(Deprecated)* Returns the coefficients of a given objective function for the columns in a given range. p. 768

`problem.getObjVal` *(Deprecated)* Returns the objective value of the solution found by the Optimizer. p. 771

`problem.getpivotorder` *(Deprecated)* Returns the pivot order of the basic variables. p. 772

`problem.getpivots` *(Deprecated)* Returns a list of potential leaving variables if a specified variable enters the basis. p. 773

`problem.getpresolvebasis` *(Deprecated)* Returns the current basis from memory into the user's data areas. p. 774

`problem.getpresolvemap` *(Deprecated)* Returns the mapping of the row and column numbers from the presolve problem back to the original problem. p. 775

`problem.getpresolvesol` *(Deprecated)* Returns the solution for the presolved problem from memory. p. 776

`problem.getprimalray` *(Deprecated)* Retrieves a primal ray (primal unbounded direction) for the current problem, if the problem is found to be unbounded. p. 777

`problem.getProbStatus` *(Deprecated)* Returns the problem status before or after a call to `problem.optimize`. p. 778

`problem.getProbStatusString` *(Deprecated)* Returns the string corresponding to the problem status before or after a call to `problem.optimize`. p. 779

`problem.getpwlcons` *(Deprecated)* Returns the piecewise linear constraints `y = f(x)` in a given range. p. 780

`problem.getqobj` *(Deprecated)* Returns a single quadratic objective function coefficient corresponding to the variable pair `(objqcol1, objqcol2)` of the Hessian matrix. p. 781

`problem.getqrowcoeff` *(Deprecated)* Returns a single quadratic constraint coefficient corresponding to the variable pair (`rowqcol1`, `rowqcol2`) of the Hessian of a given constraint. p. 782

`problem.getqrowqmatrix` *(Deprecated)* Returns the nonzeros in a quadratic constraint coefficients matrix for the columns in a given range. p. 783

`problem.getqrowqmatrixtriplets` *(Deprecated)* Returns the nonzeros in a quadratic constraint coefficients matrix as triplets (index pairs with coefficients). p. 784

`problem.getqrows` *(Deprecated)* Returns a list of row objects that have quadratic coefficients. p. 785

`problem.getRCost` *(Deprecated)* p. 786

`problem.getrhs` *(Deprecated)* Returns the right hand side elements for the rows in a given range. p. 787

`problem.getrhsrange` *(Deprecated)* Returns the right hand side range values for the rows in a given range. p. 788

`problem.getrowinfo` *(Deprecated)* Get current row information. p. 789

`problem.getrows` *(Deprecated)* Returns the nonzeros in the constraint matrix for the rows in a given range. p. 790

`problem.getrowstatus` *(Deprecated)* Retrieve the status setting of a constraint p. 791

`problem.getrowtype` *(Deprecated)* Returns the row types for the rows in a given range. p. 792

`problem.getrowwt` *(Deprecated)* Get the initial penalty error weight for a row p. 793

`problem.getscaledinfeas` *(Deprecated)* Returns a list of scaled infeasible primal and dual variables for the original problem. p. 794

`problem.getSlack` *(Deprecated)* p. 795

`problem.getslpsol` *(Deprecated)* Obtain the solution values for the most recent SLP iteration p. 796

`problem.gettolset` *(Deprecated)* Retrieve the values of a set of convergence tolerances for an SLP problem p. 797

`problem.getub` *(Deprecated)* Returns the upper bounds on the columns in a given range. p. 798

`problem.getunbvec` *(Deprecated)* Returns the index vector which causes the primal simplex or dual simplex algorithm to determine that a problem is primal or dual unbounded respectively. p. 799

`problem.getvar` *(Deprecated)* Retrieve information about an SLP variable p. 800

`problem.hasdualray` *(Deprecated)* Returns true if a dual ray (dual unbounded direction) exists for the current problem, if the problem is found to be infeasible. p. 801

`problem.hasprimalray` *(Deprecated)* Returns true if a primal ray (primal unbounded direction) exists for the current problem, if the problem is found to be unbounded. p. 802

`problem.iisall` *(Deprecated)* Performs an automated search for independent Irreducible Infeasible Sets (IIS) in an infeasible problem. p. 803

`problem.iisclear` *(Deprecated)* Resets the search for Irreducible Infeasible Sets (IIS). p. 804

`problem.iisfirst` *(Deprecated)* Initiates a search for an Irreducible Infeasible Set (IIS) in an infeasible problem. p. 805

`problem.iisisolations` *(Deprecated)* Performs the isolation identification procedure for an Irreducible Infeasible Set (IIS). p. 806

`problem.iisnext` *(Deprecated)* Continues the search for further Irreducible Infeasible Sets (IIS). p. 807

`problem.iisstatus` *(Deprecated)* Returns statistics on the Irreducible Infeasible Sets (IIS) found so far. p. 808

`problem.iiswrite` *(Deprecated)* Writes an LP/MPS/CSV file containing a given Irreducible Infeasible Set (IIS). p. 809

`problem.loadbasis` *(Deprecated)* Loads a basis as specified by the user. p. 810

`problem.loadbranchdirs` *(Deprecated)* Loads directives into the current problem to specify which MIP entities the Optimizer should continue to branch on when a node solution is integer feasible. p. 811

`problem.loadcoefs` *(Deprecated)* Load non-linear coefficients into the SLP problem p. 812

`problem.loadcuts` *(Deprecated)* Loads cuts from the cut pool into the matrix. p. 814

`problem.loaddelayedrows` *(Deprecated)* Specifies that a set of rows in the problem will be treated as delayed rows during a tree search. p. 815

`problem.loaddfs` *(Deprecated)* Load a set of distribution factors p. 816

`problem.loaddirs` *(Deprecated)* Loads directives into the problem. p. 817

`problem.loadlpsol` *(Deprecated)* Loads an LP solution for the problem into the Optimizer. p. 818

`problem.loadmipsol` *(Deprecated)* Loads a MIP solution for the problem into the Optimizer. p. 819

`problem.loadmodelcuts` *(Deprecated)* Specifies that a set of rows in the problem will be treated as model cuts. p. 820

`problem.loadpresolvebasis` *(Deprecated)* Loads a presolved basis from the user's areas. p. 821

`problem.loadpresolvedirs` *(Deprecated)* Loads directives into the presolved matrix. p. 822

`problem.loadproblem` *(Deprecated)* Load an optimization problem, possibly with quadratic objective and/or constraints, and integer variables. p. 823

`problem.loadsecurevecs` *(Deprecated)* Allows the user to mark rows and columns in order to prevent the presolve removing these rows and columns from the problem. p. 825

`problem.loadtolsets` *(Deprecated)* Load sets of standard tolerance values into an SLP problem p. 826

`problem.loadvars` *(Deprecated)* Load SLP variables defined as matrix columns into an SLP problem p. 827

`problem.lpoptimize` *(Deprecated)* Begins a search for the optimal continuous (LP) solution. p. 828

`problem.mipoptimize` *(Deprecated)* Begins a tree search for the optimal MIP solution. p. 829

`problem.msaddcustompreset` *(Deprecated)* A combined version of msaddjob and msaddpreset. p. 830

`problem.msaddjob` *(Deprecated)* Adds a multistart job to the multistart pool p. 831

`problem.msaddpreset` *(Deprecated)* Loads a preset of jobs into the multistart job pool. p. 832

`problem.msclear` *(Deprecated)* Removes all scheduled jobs from the multistart job pool p. 833

`problem.name` *(Deprecated)* Returns the name of the problem as a Python string. p. 834

`problem.nlpchgformula` *(Deprecated)* Add or replace a single matrix formula using a parsed or unparsed formula p. 835

`problem.nlpchgformulastr` *(Deprecated)* Add or replace a single matrix formula using a character string for the formula. p. 836

`problem.nlpgetformula` *(Deprecated)* Retrieve a single matrix formula split into tokens p. 837

`problem.nlpgetformulastr` *(Deprecated)* Retrieve a single matrix formula in a character string p. 839

`problem.nlpoptimize` *(Deprecated)* Solves an SLP problem p. 840

`problem.nlpsetinitval` *(Deprecated)* Set the initial value of a nonlinear variable p. 841

`problem.objsa` *(Deprecated)* Returns upper and lower sensitivity ranges for specified objective function coefficients. p. 842

`problem.postsolve` *(Deprecated)* Postsolve the current problem when it is in a presolved state. p. 843

`problem.presolve` *(Deprecated)* Perform a nonlinear presolve on the problem p. 844

`problem.presolverow` *(Deprecated)* Presolves a row formulated in terms of the original variables such that it can be added to a presolved problem. p. 845

`problem.printevalinfo` *(Deprecated)* Print a summary of any evaluation errors that may have occurred during solving a problem p. 847

`problem.printmemory` *(Deprecated)* Print the dimensions and memory allocations for a problem p. 846

`problem.read` *(Deprecated)* Read an optimization problem into a Python problem object created prior to the call. p. 848

`problem.readbasis` *(Deprecated)* Instructs the Optimizer to read in a previously saved basis from a file. p. 849

`problem.readbinsol` *(Deprecated)* Reads a solution from a binary solution file. p. 850

`problem.readdirs` *(Deprecated)* Reads a directives file to help direct the tree search. p. 851

`problem.readslxsol` *(Deprecated)* Reads an ASCII solution file (`.slx`) created by the `problem.writeSlxSol` function. p. 852

`problem.reinitialize` *(Deprecated)* Reset the SLP problem to match a just augmented system p. 853

`problem.removecbafterobjective` *(Deprecated)* Removes a user after objective callback function previously added by addcbafterobjective. p. 876

`problem.removecbbariteration` *(Deprecated)* Removes a barrier iteration callback function previously added by addcbbariteration. p. 854

`problem.removecbbarlog` *(Deprecated)* Removes a newton barrier log callback function previously added by addcbbarlog. p. 855

`problem.removecbbeforeobjective` *(Deprecated)* Removes a user before objective callback function previously added by addcbbeforeobjective. p. 875

`problem.removecbchecktime` *(Deprecated)* Removes a callback function previously added by problem.addcbchecktime. p. 856

`problem.removecbchgbranchobject` *(Deprecated)* Removes a callback function previously added by addcbchgbranchobject. p. 857

`problem.removecbcutlog` *(Deprecated)* Removes a cut log callback function previously added by addcbcutlog. p. 858

`problem.removecbcutround` *(Deprecated)* Removes a cut round callback function previously added by addcbcutround. p. 859

`problem.removecbdestroymt` *(Deprecated)* Removes a child thread destruction callback function previously added by addcbdestroymt. p. 860

`problem.removecbgapnotify` *(Deprecated)* Removes a callback function previously added by problem.addcbgapnotify. p. 861

`problem.removecbinfnode` *(Deprecated)* Removes a user infeasible node callback function previously added by addcbinfnode. p. 863

`problem.removecbintsol` *(Deprecated)* Removes an integer solution callback function previously added by addcbintsol. p. 864

`problem.removecblplog` *(Deprecated)* Removes a simplex log callback function previously added by addcblplog. p. 865

`problem.removecbmessage` *(Deprecated)* Removes a message callback function previously added by addcbmessage. p. 866

`problem.removecbmiplog` *(Deprecated)* Removes a MIP log callback function previously added by addcbmiplog. p. 862

`problem.removecbmipthread` *(Deprecated)* Removes a callback function previously added by addcbmipthread. p. 867

`problem.removecbnewnode` *(Deprecated)* Removes a new-node callback function previously added by addcbnewnode. p. 868

`problem.removecbnodecutoff` *(Deprecated)* Removes a node-cutoff callback function previously added by addcbnodecutoff. p. 869

`problem.removecbnodelpsolved` *(Deprecated)* Removes a node lp solved callback function previously added by addcbnodelpsolved. p. 870

`problem.removecboptnode` *(Deprecated)* Removes a node-optimal callback function previously added by addcboptnode. p. 871

`problem.removecbpreintsol` *(Deprecated)* Removes a pre-integer solution callback function previously added by addcbpreintsol. p. 872

`problem.removecbprenode` *(Deprecated)* Removes a preprocess node callback function previously added by addcbprenode. p. 873

`problem.removecbpresolve` *(Deprecated)* Removes a presolve callback function previously added by addcbpresolve. p. 877

`problem.removecbusersolnotify` *(Deprecated)* Removes a user solution notification callback previously added by problem.addcbusersolnotify. p. 874

`problem.repairinfeas` *(Deprecated)* Provides a simplified interface for problem.repairweightedinfeas. p. 878

`problem.repairweightedinfeas` *(Deprecated)* Attempts to identify a 'solution' that violates a selected set of constraints and bounds minimally, while satisfying all other constraints and bounds. p. 880

`problem.repairweightedinfeasbounds` *(Deprecated)* An extended version of problem.repairweightedinfeas that allows for bounding the level of relaxation allowed. p. 882

`problem.rhssa` *(Deprecated)* Returns upper and lower sensitivity ranges for specified right hand side (RHS) function coefficients. p. 884

`problem.scaling` *(Deprecated)* Analyze the current matrix for largest/smallest coefficients and ratios p. 885

`problem.setcbcascadeend` *(Deprecated)* Set a user callback to be called at the end of the cascading process, after the last variable has been cascaded p. 669

`problem.setcbcascadestart` *(Deprecated)* Set a user callback to be called at the start of the cascading process, before any variables have been cascaded p. 670

`problem.setcbcascadevar` *(Deprecated)* Set a user callback to be called after each column has been cascaded p. 671

`problem.setcbcascadevarfail` *(Deprecated)* Set a user callback to be called after cascading a column was not successful p. 672

`problem.setcbcoefevalerror` *(Deprecated)* Set a user callback to be called when an evaluation of a coefficient fails during the solve p. 673

`problem.setcbconstruct` *(Deprecated)* Set a user callback to be called during the Xpress SLP augmentation process p. 674

`problem.setcbdestroy` *(Deprecated)* Set a user callback to be called when an SLP problem is about to be destroyed p. 675

`problem.setcbdrcol` *(Deprecated)* Set a user callback used to override the update of variables with small determining column p. 676

`problem.setcbintsol` *(Deprecated)* Set a user callback to be called during MISLP when an integer solution is obtained p. 677

`problem.setcbiterend` *(Deprecated)* Set a user callback to be called at the end of each SLP iteration p. 678

`problem.setcbiterstart` *(Deprecated)* Set a user callback to be called at the start of each SLP iteration p. 679

`problem.setcbitervar` *(Deprecated)* Set a user callback to be called after each column has been tested for convergence p. 680

`problem.setcbmessage` *(Deprecated)* Set a user callback to be called whenever Xpress Nonlinear outputs a line of text p. 681

`problem.setcbmsjobend` *(Deprecated)* Set a user callback to be called every time a new multistart job finishes. p. 682

`problem.setcbmsjobstart` *(Deprecated)* Set a user callback to be called every time a new multistart job is created, and the pre-loaded settings are applied p. 683

`problem.setcbmswinner` *(Deprecated)* Set a user callback to be called every time a new multistart job is created, and the pre-loaded settings are applied p. 684

`problem.setcboptnode` *(Deprecated)* Set a user callback to be called during MISLP when an optimal SLP solution is obtained at a node p. 685

`problem.setcbprenode` *(Deprecated)* Set a user callback to be called during MISLP after the set-up of the SLP problem to be solved at a node, but before SLP optimization p. 686

`problem.setcbpreupdatelinearization` *(Deprecated)* Set a user callback to be called before the linearization is updated p. 687

`problem.setcbslpend` *(Deprecated)* Set a user callback to be called at the end of the SLP optimization p. 688

`problem.setcbslpnode` *(Deprecated)* Set a user callback to be called during MISLP after the SLP optimization at each node. p. 689

`problem.setcbslpstart` *(Deprecated)* Set a user callback to be called at the start of the SLP optimization p. 690

`problem.setcurrentiv` *(Deprecated)* Transfer the current solution to initial values p. 886

`problem.setdefaultcontrol` *(Deprecated)* Sets one control to its default value. p. 887

`problem.setdefaults` *(Deprecated)* Sets all controls to their default values. p. 888

`problem.setindicators` *(Deprecated)* Specifies that a set of rows in the problem will be treated as indicator constraints during a tree search. p. 889

`problem.setlogfile` *(Deprecated)* This directs all Optimizer output to a log file. p. 890

`problem.setmessagestatus` *(Deprecated)* Manages suppression of messages. p. 891

`problem.setprobname` *(Deprecated)* Sets the current default problem name. p. 892

`problem.slpchgcoef` *(Deprecated)* Add or change a single matrix coefficient using a parsed or unparsed formula p. 700

`problem.slpchgcoefstr` *(Deprecated)* Add or change a single matrix coefficient using a string for the formula p. 691

`problem.slpgetcoefformula` *(Deprecated)* Retrieve a single matrix formula split into tokens p. 838

`problem.slpgetcoefstr` *(Deprecated)* Retrieve a single matrix coefficient as a formula in a character string p. 893

`problem.storecuts` *(Deprecated)* Stores cuts into the cut pool, but does not apply them to the current node. p. 894

`problem.strongbranch` *(Deprecated)* Performs strong branching iterations on all specified bound changes. p. 895

`problem.strongbranchcb` *(Deprecated)* Performs strong branching iterations on all specified bound changes. p. 896

`problem.tuneprobsetfile` *(Deprecated)* Begins a tuner session for a set of problems. p. 897

`problem.tunerreadmethod` *(Deprecated)* Load a user defined tuner method from the given file. p. 898

`problem.tunerwritemethod` *(Deprecated)* Writes the current tuner method to a given file or prints it to the console. p. 899

`problem.unconstruct` *(Deprecated)* Reset the SLP problem and removes the augmentation structures p. 900

`problem.updatelinearization` *(Deprecated)* Updates the current linearization p. 901

`problem.validate` *(Deprecated)* Validate the feasibility of constraints in a converged solution p. 902

`problem.validatekkt` *(Deprecated)* Validates the first order optimality conditions also known as the Karush-Kuhn-Tucker (KKT) conditions versus the currect solution p. 903

`problem.validaterow` *(Deprecated)* Prints an extensive analysis on a given constraint of the SLP problem p. 904

`problem.validatevector` *(Deprecated)* Validate the feasibility of constraints for a given solution p. 905

`problem.write` *(Deprecated)* Writes the current problem to an MPS or LP file. p. 906

`problem.writebasis` *(Deprecated)* Writes the current basis to a file for later input into the Optimizer. p. 907

`problem.writebinsol` *(Deprecated)* Writes the current MIP or LP solution to a binary solution file for later input into the Optimizer. p. 908

`problem.writedirs` *(Deprecated)* Writes the tree search directives from the current problem to a directives file. p. 909

`problem.writeprtsol` *(Deprecated)* Writes the current solution to a fixed format ASCII file, *problem_name*`.prt`. p. 910

`problem.writeslxsol` *(Deprecated)* Creates an ASCII solution file (.slx) using a similar format to MPS files. p. 911

`problem.writesol` *(Deprecated)* Writes the current solution to a CSV format ASCII file, *problem_name*`.asc`(and `.hdr`). p. 912

`xpress.addcbmsghandler` *(Deprecated)* Declares an output callback function in the global environment, called every time a line of message text is output by any object in the library. This callback function will be called in addition to any output callbacks already added by `xpress.addcbmsghandler`. p. 924

`xpress.featurequery` *(Deprecated)* Returns `True` if the provided feature is available in the current license used by the optimizer, `False` otherwise. p. 925

`xpress.getbanner` *(Deprecated)* Returns the banner and copyright message. p. 926

`xpress.getcheckedmode` *(Deprecated)* Returns whether checking and validation of all Optimizer function calls is enabled for the current process. Checking and validation is enabled by default but can be disabled by `xpress.setCheckedMode`. p. 928

`xpress.getcomputeallowed` *(Deprecated)* Queries whether the current application is allowed to use the Insight Compute interface. p. 927

`xpress.getdaysleft` *(Deprecated)* Returns the number of days left until an evaluation license expires. p. 929

`xpress.getlasterror` *(Deprecated)* Returns the last error encountered during a call to the Xpress global environment. p. 930

`xpress.getlicerrmsg` *(Deprecated)* Returns the error message string describing the last licensing error, if any occurred. p. 931

`xpress.getversion` *(Deprecated)* Returns the full Optimizer version number as a string of the form 15.10.03, where 15 is the major release, 10 is the minor release, and 03 is the build number. p. 932

`xpress.getversionnumbers` *(Deprecated)* Returns the Optimizer version numbers split into major, minor, and build number. p. 933

`xpress.is_equal` *(Deprecated)* p. 934

`xpress.removecbmsghandler` *(Deprecated)* Removes a message callback function previously added by `xpress.addcbmsghandler`. The specified callback function will no longer be called after it has been removed. p. 935

`xpress.setarchconsistency` *(Deprecated)* Sets whether to force the same execution path on various CPU architecture extensions, in particular (pre-)AVX and AVX2. p. 936

`xpress.setcheckedmode` *(Deprecated)* Disable/enable some of the checking and validation of function calls and function call parameters for calls to the Xpress Optimizer API. This checking is relatively lightweight but disabling it can improve performance in cases where non-intensive Xpress Optimizer functions are called repeatedly in a short space of time. Please note: after disabling checking and validation for function calls, invalid usage of Xpress Optimizer functions may not be detected and may cause the Xpress Optimizer process to behave unexpectedly or crash. It is not recommended to disable function call checking and validation during application development. p. 938

`xpress.setcomputeallowed` *(Deprecated)* Set whether the current application is allowed to use the Insight Compute interface. p. 937

`xpress.setdefaultcontrol` *(Deprecated)* p. 939

`xpress.setdefaults` *(Deprecated)* p. 940

`xpress.vars` *(Deprecated)* Creates a dictionary or array of variables. p. 941

problem.addcbbariteration

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addBarIterationCallback` *instead.*
Declares a barrier iteration callback function, called after each iteration during the interior point algorithm, with the ability to access the current barrier solution/slack/duals or reduced cost values, and to ask barrier to stop. This callback function will be called in addition to any callbacks already added by addcbbariteration.

Synopsis

```
problem.addcbbariteration(callback, data, priority)
barrier_action = callback(my_prob, my_object)
```

Arguments

`callback`	The callback function itself. This takes two arguments, `my_prob` and `my_object`, and returns an integer return value. This function is called at every barrier iteration.
`my_prob`	The problem passed to the callback function, `fubi`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbbariteration`.
`barrier_action`	Defines a return value controlling barrier: `<0` continue with the next iteration; `=0` let barrier decide (use default stopping criteria) `1` barrier stops with status not defined; `2` barrier stops with optimal status; `3` barrier stops with dual infeasible status; `4` barrier stops wih primal infeasible status;
`data`	A user-defined object to be passed to the callback function, `f_bariteration`.
`priority`	An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

This simple example demonstrates how the solution might be retrieved for each barrier iteration.

```
# Barrier iteration callback
def BarrierIterCallback(my_prob, my_object):

    current_iteration = my_prob.attributes.bariter

    PrimalObj = my_prob.attributes.barprimalobj
    DualObj   = my_prob.attributes.bardualobj

    Gap = DualObj - PrimalObj

    PrimalInf        = my_prob.attributes.barprimalinf
    DualInf          = my_prob.attributes.bardualinf
    ComplementaryGap = my_prob.attributes.barcgap

    # decide if stop or continue
    barrier_action = 0
    if(current_iteration >= 50 or
        Gap <= 0.1 * max(abs(PrimalObj), abs(DualObj))):
        barrier_action = 2

    return barrier_action

# To set callback:
prob.addcbbariteration(BarrierIterCallback, myobj, 0)
```

Further information

1. Only the functions for retrieving the solution and the attribute/control value retrieving and setting routines are expected to be called from the callback.
2. Please note that these values refer to the scaled and presolved problem used by barrier, and may differ from the ones calculated from the postsolved solution returned by `problem.getCallbackSolution`.

Related topics

`problem.removeBarIterationCallback`, `problem.addBarIterationCallback`.

problem.addcbbarlog

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addBarlogCallback` *instead.*

Declares a barrier log callback function, called at each iteration during the interior point algorithm. This callback function will be called in addition to any barrier log callbacks already added by addcbbarlog.

Synopsis

```
problem.addcbbarlog(callback, data, priority)
ret = callback(my_prob, my_object)
```

Arguments

`callback`	The callback function itself. This takes two arguments, `my_prob` and `my_object`, and has an integer return value. If the value returned by `callback` is nonzero, the solution process will be interrupted. This function is called at every barrier iteration.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbbarlog`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple barrier log callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

This simple example prints a line to the screen for each iteration of the algorithm.

```
prob.addcbbarlog(barLog, None, 0)
prob.lpoptimize('b')
```

The callback function might resemble:

```
def barLog(prob, object):
    print('Next barrier iteration')
```

Further information

If the callback function returns a nonzero value, the Optimizer run will be interrupted.

Related topics

`problem.removeBarlogCallback`, `problem.addMiplogCallback`, `problem.addLplogCallback`, `problem.addMessageCallback`, `problem.addBarlogCallback`.

problem.addcbchecktime

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addCheckTimeCallback` *instead.*

Declares a callback function which is called every time the Optimizer checks if the time limit has been reached. This callback function will be called in addition to any callbacks already added by `addcbchecktime`.

Synopsis

```
problem.addcbchecktime(callback, data, priority)
ret = callback(my_prob, my_object)
```

Arguments

`callback`	The callback function which takes two arguments, `my_prob` and `my_data`, and has an integer return value. If the value returned by `callback` is nonzero, the solution process will be interrupted. This function is called every time the Optimizer checks against the time limit.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbchecktime`.
`data`	The user-defined object passed as `data` when setting up the callback with `addcbchecktime`.
`priority`	An integer that determines the order in which multiple checktime callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Further information

If the callback function returns a nonzero value the solution process will be interrupted.

Related topics

`problem.removeCheckTimeCallback`, `problem.addCheckTimeCallback`.

problem.addcbchgbranchobject

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addChangeBranchObjectCallback` instead.
Declares a callback function that will be called every time the Optimizer has selected a MIP entity for branching. Allows the user to inspect and override the Optimizer's branching choice. This callback function will be called in addition to any callbacks already added by `problem.addChangeBranchObjectCallback`.

Synopsis

```
problem.addcbchgbranchobject(callback, data, priority)
newobject = callback(my_prob, my_object, obranch)
```

Arguments

`callback`	The callback function, which takes three arguments: `my_prob`, `my_object`, and `obranch`. This function is called every time the Optimizer has selected a candidate entity for branching.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user defined object passed as `data` when setting up the callback with `addcbchgbranchobject`.
`obranch`	The candidate branching object selected by the Optimizer.
`newobject`	New branching object to replace the Optimizer's selection. Can be `None`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Further information

1. The branching object given by the Optimizer provides a linear description of how the Optimizer intends to branch on the selected candidate. This will often be one of standard MIP entities of the current problem, but can also be e.g. a split disjunction or a structural branch, if those features are turned on.
2. The functions `branchobj.getBranches`, `branchobj.getBounds` and `branchobj.getRows` can be used to inspect the given branching object.
3. Refer to the `branchobj` class to learn how to create a new branching object to replace the Optimizer's selection. Note that the new branching object should be created with a priority value no higher than the current object to guarantee it will be used for branching.

Related topics

`problem.removeChangeBranchObjectCallback`, `problem.addChangeBranchObjectCallback`.

problem.addcbcutlog

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addCutlogCallback` *instead.*

Declares a cut log callback function, called each time the cut log is printed. This callback function will be called in addition to any callbacks already added by `problem.addCutlogCallback`.

Synopsis

```
problem.addcbcutlog(callback, data, priority)
ret = callback(my_prob, my_object)
```

Arguments

`callback`	The callback function which takes two arguments, `my_prob` and `my_object`, and has an integer return value.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbcutlog`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple cut log callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Further information

The callback `callback` should return a nonzero value to stop cutting on the current node.

Related topics

`problem.removeCutlogCallback`, `problem.addCutlogCallback`.

problem.addcbcutround

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addCutRoundCallback` *instead.*

Declares a callback function that is called when the Optimizer could separate cutting planes during the branch and bound search. The callback allows a user to modify when the Optimizer should separate cutting planes. It also allows a user to tighten the local node problem. This callback function will be called in addition to any callbacks already added by problem.addcbcutmgr.

Synopsis

```
problem.addcbcutround(callback, data, priority)
action = callback(my_prob, my_object, ifxpresscuts)
```

Arguments

`callback` The callback function which takes three arguments, `cbprob`, `cbdata` and `ifxpresscuts` and has an integer return value. This function is called before each potential round of cut separation in the branch and bound search.

`my_prob` The problem passed to the callback function, `callback`.

`my_object` The user-defined object passed as object when setting up the callback with `addcbcutround`.

`ifxpresscuts` An integer set to 1 if the Optimizer will apply a round of cuts after this callback. 0 otherwise.

`action` An integer return value that specifies the action the Optimizer should take:

- `-1` Continue unchanged. The default action.
- `0` No further rounds of cuts should be applied on this node.
- `1` The Optimizer should apply one more round of cutting, regardless of the value of `ifxpresscuts`
- `2` The Optimizer should process any changes applied during this callback and fire the callback again, but skip any Optimizer cutting.

`data` A user-defined object to be passed to the callback function, `callback`.

`priority` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Further information

1. The current node problem can be modified in this callback, including adding cuts and changing bounds. If the problem is changed, the node LP relaxation will be resolved and the callback fired again.
2. It is possible to add new solutions using `problem.addMipSol`. These will be checked immediately after the callback, but adding solutions will not automatically trigger another firing of this callback.
3. If cutting planes are added to the problem or removed from the problem, or bounds tightened, a reoptimization of the node LP relaxation will automatically be triggered. The callback will only be fired again after the reoptimization if a non-zero value is returned in `p_action`.

Related topics

`problem.removeCutRoundCallback`, `problem.addCutRoundCallback`

problem.addcbdestroymt

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addMipThreadDestroyCallback` *instead.*
Declares a callback function that is called every time a MIP thread is destroyed by the parallel MIP code. This callback function will be called in addition to any callbacks already added by `addcbdestroymt`.

Synopsis

```
problem.addcbdestroymt(callback, data, priority)
callback(my_prob, my_object)
```

Arguments

`callback`	The callback function which takes two arguments, `my_prob` and `my_object`, and has no return value.
`my_prob`	The thread problem passed to the callback function.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbdestroymt`.
`data`	A user-defined object to be passed to the callback function.
`priority`	An integer that determines the order in which multiple callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Further information

This callback is useful for freeing up any user data created in the MIP thread callback.

Related topics

`problem.removeMipThreadDestroyCallback`, `problem.addMipThreadCallback`, `problem.addMipThreadDestroyCallback`.

problem.addcbgapnotify

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addGapNotifyCallback` *instead.*

Declares a gap notification callback, to be called when a MIP solve reaches a predefined target, set using the `miprelgapnotify`, `mipabsgapnotify`, `mipabsgapnotifyobj`, and/or `mipabsgapnotifybound` controls.

Synopsis

```
problem.addcbgapnotify(callback, data, priority)
(RelGapNotify, AbsGapNotify, AbsGapNotifyObj, AbsGapNotifyBound) = callback(my_prob,
       my_object)
```

Arguments

`callback` The callback function.

`data` A user-defined object that will be passed into the callback `callback`.

`priority` An integer that determines the order in which multiple gap notification callbacks will be invoked. The callback added with the higher priority will be called before a callback with a lower priority. Set to `0` if not required.

`my_prob` The current problem.

`my_object` The user-defined object passed as `data` when setting up the callback with `addcbgapnotify`.

`RelGapNotify` The value the `miprelgapnotify` control will be set to after this callback. May be modified within the callback in order to set a new notification target. Can be `None`.

`AbsGapNotify` The value the `mipabsgapnotify` control will be set to after this callback. May be modified within the callback in order to set a new notification target. Can be `None`.

`AbsGapNotifyObj` The value the `mipabsgapnotifyobj` control will be set to after this callback. May be modified within the callback in order to set a new notification target. Can be `None`.

`AbsGapNotifyBound` The value the `mipabsgapnotifybound` control will be set to after this callback. May be modified within the callback in order to set a new notification target. Can be `None`.

Example

The following example prints a message when the gap reaches 10% and 1%

```
def gapnotify(prob, object):

    obj = prob.attributes.mipobjval
    bound = prob.attributes.bestbound

    # If no solutions were found, just return a tuple of None's
    if prob.attributes.mipsols == 0:
        return None, None, None, None

    if obj != 0 and bound != 0:
        relgap = abs((obj - bound) / max(abs(obj), abs(bound)))
    else:
        relgap = 0

    newRelGapNotifyTarget = -1

    if relgap <= 0.1:
        print('Gap reached 10%')
        newRelGapNotifyTarget = 0.1

    if relgap <= 0.01:
        print('Gap reached 1%')
        newRelGapNotifyTarget = -1 # Don't call gapnotify again

    # return a quadruple with new values, or
    # None for those that should not be set
    return (newRelGapNotifyTarget, None, None, None)
```

```
prob.controls.miprelgapnotify = 0.1
prob.addcbgapnotify(gapnotify, None, 0)
prob.mipoptimize('')
```

Further information

The target values that caused the callback to be triggered will automatically be reset to prevent the same callback from being fired again.

Related topics

`MIPRELGAPNOTIFY`, `MIPABSGAPNOTIFY`, `MIPABSGAPNOTIFYOBJ`, `MIPABSGAPNOTIFYBOUND`, `problem.removeGapNotifyCallback`, `problem.addGapNotifyCallback`.

problem.addcbmiplog

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addMiplogCallback` *instead.*

Declares a MIP log callback function, called each time the MIP log is printed. This callback function will be called in addition to any callbacks already added by addcbmiplog.

Synopsis

```
problem.addcbmiplog(callback, data, priority)
ret = callback(my_prob, my_object)
```

Arguments

`callback`	The callback function which takes two arguments, `my_prob` and `my_object`, and has an integer return value. If the value returned by `callback` is nonzero, the solution process will be interrupted. This function is called whenever the MIP log is printed as determined by the `MIPLOG` control.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbmiplog`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple MIP log callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following example prints at each node of the tree search the node number and its depth:

```
prob.controls.miplog = 3
prob.addcbmiplog(mipLog, None, 0)
prob.mipoptimize('')
```

The callback function may resemble:

```
def mipLog(prob, object):

    nodedepth = prob.attributes.nodedepth
    node      = prob.attributes.currentnode

    print('Node {0} with depth {1} has been processed'.format
          (node, nodedepth))

    return 0
```

Further information

If the callback function returns a nonzero value, the tree search will be interrupted.

Related topics

`problem.removeMiplogCallback`, `problem.addBarlogCallback`, `problem.addLplogCallback`, `problem.addMessageCallback`, `problem.addMiplogCallback`.

problem.addcbinfnode

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addInfnodeCallback` *instead.*

Declares a user infeasible node callback function, called after the current node has been found to be infeasible during the Branch and Bound search. This callback function will be called in addition to any callbacks already added by addcbinfnode.

Synopsis

```
problem.addcbinfnode(callback, data, priority)
callback(my_prob, my_object)
```

Arguments

`callback`	The callback function which takes two arguments, `my_prob` and `my_object`, and has no return value. This function is called after the current node has been found to be infeasible.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbinfnode`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple user infeasible node callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following notifies the user whenever an infeasible node is found during the tree search:

```
prob.addcbinfnode(nodeInfeasible, None, 0)
prob.mipoptimize("")
```

The callback function may resemble:

```
def nodeInfeasible(prob, object):
    node = prob.attributes.currentnode
    print("Node {0} infeasible".format(node))
```

Related topics

`problem.removeInfnodeCallback`, `problem.addOptnodeCallback`, `problem.addIntsolCallback`, `problem.addNodecutoffCallback`, `problem.addInfnodeCallback`.

problem.addcbintsol

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addIntsolCallback` *instead.*
Declares a user integer solution callback function, called every time an integer solution is found by heuristics or during the Branch and Bound search. This callback function will be called in addition to any callbacks already added by addcbintsol.

Synopsis

```
problem.addcbintsol(callback, data, priority)
callback(my_prob, my_object)
```

Arguments

`callback`	The callback function which takes two arguments, `my_prob` and `my_object`, and has no return value. This function is called if the current node is found to have an integer feasible solution, i.e. every time an integer feasible solution is found.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbintsol`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple integer solution callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following example prints integer solutions as they are discovered in the tree search:

```
prob.addcbintsol(printsol, None, 0)
prob.mipoptimize("")
```

The callback function might resemble:

```
def printsol(my_prob, object):
    objval = my_prob.attributes.lpobjval
    x = my_prob.getSolution()
    print("Integer solution found:", objval, "; values:")
    print(x)
```

Further information

1. This callback is useful if the user wants to retrieve the integer solution when it is found.
2. To retrieve the integer solution, use `problem.getCallbackSolution`, `problem.getCallbackPresolveSolution` or `problem.getSolution`.
3. This callback is called after a new integer solution was found by the Optimizer. Use a callback set by `problem.addPreIntsolCallback` in order to be notified before a new integer solution is accepted by the Optimizer, which allows for the new solution to be rejected.

Related topics

`problem.removeIntsolCallback`, `problem.addPreIntsolCallback`, `problem.addIntsolCallback`.

problem.addcblplog

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addLplogCallback` *instead.* Declares a simplex log callback function which is called after every `LPLOG` iterations of the simplex algorithm. This callback function will be called in addition to any callbacks already added by addcblplog.

Synopsis

```
problem.addcblplog(callback, data, priority)
ret = callback(my_prob, my_object)
```

Arguments

`callback`	The callback function which takes two arguments, `my_prob` and `my_object`, and has an integer return value. This function is called every `LPLOG` simplex iterations including iteration `0` and the final iteration.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcblplog`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple lplog callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following code sets a callback function, `lpLog`, to be called every `10` iterations of the optimization:

```
prob.controls.lplog = 10
prob.addcblplog(lpLog, None, 0)
prob.read("problem", "")
prob.mipoptimize("")
```

The callback function may resemble:

```
def lpLog(my_prob, object):

    iter = my_prob.attributes.simplexiter
    obj  = my_prob.attributes.lpobjval

    print("At iteration {0} objval is {1}".format(iter, obj))
    return 0
```

Further information

If the callback function returns a nonzero value, the solution process will be interrupted.

Related topics

`problem.removeLplogCallback`, `problem.addBarlogCallback`, `problem.addMiplogCallback`, `problem.addMessageCallback`, `problem.addLplogCallback`.

problem.addcbmessage

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addMessageCallback` *instead.*

Declares an output callback function, called every time a text line relating to the given prob is output by the Optimizer. This callback function will be called in addition to any callbacks already added by addcbmessage.

Synopsis

```
problem.addcbmessage(callback, data, priority)
callback(my_prob, my_object, msg, msgtype)
```

Arguments

`callback`	The callback function which takes four arguments: `my_prob`, `my_object`, `msg`, and `msgtype`, and has no return value. Use a `None` value to cancel a callback function.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbmessage`.
`msg`	A string containing the message.
`msgtype`	Indicates the type of output message: `1` information messages; `2` (not used) `3` warning messages; `4` error messages. A negative value indicates that the Optimizer is about to finish and the buffers should be flushed at this time if the output is being redirected to a file.
`data`	A user-defined object to be passed to the callback function.
`priority`	An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following example simply sends all output to the screen (`stdout`):

```
prob.addcbmessage(Message, None, 0)
```

The callback function might resemble:

```
def Message(my_prob, object, msg, msgtype):

    print('{0}: {1}'.format(msgtype, msg))
```

Further information

1. By default, Optimizer messages are also printed to `stdout`. This behaviour is unaffected by user message callbacks. It can be disabled by calling `problem.setOutputEnabled` or `xpress.setOutputEnabled` with argument `False`.
2. This function offers one method of handling the messages which describe any warnings and errors that may occur during execution. Other methods are to check the return values of functions and then get the error code using the `errorcode` attribute, obtain the last error message directly using `problem.getlasterror`, or send messages direct to a log file using `problem.setLogFile`.

Related topics

`problem.removeMessageCallback`, `problem.addBarlogCallback`, `problem.addMiplogCallback`, `problem.addLplogCallback`, `problem.setLogFile`, `problem.setOutputEnabled`, `xpress.setOutputEnabled`, `problem.addMessageCallback`.

problem.addcbmipthread

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addMipThreadCallback` *instead.*
Declares a MIP thread callback function, called every time a MIP worker problem is created by the parallel MIP code. This callback function will be called in addition to any callbacks already added by addcbmipthread.

Synopsis

```
problem.addcbmipthread(callback, data, priority)
callback(my_prob, my_object, thread_prob)
```

Arguments

`callback`	The callback function which takes three arguments, `my_prob`, `my_object` and `thread_prob`, and has no return value.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed to the callback function.
`thread_prob`	The problem for the MIP thread
`data`	A user-defined object to be passed to the callback function.
`priority`	An integer that determines the order in which multiple callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following example clears the message callback for each of the MIP threads:

```
prob.addcbmipthread(mipthread, None, 0)
```

```
def mipthread(my_prob, my_object, mipthread):
    my_prob.removecbmessage(mipthread, None)
```

Further information

This function will be called when a new MIP worker problem is created. Each worker problem receives a unique identifier that can be obtained through the `MIPTHREADID` attribute. Worker problems can be matched with different system threads at different points of a solve, so the system thread that is responsible for executing the callback is not necessarily the same thread used for all subsequent callbacks for the same worker problem. On the other hand, worker problems are always assigned to a single thread at a time and the same nodes are always solved on the same worker problem in repeated runs of a deterministic MIP solve. A worker problem therefore acts as a virtual thread through the node solves.

Related topics

`problem.removeMipThreadCallback`, `problem.addMipThreadDestroyCallback`, `problem.addMipThreadCallback`.

problem.addcbnewnode

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addNewnodeCallback` *instead.*
Declares a callback function that will be called every time a new node is created during the branch and bound search. This callback function will be called in addition to any callbacks already added by addcbnewnode.

Synopsis

```
problem.addcbnewnode(callback, data, priority)
callback(my_prob, my_object, parentnode, newnode, branch)
```

Arguments

`callback`	The callback function, which takes five arguments: `myprob`, `my_object`, `parentnode`, `newnode` and `branch`. This function is called every time a new node is created through branching.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbnewnode`.
`parentnode`	Unique identifier for the parent of the new node.
`newnode`	Unique identifier assigned to the new node.
`branch`	The sequence number of the new node amongst the child nodes of `parentnode`. For regular branches on a MIP entity this will be either `0` or `1`.
`data`	A user-defined object to be passed to the callback function.
`priority`	An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Further information

1. For regular branches on a MIP entity, `branch` will be either zero or one, depending on whether the new node corresponds to branching the MIP entity up or down.
2. When branching on a `branchobject`, `branch` refers to the given branch index of the object.

Related topics

`problem.removeNewnodeCallback`, `problem.addNewnodeCallback`.

problem.addcbnodecutoff

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addNodecutoffCallback` *instead.*

Declares a user node cutoff callback function, called every time a node is cut off as a result of an improved integer solution being found during the branch and bound search. This callback function will be called in addition to any callbacks already added by addcbnodecutoff.

Synopsis

```
problem.addcbnodecutoff(callback, data, priority)
callback(my_prob, my_object, node)
```

Arguments

`callback`	The callback function, which takes three arguments, `my_prob`, `my_object` and `node`, and has no return value. This function is called every time a node is cut off as the result of an improved integer solution being found.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbnodecutoff`.
`node`	The number of the node that is cut off.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple node-optimal callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following notifies the user whenever a node is cutoff during the tree search:

```
prob.addcbnodecutoff(Cutoff, None, 0)
mipoptimize(prob, "")
```

The callback function might resemble:

```
def Cutoff(prob, object, node):

    print("Node {0} cutoff".format(node))
```

Further information

This function allows the user to keep track of the eligible nodes. Note that the LP solution will not be available from this callback.

Related topics

`problem.removeNodecutoffCallback`, `problem.addOptnodeCallback`, `problem.addInfnodeCallback`, `problem.addIntsolCallback`, `problem.addNodecutoffCallback`.

problem.addcbnodelpsolved

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addNodeLPSolvedCallback` *instead.*
Declares a node LP solved callback function, called during the branch and bound search, after the LP relaxation has been solved for the current node, but before any internal cuts and heuristics have been applied. This callback function will be called in addition to any callbacks already added by problem.addcbnodelpsolved.

Synopsis

```
problem.addcbnodelpsolved(callback, data, priority)
callback(my_prob, my_object)
```

Arguments

`callback`	The callback function, which takes two arguments, `my_prob` and `my_object`, and has no return value.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbnodecutoff`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple node-optimal callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Related topics

`problem.removeNodeLPSolvedCallback`, `problem.addOptnodeCallback`, `problem.addNodeLPSolvedCallback`

problem.addcboptnode

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addOptnodeCallback` *instead.*

Declares an optimal node callback function, called during the branch and bound search, after the LP relaxation has been solved for the current node, and after any internal cuts and heuristics have been applied, but before the Optimizer checks if the current node should be branched. This callback function will be called in addition to any callbacks already added by addcboptnode.

Synopsis

```
problem.addcboptnode(callback, data, priority)
infeas = callback(my_prob, my_object)
```

Arguments

`callback`	The callback function which takes two arguments, `my_prob` and `my_object`, and returns an integer. If the value returned by `callback` is nonzero, the solution process will be interrupted.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcboptnode`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple node-optimal callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following prints the optimal objective value of the node LP relaxations:

```
prob.addcboptnode(nodeOptimal, None, 0)
prob.mipoptimize("")
```

The callback function might resemble:

```
def nodeOptimal(prob, object):

    node = prob.attributes.currentnode
    print("NodeOptimal: node number", node)
    objval = prob.attributes.lpobjval
    print("Objective function value =", objval)
    return 0
```

Related topics

`problem.removeOptnodeCallback`, `problem.addInfnodeCallback`, `problem.addIntsolCallback`, `problem.addNodecutoffCallback`, `problem.addOptnodeCallback`, `CALLBACKCOUNT_OPTNODE`.

problem.addcbpreintsol

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addPreIntsolCallback` *instead.*

Declares a user integer solution callback function, called when an integer solution is found by heuristics or during the branch and bound search, but before it is accepted by the Optimizer. This callback function will be called in addition to any integer solution callbacks already added by addcbpreintsol.

Synopsis

```
problem.addcbpreintsol(callback, data, priority)
(ifreject, newcutoff) = callback(my_prob, my_object, soltype, cutoff)
```

Arguments

`callback` The callback function which takes four arguments, `my_prob`, `my_object`, `soltype` and `cutoff`, returns a tuple of two elements. This function is called when an integer solution is found, but before the solution is accepted by the Optimizer, allowing the user to reject the solution.

`my_prob` The problem passed to the callback function, `callback`.

`my_object` The user-defined object passed as object when setting up the callback with `addcbpreintsol`.

`soltype` The type of MIP solution that has been found.

- `0` The continuous relaxation solution to the current node of the tree search, which has been found to be integer feasible.
- `1` A MIP solution found by a heuristic.
- `2` A MIP solution provided by the user.
- `3` A solution resulting from refinement of primal or dual violations of a previous MIP solution.

`cutoff` The current `cutoff` value.

`ifreject` If a nonzero value is returned in the first position of the tuple, the solution will be rejected.

`newcutoff` A new `cutoff` value can be returned in the second position of the tuple, to be used by the Optimizer if the solution is accepted. The returned `newcutoff` value will not be updated if the solution is rejected.

`data` A user-defined object to be passed to the callback function, `callback`.

`priority` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Further information

1. If a solution is rejected, the Optimizer will drop the found solution without updating any attributes, including the cutoff value. To change the cutoff value when rejecting a solution, the control `MIPABSCUTOFF` should be set instead.
2. When a node solution is rejected (`isheuristic = 0`), the node itself will be dropped without further branching.
3. To retrieve the integer solution, use either `problem.getCallbackSolution` or `problem.getCallbackPresolveSolution`. `problem.getSolution` will not return the newly found solution because it has not been saved at this point.

Related topics

`problem.removePreIntsolCallback`, `problem.addIntsolCallback`, `problem.addPreIntsolCallback`.

problem.addcbprenode

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addPrenodeCallback` *instead.*
Declares a preprocess node callback function, called before the LP relaxation of a node has been optimized, so the solution at the node will not be available. This callback function will be called in addition to any callbacks already added by addcbprenode.

Synopsis

```
problem.addcbprenode(callback, data, priority)
nodinfeas = callback(my_prob, my_object)
```

Arguments

`callback`	The callback function, which takes two arguments, `my_prob`, `my_object` and and returns an integer. This function is called before a node is reoptimized and the node may be made infeasible by returning `1` from the callback.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbprenode`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple preprocess node callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following example notifies the user before each node is processed:

```
prob.addcbprenode(preNode, None, 0)
prob.mipoptimize("")
```

The callback function might resemble:

```
def preNode(prob, object):

    return 0 # set to 1 if node is infeasible
```

Related topics

`problem.removePrenodeCallback`, `problem.addInfnodeCallback`, `problem.addIntsolCallback`, `problem.addNodecutoffCallback`, `problem.addOptnodeCallback`, `problem.addPrenodeCallback`.

problem.addcbusersolnotify

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addUserSolNotifyCallback` instead.
Declares a callback function to be called each time a solution added by `problem.addMipSol` has been processed. This callback function will be called in addition to any callbacks already added by `addcbusersolnotify`.

Synopsis

```
problem.addcbusersolnotify(callback, data, priority)
callback(my_prob, my_object, solname, status)
```

Arguments

`callback` The callback function which takes four arguments, `my_prob`, `my_object`, `id` and `status` and has no return value.

`my_prob` The problem passed to the callback function, `callback`.

`my_object` The user-defined object passed as object when setting up the callback with `addcbusersolnotify`.

`solname` The string name assigned to the solution when it was loaded into the Optimizer using `problem.addMipSol`.

`status` One of the following status values:

- `0` An error occured while processing the solution.
- `1` Solution is feasible.
- `2` Solution is feasible after reoptimizing with fixed MIP entities.
- `3` A local search heuristic was applied and a feasible solution discovered.
- `4` A local search heuristic was applied but a feasible solution was not found.
- `5` Solution is infeasible and a local search could not be applied.
- `6` Solution is partial and a local search could not be applied.
- `7` Failed to reoptimize the problem with MIP entities fixed to the provided solution. Likely because a time or iteration limit was reached.
- `8` Solution is dropped. This can happen if the MIP problem is changed or solved to completion before the solution could be processed.

`data` A user-defined object to be passed to the callback function, `callback`.

`priority` An integer that determines the order in which multiple callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to `0` if not required.

Further information

If presolve is turned on, any solution added with `problem.addMipSol` will first be presolved before it can be checked. The value returned in `status` refers to the presolved solution, which might have had values adjusted due to bound changes, fixing of variables, etc.

Related topics

`problem.removeUserSolNotifyCallback`, `problem.addMipSol`, `problem.addUserSolNotifyCallback`.

problem.addcbbeforeobjective

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addBeforeObjectiveCallback` *instead.*
Declares a callback which will be called before each objective in a multi-objective problem is solved. This callback function will be called in addition to any callbacks already added by addcbbeforeobjective.

Synopsis

```
problem.addcbbeforeobjective(callback, data, priority)
callback(my_prob, my_object)
```

Arguments

`callback`	The callback function which takes two arguments, `my_prob` and `my_object`, and has no return value. This function is called before every solve.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbbeforeobjective`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple before objective callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

This example sets a node limit for the second multi-objective solve:

```
def beforeobjective(prob, object):
  if prob.attributes.solvedobjs == 1:
    prob.controls.maxnode = 100

prob.addcbbeforeobjective(beforeobjective, None, 0)
prob.optimize("")
```

Related topics

`problem.removeBeforeObjectiveCallback`, `problem.addAfterObjectiveCallback`, `problem.addBeforeObjectiveCallback`.

problem.addcbafterobjective

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addAfterObjectiveCallback` instead.
Declares a callback which will be called after each objective in a multi-objective problem is solved. This callback function will be called in addition to any callbacks already added by addcbafterobjective.

Synopsis

```
problem.addcbafterobjective(callback, data, priority)
callback(my_prob, my_object)
```

Arguments

`callback`	The callback function which takes two arguments, `my_prob` and `my_object`, and has no return value. This function is called after every solve.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbafterobjective`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple after objective callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

This example logs a message after each objective is solved:

```
def afterobjective(prob, object):
  print("Completed solve {}".format(prob.attributes.solvedobjs))

prob.addcbafterobjective(afterobjective, None, 0)
prob.optimize("")
```

Related topics

`problem.removeAfterObjectiveCallback`, `problem.addBeforeObjectiveCallback`, `problem.addAfterObjectiveCallback`.

problem.addcbpresolve

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addPresolveCallback` *instead.*
Declares a callback to be called after presolve has been performed.

Synopsis

```
problem.addcbpresolve(callback, data, priority)
callback(my_prob, my_object)
```

Arguments

`callback`	The callback function which takes two arguments, `my_prob` and `my_object`, and has no return value. This function is called after presolve is complete.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbpresolve`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple presolve callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Related topics

`problem.removePresolveCallback`, `problem.addPresolveCallback`.

problem.addcoefs

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpAddCoefs` instead.
Add non-linear coefficients to the SLP problem

Synopsis

```
problem.addcoefs(rowindex, colindex, factor, fstart, parsed, type, value)
```

Arguments

`rowindex`	Array holding the rows (or their indices or names) for the coefficient.
`colindex`	Array holding the columns (or their indices or names) for the coefficient.
`factor`	Array holding factor by which formula is scaled. If `None`, a value of 1.0 will be used.
`fstart`	Integer array holding the start position in the arrays `Type` and `Value` of the formula for the coefficients. `fstart` should have an extra entry containing the next position after the end of the last formula.
`parsed`	Boolean indicating whether the token arrays are formatted as internal unparsed (`parsed=False`) or internal parsed reverse Polish (`parsed=True`).
`type`	Array of token types providing the formula for each coefficient.
`value`	Array of values corresponding to the types in `Type`.

Example

Assume that the rows and columns of `Prob` are named `Row1`, `Row2` ..., `Col1`, `Col2` ..., respectively. The following example adds coefficients representing:
`Col2 * Col3 + Col6 * Col2^2` into `Row1` and
`Col2 ^ 2` into `Row3`.

```
rowindex = [Row1,Row1,Row3]
colindex = [Col2,Col6,Col2]

formulastart = []
type = []
value = []

formulastart.append(len(type))
type.append(xp.constants.TOK_COL); value.append(3)
type.append(xp.constants.TOK_EOF); value.append(0)

formulastart.append(len(type))
type.append(xp.constants.TOK_COL); value.append(2)
type.append(xp.constants.TOK_COL); value.append(2)
type.append(xp.constants.TOK_OP);  value.append(xp.constants.OP_MULTIPLY)
type.append(xp.constants.TOK_EOF); value.append(0)

formulastart.append(len(type))
type.append(xp.constants.TOK_COL); value.append(2)
type.append(xp.constants.TOK_EOF); value.append(0)

formulastart.append(len(type))

p.addcoefs(rowindex, colindex,  None, formulastart, True, Type, Value)
```

The first coefficient in `Row1` is in `Col2` and has the formula `Col3`, so it represents `Col2 * Col3`.
The second coefficient in `Row1` is in `Col6` and has the formula `Col2 * Col2` so it represents `Col6 * Col2^2`.
The formulae are described as *parsed* (`parsed=True`), so the formula is written as
`Col2 Col2 *`
rather than the unparsed form
`Col2 * Col2`
The last coefficient, in `Row3`, is in `Col2` and has the formula `Col2`, so it represents `Col2 * Col2`.

Further information

The j^{th} coefficient is made up of two parts: `Factor` and `Formula`. `Factor` is a constant multiplier, which can be

provided in the `Factor` array. If Xpress Nonlinear can identify a constant factor in `Formula`, then it will use that as well, to minimize the size of the formula which has to be calculated. `Formula` is made up of a list of tokens in `Type` and `Value` starting at `formulastart[j]`. The tokens follow the rules for parsed or unparsed formulae as indicated by the setting of `parsed`. The formula must be terminated with an `xslp_op_eof` token. If several coefficients share the same formula, they can have the same value in `fstart`. For possible token types and values see the chapter on Formula Parsing in the SLP reference manual.

The `addcoef` function loads additional items into the SLP problem. The corresponding `loadcoefs` function deletes any existing items first.

The behaviour for existing coefficients is additive: the formula defined in the parameters are added to any existing formula coefficients. However, due to performance considerations, such duplications should be avoided when possible.

Related topics

`problem.slpChgCoef`, `problem.slpChgCoefStr`, `problem.slpDelCoefs`, `problem.slpGetCoefStr`,`problem.slpLoadCoefs`, `problem.slpAddCoefs`

problem.addcols

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addCols` *instead.*
Add columns to the problem after passing it to the Optimizer using the input routines.

Synopsis

```
problem.addcols(objcoef, start, rowind, rowcoef, lb, ub, names, types,
       unlinked=False)
```

Arguments

`objcoef` Array containing the objective function coefficients of the new columns.

`start` Array containing the offsets in the `rowind` and `rowcoef` arrays of the start of the elements for each column. The array must contain one more element than the number of columns to add, with the last element being the total number of coefficients across all columns.

`rowind` Array containing the rows (i.e. `xpress.constraint` objects, indices, or names) for the elements in each column.

`rowcoef` Array containing the element values.

`lb` Array containing the lower bounds on the added columns.

`ub` Array containing the upper bounds on the added columns.

`names` (optional) Array containing the names of the columns added.

`types` (optional) Array of characters containing the types of the newly added columns:

- `C` indicates a continuous variable (default);
- `I` indicates an integer variable;
- `B` indicates a binary variable;
- `S` indicates a semi-continuous variable;
- `R` indicates a semi-continuous integer variable;
- `P` indicates a partial integer variable.

`unlinked` (optional) If `True`, unlinked variables will be created (deprecated).

Example

In this example, we consider the two problems:

(a)	maximize:	2x + y			(b)	maximize:	2x + y + 3z		
	subject to:	x + 4y	$\leqslant$	24		subject to:	x + 4y + 2z	$\leqslant$	24
		y	$\leqslant$	5			y + z	$\leqslant$	5
		3x + y	$\leqslant$	20			3x + y	$\leqslant$	20
		x + y	$\leqslant$	9			x + y + 3z	$\leqslant$	9
							z	$\leqslant$	12

Using `addcols`, the following transforms (a) into (b):

```
p = xpress.problem()

p.read("example.lp")

# assume this problem has at least four constraints
p.addcols(obj=[3], start=[0,3], rowind=[0, 1, 3],
          matval=[2,1,3], lb=[-xpress.infinity], ub=[12],
          names=['john_cleese'], types=['C'])
```

Further information

1. The constant `xpress.infinity` can be used to represent infinite bounds.
2. If the columns are added to a MIP problem, then they will be continuous variables unless `types` is specified. Use `problem.chgColType` to impose integrality conditions on such new columns.

Related topics

`problem.addRows`, `problem.chgColType`, `problem.addCols`.

problem.addcuts

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addCuts` *instead.*
Adds cuts directly to the matrix at the current node. Any cuts added to the matrix at the current node and not deleted at the current node will be automatically added to the cut pool. The cuts added to the cut pool will be automatically restored at descendant nodes.

Synopsis

```
problem.addcuts(cuttype, rowtype, rhs, start, colind, cutcoef)
```

Arguments

`cuttype`	Array containing the user assigned cut types. The cut types can be any integer chosen by the user, and are used to identify the cuts in other cut manager routines using user supplied parameters. The cut type can be interpreted as an integer or a bitmap - see `problem.delCuts`.
`rowtype`	Character array containing the row types: `L` indicates a $\leqslant$ row; `G` indicates $\geqslant$ row; `E` indicates an = row.
`rhs`	Array containing the right hand side elements for the cuts.
`start`	Array containing offset into the `colind` and `cutcoef` arrays indicating the start of each cut. This array is of length `ncuts+1` with the last element, `start[ncuts]`, being where cut `ncuts+1` would start.
`colind`	Array containing the columns (i.e. `xpress.var` objects, indices, or names) in the cuts.
`cutcoef`	Array containing the matrix values for the cuts.

Further information

1. The columns and elements of the cuts must be stored contiguously in the `colind` and `cutcoef` arrays passed to `addcuts`. The starting point of each cut must be stored in the `start` array. To determine the length of the final cut, the `start` array must be of length `ncuts+1` with the last element of this array containing the position in `colind` and `cutcoef` where the cut `ncuts+1` would start. `start[ncuts]` denotes the number of nonzeros in the added cuts.
2. The cuts added to the matrix are always added at the end of the matrix and the number of rows is always set to the original number of cuts added. If `ncuts` have been added, then the rows `0`,...,`ROWS-ncuts-1` are the original rows, whilst the rows `ROWS-ncuts`,...,`ROWS-1` are the added cuts. The number of cuts can be found by consulting the `CUTS` problem attribute.
3. This function should be called only from within callback functions set by either `problem.addNodeLPSolvedCallback`, `problem.addOptnodeCallback`, `problem.addPreIntsolCallback` or `problem.addPrenodeCallback`.

Related topics

`problem.addRows`, `problem.delCPCuts`, `problem.delCuts`, `problem.getCPCutList`, `problem.getCutList`, `problem.loadCuts`, `problem.storeCuts`, `problem.addCuts`, Section "Working with the cut manager" of the Xpress Optimizer reference manual.

problem.adddfs

Purpose

This subroutine is deprecated and will be removed in a future release.
Add a set of distribution factors

Synopsis

```
problem.adddfs(colindex, rowindex, value)
```

Arguments

`colindex`	Array of columns (i.e. `xpress.var` objects, indices, or names) whose distribution factor is to be changed.
`rowindex`	Array of rows (i.e. `xpress.constraint` objects, indices, or names) where each distribution factor applies.
`value`	Array holding the new values of the distribution factors.

Example

The following example adds distribution factors as follows:
column 282 in row 134 = 0.1
column 282 in row 136 = 0.15
column 285 in row 133 = 1.0.

```
colindex = [282, 282, 285]
rowindex = [134, 136, 133]
value    = [0.1, 0.15,  1]
p.adddfs(colindex,rowindex,value)
```

Further information

The *distribution factor* of a column in a row is the matrix coefficient of the corresponding delta vector in the row. Distribution factors are used in conventional recursion models, and are essentially normalized first-order derivatives. Xpress SLP can accept distribution factors instead of initial values, provided that the values of the variables involved can all be calculated after optimization using determining rows, or by a callback.

The `problem.adddfs` functions load additional items into the SLP problem. The corresponding `problem.loaddfs` functions delete any existing items first.

Related topics

`problem.chgdf`, `problem.getdf`, `problem.loaddfs`

problem.addgencons

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addGenCons` *instead.*
Adds one or more general constraints to the problem. Each general constraint `y = f(x1, ..., xn, c1, ..., cn)` consists of one or more (input) columns xi, zero or more constant values ci and a resultant (output column) y. General constraints can be defined using operators such as `maximum` and `minimum` (at least one input column of any contype and arbitrary number of input values), `and` and `or` (at least one binary input column, no constant values, binary resultant) and `absolute value` (exactly one input column of arbitrary contype, no constant values).

Synopsis

```
problem.addgencons (contype, resultant, colstart, colind, valstart, val)
```

Arguments

`contype`	list or array containing the types of the general constraints: `xpress.GenConsType.MAX (0)` indicates a `maximum` constraint; `xpress.GenConsType.MIN (1)` indicates a `minimum` constraint; `xpress.GenConsType.AND (2)` indicates an `and` constraint; `xpress.GenConsType.OR (3)` indicates an `or` constraint; `xpress.GenConsType.ABS (4)` indicates an `absolute value` constraint.
`resultant`	Array/list containing the output variables (or indices thereof) of the general constraints.
`colstart`	Array/list containing the start index of each general constraint in the `colind` array.
`colind`	Array/list containing the input variables in all general constraints.
`valstart`	Array/list containing the start index of each general constraint in the `val` array (may be `None`).
`val`	Array/list containing the constant values in all general constraints (may be `None`).

Example

This adds two new general constraints `x2 = max(x0, x1, 5)` and `x3 = |x1|`:

```
contype = [xpress.GenConsType.MAX, xpress.GenConsType.ABS]
resultant = [2, 3]
colstart = [0, 2]
colind = [0, 1, 1]
valstart = [0, 1]
val = [5.0]

prob.addgencons(contype, resultant, colstart, colind, valstart, val)
prob.optimize()
```

Further information

General constraints must be set up before solving the problem. They are converted to additional binary variables, indicator and linear constraints with the exact formulation and number of added entities depending on the performed presolving.

Note that using non-binary variables in `and`/`or` constraints or adding constant values to them or `absolute value` constraints will give an error at solve time.

Related topics

`problem.getGenCons`, `problem.delGenCons`, `xpress.And`, `xpress.Or`, `xpress.max`, `xpress.min`, `xpress.abs`, `problem.addGenCons`.

problem.addmipsol

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addMipSol` *instead.*
Adds a new feasible, infeasible or partial MIP solution for the problem to the Optimizer.

Synopsis

```
problem.addmipsol(solval, colind, name)
```

Arguments

`solval`	Array containing solution values.
`colind`	Optional integer array containing the columns (i.e. `xpress.var` objects, indices, or names) for the solution values provided in `solval`. It is optional when the length of solval is equal to `COLS`, in which case it is assumed that `solval` provides a complete solution vector.
`name`	An optional name to associate with the solution.

Further information

1. The function returns immediately after passing the solution to the Optimizer. The solution is placed in a pool until the Optimizer is able to analyze the solution during a MIP solve.
2. If the provided solution is found to be infeasible, a limited local search heuristic will be run in an attempt to find a close feasible integer solution.
3. If a partial solution is provided, discrete columns will be fixed to any provided values and a limited local search will be run in an attempt to find integer feasible values for the remaining unspecified columns. Values provided for continuous column in partial solutions are currently ignored.
4. The `problem.addUserSolNotifyCallback` callback function can be used to discover the outcome of a loaded solution. The optional name provided as `name` will be returned in the callback function.
5. If one or more solutions are loaded during the `problem.addOptnodeCallback` callback, the Optimizer will process all loaded solutions and fire the callback again. This will be repeated as long as new solutions are loaded during the callback.

Related topics

`problem.addUserSolNotifyCallback`, `problem.addOptnodeCallback`, `problem.addMipSol`.

problem.addnames

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addNames` *instead.*
Associates names with rows, columns, sets, piecewise linear constraints, general constraints or objectives.

Synopsis

```
problem.addnames(type, names, first, last)
```

Arguments

`names`	Array containing names.	
`type`	`xpress.Namespaces.ROW`	for row names;
	`xpress.Namespaces.COLUMN`	for column names;
	`xpress.Namespaces.SET`	for set names;
	`xpress.Namespaces.PWLCONS`	for piecewise linear constraint names;
	`xpress.Namespaces.GENCONS`	for general constraint names;
	`xpress.Namespaces.OBJECTIVE`	for objective names.
`first`	Start of the range of rows, columns, sets, piecewise linear constraints, general constraints or objectives.	
`last`	End of the range of rows, columns, sets, piecewise linear constraints, general constraints or objectives.	

Example

Assign names to an array of constraints:

```
p = xp.problem()
x = p.addVariables(10)
m = np.random.rand(10, 10)
ctrs = xp.Dot(m, x) == 1
first_ctr = prob.attributes.rows
prob.addConstraint(ctrs)
last_ctr = prob.attributes.rows - 1
names = np.array(['ctr{}'.format(i) for i in range(len(ctrs))])
p.addnames(xp.Namespaces.ROW, names, first_ctr, last_ctr)
```

Related topics

`problem.addCols`, `problem.addRows`, `problem.getNameList`, `problem.addNames`.

problem.addobj

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addObj` *instead.*
Appends an objective function with the given coefficients to a multi-objective problem. The weight and priority of the objective are set to the given values.

Synopsis

```
problem.addobj(colind, objcoef, priority=0, weight=1)
```

Arguments

`colind`	Integer array of length `ncols` containing the indices of the columns whose objective coefficients will change. An index of `-1` indicates that the fixed part of the objective function on the right hand side should change.
`objcoef`	Double array of length `ncols` giving the new objective function coefficients.
`priority`	The priority for the objective function. During optimization, objectives with the same priority are combined together in a weighted sum.
`weight`	The weight for the objective function. If the weight is negative, the sense of this objective is reversed.

Example

Adding a second objective function to a problem:

```
colind = [0, 2, 5]
objcoef = [25.0, 5.3, 0.0]
p.addobj(colind, objcoef, 1, 1)
```

Related topics

`problem.addObjective`, `problem.setObjective`, `problem.chgObjN`, `problem.getObjN`, `problem.delObj`, `problem.chgObj`, `problem.addObj`.

problem.addpwlcons

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addPwlCons` *instead.*
Adds one or more piecewise linear constraints to the problem. Each piecewise linear constraint `y = f(x)` consists of an (input) column x, a resultant (output column) y and a piecewise linear function f. The piecewise linear function f is described by a number of breakpoints, which are given as combinations of x- and y-values. Discontinuous piecewise linear functions are supported, in this case both the left and right limit at a given point need to be entered as breakpoints. To differentiate between left and right limit, the breakpoints need to be given as a list with non-decreasing x-values.

Synopsis

```
problem.addpwlcons(colind, resultant, start, xval, yval)
```

Arguments

`colind`	Integer array (or list) containing the input variables x of the piecewise linear functions.
`resultant`	Integer array containing the output variables y of the piecewise linear functions.
`start`	Integer array containing the start index of each piecewise linear constraint in the `xval` and `yval` arrays.
`xval`	Array containing the x-values of the breakpoints.
`yval`	Array containing the y-values of the breakpoints.

Example

This adds a new piecewise linear constraint `y = f(x)`, where

```
f(x) = -x            if x < 0
f(x) = 1      if 0 <= x <= 2
f(x) = 2x-3          if x > 2
```

This function can be defined using the breakpoints `(x = -1, y = 1), (0,0), (0,1), (2,1), (3,3)` (note that the first breakpoint could also be replaced, e.g., by (x = -2, y = 2), similarly for the last):

```
colind = [x]
resultant = [y]
start = [0]
xval = [-1, 0, 0, 2, 3]
yval = [ 1, 0, 1, 1, 3]

prob.addpwlcons(colind, resultant, start, xval, yval)
prob.setObjective(y)   # the piecewise linear function is to be minimized
prob.mipoptimize()
```

Further information

Piecewise linear constraints must be set up before solving the problem. They are converted to additional linear constraints, continuous variables and SOS2 constraints, with the exact formulation and number of added entities depending on the convexity of the piecewise linear function and some presolving steps that are applied.

Related topics

`problem.getPwlCons`, `problem.delPwlCons`, `xpress.pwl`, `problem.addPwlCons`.

problem.addqmatrix

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addQMatrix` *instead.*
Adds a new quadratic matrix into a row defined by triplets.

Synopsis

```
problem.addqmatrix(row, rowqcol1, rowqcol2, rowqcoef)
```

Arguments

`row`	Row (i.e. `xpress.constraint` object, index, or name) where the quadratic matrix is to be added.
`rowqcol1`	Array with first variables (i.e. `xpress.var`objects, indices, or names) in the triplets.
`rowqcol2`	Array with second variables (i.e. `xpress.var`objects, indices, or names) index in the triplets.
`rowqcoef`	Array of coefficients in the triplets.

Further information

The triplets should define the upper triangular part of the quadratic expression. This means that to add `[x^2 + 6 xy]` the `rowqcoef` array shall contain the coefficients 1 and 3.

Related topics

`problem.loadproblem`, `problem.getQRowCoeff`, `problem.chgQRowCoeff`, `problem.getQRowQMatrix`, `problem.getQRowQMatrixTriplets`, `problem.getQRows`, `problem.chgQObj`, `problem.chgMQObj`, `problem.getQObj`, `problem.addQMatrix`.

problem.addrows

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addRows` *instead.*
Adds rows and their coefficient to the problem.

Synopsis

```
problem.addrows(rowtype, rhs, start, colind, rowcoef, range=None, names=None,
       unlinked=False)
```

Arguments

`rowtype`	Character array containing the row types: `L` indicates a $\leqslant$ row; `G` indicates $\geqslant$ row; `E` indicates an = row. `R` indicates a range constraint; `N` indicates a nonbinding constraint.
`rhs`	Array containing the right hand side elements.
`start`	Array containing the offsets in the `colind` and `rowcoef` arrays of the start of the elements for each row. The array must contain one more element than the number of rows to add, with the last element being the total number of coefficients across all rows.
`colind`	Array containing the (contiguous) columns (i.e. `xpress.var`objects, indices, or names) for the elements in each row.
`rowcoef`	Array containing the (contiguous) element coefficients.
`range`	(optional) Array containing the row range elements. The values in the `range` array will only be read for `'R'` type rows. The entries for other type rows will be ignored.
`names`	(optional) Array of names to be assigned to each new row.
`unlinked`	(optional) If `True`, unlinked constraints will be created (deprecated).

Example

Suppose the current problem is:

maximize:	2x + y + 3z		
subject to:	x + 4y + 2z	$\leqslant$	24
	y + z	$\leqslant$	5
	3x + y	$\leqslant$	20
	x + y + 3z	$\leqslant$	9

Then the following adds the row $8x + 9y + 10z \leqslant 25$ to the problem and names it `NewRow`:

```
p = xpress.problem()
p.addrows(['L'], [25], [0,3], [0,1,2],
          rowcoef=[8, 9, 10], range=None, names=['NewRow'])
```

Further information

Range rows are automatically converted to type `L`, with an upper bound in the slack. This must be taken into consideration, when retrieving row type, right–hand side values or range information for rows.

Related topics

`problem.addCols`, `problem.addCuts`, `problem.addRows`.

problem.addsetnames

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addNames` instead.
When a model with MIP entities is loaded, any special ordered sets may not have names associated with them. If you wish names to appear in the ASCII solutions files, the names for a range of sets can be added with this function.

Synopsis

```
problem.addsetnames(names, first=0, last=problem.attributes.sets - 1)
```

Arguments

`names`	A list of strings contatining all names to be assigned.
`first`	(Optional) first of the set range.
`last`	(Optional) last of the set range.

Example

Add set names (`set1` and `set2`) to a problem:

```
snames = ["set1", "set2"]
...
p.addsetnames(snames, 0, 1)
```

Further information

If `first` is not provided, it is considered equal to 0; if `last` is omitted, a value of `problem.attributes.sets - 1` is used.

Related topics

`problem.loadproblem`.

problem.addtolsets

Purpose

This subroutine is deprecated and will be removed in a future release.
Add sets of standard tolerance values to an SLP problem

Synopsis

```
problem.addtolsets(tol)
```

Argument

`tol` Array of 9h elements containing the 9 tolerance values for each set in order.

Example

The following example creates two tolerance sets: the first has values of 0.005 for all tolerances; the second has values of 0.001 for relative tolerances (numbers 2,4,6,8), values of 0.01 for absolute tolerances (numbers 1,3,5,7) and zero for the closure tolerance (number 0).

```
tol = 9*[0.005]+[0]+[0.01,0.001]*4
p.addtolsets(tol)
```

Further information

A tolerance set is an array of 9 values containing the following tolerances:

Entry / Bit	Tolerance	XSLP constant	XSLP bit constant
0	Closure tolerance (TC)	`xslp_TOLSET_TC`	`xslp_TOLSETBIT_TC`
1	Absolute delta tolerance (TA)	`xslp_TOLSET_TA`	`xslp_TOLSETBIT_TA`
2	Relative delta tolerance (RA)	`xslp_TOLSET_RA`	`xslp_TOLSETBIT_RA`
3	Absolute coefficient tolerance (TM)	`xslp_TOLSET_TM`	`xslp_TOLSETBIT_TM`
4	Relative coefficient tolerance (RM)	`xslp_TOLSET_RM`	`xslp_TOLSETBIT_RM`
5	Absolute impact tolerance (TI)	`xslp_TOLSET_TI`	`xslp_TOLSETBIT_TI`
6	Relative impact tolerance (RI)	`xslp_TOLSET_RI`	`xslp_TOLSETBIT_RI`
7	Absolute slack tolerance (TS)	`xslp_TOLSET_TS`	`xslp_TOLSETBIT_TS`
8	Relative slack tolerance (RS)	`xslp_TOLSET_RS`	`xslp_TOLSETBIT_RS`

The xslp_TOLSET constants can be used to access the corresponding entry in the value arrays, while the xslp_TOLSETBIT constants are used to set or retrieve which tolerance values are used for a given SLP variable. Once created, a tolerance set can be used to set the tolerances for any SLP variable. If a tolerance value is zero, then the default tolerance will be used instead. To force the use of a tolerance, use the `problem.chgtolset` function and set the `Status` variable appropriately. See the section "Convergence criteria" of the SLP Reference Manual for a fuller description of tolerances and their uses. The `problem.addtolsets` functions load additional items into the SLP problem. The corresponding `problem.loadtolsets` functions delete any existing items first.

Related topics

`problem.chgtolset`, `problem.deltolsets`, `problem.gettolset`, `problem.loadtolsets`

problem.addvars

Purpose

This subroutine is deprecated and will be removed in a future release.
Add SLP variables defined as matrix columns to an SLP problem

Synopsis

```
problem.addvars(colindex, vartype, detrow, seqnum, tolindex, initvalue, stepbound)
```

Arguments

`colindex` Integer array holding the index of the matrix column corresponding to each SLP variable.

`vartype` Bitmap giving information about the SLP variables, compare the variable status keys in the C interface, in particular:
`Bit 2` Variable has an initial value;
May be `None` if not required.

`detrow` Integer array holding the index of the determining row for each SLP variable (a negative value means there is no determining row)
May be `None` if not required.

`seqnum` Integer array holding the index sequence number for cascading for each SLP variable (a zero value means there is no pre-defined order for this variable)
May be `None` if not required.

`tolindex` Integer array holding the index of the tolerance set for each SLP variable (a zero value means the default tolerances are used)
May be `None` if not required.

`initvalue` Array holding the initial value for each SLP variable (use the `VarType` bit map to indicate if a value is being provided)
May be `None` if not required.

`stepbound` Array holding the initial step bound size for each SLP variable (a zero value means that no initial step bound size has been specified). If a value of `xpress.infinity` is used for a value in `stepbound`, the delta will never have step bounds applied, and will almost always be regarded as converged.
May be `None` if not required.

Example

The following example loads two SLP variables into the problem. They correspond to columns 23 and 25 of the underlying LP problem. Column 25 has an initial value of 1.42; column 23 has no specific initial value

```
colindex  = [23,25]
vartype   = [0,4]
initvalue = [0,1.42]

p.addvars(colindex, vartype, None, None, None, initvalue, None)
```

Note that the initial value for the first variable will not actually be used, because the `XSLP_HASIV` flag is not set (`vartype` = 0). Bit 2 of `vartype` is set for the second variable to indicate that the initial value has been set. The arrays for determining rows, sequence numbers, tolerance sets and step bounds are not used at all, and so have been passed to the function as `None`.

Further information

The `addvars` functions load additional items into the SLP problem. The corresponding `loadvars` functions delete any existing items first.

Related topics

`problem.chgvar`, `problem.delvars`, `problem.getvar`, `problem.loadvars`

problem.basisstability

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.basisStability` *instead.*
Returns various measures for the stability of the current basis, including the basis condition number.

Synopsis

```
x = problem.basisstability(type, norm, scaled)
```

Arguments

`type`	`0`	Condition number of the basis.
	`1`	Stability measure for the solution relative to the current basis.
	`2`	Stability measure for the duals relative to the current basis.
	`3`	Stability measure for the right hand side relative to the current basis.
	`4`	Stability measure for the basic part of the objective relative to the current basis.
`norm`	`0`	Use the infinity norm.
	`1`	Use the 1 norm.
	`2`	Use the Euclidian norm for vectors and the Frobenius norm for matrices.
`scaled`	If the stability values are to be calculated in the scaled or the unscaled matrix.	

Return value

`x`	The calculated value

Further information

1. The condition number (`type` = `0`) of an invertible matrix is the norm of the matrix multiplied with the norm of its inverse. This number is an indication of how accurate the solution can be calculated and how sensitive it is to small changes in the data. The larger the condition number is, the less accurate the solution is likely to become.
2. The stability measures (`type` = `1...4`) are using the original matrix and the basis to recalculate the various vectors related to the solution and the duals. The returned stability measure is the norm of the difference of the recalculated vector to the original one.

problem.bndsa

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.bndSA` *instead.*
Returns upper and lower sensitivity ranges for specified variables' lower and upper bounds. If the bounds are varied within these ranges the current basis remains optimal and feasible.

Synopsis

```
problem.bndsa(colind, lblower=None, lbupper=None, ublower=None, ubupper=None)
```

Arguments

`colind`	A list or NumPy array of the variables (or their indices or names), for which the sensitivity range is requested.
`lblower`	Array (to be passed as a list, possibly empty) that will contain the variable lower bound lower ranges.
`lbupper`	Array for the variable lower bound upper ranges.
`ublower`	Array for the variable upper bound lower ranges.
`ubupper`	Array for the variable upper bound upper ranges.

Example

```
p = xp.problem()
x = p.addVariables(10)
[...]
ll, lu, ul, uu = [], [], [], []
p.bndsa(x, ll, lu, ul, uu)
print("ranges:", ll, lu, ul, uu)
```

Further information

`problem.bndsa` can only be called when an optimal solution to the current LP has been found. It cannot be used when the problem is MIP presolved.

Related topics

`problem.bndSA`, `problem.rhsSA`, `problem.objSA`.

problem.btran

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.BTran` instead.
Post-multiplies a (row) vector provided by the user by the inverse of the current basis.

Synopsis

```
problem.btran(vec)
```

Argument

`vec`	Array of length `problem.attributes.rows` containing the values by which the basis inverse is to be multiplied. The transformed values will also be returned in this array.

Example

Get the (unscaled) tableau row `z` of constraint number `irow`, assuming that all arrays have been dimensioned.

```
y = [0,1,0,0]
p.btran(y)
print("btran result:", y)
```

Further information

If the problem is in a presolved state, `btran` works with the basis for the presolved problem.

Related topics

`problem.FTran`, `problem.BTran`.

problem.calcobjn

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.calcObjN` *instead.*
Returns the value of a given objective. A solution can optionally be provided, otherwise the current solution will be used.

Synopsis

```
objval = problem.calcobjn(objidx, solution)
```

Arguments

`objidx`	Index of the objective to calculate.
`solution`	Array of length `problem.attributes.cols` that holds the solution.

Further information

The calculations are always carried out in the original problem, even if the problem is currently presolved.

Related topics

`problem.setObjective`, `problem.calcObjN`

problem.calcobjective

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.calcObjective` *instead.*
Returns the objective value of a given solution.

Synopsis

```
objval = problem.calcobjective(solution)
```

Argument

`solution` Array of length `problem.attributes.cols` that holds the solution.

Further information

The calculations are always carried out in the original problem, even if the problem is currently presolved.

Related topics

`problem.calcSlacks`, `problem.calcReducedCosts`, `problem.calcObjective`.

problem.calcreducedcosts

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.calcReducedCosts` *instead.* Returns the reduced cost values for a given (row) dual solution.

Synopsis

```
problem.calcreducedcosts(duals, solution, djs)
```

Arguments

`duals`	Array of length `problem.attributes.rows` that holds the dual solution to calculate the reduced costs for.
`solution`	Optional array of length `problem.attributes.cols` that holds the primal solution. This is necessary for quadratic problems.
`djs`	Array of length `problem.attributes.cols` in which the calculated reduced costs are returned.

Example

```
p = xpress.problem()
p.read("silly_walks.lp") # assume problem has 4 constraints
dj = []
p.calcreducedcosts([0,1,1,1], None, dj)
print("red. cost:", dj)
```

Further information

1. The calculations are always carried out in the original problem, even if the problem is currently presolved.
2. If using the function during a solve (e.g. from a callback), use ORIGINALCOLS and ORIGINALROWS to retrieve the non-presolved dimensions of the problem.

Related topics

`problem.calcSlacks`, `problem.calcObjective`, `problem.calcReducedCosts`.

problem.calcslacks

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.calcSlacks` *instead.*
Calculates the row slack values for a given solution.

Synopsis

```
problem.calcslacks(solution, slacks)
```

Arguments

`solution`	Array of length `problem.attributes.cols` that holds the solution to calculate the slacks for.
`slacks`	Array of length `problem.attributes.rows` in which the calculated row slacks are returned.

Further information

1. The calculations are always carried out in the original problem, even if the problem is currently presolved.
2. If using the function during a solve (e.g. from a callback), use ORIGINALCOLS and ORIGINALROWS to retrieve the non-presolved dimensions of the problem.

Related topics

`problem.calcReducedCosts`, `problem.calcObjective`, `problem.calcSlacks`.

problem.calcsolinfo

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.calcSolInfo` *instead.*
Returns the required property of a solution, like maximum infeasibility of a given primal and duals solution.

Synopsis

```
val = problem.calcsolinfo(solution, duals, property)
```

Arguments

`solution`	Array of length `problem.attributes.cols` that holds the solution.
`duals`	Array of length `problem.attributes.rows` that holds the duals solution.
`property`	`xpress.SolInfo.ABSPRIMALINFEAS` absolute primal infeasibility. `xpress.SolInfo.RELPRIMALINFEAS` relative primal infeasibility. `xpress.SolInfo.ABSDUALINFEAS` absolute duals infeasibility. `xpress.SolInfo.RELDUALINFEAS` relative duals infeasibility. `xpress.SolInfo.MAXMIPFRACTIONAL` absolute MIP infeasibility (fractionality).

Further information

The calculations are always carried out in the original problem, even if the problem is currently presolved.

Related topics

`problem.calcSlacks`, `problem.calcObjective`, `problem.calcReducedCosts`, `problem.calcSolInfo`.

problem.cascade

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpCascadeSol` *instead.*
Re-calculate consistent values for SLP variables based on the current values of the remaining variables

Synopsis

```
problem.cascade()
```

Example

The following example changes the solution value for column 91, and then re-calculates the values of those dependent on it.

```
colnum = 91
(a,b,c,d,e,f,value,h,i,j,k,l,m,n,o) = p.getvar(colnum)

value += 1.42

p.chgvar(col=colnum)

p.cascade()
```

`problem.getvar` and `problem.chgvar` are being used to get and change the current value of a single variable. Provided no other values have been changed since the last execution of `cascade`, values will be changed only for variables which depend on column 91.

Further information

See the section on cascading for an extended discussion of the types of cascading which can be performed.

`cascade` is called automatically during the SLP iteration process and so it is not normally necessary to perform an explicit cascade calculation.

The variables are re-calculated in accordance with the order generated by `problem.slpCascadeOrder`.

Related topics

`problem.slpCascadeOrder`, `problem.slpCascadeSol`

problem.cascadeorder

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpCascadeOrder` *instead.*
Establish a re-calculation sequence for SLP variables with determining rows.

Synopsis

```
problem.cascadeorder()
```

Example

Assuming that all variables are SLP variables, the following example sets default values for the variables, creates the re-calculation order and then calls `problem.slpCascadeSol` to calculate consistent values for the dependent variables.

```
for colnum in range(1, nCol):
  p.chgvar(col=colnum, value=DefaultValue[ColNum])
p.cascadeorder()
p.cascade()
```

Further information

`cascadeorder` is called automatically at the start of the SLP iteration process and so it is not normally necessary to perform an explicit cascade ordering.

Related topics

`problem.slpCascadeSol`, `problem.slpCascadeOrder`

problem.chgbounds

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.chgBounds` *instead.*
Changes the bounds on columns in the problem.

Synopsis

```
problem.chgbounds(colind, bndtype, bndval)
```

Arguments

`colind` Array containing the columns (i.e. `xpress.var` objects, indices, or names) on which the bounds will change.

`bndtype` Character array indicating the type of bound to change:
- `U` indicates a change in the upper bound;
- `L` indicates a change in the lower bound;
- `B` indicates a change in both bounds, i.e. the column is fixed.

`bndval` Array giving the new bound values.

Example

The following changes the lower bound of variable `v1` to 2, upper bound of variable `v2` to 5, and fixes variable `v3` to 3:

```
p.chgbounds([v1,v2,v3],['L','U','B'],[2,5,3])
```

Further information

1. A column may appear twice in the `colind` array so it is possible to change both the upper and lower bounds on a variable in one go.
2. `chgbounds` may be applied to the problem in a presolved state, in which case it expects references to the presolved problem.
3. The constant `xpress.infinity` can be used to represent plus and minus infinity in the bound (`bndval`) array.
4. If the upper bound on a binary variable is changed to be greater than 1 or the lower bound is changed to be less than 0 then the variable will become an integer variable.

Related topics

`problem.getLB`, `problem.getUB`, `problem.chgBounds`.

problem.chgcoef

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.chgCoef` *instead.*
Changes a single coefficient in the problem. If the coefficient does not already exist, a new coefficient will be added to the problem. If many coefficients are being added to a row of the problem, it may be more efficient to delete the old row and add a new row.

Synopsis

```
problem.chgcoef(row, col, coef)
```

Arguments

`row`	Row (i.e. `xpress.constraint` object, index, or name) for the coefficient.
`col`	Column (i.e. `xpress.var` object, index, or name) for the coefficient.
`coef`	New value for the coefficient. If `coef` is zero, any existing coefficient will be deleted.

Example

In the following, the constraint is introduced in the problem and then its linear coefficient for `x` is changed to 3:

```
p = xpress.problem()
x = p.addVariable()
c = x + x**2 <= 3
p.addConstraint(c)
p.chgcoef(c,x,3)
```

Further information

`problem.chgMCoef` is more efficient than multiple calls to `chgcoef` and should be used in its place in such circumstances.

Related topics

`problem.addCols`, `problem.addRows`, `problem.chgMCoef`, `problem.chgMQObj`, `problem.chgObj`, `problem.chgQObj`, `problem.chgRHS`, `problem.getCols`, `problem.getRows`, `problem.chgCoef`.

problem.chgcoltype

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.chgColType` *instead.*
Changes the type of a column in the problem.

Synopsis

```
problem.chgcoltype(colind, coltype)
```

Arguments

`colind` Array containing the columns (i.e. `xpress.var` objects, indices, or names) whose type is to be changed.

`coltype` Character array giving the new column types:

- `C` indicates a continuous column;
- `B` indicates a binary column;
- `I` indicates an integer column.
- `S` indicates a semi–continuous column. The semi–continuous lower bound will be set to `1.0`.
- `R` indicates a semi–integer column. The semi–integer lower bound will be set to `1.0`.
- `P` indicates a partial integer column. The partial integer limit will be set to `1.0`.

Example

The following changes the type of variable `x` from binary to integer:

```
p = xpress.problem()
x = p.addVariable(vartype=xp.binary)
p.chgcoltype([x],['I'])
```

Further information

1. The column types can only be changed before the tree search is started.
2. Calling `chgcoltype` to change any variable into a binary variable causes the bounds previously defined for the variable to be deleted and replaced by bounds of 0 and 1.
3. Calling `chgcoltype` to change a continuous variable into an integer variable cause its lower bound to be rounded up to the nearest integer value and its upper bound to be rounded down to the nearest integer value.
4. After changing the type of a column to semi-continuous, semi-integer or partial integer, call `problem.chgGlbLimit` to set the limit value.

Related topics

`problem.addCols`, `problem.chgRowType`, `problem.getColType`, `problem.chgGlbLimit`, `problem.chgColType`.

problem.chgcascadenlimit

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpChgCascadeNLimit` *instead.*

Set a variable specific cascade iteration limit

Synopsis

```
problem.chgcascadenlimit(col, limit)
```

Arguments

`col`	The column corresponding to the SLP variable for which the cascading limit is to be imposed.
`limit`	The new cascading iteration limit.

Further information

A value set by this function will overwrite the value of the control `xslp_cascadenlimit` for this variable. To remove any previous value set by this function, use an iteration limit of 0.

Related topics

`problem.slpCascadeOrder`, `problem.slpChgCascadeNLimit`

problem.setcbcascadeend

Purpose

This subroutine is deprecated and will be removed in a future release.
Set a user callback to be called at the end of the cascading process, after the last variable has been cascaded

Synopsis

```
problem.setcbcascadeend(callback, data)
value = callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called at the end of the cascading process. `callback` returns an integer value. The return value is noted by Xpress SLP but it has no effect on the optimization.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbcascadeend`.
`data`	User-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Example

The following example sets up a callback to be executed at the end of the cascading process which checks if any of the values have been changed significantly:

```
csol = [1,2,3,4]
p.setcbcascadeend(CBCascEnd, csol)
```

A suitable callback function might resemble this:

```
def CBCascEnd(prob, obj):
  for iCol in range(prob.controls.cols):
    (a,b,c,s,d,e,f,value,g,h,i,j,k,l,m,n) = prob.getvar(iCol)
    if abs(value - obj[iCol]) > .01:
      print("Col {0} changed from {1} to {2}".format(iCol, obj[iCol], value)
  return 0
```

The `obj` argument is used here to hold the original solution values.

Further information

This callback can be used at the end of the cascading, when all the solution values have been recalculated.

Related topics

`problem.slpCascadeSol`, `problem.setcbcascadestart`, `problem.setcbcascadevar`, `problem.setcbcascadevarfail`

problem.setcbcascadestart

Purpose

This subroutine is deprecated and will be removed in a future release.
Set a user callback to be called at the start of the cascading process, before any variables have been cascaded

Synopsis

```
problem.setcbcascadestart(callback, data)
retval = callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called at the start of the cascading process. `callback` returns an integer value. If the return value is nonzero, the cascading process will be omitted for the current SLP iteration, but the optimization will continue.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbcascadestart`.
`data`	A user-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Further information

This callback can be used at the start of the cascading, before any of the solution values have been recalculated.

Related topics

`problem.slpCascadeSol`, `problem.setcbcascadeend`, `problem.setcbcascadevar`, `problem.setcbcascadevarfail`

problem.setcbcascadevar

Purpose

This subroutine is deprecated and will be removed in a future release.
Set a user callback to be called after each column has been cascaded

Synopsis

```
problem.setcbcascadevar(callback, data)
retval = callback(my_prob, my_object, colindex)
```

Arguments

`callback`	The function to be called after each column has been cascaded. `callback` returns an integer value. If the return value is nonzero, the cascading process will be omitted for the remaining variables during the current SLP iteration, but the optimization will continue.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `problem.setcbcascadevar`.
`colindex`	The number of the column which has been cascaded.
`data`	User-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Example

The following example sets up a callback to be executed after each variable has been cascaded:

```
obj = []
p.setcbcascadevar(CBCascVar, obj)
```

The following sample callback function resets the value of the variable if the cascaded value is of the opposite sign to the original value:

```
def CBCascVar(myprob, obj, iCol):
(a,b,c,d,e,f,value,g,h,i,j,k,l,m,n) = myprob.getvar(iCol)
  if value * obj[iCol] < 0:
    p.chgvar(col=ColNum, value=obj[iCol])
  return 0
```

The `data` argument is used here to hold the array `cSol` which we assume has been populated with the original solution values.

Further information

This callback can be used after each variable has been cascaded and its new value has been calculated.

Related topics

`problem.slpCascadeSol`, `problem.setcbcascadeend`, `problem.setcbcascadestart`, `problem.setcbcascadevarfail`

problem.setcbcascadevarfail

Purpose

This subroutine is deprecated and will be removed in a future release.
Set a user callback to be called after cascading a column was not successful

Synopsis

```
problem.setcbcascadevarfail(callback, data)
retval = callback(my_prob, my_object, colindex)
```

Arguments

`callback`	The function to be called after cascading a column was not successful. `callback` returns an integer value. If the return value is nonzero, the cascading process will be omitted for the remaining variables during the current SLP iteration, but the optimization will continue.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbcascadevarfail`.
`colindex`	The number of the column which has been cascaded.
`data`	A user-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Further information

This callback can be used to provide user defined updates for SLP variables having a determining row that were not successfully cascaded due to the determining row being close to singular around the current values. This callback will always be called in place of the cascadevar callback in such cases, and in no situation will both the cascadevar and the cascadevarfail callback be called in the same iteration for the same variable.

Related topics

`problem.slpCascadeSol`, `problem.setcbcascadeend`, `problem.setcbcascadestart`, `problem.setcbcascadevar`

problem.setcbcoefevalerror

Purpose

This subroutine is deprecated and will be removed in a future release.
Set a user callback to be called when an evaluation of a coefficient fails during the solve

Synopsis

```
problem.setcbcoefevalerror(callback, data)
retval = callback(my_prob, my_object, rowindex, colindex)
```

Arguments

`callback`	The function to be called when an evaluation fails.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbcoefevalerror`.
`rowindex`	The row position of the coefficient.
`colindex`	The column position of the coefficient.
`data`	A user-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Further information

This callback can be used to capture when an evaluation of a coefficient fails. The callback is called only once for each coefficient.

Related topics

`problem.nlpPrintEvalInfo`

problem.setcbconstruct

Purpose

This subroutine is deprecated and will be removed in a future release.
Set a user callback to be called during the Xpress SLP augmentation process

Synopsis

```
problem.setcbconstruct(callback, data)
retval = callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called during problem augmentation. `callback` returns an integer value. See below for an explanation of the values.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbconstruct`.
`data`	A user-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Example

The following example sets up a callback to be executed during the Xpress SLP problem augmentation:

```
value = []
p.setcbconstruct(CBConstruct, value)
```

The following sample callback function sets values for the variables the first time the function is called and returns to `problem.slpConstruct` to recalculate the initial matrix. The second time it is called it frees the allocated memory and returns to `problem.slpConstruct` to proceed with the rest of the augmentation.

```
def CBConstruct(myprob, obj):
  if obj is None:
    n = myprob.attributes.cols
    cValue = n * [0]
    # initialize with values (not shown here)
    for i in range(n):
    # store into SLP structures
    myprob.chgvar(col=i, value=cValue[i])
    # set Object non-None to indicate we have processed data
    obj = cValue
    return -1
  else:
    obj = None
  return 0
```

Further information

This callback can be used during the problem augmentation, generally (although not exclusively) to change the initial values for the variables.

The following return codes are accepted:

`0`	Normal return: augmentation continues
`-1`	Return to recalculate matrix values
`-2`	Return to recalculate row weights and matrix entries
`other`	Error return: augmentation terminates, `problem.slpConstruct` terminates with a nonzero error code.

The return values -1 and -2 will cause the callback to be called a second time after the matrix has been recalculated. It is the responsibility of the callback to ensure that it does ultimately exit with a return value of zero.

Related topics

`problem.slpConstruct`

problem.setcbdestroy

Purpose

This subroutine is deprecated and will be removed in a future release.
Set a user callback to be called when an SLP problem is about to be destroyed

Synopsis

```
problem.setcbdestroy(callback, data)
callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called when the SLP problem is about to be destroyed. `callback` returns an integer value. At present the return value is ignored.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbdestroy`.
`data`	A user-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Example

The following example sets up a callback to be executed before the SLP problem is destroyed:

```
p.setcbdestroy(CBDestroy, cSol)
```

The following sample callback function frees the memory associated with the user-defined object:

```
def CBDestroy(myprob, Obj):
  if Obj is not None:
    Obj.inuse = 0
  return 0
```

The `Obj` argument is used here to hold the array `cSol` which we assume was assigned using one of the `malloc` functions.

Further information

This callback can be used when the problem is about to be destroyed to free any user-defined resources which were allocated during the life of the problem.

problem.setcbdrcol

Purpose

This subroutine is deprecated and will be removed in a future release.
Set a user callback used to override the update of variables with small determining column

Synopsis

```
problem.setcbdrcol(callback, data)
newvalue = callback(my_prob, my_object, colindex, drcolindex, drcolvalue, vlb, vub)
```

Arguments

`callback`	The function to be called after each column has been cascaded. `callback` returns an integer value. If the return value is positive, it will indicate that the value has been fixed, and cascading should be omitted for the variable. A negative value indicates that a previously fixed value has been relaxed. If no action is taken, a 0 return value should be used.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbcascadevar`.
`ColIndex`	The column (i.e., `xpress.var` object, index, or name) for which the determining columns is checked.
`DrColIndex`	The index of the determining column for the column that is being updated.
`DrColValue`	The value of the determining column in the current SLP iteration.
`NewValue`	Used to return the new value for column `ColIndex`, should it need to be updated, in which case the callback must return a positive value to indicate that this value should be used.
`VLB`	The original lower bound of column `ColIndex`. The callback provides this value as a reference, should the bound be updated or changed during the solution process.
`VUB`	The original upper bound of column `ColIndex`. The callback provides this value as a reference, should the bound be updated or changed during the solution process.
`data`	A user-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Further information

If set, this callback is called as part of the cascading procedure. Please see the chapter on cascading of the SLP Reference Manual for more information.

Related topics

`xslp_DRCOLTOL`, `problem.slpCascadeSol`, `problem.setcbcascadeend`, `problem.setcbcascadestart`

problem.setcbintsol

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addIntsolCallback` *instead.*

Set a user callback to be called during MISLP when an integer solution is obtained

Synopsis

```
problem.setcbintsol(callback, data)
callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called when an integer solution is obtained.
`data`	A user-defined object, which can be used for any purpose. `data` is passed to `callback` as `my_object`.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbintsol`.

Example

The following example sets up a callback to be executed whenever an integer solution is found during MISLP:

```
cSol = []
p.setcbintsol(CBIntSol, cSol)
```

The following sample callback function saves the solution values for the integer solution just found:

```
def CBIntSol(prob, cSol):
  cSol[:] = prob.getCallbackSolution()
```

Related topics

`problem.addOptnodeCallback`, `problem.addPrenodeCallback`, `problem.addIntsolCallback`

problem.setcbiterend

Purpose

This subroutine is deprecated and will be removed in a future release.
Set a user callback to be called at the end of each SLP iteration

Synopsis

```
problem.setcbiterend(callback, data)
retval = callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called at the end of each SLP iteration. `callback` returns an integer value. If the return value is nonzero, the SLP iterations will stop.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbiterend`.
`data`	User-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Example

The following example sets up a callback to be executed at the end of each SLP iteration. It records the number of LP iterations in the latest optimization and stops if there were fewer than 10:

```
p.setcbiterend(CBIterEnd, None)
```

A suitable callback function might resemble this:

```
def CBIterEnd(MyProb, Obj):
  niter = MyProb.attributes.simplexiter
  return (niter < 10)
```

The `Obj` argument is not used here, and so is passed as `None`.

Further information

This callback can be used at the end of each SLP iteration to carry out any further processing and/or stop any further SLP iterations.

Related topics

`problem.setcbiterstart`, `problem.setcbitervar`

problem.setcbiterstart

Purpose

This subroutine is deprecated and will be removed in a future release.
Set a user callback to be called at the start of each SLP iteration

Synopsis

```
problem.setcbiterstart(callback, data)
retval = callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called at the start of each SLP iteration. `callback` returns an integer value. If the return value is nonzero, the SLP iterations will stop.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbiterstart`.
`data`	User-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Example

The following example sets up a callback to be executed at the start of the optimization to save to save the values of the variables from the previous iteration:

```
p.setcbiterstart(CBIterStart, cSol)
```

A suitable callback function might resemble this:

```
def CBIterStart(MyProb, Obj):
  niter = MyProb.attributes.xslp_iter
  if nIter == 0:
    return 0 # no previous solution
  Obj[:] = MyProb.getCallbackSolution()
  return 0
```

The `Obj` argument is used here to hold the array `cSol` which we populate with the solution values.

Further information

This callback can be used at the start of each SLP iteration before the optimization begins.

Related topics

`problem.setcbiterend`, `problem.setcbitervar`

problem.setcbitervar

Purpose

This subroutine is deprecated and will be removed in a future release.
Set a user callback to be called after each column has been tested for convergence

Synopsis

```
problem.setcbitervar(callback, data)
retval = callback(my_prob, my_object, colindex)
```

Arguments

`callback` The function to be called after each column has been tested for convergence. `callback` returns an integer value. The return value is interpreted as a convergence status. The possible values are:

`< 0`	The variable has not converged;
`0`	The convergence status of the variable is unchanged;
`1 to 10`	The column has converged on a system-defined convergence criterion (these values should not normally be returned);
`> 10`	The variable has converged on user criteria.

`my_prob` The problem passed to the callback function.

`my_object` The user-defined object passed as `data` to `setcbitervar`.

`ColIndex` The number of the column which has been tested for convergence.

`data` A user-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Example

The following example sets up a callback to be executed after each variable has been tested for convergence. The user object `Important` is an integer array which has already been set up and holds a flag for each variable indicating whether it is important that it converges.

```
Obj = None
p.setcbitervar(CBIterVar, Obj)
```

The following sample callback function tests if the variable is already converged. If not, then it checks if the variable is important. If it is not important, the function returns a convergence status of 99.

```
def CBIterVar(MyProb, Obj, iCol):
  (a,b,c,d,e,f,g,h,i,converged,j,k,l,m,n) = MyProb.getvar(iCol)
  if converged:
    return 0
  if Obj[iCol]:
    return 99
  return -1
```

The `object` argument is used here to hold the array `Important`.

Further information

This callback can be used after each variable has been checked for convergence, and allows the convergence status to be reset if required.

Related topics

`problem.setcbiterend`, `problem.setcbiterstart`

problem.setcbmessage

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addMessageCallback` *instead.*

Set a user callback to be called whenever Xpress Nonlinear outputs a line of text

Synopsis

```
problem.setcbmessage(callback, data)
callback(my_prob, my_object, msg, msgtype)
```

Arguments

`callback` The function to be called whenever Xpress Nonlinear outputs a line of text. `callback` does not return a value.

`my_prob` The problem passed to the callback function.

`my_object` The user-defined object passed as `data` to `setcbmessage`.

`msg` String to be output.

`msgtype` Type of message. The following are system-defined:

`1`	Information message
`3`	Warning message
`4`	Error message

A negative value indicates that the Optimizer is about to finish and any buffers should be flushed at this time. User-defined values are also possible for `msgtype`.

`data` A user-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Example

The following example creates a log file into which all messages are placed. System messages are also printed on standard output:

```
log = ''
p.setcbmessage(CBMessage, log)
```

A suitable callback function could resemble the following:

```
def CBMessage(Obj, msg, msgtype):
  if msgtype < 0:
    print(log)
    log = ''
    return
  if msgtype >= 1 and msgtype <= 4:
    print(msg)
  else:
    log += msg + ';'
```

Further information

If a user message callback is defined, screen output is automatically disabled.

Output can be directed into a log file by using `problem.setLogFile`.

Also, because of Python's garbage collection functions, it is advised to explicitly delete a problem at the end of its use (with the `del` statement) if a message callback was set for that problem using `setcbmessage`. def CBMessage(Obj, msg, msgtype): pass

p = xp.problem() p.setcbmessage(CBMessage, None) [...] del p

Related topics

`problem.setLogFile`, `problem.addMessageCallback`

problem.setcbmsjobend

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addMsJobEndCallback` *instead.*

Set a user callback to be called every time a new multistart job finishes. Can be used to overwrite the default solution ranking function

Synopsis

```
problem.setcbmsjobend(callback, data)
status = callback(my_prob, my_object, job_object, description)
```

Arguments

`callback`	The function to be called when a new multistart job is created
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbmsjobend`.
`job_object`	Job specific user-defined object, as specified in by the multistart job creating API functions.
`description`	The description of the problem as specified in by the multistart job creating API functions.
`status`	User return status variable: 0 - use the default evaluation of the finished job 1 - disregard the result and continue 2 - stop the multistart search

Further information

The multistart pool is dynamic, and this callback can be used to load new multistart jobs using the normal API functions.

Related topics

`problem.addMsJobStartCallback`, `problem.addMsWinnerCallback`, `problem.addMsJobEndCallback`

problem.setcbmsjobstart

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addMsJobStartCallback` *instead.*

Set a user callback to be called every time a new multistart job is created, and the pre-loaded settings are applied

Synopsis

```
problem.setcbmsjobstart(callback, data)
status = callback(my_prob, my_object, job_object, description)
```

Arguments

`callback`	The function to be called when a new multistart job is created;
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbmsjobstart`.
`job_object`	Job specific user-defined object, as specified in by the multistart job creating API functions.
`description`	The description of the problem as specified in by the multistart job creating API functions.
`status`	User return status variable: 0 - normal return, solve the job, 1 - disregard this job and continue, 2 - Stop multistart.

Further information

All mulit-start jobs operation on an independent copy of the original problem, and any modification to the problem is allowed, including structural changes. Please note however, that any modification will be carried over to the base problem, should a modified problem be declared the winner prob.

Related topics

`problem.addMsJobEndCallback`, `problem.addMsWinnerCallback`, `problem.addMsJobStartCallback`

problem.setcbmswinner

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addMsWinnerCallback` *instead.*

Set a user callback to be called every time a new multistart job is created, and the pre-loaded settings are applied

Synopsis

```
problem.setcbmswinner(callback, data)
callback(my_prob, my_object, job_object, description)
```

Arguments

`callback`	The function to be called when a new multistart job is created
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbmswinner`.
`job_object`	Job specific user-defined object, as specified in by the multistart job creating API functions.
`description`	The description of the problem as specified in by the multistart job creating API functions.

Further information

The multistart pool is dynamic, and this callback can be used to load new multistart jobs using the normal API functions.

Related topics

`problem.addMsJobStartCallback`, `problem.addMsJobEndCallback`, `problem.addMsWinnerCallback`

problem.setcboptnode

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addOptnodeCallback` *instead.*

Set a user callback to be called during MISLP when an optimal SLP solution is obtained at a node

Synopsis

```
problem.setcboptnode(callback, data)
infeas = callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called when an optimal SLP solution is obtained at a node. It must returns an integer value. If the return value is nonzero, or if the feasibility flag is set nonzero, then further processing of the node will be terminated (it is declared infeasible).
`data`	The user-defined object passed as `my_object` to `setcboptnode`.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed to `setcboptnode`.
`infeas`	Integer containing the feasibility flag. If nonzero, the node is declared infeasible.

Example

The following example defines a callback function to be run at each node when an SLP optimal solution is found. If there are significant penalty errors in the solution, the node is declared infeasible.

```
p.setcboptnode(CBOptNode, None)
```

A suitable callback function might resemble the following:

```
def CBOptNode(prob, data) {
  total = prob.attributes.xslp_errorcosts
  objval = prob.attributes.xslp_objval
  if abs(total) > abs(objval) * 0.001 and abs(total) > 1:
    return 1
  else:
    return 0
```

Further information

If a node is declared infeasible from the callback function, the cost of exploring the node further will be avoided.

This callback must be used in place of `setcboptnode` when optimizing with MISLP.

Related topics

`problem.addPrenodeCallback`, `problem.setcbslpnode`, `problem.addOptnodeCallback`

problem.setcbprenode

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addPrenodeCallback` *instead.*

Set a user callback to be called during MISLP after the set-up of the SLP problem to be solved at a node, but before SLP optimization

Synopsis

```
problem.setcbprenode(callback, data)
feas = callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called after the set-up of the SLP problem to be solved at a node. `callback` returns an integer value. If the return value is nonzero, then further processing of the node will be terminated (it is declared infeasible).
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbprenode`.
`feas`	feasibility flag. If `callback` return a nonzero, the node is declared infeasible.

Example

The following example sets up a callback function to be executed at each node before the SLP optimization starts. The array `IntList` contains a list of integer variables, and the function prints the bounds on these variables.

```
IntList = [...]
prob.setcbprenode(CBPreNode, IntList)
```

A suitable callback function might resemble the following:

```
def CBPreNode(myProb, intlist):
  for i in intlist:
    LO,UP = [],[]
    myProb.getlb(LO,i,i)
    myProb.getub(UP,i,i)
    lb,ub = LO[0], UP[0]
    if lb > 0 or ub < xp.infinity:
      print("Col {0}: {1} <= {2}".format(i,lb,ub)
  return 0
```

Further information

If a node can be identified as infeasible by the callback function, then the initial optimization at the current node is avoided, as well as further exploration of the node.

Related topics

`problem.addOptnodeCallback`, `problem.setcbslpnode`, `problem.addPrenodeCallback`

problem.setcbpreupdatelinearization

Purpose

This subroutine is deprecated and will be removed in a future release.
Set a user callback to be called before the linearization is updated

Synopsis

```
problem.setcbpreupdatelinearization(callback, data)
ifRepeat = callback(my_prob, my_object, when)
```

Arguments

`callback`	The function to be called before the linearization is updated. If `callback` returns `True`, another call to the callback will be scheduled. If it returns `False`, a final call with `when == -1` will be made.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbpreupdatelinearization`.
`when`	Indicates the call number, starting at 1. A value of -1 indicates the end of the linearization update.

Further information

When the linearization is updated, all user functions are evaluated and their derivatives calculated at the current base point. In some models, it is cheaper to compute the derivatives for all user functions at the same time, thereby avoiding repeated calculations for each function. This callback is intended to be used in such cases.

During each SLP iteration, the callback is invoked repeatedly, with `when` indicating the current call number (starting from 1), until the callback indicates that no further calls are needed, by returning `False`. Between each callback invocation, the solver evaluates the user functions without requesting derivatives. After the callback has returned `False`, the user functions are evaluated one more time, this time requesting derivatives, and then finally the callback is called with `when == -1`, marking the end of the linearization update. The only time derivatives will be requested outside of this sequence is during KKT validation. This can be disabled during the solve by clearing the `XSLP_CONVERGEBIT_VALIDATION_K` bit in `XSLP_CONVERGENCEOPS`, ensuring that derivatives can always be precomputed.

One way that this callback can be used to precompute derivatives for user functions is as follows:

1. On each SLP iteration, the callback is first called with `when = 1`. This is a signal that derivatives will be needed soon. The callback sets a flag to indicate that user functions should capture their input values, and returns `True` to request another call.
2. When the callback returns, the user functions are evaluated without requesting derivatives. Each user function captures its input values somewhere, and returns the correct function value.
3. The callback is called again, with `when == 2`. The callback now computes derivates for all user functions using the captured input values. The callback clears the flag so that user functions no longer capture their input values, and returns `False` to indicate that no further calls are needed.
4. When the callback returns, the user functions are evaluated again. Derivatives are requested, and the user functions return the precomputed derivative values.
5. The callback is invoked one more time for this iteration with `when == -1`, marking the end of the linearization update. User functions should behave normally from this point.

problem.setcbslpend

Purpose

This subroutine is deprecated and will be removed in a future release.
Set a user callback to be called at the end of the SLP optimization

Synopsis

```
problem.setcbslpend(callback, data)
callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called at the end of the SLP optimization. `callback` returns an integer value. If the return value is nonzero, the optimization will return an error code and the "User Return Code" error will be set.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbslpend`.
`data`	A user-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Example

The following example sets up a callback to be executed at the end of the SLP optimization. It frees the memory allocated to the object created when the optimization began:

```
ObjData = None
p.setcbslpend(CBSlpEnd, ObjData)
```

A suitable callback function might resemble this:

```
def CBSlpEnd(MyProb, Obj):
  if Obj is not None:
    Obj = []
  return 0
```

Further information

This callback can be used at the end of the SLP optimization to carry out any further processing or housekeeping before the optimization function returns.

Related topics

`problem.setcbslpstart`

problem.setcbslpnode

Purpose

This subroutine is deprecated and will be removed in a future release.
Set a user callback to be called during MISLP after the SLP optimization at each node.

Synopsis

```
problem.setcbslpnode(callback, data)
(retval, infeas) = callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called after the set-up of the SLP problem to be solved at a node. `callback` returns an integer value. If the return value is nonzero, or if the feasibility flag is set nonzero, then further processing of the node will be terminated (it is declared infeasible).
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbslpnode`.
`infeas`	An integer containing the feasibility flag. If `callback` sets the flag nonzero, the node is declared infeasible.
`data`	A user-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Example

The following example sets up a callback function to be executed at each node after the SLP optimization finishes. If the solution value is worse than a target value (referenced through the user object), the node is cut off (it is declared infeasible).

```
objtarget = []
p.setcbslpnode(CBSLPNode, objtarget)
```

A suitable callback function might resemble the following:

```
def CBSLPNode(my_prob, my_obj):
  lpval = my_prob.attributes.lpobjval
  return (0, (lpval < my_obj))
```

Further information

If a node can be cut off by the callback function, then further exploration of the node is avoided.

problem.setcbslpstart

Purpose

This subroutine is deprecated and will be removed in a future release.
Set a user callback to be called at the start of the SLP optimization

Synopsis

```
problem.setcbslpstart(callback, data)
retval = callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called at the start of the SLP optimization. `callback` returns an integer value. If the return value is nonzero, the optimization will not be carried out.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbslpstart`.
`data`	User-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Example

The following example sets up a callback to be executed at the start of the SLP optimization:

```
Objdata = []
p.setcbslpstart(CBSlpStart, Objdata)
```

A suitable callback function might resemble this:

```
def CBSlpStart(object):
  object.append(1)
  return 0
```

Further information

This callback can be used at the start of the SLP optimization to carry out any housekeeping before the optimization actually starts. Note that a nonzero return code from the callback will terminate the optimization immediately.

Related topics

`problem.setcbslpend`

problem.slpchgcoefstr

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpChgCoefStr` *instead.*
Add or change a single matrix coefficient using a string for the formula

Synopsis

```
problem.slpchgcoefstr(row, col, factor, formula)
```

Arguments

`row`	The row (i.e. `xpress.constraint` object, index, or name) for the coefficient.
`col`	The column (i.e. `xpress.var` object, index, or name) for the coefficient.
`factor`	Constant multiplier for the formula. If `factor` is `None`, a value of 1.0 will be used.
`formula`	String holding the formula, with the tokens separated by spaces.

Example

Assuming that the columns of the matrix are named `Col1`, `Col2`, etc, the following example puts the formula `2.5*sin(Col1)` into the coefficient in row 1, column 3.

```
Formula = "sin ( Col1 )"
Factor = 2.5
p.slpchgcoefstr(1, 3, Factor, Formula)
```

Note that all the tokens in the formula (including mathematical operators and separators) are separated by one or more spaces.

Further information

If the coefficient already exists as a constant or formula, it will be changed into the new coefficient. If it does not exist, it will be added to the problem.

A coefficient is made up of two parts: `Factor` and `Formula`. `Factor` is a constant multiplier which can be provided in the `Factor` variable. If Xpress Nonlinear can identify a constant factor in the Formula, then it will use that as well, to minimize the size of the formula which has to be calculated.

This function can only be used if all the operands in the formula can be correctly identified as constants, existing columns, character variables or functions. Therefore, if a formula refers to a new column, that new item must be added to the Xpress Nonlinear problem first.

Related topics

`problem.slpAddCoefs`, `problem.slpDelCoefs`, `problem.slpChgCoef`, `problem.slpGetCoefStr`, `problem.slpLoadCoefs`, `problem.slpChgCoefStr`

problem.chgccoef

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpChgCoefStr` *instead.*
Add or change a single matrix coefficient using a string for the formula

Synopsis

```
problem.chgccoef(row, col, factor, formula)
```

Arguments

`row`	The row (i.e. `xpress.constraint` object, index, or name) for the coefficient.
`col`	The column (i.e. `xpress.var` object, index, or name) for the coefficient.
`factor`	Constant multiplier for the formula. If `factor` is `None`, a value of 1.0 will be used.
`formula`	String holding the formula, with the tokens separated by spaces.

Example

Assuming that the columns of the matrix are named `Col1`, `Col2`, etc, the following example puts the formula `2.5*sin(Col1)` into the coefficient in row 1, column 3.

```
Formula = "sin ( Col1 )"
Factor = 2.5
p.chgccoef(1, 3, Factor, Formula)
```

Note that all the tokens in the formula (including mathematical operators and separators) are separated by one or more spaces.

Further information

If the coefficient already exists as a constant or formula, it will be changed into the new coefficient. If it does not exist, it will be added to the problem.

A coefficient is made up of two parts: `Factor` and `Formula`. `Factor` is a constant multiplier which can be provided in the `Factor` variable. If Xpress Nonlinear can identify a constant factor in the Formula, then it will use that as well, to minimize the size of the formula which has to be calculated.

This function can only be used if all the operands in the formula can be correctly identified as constants, existing columns, character variables or functions. Therefore, if a formula refers to a new column, that new item must be added to the Xpress Nonlinear problem first.

Related topics

`problem.slpChgCoefStr`, `problem.slpAddCoefs`, `problem.slpDelCoefs`, `problem.slpChgCoef`, `problem.slpGetCoefStr`, `problem.slpLoadCoefs`

problem.chgdeltatype

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpChgDeltaType` *instead.*
Changes the type of the delta assigned to a nonlinear variable

Synopsis

```
problem.chgdeltatype(varind, deltatypes, values)
```

Arguments

`varind` Array containing the columns (i.e. xpress.var objects, indices, or names) to change the deltas for.

`deltatypes` Type of the delta variable:

`0`	Differentiable variable, default.
`1`	Variable where a minimum perturbation size given in `values` may be required before a significant change in the problem is achieved.
`2`	Variable defined over the grid size given in `values`.
`3`	Variable where a meaningful step size should automatically be detected, with an upper limit given in `values`.

`values` Grid or minimum step sizes for the variables.

Example

The following example forces the variables `b1` and `b2` to be evaluated for integer points:

```
problem.nlpsetinitval([b1, b2], [2, 2], [1.0, 1.0])
```

Further information

Changing the delta type of a variables makes the variable nonlinear.

Related topics

`,problem.slpChgDeltaType`

problem.chgdf

Purpose

This subroutine is deprecated and will be removed in a future release.
Set or change a distribution factor

Synopsis

```
problem.chgdf(col, row, value)
```

Arguments

`col`	The column (i.e., `xpress.var` object, index, or name) whose distribution factor is to be set or changed.
`row`	The row (i.e., `xpress.constraint` object, index, or name) where the distribution applies.
`value`	The new value of the distribution factor. May be `None` if not required.

Example

The following example retrieves the value of the distribution factor for column 282 in row 134 and changes it to be twice as large.

```
value = p.getdf(282,134)
value *= 2
p.chgdf(282,134,value)
```

Further information

The *distribution factor* of a column in a row is the matrix coefficient of the corresponding delta vector in the row. Distribution factors are used in conventional recursion models, and are essentially normalized first-order derivatives. Xpress Nonlinear can accept distribution factors instead of initial values, provided that the values of the variables involved can all be calculated after optimization using determining rows, or by a callback.

Related topics

`problem.adddfs`, `problem.getdf`, `problem.loaddfs`

problem.chgglblimit

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.chgGlbLimit` *instead.*
Changes semi-continuous or semi-integer lower bounds, or upper limits on partial integers.

Synopsis

```
problem.chgglblimit(colind, limit)
```

Arguments

`colind`	Array containing the indices of the semi-continuous, semi-integer or partial integer columns that should have their limits changed.
`limit`	Array giving the new limit values.

Further information

1. The new limits are not allowed to be negative.
2. Partial integer limits can be at most 2^{28}.

Related topics

`problem.chgColType`, `problem.getMIPEntities`, `problem.chgGlbLimit`.

problem.chgmcoef

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.chgMCoef` *instead.* Change multiple coefficients in the problem. The coefficients that do not exist yet will be added to the problem. If many coefficients are being added to a row of the matrix, it may be more efficient to delete the old row of the matrix and add a new one.

Synopsis

```
problem.chgmcoef(rowind, colind, rowcoef)
```

Arguments

`rowind`	Array containing the rows (i.e., `xpress.constraint` objects, indices, or names) of the coefficients to be changed.
`colind`	Array containing the columns (i.e., `xpress.var` objects, indices, or names) of the coefficients to be changed.
`rowcoef`	Array containing the new coefficient values. If an element of `rowcoef` is zero, the coefficient will be deleted.

Example

```
con1 = x + y + z <= 2
con2 = x + y >= 1
con3 = x + 3*y == 1
p.addVariable(x,y,z)
p.addConstraint(con1, con2, con3)
p.chgmcoef([con1,con1,con1,con2,con3], [x,y,z,x,x], [-2, -3, -3.2, 1, 3])
```

This changes five coefficients, three of which in the first constraint and one in each of the second and third constraints.

Further information

`chgmcoef` is more efficient than repeated calls to `problem.chgCoef` and should be used in its place if many coefficients are to be changed.

Related topics

`problem.chgCoef`, `problem.chgMQObj`, `problem.chgObj`, `problem.chgQObj`, `problem.chgRHS`, `problem.getCols`, `problem.getRHS`, `problem.chgMCoef`.

problem.chgobjn

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.chgObjN` instead.
Modifies one or more coefficients of an objective function in a multi-objective problem. If the objective already exists, any coefficients not present in the `colind` and `objcoef` arrays will unchanged. If the objective does not exist, it will be added to the problem.

Synopsis

```
problem.chgobjn(objidx, colind, objcoef)
```

Arguments

`objidx`	Index of the objective function to add or modify.
`colind`	Integer array of length `ncols` containing the indices of the columns whose objective coefficients will change. An index of `-1` indicates that the fixed part of the objective function on the right hand side should change.
`objcoef`	Double array of length `ncols` giving the new objective function coefficients.

Example

Changing three coefficients of the first objective function:

```
colind = [0, 2, 5]
objcoef = [25.0, 5.3, 0.0]
p.chgobjn(0, colind, objcoef)
```

Further information

1. When `objidx=0`, this function is equivalent to `problem.chgObj`.
2. Any objectives with `idx < objidx` that do not already exist will be added to the problem with all zero coefficients.

Related topics

`problem.addObjective`, `problem.setObjective`, `problem.addObj`, `problem.getObjN`, `problem.delObj`, `problem.chgObj`, `problem.chgObjN`.

problem.chgmqobj

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.chgMQObj` *instead.*
Change multiple quadratic coefficients in the objective function. If any of the coefficients does not exist already, new coefficients will be added to the objective function.

Synopsis

```
problem.chgmqobj(objqcol1, objqcol2, objqcoef)
```

Arguments

`objqcol1`	Array containing the column index of the first variable in each quadratic term.
`objqcol2`	Array containing the column index of the second variable in each quadratic term.
`objqcoef`	New values for the coefficients. If an entry in `objqcoef` is `0`, the corresponding entry will be deleted. These are the coefficients of the lower triangular part of the Hessian of the objective function.

Example

The following code results in an objective function with terms: $[15x_1^2 + 7x_1x_2]/2$

```
p.chgmqobj([x1,x1], [x1,x2], [15,3.5])
```

Further information

1. If `objqcol1[t]` is not equal to `objqcol2[t]`, then both the matrix elements `(objqcol1[t], objqcol2[t])` and `(objqcol2[t], objqcol1[t])` are changed to leave the Hessian symmetric.
2. The quadratic matrix coefficients are implicitly divided by two. But since the coefficients for $x_i x_j$ (`i` not equal to `j`) appear twice, only the coefficients for x_i^2 should be multiplied by two in the `objqcoef` array to account for the implicit division. See the example above.
3. `chgmqobj` is more efficient than repeated calls to `problem.chgQObj` and should be used in its place when several coefficients are to be changed.

Related topics

`problem.chgCoef`, `problem.chgMCoef`, `problem.chgObj`, `problem.chgQObj`, `problem.getQObj`, `problem.chgMQObj`.

problem.chgnlcoef

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpChgCoef` *instead.*
Add or change a single matrix coefficient using a parsed or unparsed formula

Synopsis

```
problem.chgnlcoef(row, col, factor, parsed, type, value)
```

Arguments

`row`	The index of the matrix row for the coefficient.
`col`	The index of the matrix column for the coefficient.
`factor`	The constant multiplier for the formula. If `factor` is `None`, a value of 1.0 will be used.
`parsed`	Boolean indicating the whether the token arrays are formatted as internal unparsed (`parsed=False`) or internal parsed reverse Polish (`parsed=True`).
`type`	Array of token types providing the description and formula for each item.
`value`	Array of values corresponding to the types in `type`.

Example

Assuming that the columns of the matrix are named `Col1`, `Col2`, etc, the following example puts the formula `2.5*sin(Col1)` into the coefficient in row 1, column 3.

```
type  = [xp.constants.TOK_IFUN, xp.constants.TOK_COL, xp.constants.TOK_RB, xp.constants.TOK_EO
value = [xp.constants.IFUN_SIN, 1,          0,          0]

Factor = 2.5
p.chgnlcoef(1, 3, Factor, False, type, value)
```

The formula is written in unparsed form (`parsed=False`) and so it is provided as tokens in the same order as they would appear if the formula were written in string form.

Further information

If the coefficient already exists as a constant or formula, it will be changed into the new coefficient. If it does not exist, it will be added to the problem.

A coefficient is made up of two parts: `Factor` and `Formula`. `Factor` is a constant multiplier which can be provided in the `factor` variable. If Xpress Nonlinear can identify a constant factor in the Formula, then it will use that as well, to minimize the size of the formula which has to be calculated.

Related topics

`problem.slpChgCoef`, `problem.slpAddCoefs`, `problem.slpChgCoefStr`, `problem.slpDelCoefs`, `problem.slpGetCoefStr`, `problem.slpLoadCoefs`,`problem.slpChgCoef`

problem.slpchgcoef

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpChgCoef` *instead.*
Add or change a single matrix coefficient using a parsed or unparsed formula

Synopsis

```
problem.slpchgcoef(row, col, factor, parsed, type, value)
```

Arguments

`row`	The index of the matrix row for the coefficient.
`col`	The index of the matrix column for the coefficient.
`factor`	The constant multiplier for the formula. If `factor` is `None`, a value of 1.0 will be used.
`parsed`	Boolean indicating the whether the token arrays are formatted as internal unparsed (`parsed=False`) or internal parsed reverse Polish (`parsed=True`).
`type`	Array of token types providing the description and formula for each item.
`value`	Array of values corresponding to the types in `type`.

Example

Assuming that the columns of the matrix are named `Col1`, `Col2`, etc, the following example puts the formula `2.5*sin(Col1)` into the coefficient in row 1, column 3.

```
type  = [xp.constants.TOK_IFUN, xp.constants.TOK_COL, xp.constants.TOK_RB, xp.constants.TOK_EO
value = [xp.constants.IFUN_SIN, 1,          0,          0]

Factor = 2.5
p.slpchgcoef(1, 3, Factor, 0, type, value)
```

The formula is written in unparsed form (`parsed=False`) and so it is provided as tokens in the same order as they would appear if the formula were written in string form.

Further information

If the coefficient already exists as a constant or formula, it will be changed into the new coefficient. If it does not exist, it will be added to the problem.

A coefficient is made up of two parts: `Factor` and `Formula`. `Factor` is a constant multiplier which can be provided in the `factor` variable. If Xpress Nonlinear can identify a constant factor in the Formula, then it will use that as well, to minimize the size of the formula which has to be calculated.

Related topics

`problem.slpAddCoefs`, `problem.slpChgCoefStr`, `problem.slpDelCoefs`, `problem.slpGetCoefStr`, `problem.slpLoadCoefs`, `problem.slpChgCoef`

problem.chgobj

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.chgObj` *instead.*
Change the objective function coefficients.

Synopsis

```
problem.chgobj(colind, objcoef)
```

Arguments

`colind`	Array containing the columns (i.e., `xpress.var` objects, indices, or names) on which the range elements will change. An index of `-1` indicates that the fixed part of the objective function on the right hand side should change.
`objcoef`	Array giving the new objective function coefficient.

Example

Changing three coefficients of the objective function with `chgobj`:

```
p.chgobj([x1,x2,x3,-1], [3.5, -2, 0, 224])
```

Further information

The value of the fixed part of the objective function can be obtained using the `OBJRHS` problem attribute.

Related topics

`problem.chgCoef`, `problem.chgMCoef`, `problem.chgMQObj`, `problem.chgQObj`, `problem.getObj`, `problem.chgObj`.

problem.chgobjsense

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.chgObjSense` *instead.*
Changes the problem's objective function objsense to minimize or maximize.

Synopsis

```
problem.chgobjsense(objsense)
```

Argument

`objsense` `xpress.ObjSense.MINIMIZE` or `xpress.ObjSense.MAXIMIZE` to change into a minimization or maximization problem, respectively.

Example

Changing three coefficients of the objective function with `chgobj`:

```
p.chgobjsense(xpress.ObjSense.MAXIMIZE) # optimize in this general direction
```

Related topics

`problem.lpOptimize`, `problem.mipOptimize`, `problem.chgObjSense`.

problem.chgqobj

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.chgQObj` *instead.*
Change a single quadratic coefficient in the objective function corresponding to the variable pair `(objqcol1,objqcol2)` of the Hessian matrix.

Synopsis

```
problem.chgqobj(objqcol1, objqcol2, objqcoef)
```

Arguments

`objqcol1`	Column index for the first variable in the quadratic term.
`objqcol2`	Column index for the second variable in the quadratic term.
`objqcoef`	New value for the coefficient in the quadratic Hessian matrix. If an entry in `objqcoef` is `0`, the corresponding entry will be deleted.

Example

The following code adds the terms $[15x_1^2 + 7x_1x_2]/2$ to the objective function:

```
p.chgqobj(x1, x1, 15)
p.chgqobj(x1, x2, 3.5)
```

Further information

1. If `objqcol1` is not equal to `objqcol2`, then both the matrix elements `(objqcol1, objqcol2)` and `(objqcol2, objqcol1)` are changed to leave the Hessian symmetric.
2. The quadratic matrix coefficients are implicitly divided by two. But since the coefficients for $x_i x_j$ (`i` not equal to `j`) appear twice, only the coefficients for x_i^2 should be multiplied by two in the `objqcoef` argument to account for the implicit division. See the example above.

Related topics

`problem.chgCoef`, `problem.chgMCoef`, `problem.chgMQObj`, `problem.chgObj`, `problem.getQObj`, `problem.chgQObj`.

problem.chgqrowcoeff

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.chgQRowCoeff` *instead.*
Changes a single quadratic coefficient in a row.

Synopsis

```
problem.chgqrowcoeff(row, rowqcol1, rowqcol2, rowqcoef)
```

Arguments

`row`	Row (i.e., `xpress.constraint` object, index, or name) where the quadratic matrix is to be changed.
`rowqcol1`	First index of the coefficient to be changed.
`rowqcol2`	Second index of the coefficient to be changed.
`rowqcoef`	The new coefficient.

Further information

1. This function may be used to add new nonzero coefficients, or even to define the whole quadratic expression with it. Doing that, however, is significantly less efficient than adding the whole expression with `problem.addQMatrix`.
2. The row must not be an equality or a ranged row.

Related topics

`problem.loadproblem`, `problem.getQRowCoeff`, `problem.addQMatrix`, `problem.chgQRowCoeff`, `problem.getQRowQMatrix`, `problem.getQRowQMatrixTriplets`, `problem.getQRows`, `problem.chgQObj`, `problem.chgMQObj`, `problem.getQObj`.

problem.chgrhs

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.chgRHS` instead.
Changes right-hand side values of the problem.

Synopsis

```
problem.chgrhs(rowind, rhs)
```

Arguments

`rowind`	Array containing the rows (i.e., `xpress.constraint` objects, indices, or names) whose right hand side will change.
`rhs`	Array containing the right hand side values.

Example

Here we change the three right hand sides in rows `2`, `6`, and `8` to new values:

```
p.chgrhs([2,8,6], [5, 3.8, 5.7])
```

Related topics

`problem.chgCoef`, `problem.chgMCoef`, `problem.chgRHSRange`, `problem.getRHS`, `problem.getRHSRange`, `problem.chgRHS`.

problem.chgrhsrange

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.chgRHSRange` *instead.*
Change the range for one or more rows of the problem.

Synopsis

```
problem.chgrhsrange(rowind, rng)
```

Arguments

`rowind`	Array containing the rows (i.e., `xpress.constraint` objects, indices, or names) on which the range elements will change.
`rng`	Array containing the range values.

Example

Here, the constraint `cons1` x + y $\leqslant$ 10 is changed to 8 $\leqslant$ x + y $\leqslant$ 10:

```
p.chgrhsrange([cons1], [2])
```

Further information

If the range specified on the row is r, what happens depends on the row type and value of r. It is possible to convert non-range rows using this routine.

Value of r	Row type	Effect
$r \geqslant 0$	$= b, \leqslant b$	$b - r \leqslant \sum a_j x_j \leqslant b$
$r \geqslant 0$	$\geqslant b$	$b \leqslant \sum a_j x_j \leqslant b + r$
$r < 0$	$= b, \leqslant b$	$b \leqslant \sum a_j x_j \leqslant b - r$
$r < 0$	$\geqslant b$	$b + r \leqslant \sum a_j x_j \leqslant b$

Related topics

`problem.chgCoef`, `problem.chgMCoef`, `problem.chgRHS`, `problem.getRHSRange`, `problem.chgRHSRange`.

problem.chgrowstatus

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpChgRowStatus` *instead.*
Change the status setting of a constraint

Synopsis

```
problem.chgrowstatus(row, status)
```

Arguments

`row`	The index of the matrix row to be changed.
`status`	The bitmap with the new status settings. If the status is to be changed, always get the current status first (use `problem.slpGetRowStatus`) and then change settings as required. The only settings likely to be changed are: `Bit 11` Set if row must not have a penalty error vector. This is the equivalent of an enforced constraint (SLPDATA type EC).

Example

The following example changes the status of row 9 to be an enforced constraint.

```
status = p.getrowstatus(9)
status = status | (1<<11)
p.chgrowstatus(9, status)
```

Further information

If `status` is `None` the current status will remain unchanged.

Related topics

`problem.slpGetRowStatus`, `problem.slpChgRowStatus`

problem.chgrowtype

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.chgRowType` *instead.*
Changes the type of a row in the problem.

Synopsis

```
problem.chgrowtype(rowind, rowtype)
```

Arguments

`rowind` Array containing the rows (i.e., `xpress.constraint` objects, indices, or names).

`rowtype` Character array giving the new row types:

- `L` indicates a $\leqslant$ row;
- `E` indicates an = row;
- `G` indicates a $\geqslant$ row;
- `R` indicates a range row;
- `N` indicates a free row.

Example

Here two rows are changed to an equality and a free row, respectively:

```
p.chgrowtype([con1, con2], ['E', 'N'])
```

Further information

A row can be changed to a range type row by first changing the row to an `R` or `L` type row and then changing the range on the row using `problem.chgRHSRange`.

Related topics

`problem.addRows`, `problem.chgColType`, `problem.chgRHS`, `problem.chgRHSRange`, `problem.getRowType`, `problem.chgRowType`.

problem.chgrowwt

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpChgRowWt` *instead.*
Set or change the initial penalty error weight for a row

Synopsis

```
problem.chgrowwt(row, weight)
```

Arguments

`row`	The row (i.e., `xpress.constraint` object, index, or name) whose weight is to be set or changed.
`weight`	The new value of the weight. May be `None` if not required.

Example

The following example sets the initial weight of row number 2 to a fixed value of 3.6 and the initial weight of row 4 to a value twice the calculated default value.

```
p.chgrowwt(2, -3.6)
p.chgrowwt(4,2)
```

Further information

A positive value is interpreted as a multiplier of the default row weight calculated by Xpress SLP.

A negative value is interpreted as a fixed value: the absolute value is used directly as the row weight.

The initial row weight is used only when the augmented structure is created. After that, the current weighting can be accessed and changed using `problem.slpGetRowInfo`.

Related topics

`problem.slpGetRowWT`, `problem.slpGetRowInfo`, `problem.slpChgRowWt`

problem.chgtolset

Purpose

This subroutine is deprecated and will be removed in a future release.
Add or change a set of convergence tolerances used for SLP variables

Synopsis

```
problem.chgtolset(tolset, status, tols)
```

Arguments

`tolset`	Tolerance set for which values are to be changed. A zero value for `tolset` will create a new set.
`status`	A bitmap describing which tolerances are active in this set. See below for the settings.
`tols`	Array of 9 values holding the values for the corresponding tolerances.

Example

The following example creates a new tolerance set with the default values for all tolerances except the relative delta tolerance, which is set to 0.005. It then changes the value of the absolute delta and absolute impact tolerances in tolerance set 6 to 0.015

```
Tols = 9*[0]
Tols[2] = 0.005
Status = 1<<2

p.chgtolset(0, 1<<2, Tols)
Tols[1] = 0.015
Tols[5] = 0.015
Status = 1<<1 | 1<<5
p.chgtolset(6, Status, Tols)
```

Further information

The bits in `status` are set to indicate that the corresponding tolerance is to be changed in the tolerance set. The meaning of the bits is as follows:

Entry / Bit	Tolerance	XSLP constant	XSLP bit constant
0	Closure tolerance (TC)	`xslp_TOLSET_TC`	`xslp_TOLSETBIT_TC`
1	Absolute delta tolerance (TA)	`xslp_TOLSET_TA`	`xslp_TOLSETBIT_TA`
2	Relative delta tolerance (RA)	`xslp_TOLSET_RA`	`xslp_TOLSETBIT_RA`
3	Absolute coefficient tolerance (TM)	`xslp_TOLSET_TM`	`xslp_TOLSETBIT_TM`
4	Relative coefficient tolerance (RM)	`xslp_TOLSET_RM`	`xslp_TOLSETBIT_RM`
5	Absolute impact tolerance (TI)	`xslp_TOLSET_TI`	`xslp_TOLSETBIT_TI`
6	Relative impact tolerance (RI)	`xslp_TOLSET_RI`	`xslp_TOLSETBIT_RI`
7	Absolute slack tolerance (TS)	`xslp_TOLSET_TS`	`xslp_TOLSETBIT_TS`
8	Relative slack tolerance (RS)	`xslp_TOLSET_RS`	`xslp_TOLSETBIT_RS`

The xslp_TOLSET constants can be used to access the corresponding entry in the value arrays, while the xslp_TOLSETBIT constants are used to set or retrieve which tolerance values are used for a given SLP variable. The members of the `Tols` array corresponding to nonzero bit settings in `Status` will be used to change the tolerance set. So, for example, if bit 3 is set in `Status`, then `Tols[3]` will replace the current value of the absolute coefficient tolerance. If a bit is not set in `Status`, the value of the corresponding element of `Tols` is unimportant.

Related topics

`problem.addtolsets`, `problem.deltolsets`, `problem.gettolset`, `problem.loadtolsets`

problem.chgvar

Purpose

This subroutine is deprecated and will be removed in a future release.
Define a column as an SLP variable or change the characteristics and values of an existing SLP variable

Synopsis

```
problem.chgvar(col=None, detrow=None, initstepbound=None, stepbound=None,
       penalty=None, damp=None, initial=None, value=None, tolset=None, history=None,
       converged=None, vartype=None)
```

Arguments

`col`	The index of the matrix column.
`detrow`	An integer holding the index of the determining row. Use -1 if there is no determining row. May be `None` if not required.
`initstepbound`	The initial step bound size. May be `None` if not required.
`stepbound`	The current step bound size. Use zero to disable the step bounds. May be `None` if not required.
`penalty`	The weighting of the penalty cost for exceeding the step bounds. May be `None` if not required.
`damp`	The damping factor for the variable. May be `None` if not required.
`initial`	The initial value for the variable. May be `None` if not required.
`value`	The current value for the variable. May be `None` if not required.
`tolset`	The index of the tolerance set for this variable. Use zero if there is no specific tolerance set. May be `None` if not required.
`history`	The history value for this variable. May be `None` if not required.
`converged`	The convergence status for this variable. May be `None` if not required.
`vartype`	A bitmap defining the existence of certain properties for this variable: `Bit 1:` Variable has a delta vector `Bit 2:` Variable has an initial value `Bit 14:` Variable is the reserved "=" column May be `None` if not required.

Example

The following example sets an initial value of 1.42 and tolerance set 2 for column 25 in the matrix.

```
p.chgvar(col=25, initial=1.42, tolset=2,
           vartype=1<<1 | 1<<2)
```

Note that bits 1 and 2 of `vartype` are set, indicating that the variable has a delta vector and an initial value. For columns already defined as SLP variables, use `problem.getvar` to obtain the current value of `vartype` because other bits may already have been set by the system.

Further information

If any of the arguments is `None` then the corresponding information for the variable will be left unaltered. If the information is new (i.e., the column was not previously defined as an SLP variable) then the default values will be used.

Changing `Value`, `History` or `Converged` is only effective during SLP iterations.

Changing `initvalue` and `initstepbound` is only effective before `problem.slpConstruct`. If a value of `xpress.infinity` is used in the value for `stepbound` or `initstepbound`, the delta will never have step bounds applied, and will almost always be regarded as converged.

Related topics

`problem.addvars`, `problem.delvars`, `problem.getvar`, `problem.loadvars`

problem.construct

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpConstruct` *instead.* Create the full augmented SLP matrix and data structures, ready for optimization

Synopsis

```
problem.construct()
```

Example

The following example constructs the augmented matrix and then outputs the result in MPS format to a file called augment.mat

```
# creation and/or loading of data
# precedes this segment of code
p.construct()
p.write("augment","l")
```

The "`l`" flag causes output of the current linear problem (which is now the augmented structure and the current linearization) rather than the original nonlinear problem.

Further information

`construct` adds new rows and columns to the SLP matrix and calculates initial values for the non-linear coefficients. Which rows and columns are added will depend on the setting of `xslp_augmentation`. Names for the new rows and columns are generated automatically, based on the existing names and the string control variables `xslp_xxxformat`.

Once `construct` has been called, no new rows, columns or non-linear coefficients can be added to the problem. Any rows or columns which will be required must be added first. Non-linear coefficients must not be changed; constant matrix elements can generally be changed after `construct`, but not after `problem.presolve` if used.

`construct` is called automatically by the SLP optimization procedure, and so only needs to be called explicitly if changes need to be made between the augmentation and the optimization.

Related topics

`problem.presolve`, `problem.slpConstruct`

problem.copycontrols

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.copyControls` *instead.*
Copies controls defined for one problem to another.

Synopsis

```
problem.copycontrols(src)
```

Argument

`src` The problem from which the controls are copied.

Example

The following turns off presolve for problem `prob1` and then copies this and other control values to the problem `prob2`:

```
prob1 = xpress.problem()
prob2 = xpress.problem()
prob1.controls.presolve = 0
prob2.copycontrols(prob1)
```

Related topics

`problem.copyControls`, `problem.copycallbacks`.

problem.crossoverlpsol

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.crossoverLpSol` *instead.*
Provides a basic optimal solution for a given solution of an LP problem. This function behaves like the crossover after the barrier algorithm.

Synopsis

```
status = problem.crossoverlpsol()
```

Argument

`status` One of:

`0`	The crossover was successful.
`1`	The crossover was not performed because the problem has no solution.

Example

This example loads a problem, loads a solution for the problem and then uses `crossoverlpsol` to find a basic optimal solution.

```
p = xp.problem()
p.read('problem.mps')
status = p.loadlpsol(x, None, dual, None)
status = p.crossoverlpsol()
```

A solution can also be loaded from an ASCII solution file using `problem.readslxsol`.

Further information

1. The crossover performs two phases: a crossover phase for finding a basic solution and a clean-up phase for finding a basic optimal solution. Setting `algaftercrossover` to `0` will allow the crossover to skip the clean-up phase.
2. The given solution is expected to be feasible or nearly feasible, otherwise the crossover may take a long time to find a basic feasible solution. More importantly, the given solution is expected to have a small duality gap. A small duality gap indicates that the given solution is close to the optimal solution. If the given solution is far away from the optimal solution, the clean-up phase may need many simplex iterations to move to a basic optimal solution.

Related topics

`problem.loadLPSol`, `problem.readSlxSol`, `problem.crossoverLpSol`

problem.delcoefs

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpDelCoefs` *instead.*
Delete coefficients from the current problem

Synopsis

```
problem.delcoefs(rowind, colind)
```

Arguments

`rowind`	rows (i.e., `xpress.constraint` objects, indices, or names) of the SLP coefficients to delete.
`colind`	columns (i.e., `xpress.var` objects, indices, or names) of the SLP coefficients to delete.

Related topics

`problem.slpAddCoefs`, `problem.slpChgCoef`, `problem.slpChgCoefStr`, `problem.slpGetCoefStr`,`problem.slpLoadCoefs`, `problem.slpDelCoefs`

problem.delcpcuts

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.delCPCuts` *instead.*
During the branch and bound search, cuts are stored in the cut pool to be applied at descendant nodes. These cuts may be removed from a given node using `problem.delCuts`, but if this is to be applied in a large number of cases, it may be preferable to remove the cut completely from the cut pool. This is achieved using `delcpcuts`.

Synopsis

```
problem.delcpcuts(cuttype, interp, cutind)
```

Arguments

`cuttype`	User defined cut type to match against.
`interp`	Way in which the cut `cuttype` is interpreted: `-1` match all cut types; `1` treat cut types as numbers; `2` treat cut types as bit maps - delete if any bit matches any bit set in `cuttype`; `3` treat cut types as bit maps - delete if all bits match those set in `cuttype`.
`cutind`	Array containing the cuts which are to be deleted.

Related topics

`problem.addCuts`, `problem.delCuts`, `problem.loadCuts`, `problem.delCPCuts`, Section "Working with the cut manager" of the Xpress Optimizer reference manual.

problem.delcuts

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.delCuts` *instead.*
Deletes cuts from the matrix at the current node. Cuts from the parent node which have been automatically restored may be deleted as well as cuts added to the current node using `problem.addCuts` or `problem.loadCuts`. The cuts to be deleted can be specified in a number of ways. If a cut is ruled out by any one of the criteria it will not be deleted.

Synopsis

```
problem.delcuts(basis, cuttype, interp, delta, cutind)
```

Arguments

`basis`	Ensures the basis will be valid if set to `1`. If set to `0`, cuts with non-basic slacks may be deleted.
`cuttype`	User defined type of the cut to be deleted.
`interp`	Way in which the cut `cuttype` is interpreted: `-1` match all cut types; `1` treat cut types as numbers; `2` treat cut types as bit maps - delete if any bit matches any bit set in `cuttype`; `3` treat cut types as bit maps - delete if all bits match those set in `cuttype`.
`delta`	Only delete cuts with an absolute slack value greater than `delta`. To delete all the cuts, this argument should be set to `-xpress.infinity`.
`cutind`	Array containing the cuts which are to be deleted.

Further information

1. It is usually best to drop only those cuts with basic slacks, otherwise the basis will no longer be valid and it may take many iterations to recover an optimal basis. If the `basis` parameter is set to `1`, this will ensure that cuts with non-basic slacks will not be deleted even if the other parameters specify that these cuts should be deleted. It is highly recommended that the `basis` parameter is always set to `1`.
2. The cuts to be deleted can also be specified by the size of the slack variable for the cut. Only those cuts with a slack value greater than the `delta` parameter will be deleted.
3. A list of indices of the cuts to be deleted can also be provided. The list of active cuts at a node can be obtained with the `problem.getCutList` function.
4. This function should be called only from within callback functions set by either `problem.addOptnodeCallback`, `problem.addPrenodeCallback`, `problem.addPreIntsolCallback` or `problem.addNodeLPSolvedCallback`.

Related topics

`problem.addCuts`, `problem.delCPCuts`, `problem.getCutList`, `problem.loadCuts`, `problem.delCuts`, Section "Working with the cut manager" of the Xpress Optimizer reference manual.

problem.delgencons

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.delGenCons` *instead.*
Delete general constraints from a problem.

Synopsis

```
problem.delgencons(conind)
```

Argument

`conind` An integer array containing the general constraints to delete.

Example

In this example, general constraints `0` and `2` are deleted from the problem:

```
conind = [0, 2]
prob.delgencons(conind)
```

Further information

After general constraints have been deleted from a problem, the indices of the remaining constraints are reduced down so that the general constraints are always numbered from `0` to `prob.attributes.gencons - 1` where `prob.attributes.gencons` contains the number of non-deleted general constraints in the problem.

Related topics

`problem.addGenCons`, `problem.getGenCons`, `xpress.And`, `xpress.Or`, `xpress.max`, `xpress.min`, `xpress.abs`, `problem.delGenCons`.

problem.delindicators

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.delIndicators` *instead.*
Delete indicator constraints. This turns the specified rows into normal rows (not controlled by indicator variables).

Synopsis

```
problem.delindicators(first=None, last=None)
```

Arguments

`first`	First row in the range.
`last`	Last row in the range (inclusive).

Example

In this example, if any of the first two rows of the matrix is an indicator constraint, they are turned into normal rows:

```
prob.delindicators(0,1)
```

Further information

This function has no effect on rows that are not indicator constraints.

Related topics

`problem.getIndicators`, `problem.setIndicators`, `problem.delIndicators`.

problem.delpwlcons

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.delPwlCons` *instead.*
Delete piecewise linear constraints from a problem.

Synopsis

```
problem.delpwlcons(pwlind)
```

Argument

`pwlind` An integer array containing the piecewise linear constraints to delete.

Example

In this example, piecewise linear constraints `0` and `2` are deleted from the problem:

```
pwlind = [0,2]
prob.delpwlcons(pwlind)
```

Further information

After piecewise linear constraints have been deleted from a problem, the indices of the remaining constraints are reduced so that the piecewise linear constraints are always numbered from `0` to `problem.attributes.pwlcons - 1` where `problem.attributes.pwlcons` is the problem attribute containing the number of non-deleted piecewise linear constraints in the problem.

Related topics

`problem.addPwlCons`, `problem.getPwlCons`, `xpress.pwl`, `problem.delPwlCons`.

problem.delobj

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.delObj` *instead.*
Removes an objective function from a multi-objective problem. Any objectives with `index > objidx` will be shifted down. Deleting the last objective function in the problem causes all the objective coefficients to be zeroed, but `OBJECTIVES` remains set to `1`.

Synopsis

```
problem.delobj(objidx)
```

Argument

`objidx` Index of the objective to remove.

Example

Removing the second objective function from a problem:

```
p.delobj(1)
```

Related topics

`problem.addObjective`, `problem.setObjective`, `problem.chgObjN`, `problem.addObj`, `problem.getObjN`, `problem.delObj`.

problem.delqmatrix

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.delQMatrix` *instead.*
Deletes the quadratic part of a row or of the objective function.

Synopsis

```
problem.delqmatrix(row)
```

Argument

`row`	Index of row from which the quadratic part is to be deleted.

Further information

If a row index of `-1` is used, the function deletes the quadratic coefficients from the objective function.

Related topics

`problem.addRows`, `problem.delQMatrix`.

problem.deltolsets

Purpose

This subroutine is deprecated and will be removed in a future release.
Delete tolerance sets from the current problem

Synopsis

```
problem.deltolsets(tolind)
```

Argument

`tolind`	Indices of tolerance sets to delete.

Related topics

`problem.addtolsets`, `problem.chgtolset`, `problem.gettolset`, `problem.loadtolsets`

problem.delvars

Purpose

This subroutine is deprecated and will be removed in a future release.
Convert SLP variables to normal columns. Variables must not appear in SLP structures

Synopsis

```
problem.delvars(colind)
```

Argument

`colind` Columns to be converted to linear ones.

Further information

The SLP variables to be converted to linear, non SLP columns must not be in use by any other SLP structure (coefficients, initial value formulae, delayed columns). Use the appropriate deletion or change functions to remove them first.

Related topics

`problem.addvars`, `problem.chgvar`, `problem.getvar`, `problem.loadvars`

problem.dumpcontrols

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.dumpControls` *instead.*
Displays the list of controls and their current value for those controls that have been set to a non default value.

Synopsis

```
problem.dumpcontrols()
```

Related topics

`problem.setDefaults`, `problem.dumpControls`

problem.estimaterowdualranges

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.estimateRowDualRanges` *instead.*

Performs a dual side range sensitivity analysis, i.e., calculates estimates for the possible ranges for dual values.

Synopsis

```
problem.estimaterowdualranges(rowind, iterlim, mindual, maxdual)
```

Arguments

`rowind`	rows (i.e., `xpress.constraint` objects, indices, or names) to analyze.
`iterlim`	Effort limit expressed as simplex iterations per row.
`mindual`	Estimated lower bounds on the possible dual ranges.
`maxdual`	Estimated upper bounds on the possible dual ranges.

Further information

This function may provide better results for individual row dual ranges when called for a larger number of rows.

Related topics

`problem.lpOptimize`, `problem.strongBranch`, `problem.estimateRowDualRanges`

problem.evaluatecoef

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpEvaluateCoef` *instead.*
Evaluate a coefficient using the current values of the variables

Synopsis

```
value = problem.evaluatecoef(row, col)
```

Arguments

`row`	Row (i.e., `xpress.constraint` object, index, or name).
`col`	Column (i.e., `xpress.var` object, index, or name).
`value`	The result of the calculation.

Example

The following example sets the value of column 5 to 1.42 and then calculates the coefficient in row 2, column 3. If the coefficient depends on column 5, then a value of 1.42 will be used in the calculation.

```
p.chgvar(col=5, value=1.42)
value = p.evaluatecoef(2, 3)
```

Further information

The values of the variables are obtained from the solution, or from the `Value` setting of an SLP variable (see `problem.chgvar` and `problem.getvar`).

Related topics

`problem.chgvar`, `problem.nlpEvaluateFormula`, `problem.getvar`, `problem.slpEvaluateCoef`

problem.evaluateformula

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.nlpEvaluateFormula` instead.

Evaluate a formula using the current values of the variables

Synopsis

```
result = problem.evaluateformula(parsed, type, values)
```

Arguments

`parsed`	integer indicating whether the formula of the item is in internal unparsed format (`parsed=False`) or parsed (reverse Polish) format (`parsed=True`).
`type`	Integer array of token types for the formula.
`values`	Array of values corresponding to `Type`.
`result`	The result of the calculation.

Example

The following example calculates the value of column 3 divided by column 6.

```
type   = [xp.constants.TOK_COL, xp.constants.TOK_COL, xp.constants.TOK_OP,    xp.constants.TOK
values = [3,          6,          xp.constants.OP_DIVIDE, 0]

result  = p.evaluateformula(1, type, values)
```

Further information

The formula in `type` and `values` must be terminated by an `xslp_op_eof` token.

The formula cannot include ”complicated” functions, such as user functions which return more than one value.

Related topics

`problem.slpEvaluateCoef`, `problem.nlpEvaluateFormula`

problem.fixmipentities

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.fixMIPEntities` *instead.* Fixes all the MIP entities to the values of the last found MIP solution. This is useful for finding the reduced costs for the continuous variables after the MIP entities have been fixed to their optimal values.

Synopsis

```
problem.fixmipentities(options)
```

Argument

`options` Options for fixing the MIP entities, evaluated as a bit string whose bits have the following meaning:

Bit	Meaning
`0`	If all MIP entities should be rounded to the nearest discrete value in the solution before being fixed.
`1`	If piecewise linear and general constraints should be kept in the problem with only the non-convex decisions (i.e., which part of a non-convex piecewise linear function or which variable attains a maximum) fixed. Otherwise all variables appearing in piecewise linear or general constraints will be fixed.

Example

This example performs a tree search on problem `myprob` and then uses `fixmipentities` before solving the remaining linear problem:

```
p.read("myprob", "")
p.mipoptimize()
p.fixmipentities(1)
p.lpoptimize()
p.writeprtsol()
```

Further information

1. Because of tolerances, it is possible for e.g. a binary variable to be slightly fractional in the MIP solution, where it might have the value `0.999999` instead of being at exactly `1.0`. With `ifround = 0`, such a binary will be fixed at `0.999999`, but with `ifround = 1`, it will be fixed at `1.0`.
2. This command is useful for inspecting the reduced costs of the continuous variables in a problem after the MIP entities have been fixed. Sensitivity analysis can also be performed on the continuous variables in a MIP problem using `problem.rhsSA` or `problem.objSA` after calling `fixmipentities`.

Related topics

`problem.mipOptimize`, `problem.fixMIPEntities`.

problem.fixpenalties

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpFixPenalties` *instead.*
Fixes the values of the error vectors

Synopsis

```
status = problem.fixpenalties()
```

Argument

`status` Return status after fixing the penalty variables: 0 is successful, nonzero otherwise.

Further information

The function fixes the values of all error vectors on their current values. It also removes their objective cost contribution.

The function is intended to support post optimization analysis, by removing any possible direct effect of the error vectors from the dual and reduced cost values.

The `fixpenalties` function will automatically reoptimize the linearization. However, as the XSLP convergence and infeasibility checks (regarding the original non-linear problem) will not be carried out, this function will not update the SLP solution itself. The updated values will be accessible using `problem.getLpSol` instead.

problem.ftran

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.FTran` *instead.*
Pre-multiplies a (column) vector provided by the user by the inverse of the current matrix.

Synopsis

```
problem.ftran(vec)
```

Argument

`vec`	Array of length `problem.attributes.rows` containing the values which are to be multiplied by the basis inverse. The transformed values appear in the array.

Example

To get the (unscaled) tableau column of structural variable number `jcol`, assuming that all arrays have been dimensioned, do the following:

```
y = [0,1,0,0]
p.ftran(y)
print("ftran result:", y)
```

Further information

If the problem is in a presolved state, the function will work with the basis for the presolved problem.

Related topics

`problem.BTran`, `problem.FTran`.

problem.getattribinfo

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getAttribInfo` *instead.* Accesses the id number and the type information of an attribute given its name. An attribute name may be for example `'rows'`. The function will return an id number of 0 and a type value of `notdefined` if the name is not recognized as an attribute name. Note that this will occur if the name is a control name and not an attribute name.

Synopsis

```
(id,type) = problem.getattribinfo(name)
```

Argument

`name`	The name of the attribute to be queried. Names are case-insensitive. A full list of all attributes may be found in the Xpress Optimizer reference manual.

Related topics

`problem.getControlInfo`, `problem.getAttribInfo`.

problem.getbasis

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getBasis` *instead.*
Returns the current basis into the user's data arrays.

Synopsis

```
problem.getbasis(rowstat, colstat)
```

Arguments

`rowstat` Array of length `problem.attributes.rows` to the basis status of the slack, surplus or artificial variable associated with each row. The status will be one of:

- `0` slack, surplus or artificial is non-basic at lower bound;
- `1` slack, surplus or artificial is basic;
- `2` slack or surplus is non-basic at upper bound.
- `3` slack or surplus is super-basic.

May be `None` if not required.

`colstat` Array of length `problem.attributes.cols` to hold the basis status of the columns in the constraint matrix. The status will be one of:

- `0` variable is non-basic at lower bound, or superbasic at zero if the variable has no lower bound;
- `1` variable is basic;
- `2` variable is non-basic at upper bound;
- `3` variable is super-basic.

May be `None` if not required.

Example

The following example minimizes a problem before saving the basis for later:

```
rstatus = []
cstatus = []
p.lpoptimize()
p.getbasis(rstatus, cstatus)
```

Related topics

`problem.getPresolveBasis`, `problem.loadBasis`, `problem.loadPresolveBasis`, `problem.getBasis`.

problem.getbasisval

Purpose

This subroutine is deprecated and will be removed in a future release.
Returns the current basis status for a specific col or row.

Synopsis

```
rstatus, cstatus = problem.getbasisval(row=None, col=None)
```

Arguments

`row`	Row index to get the row basis status for.
`col`	Column index to get the col basis status for.
`rstatus`	The row basis status will be returned, or 0 if `row` was passed as `None`.
`cstatus`	The value of the col basis status, or 0 if `col==None`.

Related topics

`problem.getBasis`, `problem.getPresolveBasis`, `problem.loadBasis`, `problem.loadPresolveBasis`

problem.getccoef

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpGetCoefStr` instead.
Retrieve a single nonlinear matrix coefficient as a formula in a string.

Synopsis

```
(factor, formula) = problem.getccoef(row, col, maxbytes)
```

Arguments

`row`	Integer holding the row index for the coefficient.
`col`	Integer holding the column index for the coefficient.
`maxbytes`	Maximum length of returned formula.

Return value

`factor`	The value of the constant factor multiplying the formula in the coefficient.
`formula`	String containing the formula, in the same format as used for input from a file.

Example

The following example displays the formula for the coefficient in row 2, column 3:

```
(factor, formula) = p.getccoef(2, 3, 60)
```

Further information

If the requested coefficient is constant, then `factor` will be set to 1.0 and the value will be formatted in `formula`.
If the length of the formula would exceed `maxbytes - 1`, the formula is truncated to the last token that will fit.

Related topics

`problem.slpGetCoefStr`, `problem.slpChgCoefStr`, `problem.slpChgCoef`.

problem.getcoef

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getCoef` *instead.*
Returns a single coefficient in the constraint matrix.

Synopsis

```
coef = problem.getcoef(row, col)
```

Arguments

`row`	Row of the constraint matrix.
`col`	Column of the constraint matrix.

Further information

It is quite inefficient to get several coefficients with the `getcoef` function. It is better to use `getcols` or `getrows`.

Related topics

`problem.getCols`, `problem.getRows`, `problem.getCoef`.

problem.getcoefformula

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpGetCoefStr` *instead.*
Retrieve a single nonlinear matrix coefficient as a formula split into tokens

Synopsis

```
(factor, tokencount, type, value) = problem.getcoefformula(row, col, parsed,
       maxtypes)
```

Arguments

`row`	The row index for the coefficient.
`col`	The column index for the coefficient.
`parsed`	Integer indicating whether the formula of the item is to be returned in internal unparsed format (`parsed=False`) or parsed (reverse Polish) format (`parsed=True`).
`maxtypes`	Maximum number of tokens to return, i.e., length of the `type` and `value` arrays.

Return value

`factor`	The value of the constant factor multiplying the formula in the coefficient.
`tokencount`	Number of tokens returned in `type` and `value`.
`type`	Array holding the token types for the formula.
`value`	Array of values corresponding to `type`.

Example

The following example displays the formula for the coefficient in row 2, column 3 in unparsed form:

```
(fac, tc, type, value) = p.getcoefformula(2, 3, 0, 10)
```

Further information

The `type` and `value` arrays are terminated by an `xslp_op_eof` token.
If the requested coefficient is constant, then `factor` will be set to 1.0 and the value will be returned with token type `xslp_op_con`.

Related topics

`problem.slpChgCoefStr`, `problem.slpChgCoef`, `problem.slpGetCoefStr`

problem.getcoefs

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpGetCoefs` *instead.*
Retrieve the list of positions of the nonlinear coefficients in the problem

Synopsis

```
problem.getcoefs(rowind, colind)
```

Arguments

`rowind`	Row positions of the coefficients. May be `None` if not required.
`colind`	Column positions of the coefficients. May be `None` if not required.

Related topics

`problem.slpGetCoefStr`,`problem.slpGetCoefs`

problem.getcolinfo

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpGetColInfo` *instead.*
Get current column information.

Synopsis

```
problem.getcolinfo(infotype, colindex)
```

Arguments

`infotype` Type of information (see below).

`colindex` Column (i.e., `xpress.var` object, index, or name) whose information is to be handled.

Further information

If the data is not available, the type of the returned Info is set to `None`.

The following constants are provided for column information handling:

`xpress.constants.SLPCOLINFO_VALUE` Get the current value of the column

`xpress.constants.SLPCOLINFO_RDJ` Get the current reduced cost of the column

`xpress.constants.SLPCOLINFO_DELTAINDEX` Get the delta variable index associated to the column

`xpress.constants.SLPCOLINFO_DELTA` Get the delta value (change since previous value) of the column

`xpress.constants.SLPCOLINFO_DELTADJ` Get the delta variables reduced cost

`xpress.constants.SLPCOLINFO_UPDATEROW` Get the index of the update (or step bound) row associated to the column

`xpress.constants.SLPCOLINFO_SB` Get the step bound on the variable

`xpress.constants.SLPCOLINFO_SBDUAL` Get the dual multiplier of the step bound row for the variable

problem.getcols

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getCols` instead.
Returns the nonzeros in the constraint matrix for the columns in a given range.

Synopsis

```
problem.getcols (start, rowind, rowcoef, maxcoefs, first, last)
```

Arguments

`start`	Array which will be filled with the indices indicating the starting offsets in the `rowind` and `rowcoef` arrays for each requested column. It must be of length at least `last-first+2`. Column `i` starts at position `start[i]` in the `rowind` and `rowcoef` arrays, and has `start[i+1]-start[i]` elements in it. May be `None` if not required, but must be specified.
`rowind`	Array of length `maxcoefs` which will be filled with the rows of the nonzero coefficents for each column. May be `None` if not required, but must be specified.
`rowcoef`	Array of length `maxcoefs` which will be filled with the nonzero coefficient values. May be `None` if not required, but must be specified.
`maxcoefs`	The size of the `rowind` and `rowcoef` arrays. This is the maximum number of nonzero coefficients that the Optimizer is allowed to return.
`first`	First column in the range.
`last`	Last column in the range.

Example

The following examples retrieves the `start` vector of the problem:

```
p = xpress.problem()
p.read("example", "l")
start = []
p.getcols(start, rowind=None, rowcoef=None, maxcoefs=100, first=0, last=p.attributes.cols - 1)
```

Further information

It is possible to obtain just the number of elements in the range of columns by replacing `start`, `rowind` and `rowcoef` by `None`, as in the example. In this case, `maxcoefs` must be set to `0` to indicate that the length of arrays passed is zero. This is demonstrated in the example above.

Related topics

`problem.getRows`, `problem.getCols`.

problem.getcoltype

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getColType` *instead.*
Returns the column types for the columns in a given range.

Synopsis

```
problem.getcoltype(coltype, first, last)
```

Arguments

`coltype`	Character array of length `last-first+1` where the column types will be returned: `C` indicates a continuous variable; `I` indicates an integer variable; `B` indicates a binary variable; `S` indicates a semi-continuous variable; `R` indicates a semi-continuous integer variable; `P` indicates a partial integer variable.
`first`	First column in the range.
`last`	Last column in the range.

Example

This example finds the types for all columns in the matrix and prints them:

```
coltype = []
p.getcoltype(coltype, 0, p.attributes.cols - 1)
print("coltypes:", coltype)
```

Related topics

`problem.chgColType`, `problem.getRowType`, `problem.getColType`.

problem.getcontrolinfo

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getControlInfo` *instead.* Accesses the id number and the type information of a control given its name. A control name may be for example `'presolve'`. The function will return an id number of 0 and a type value of `notdefined` if the name is not recognized as a control name. Note that this will occur if the name is an attribute name rather than a control name.

Synopsis

```
(id,type) = problem.getcontrolinfo(name)
```

Argument

`name`	The name of the control to be queried. Names are case-insensitive. A full list of all control may be found in the Xpress Optimizer reference manual.

Related topics

`problem.getAttribInfo`, `problem.getControlInfo`.

problem.getcpcutlist

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getCPCutList` *instead.*
Returns a list of cut indices from the cut pool.

Synopsis

```
ncuts = problem.getcpcutlist(cuttype, interp, delta, maxcuts, cutind, viol)
```

Arguments

`cuttype`	The user defined type of the cuts to be returned.
`interp`	Way in which the cut type is interpreted: `-1` get all cuts; `1` treat cut types as numbers; `2` treat cut types as bit maps - get cut if any bit matches any bit set in `cuttype`; `3` treat cut types as bit maps - get cut if all bits match those set in `cuttype`.
`delta`	Only those cuts with a signed violation greater than delta will be returned.
`maxcuts`	Maximum number of cuts to be returned.
`cutind`	Array of length `maxcuts` where the cuts will be returned.
`viol`	Array of length `maxcuts` where the values of the signed violations of the cuts will be returned.

Further information

1. The violated cuts can be obtained by setting the `delta` parameter to the maxcuts of the (signed) violation required. If unviolated cuts are required as well, `delta` may be set to `-xpress.infinity`.
2. If the number of active cuts is greater than `maxcuts`, only `maxcuts` cuts will be returned. Otherwise only the existing cuts will be used to fill in the positions of `cutind`.
3. In case of a cut of type 'L', the violation equals the negative of the slack associated with the row of the cut. In case of a cut of type 'G', the violation equals the slack associated with the row of the cut. For cuts of type 'E', the violation equals the absolute value of the slack.
4. Please note that the violations returned are absolute violations, while feasibility is checked by the Optimizer in the scaled problem.

Related topics

`problem.delCPCuts`, `problem.getCPCuts`, `problem.getCutList`, `problem.loadCuts`, `problem.getCutMap`, `problem.getCutSlack`, `problem.getCPCutList`, Section "Working with the cut manager" of the Xpress Optimizer reference manual.

problem.getcpcuts

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getCPCuts` *instead.* Returns cuts from the cut pool. A list of cuts in the array `mindex` must be passed to the routine. The columns and elements of the cut will be returned in the regions pointed to by the `colind` and `cutcoef` parameters. The columns and elements will be stored contiguously and the starting point of each cut will be returned in the region pointed to by the `start` parameter.

Synopsis

```
problem.getcpcuts(rowind, maxcoefs, cuttype, rowtype, start, colind, cutcoef, rhs)
```

Arguments

`rowind`	List containing the cuts.
`maxcoefs`	Maximum number of columns of the cuts to be returned.
`cuttype`	List where the cut types will be returned.
`rowtype`	Character list where the sense of the cuts (`L`, `G`, or `E`) will be returned.
`start`	Array containing the offsets into the `colind` and `cutcoef` arrays. The last element indicates the total number of elements.
`colind`	Array where the columns of the cuts will be returned.
`cutcoef`	Array where the coefficients will be returned.
`rhs`	Array where the right hand side elements for the cuts will be returned.

Related topics

`problem.getCPCutList`, `problem.getCutList`, `problem.getCPCuts`, Section "Working with the cut manager" of the Xpress Optimizer reference manual.

problem.getcutlist

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getCutList` *instead.*
Retrieves a list of cuts for the cuts active at the current node.

Synopsis

```
problem.getcutlist(cuttype, interp, maxcuts, cutind)
```

Arguments

`cuttype`	User defined type of the cuts to be returned. A value of `-1` indicates return all active cuts.
`interp`	Way in which the cut type is interpreted: `-1` get all cuts; `1` treat cut types as numbers; `2` treat cut types as bit maps - get cut if any bit matches any bit set in `cuttype`; `3` treat cut types as bit maps - get cut if all bits match those set in `cuttype`.
`maxcuts`	Maximum number of cuts to be retrieved.
`cutind`	Array of length `maxcuts` where the cuts will be returned.

Further information

If the number of active cuts is greater than `maxcuts`, then `maxcuts` cuts will be returned. Otherwise only the positions corresponding to the number of active cuts will be filled in `cutind`.

Related topics

`problem.getCPCutList`, `problem.getCPCuts`, `problem.getCutList`, Section "Working with the cut manager" of the Xpress Optimizer reference manual.

problem.getcutmap

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getCutMap` *instead.* Returns in which rows a list of cutind are currently loaded into the Optimizer. This is useful for example to retrieve the duals associated with active cutind.

Synopsis

```
problem.getcutmap(cutind, cutmap)
```

Arguments

`cutind`	Array with the cutind for which the row index is requested.
`cutmap`	Array where the rows are returned.

Further information

For cutind currently not loaded into the problem, a row index of `-1` is returned.

Related topics

`problem.getCPCutList`, `problem.delCPCuts`, `problem.getCutList`, `problem.loadCuts`, `problem.getCutSlack`, `problem.getCPCuts`, `problem.getCutMap`, Section "Working with the cut manager" of the Xpress Optimizer reference manual.

problem.getcutslack

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getCutSlack` *instead.* Used to calculate the slack value of a cut with respect to the current LP relaxation solution. The slack is calculated from the cutind itself, and might be requested for any cutind (even if it is not currently loaded into the problem).

Synopsis

```
slack = problem.getcutslack(cutind)
```

Arguments

`cutind`	Cut object for which the slack is to be calculated.
`slack`	Value of the slack.

Related topics

`problem.getCPCutList`, `problem.delCPCuts`, `problem.getCutList`, `problem.loadCuts`, `problem.getCutMap`, `problem.getCPCuts`, `problem.getCutSlack`, Section "Working with the cutind manager" of the Xpress Optimizer reference manual.

problem.getdirs

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getDirs` *instead.*
Returns the directives that have been loaded into a problem. Priorities, forced branching directions and pseudo costs can be returned. If called after presolve, `getdirs` will get the directives for the presolved problem.

Synopsis

```
problem.getdirs(indices, prios, branchdirs, uppseudo, downpseudo)
```

Arguments

`indices`	Array containing the column numbers (`0`, `1`, `2`,...) or negative values corresponding to special ordered sets (the first set numbered `-1`, the second numbered `-2`,...). May be `None` if not required.
`prios`	Array containing the priorities for the columns and sets. May be `None` if not required.
`branchdirs`	Character array with the branching direction for each column or set: `U` the entity is to be forced up; `D` the entity is to be forced down; `N` not specified. May be `None` if not required.
`uppseudo`	Array containing the up pseudo costs for the columns and sets. May be `None` if not required.
`downpseudo`	Array containing the down pseudo costs for the columns and sets. May be `None` if not required.

Further information

The size of all lists is at most `MIPENTS`, obtainable from `problem.attributes.mipents`.

Related topics

`problem.loadDirs`, `problem.loadPresolveDirs`, `problem.getDirs`.

problem.getdf

Purpose

This subroutine is deprecated and will be removed in a future release.
Get a distribution factor

Synopsis

```
value = problem.getdf(col, row)
```

Arguments

`col`	The column (i.e., `xpress.var` object, index, or name) whose distribution factor is to be retrieved.
`row`	The row (i.e., `xpress.constraint` object, index, or name) from which the distribution factor is to be taken.
`value`	The value of the distribution factor.

Example

The following example retrieves the value of the distribution factor for column 282 in row 134 and changes it to be twice as large.

```
value = p.getdf(282,134)
value *= 2
p.chgdf(282,134,calue)
```

Further information

The *distribution factor* of a column in a row is the matrix coefficient of the corresponding delta vector in the row. Distribution factors are used in conventional recursion models, and are essentially normalized first-order derivatives. Xpress SLP can accept distribution factors instead of initial values, provided that the values of the variables involved can all be calculated after optimization using determining rows, or by a callback.

Related topics

`problem.adddfs`, `problem.chgdf`, `problem.loaddfs`

problem.getDual

Purpose

This subroutine is deprecated and will be removed in a future release. See `problem.getDuals`.

problem.getdualray

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getDualRay` *instead.*
Retrieves a dual ray (dual unbounded direction) for the current problem, if the problem is found to be infeasible.

Synopsis

```
problem.getdualray(ray)
```

Argument

`ray` Array of length `problem.attributes.rows` to hold the ray. May be `None` if not required.

Example

The following code tries to retrieve a dual ray:

```
if not p.hasdualray():
    print("Could not retrieve a dual ray")
else:
    dray = []
    p.getdualray(dray)
    print("dual ray:", dray)
```

Further information

1. It is possible to retrieve a dual ray only when, after solving an LP problem, the final status is `xpress.LPStatus.INFEAS`.
2. Dual rays are not post-solved. If the problem is in a presolved state, the dual ray that is returned will be for the presolved problem. If the problem was solved with presolve on and has been restored to the original state (the default behavior), this function will not be able to return a ray. To ensure that a dual ray can be obtained, it is recommended to solve a problem with presolve turned off (`presolve` = 0).

Related topics

`problem.getPrimalRay`, `problem.getDualRay`.

problem.getgencons

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getGenCons` *instead.*
Returns the general constraints `y = f(x1, ..., xn, c1, ..., cm)` in a given range.

Synopsis

```
(ncol, nval) = problem.getgencons(contype, resultant, colstart, colind, maxcols,
       valstart, val, maxvals, first, last)
```

Arguments

`contype`	`None` if not required, otherwise a list which will be filled with the types of the general constraints: `xpress.GenConsType.MAX (0)` indicates a `maximum` constraint; `xpress.GenConsType.MIN (1)` indicates a `minimum` constraint; `xpress.GenConsType.AND (2)` indicates an `and` constraint. `xpress.GenConsType.OR (3)` indicates an `or` constraint; `xpress.GenConsType.ABS (4)` indicates an `absolute value` constraint.
`resultant`	List/array which will be filled with the output variables `y`. May be `None` if not required.
`colstart`	List/array which will be filled with the start index of each general constraint in the `colind` array. May be `None` if not required.
`colind`	Integer array which will be filled with the indices of the input variables `xi`. May be `None` if not required.
`maxcols`	Maximum number of input columns to be retrieved.
`valstart`	Integer array of length at least `last-first+1` which will be filled with the start index of each general constraint in the `val` array. May be `None` if not required.
`val`	Integer array which will be filled with the constant values `ci`. May be `None` if not required.
`maxvals`	Maximum number of constant values to be retrieved.
`first`	First general constraint in the range.
`last`	Last general constraint in the range.
`ncol`	Number of values in the `colind` list if not `None`.
`nval`	Number of values in the `coef` list if not `None`.

Example

The following example retrieves all general constraints:

```
contype, resultant, colstart, colind, valstart, val = [], [], [], [], [], []
prob.getgencons(contype, resultant, colstart, colind, 1e9, valstart, val, 1e9, 0, prob.attribu
```

Further information

It is possible to obtain just the number of input columns and/or constant values in the range of general constraints by calling this function with `maxcols` and `maxvals` set to `0`, in which case the required size for the arrays will be returned as a tuple with `ncols` and `nvals`.

Related topics

`problem.addGenCons`, `problem.delGenCons`, `xpress.And`, `xpress.Or`, `xpress.max`, `xpress.min`, `xpress.abs`, `problem.getGenCons`.

problem.getmipentities

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getMIPEntities` *instead.* Retrieves MIP entity information about a problem. It must be called before `problem.mipOptimize` if the presolve option is used.

Synopsis

```
problem.getmipentities(coltype, colind, limit, settype, start, setcols, refval)
```

Arguments

`coltype` Character array where the entity types will be returned. The types will be one of:

- `B` binary variables;
- `I` integer variables;
- `P` partial integer variables;
- `S` semi-continuous variables;
- `R` semi-continuous integer variables.

`colind` Array where the columns of the MIP entities will be returned.

`limit` Array where the limits for the partial integer variables and lower bounds for the semi-continuous and semi-continuous integer variables will be returned (any entries in the positions corresponding to binary and integer variables will be meaningless).

`settype` Character array where the set types will be returned. The set types will be one of:

- `1` SOS1 type sets;
- `2` SOS2 type sets.

`start` Array where the offsets into the `setcols` and `refval` arrays indicating the start of the sets will be returned. This array must be of length `SETMEMBERS+1`: the final element contains the length of the `setcols` and `refval` arrays.

`setcols` Array of length `problem.attributes.setmembers` where the columns in each set will be returned.

`refval` Array of length `problem.attributes.setmembers` where the reference row entries for each member of the sets will be returned.

Example

The following obtains the SOS information:

```
settype = []
mstart = []
setcols = []
refval   = []
p.getmipentities(None, None, None, settype, mstart, setcols, refval)
```

Further information

All arguments may be `None` if not required.

Related topics

`problem.loadproblem`, `problem.getMIPEntities`.

problem.getiisdata

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getIISData` *instead.*
Returns information for an Irreducible Infeasible Set: size, variables (row and column vectors) and conflicting sides of the variables, duals and reduced costs.

Synopsis

```
problem.getiisdata(iis, rowind, colind, contype, bndtype, duals, djs, isolationrows,
       isolationcols)
```

Arguments

`iis` The ordinal number of the IIS to get data for.

`rowind` Indices of rows in the IIS. Can be `None` if not required.

`colind` Indices of bounds (columns) in the IIS. Can be `None` if not required.

`contype` Sense of rows in the IIS:
- `L` for less or equal row;
- `G` for greater or equal row.
- `E` for an equality row (for a non LP IIS);
- `1` for a SOS1 row;
- `2` for a SOS2 row;
- `W` for a piecewise linear constraint;
- `X` for a general constraint;
- `I` for an indicator row.

Can be `None` if not required.

`bndtype` Sense of bound in the IIS:
- `U` for upper bound;
- `L` for lower bound.
- `F` for fixed columns (for a non LP IIS);
- `B` for a binary column;
- `I` for an integer column;
- `P` for a partial integer columns;
- `S` for a semi-continuous column;
- `R` for a semi-continuous integer column.

Can be `None` if not required.

`duals` The >dual multipliers associated with the rows. Can be `None` if not required.

`djs` The dual multipliers (reduced costs) associated with the bounds. Can be `None` if not required.

`isolationrows` The isolation status of the rows:
- `-1` if isolation information is not available for row (run iis isolations);
- `0` if row is not in isolation;
- `1` if row is in isolation.

Can be `None` if not required.

`isolationcols` The isolation status of the bounds:
- `-1` if isolation information is not available for column (run iisisolations);
- `0` if column is not in isolation;
- `1` if column is in isolation. Can be `None` if not required.

Example

This example first retrieves the size of IIS 1, then gets the detailed information for the IIS.

```
rowind = []
colind = []
contype = []
bndtype = []
duals = []
djs = []
isolationrows = []
isolationcols = []
p.getiisdata(1, rowind, colind, contype, bndtype,
             duals, djs, isolationrows, isolationcols)
```

Further information

1. IISs are numbered from 1 to `NUMIIS`. Index number 0 refers to the IIS approximation.
2. If `rowind` and `colind` both are `None`, only the `rownumber` and `colnumber` are returned.
3. The arrays may be `None` if not required. However, arrays `contype`, `duals` and `isolationrows` are only returned if `rowind` is not `None`. Similarly, arrays `bndtype`, `djs` and `isolationcols` are only returned if `colind` is not `None`.
4. For the initial IIS approximation (`iis` = 0) the number of rows and columns with a nonzero Lagrange multiplier (dual/reduced cost respectively) are returned. Please note that in such cases, it might be necessary to call `problem.IISStatus` to retrieve the necessary size of the return arrays.
5. If there are Special Ordered Sets in the IIS, their number is included in the `rowind` array.
6. For non-LP IISs, some column indices may appear more than once in the `colind` array, for example an integrality and a bound restriction for the same column.
7. Duals, reduced cost and isolation information is not available for nonlinear IIS problems, and for those the arrays are filled with zero values in case they are provided.

Related topics

`problem.IISAll`, `problem.clearIIS`, `problem.firstIIS`, `problem.IISIsolations`, `problem.nextIIS`, `problem.IISStatus`, `problem.writeIIS`, `problem.getIISData`.

problem.getIndexFromName

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getIndex` instead.
Returns the index for a specified row or column name.

Synopsis

```
ind = problem.getIndexFromName(type, name)
```

Arguments

`type`	`xpress.Namespaces.ROW` if a row index is required; `xpress.Namespaces.COLUMN` if a column index is required.
`name`	String containing name of the item sought.

Return value

`ind` The row or column index number. A value of -1 will be returned if the row or column does not exist.

Example

The following example retrieves the index of column "`xnew`":

```
p = xpress.problem()
x = p.addVariable(name='xnew')
[...]
print("variable's index: ", p.getIndexFromName(xpress.Namespaces.COLUMN, 'xnew'))
```

Related topics

`problem.getVariable`, `problem.getConstraint`.

problem.getindicators

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getIndicators` *instead.* Returns the indicator constraint condition (indicator variable and complement flag) associated to the rows in a given range.

Synopsis

```
problem.getindicators(colind, complement, first, last)
```

Arguments

`colind`	Array of length `last-first+1` where the indicator variables are to be placed.
`complement`	Array of length `last-first+1` where the indicator complement flags will be returned: `0` not an indicator constraint (in this case the corresponding entry in the `colind` array is ignored); `1` for indicator constraints with condition "`bin = 1`"; `-1` for indicator constraints with condition "`bin = 0`";
`first`	First row in the range.
`last`	Last row in the range (inclusive).

Example

The following example retrieves information about three indicator constraints in the problem and prints a list of their indices.

```
colind = []
complement = []
p.getindicators(colind, complement, 2, 4)
print("indices:",          colind)
print("complement flags:", complement)
```

Related topics

`problem.setIndicators`, `problem.getIndicators`.

problem.getinfeas

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getInfeas` instead.
Returns a list of infeasible primal and dual variables.

Synopsis

```
problem.getinfeas(x, slack, duals, djs)
```

Arguments

`x`	Array to store the primal infeasible variables. May be `None` if not required.
`slack`	Array to store the primal infeasible rows. May be `None` if not required.
`duals`	Array to store the dual infeasible rows. May be `None` if not required.
`djs`	Array to store the dual infeasible variables. May be `None` if not required.

Example

```
x = []
slack = []
p.getinfeas(x, slack, None, None)
print("getinfeas --> x and slack:", x, slack)
```

Further information

To find the infeasibilities in a previously saved solution, the solution must first be loaded into memory with the `problem.readBinSol` function.

Related topics

`problem.getScaledInfeas`, `problem.getIISData`, `problem.IISAll`, `problem.clearIIS`, `problem.firstIIS`, `problem.IISIsolations`, `problem.nextIIS`, `problem.IISStatus`, `problem.writeIIS`, `problem.getInfeas`.

problem.getlastbarsol

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getLastBarSol` *instead.* Obtains the last barrier solution values following optimization that used the barrier solver.

Synopsis

```
barsolstatus = problem.getastbarsol(x=None, slack=None, duals=None, djs=None)
```

Arguments

`x` Array of length `problem.attributes.cols` where the values of the primal variables will be returned. May be `None` if not required.

`slack` Array of length `problem.attributes.rows` where the values of the slack variables will be returned. May be `None` if not required.

`duals` Array of length `problem.attributes.rows` where the values of the dual variables ($c_B^T B^{-1}$) will be returned. May be `None` if not required.

`djs` Array of length `problem.attributes.cols` where the reduced cost for each variable ($c^T - c_B^T B^{-1} A$) will be returned. May be `None` if not required.

`barsolstatus` Status of the last barrier solve. Value matches that of the `lpstatus` attribute if the solve stopped immediately after the barrier.

Further information

1. If the barrier solver has not been used, `barsolstatus` will return `xpress.LPStatus.UNSOLVED`.
2. The barrier solution or the solution candidate is always available if the status is not `xpress.LPStatus.UNSOLVED`.
3. The last barrier solution is available until the next solve, and is not invalidated by otherwise working with the problem.

Related topics

`problem.getSolution`, `problem.getLastBarSol`

problem.getlasterror

Purpose

This subroutine is deprecated and will be removed in a future release.
Returns the error message corresponding to the last error triggered by a library function.

Synopsis

```
s = problem.getlasterror()
```

Example 1

The following shows how this function might be used in error-checking:

```
p.optimize()
print("Current error status:", p.getlasterror())
```

Further information

The problem.getlasterror() function is an API wrapper for the `XPRSgetlasterror()` function in the Xpress C API. For this reason, errors that occur within the Xpress API are reported by getlasterror(). Errors that occur at the level of the Python interface are *not* reported by `getlasterror`. Both classes of errors can be handled with a `try/except` construct. In the two examples below, the first is an error that is detected by the Xpress API and propagated to a Python error, while the second is an incorrect statement for the Python module. They both trigger a RuntimeError exception.

Example 2

```
p = xpress.problem()

try:
    x = p.addVariable()
    y = p.addVariable()
except RuntimeError as e:
    print(e)
```

Example 3

```
try:
    p.addVariable('John Cleese')
except RuntimeError as e:
    print(e)
```

Related topics

`problem.addMessageCallback`, `problem.setLogFile`.

problem.getlb

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getLB` instead.
Returns the lower bounds on the columns in a given range.

Synopsis

```
problem.getlb(lb, first, last)
```

Arguments

`lb`	Array where the lower bounds are to be placed.
`first`	(optional, default 0) First column in the range.
`last`	(optional, default COLS - 1) Last column in the range.

Example

The following example retrieves the lower bounds for the columns of the current problem:

```
newlb = []
p.getlb(newlb, 0, 4)
print("lb: ", newlb)
```

Further information

Values greater than or equal to `xpress.infinity` should be interpreted as infinite; values less than or equal to –`xpress.infinity` should be interpreted as negative infinite.

Related topics

`problem.chgBounds`, `problem.getUB`, `problem.getLB`.

problem.getlpsol

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getLpSol` *instead.*
Obtains the LP solution values following optimization.

Synopsis

```
problem.getlpsol(x, slack, duals, djs)
```

Arguments

`x`	Array to store the values of the primal variables. May be `None` if not required.
`slack`	Array to store the values of the slack variables. May be `None` if not required.
`duals`	Array to store the values of the dual variables ($c_B^T B^{-1}$). May be `None` if not required.
`djs`	Array to store the reduced cost for each variable ($c^T - c_B^T B^{-1} A$). May be `None` if not required.

Example

The following sequence of function calls will get the LP solution (`x`) at the top node of a MIP and the optimal MIP solution (`y`):

```
p.controls.mipstopstage = xp.MIPStopStage.INITIALRELAXATION # only solve the LP relaxation
p.mipoptimize()
p.setDefaultControl('mipstopstage')
x = []
p.getlpsol(x)
print("root LP solution:", x)
p.mipoptimize() # solve the MIP problem
x = p.getSolution()
print("final MIP solution", x)
```

Further information

1. After solving an LP, it is recommended to use `problem.getSolution` to retrieve the solution.
2. When an integer solution is found during a tree search, it is always set up as a solution to the current node; therefore the integer solution is available as the current node. It is recommended to retrieved the solution with `problem.getCallbackSolution` and `problem.getCallbackPresolveSolution`.
3. If the problem is modified after calling `lpoptimize`, then the solution will no longer be available.
4. If the problem has been presolved, then `getlpsol` returns the solution to the original problem. The only way to obtain the presolved solution is to call the related function, `problem.getPresolveSol`.

Related topics

`problem.getSolution`, `problem.getCallbackSolution`, `problem.getCallbackPresolveSolution`, `problem.writePrtSol`, `problem.writeSol`, `problem.getLpSol`.

problem.getlpsolval

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getSolution` *or* `problem.getCallbackSolution` *and related functions instead.*
Used to obtain a single LP solution value following optimization.

Synopsis

```
x, slack, dual, dj = problem.getlpsolval(col=None, row=None)
```

Arguments

`col`	Column of the variable for which to return the solution value.
`row`	Row of the constraint for which to return the solution value.
`x`	The returned value of the primal variable.
`slack`	The returned value of the slack variable.
`dual`	The returned value of the dual variable ($c_B^T B^{-1}$).
`dj`	The returned reduced cost for the variable ($c^T - c_B^T B^{-1} A$).

Further information

1. This function is currently not supported if the problem is in a presolved state.
2. If `col` or `row` are `None`, the corresponding output is set to `-xpress.infinity`.

Related topics

`problem.getSolution`, `problem.getCallbackSolution`, `problem.getCallbackPresolveSolution`, `problem.writePrtSol`, `problem.writeSol`.

problem.getmessagestatus

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getMessageStatus` *instead.*
Returns the current suppression status of a message: nonzero if the message is not suppressed; `0` otherwise.

Synopsis

```
status = problem.getmessagestatus(msgcode)
```

Argument

`msgcode`	The id number of the message. Refer to the Xpress Optimizer reference manual for a list of possible message numbers.

Further information

If a message is suppressed globally then the message will always have `status` return zero from `getmessagestatus`.

Related topics

`problem.setMessageStatus`, `problem.getMessageStatus`.

problem.getmipsol

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getSolution` and `problem.getSlacks` instead.
Obtains the solution values of the last MIP solution that was found.

Synopsis

```
problem.getmipsol(x, slack)
```

Arguments

`x`	Array to store the values of the primal variables. May be `None` if not required.
`slack`	Array to store the values of the slack variables. May be `None` if not required.

Example

The following sequence of function calls will get the solution (`x`) of the last MIP solution for a problem:

```
x = []
p.mipoptimize()
p.getmipsol(x)
print("solution:", x)
```

Further information

1. During a `intsol` or `preintsol` callback, in order to retrieve the corresponding integer solution, use either `problem.getCallbackSolution` or `problem.getCallbackPresolveSolution`, not `problem.getmipsol` (see the documentation of these callbacks for an explanation).
2. After solving the root LP of a MIP with setting `MIPSTOPSTAGE` to `INITIALRELAXATION` and calling `problem.mipoptimize()`, you can access the solution by calling `problem.getLpSol`.

Related topics

`problem.getSolution`, `problem.getSlacks`, `problem.getLpSol`, `problem.writePrtSol`, `problem.writeSol`.

problem.getmipsolval

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getSolution` *and* `problem.getSlacks` *instead.*
Used to obtain a single solution value of the last MIP solution that was found.

Synopsis

```
x, slack = problem.getmipsolval(col=None, row=None)
```

Arguments

`col`	Column index of the variable for which to return the solution value. May be `None`.
`row`	Row index of the constraint for which to return the solution value. May be `None`.
`x`	The returned value of the primal variable, or `-xpress.infinity` if `col` is `None`.
`slack`	The returned value of the slack variable, or `-xpress.infinity` if `row` is `None`.

Related topics

`problem.getSolution`, `problem.getSlacks`, `problem.writePrtSol`, `problem.writeSol`.

problem.getmqobj

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getMQObj` *instead.*
Returns the nonzeros in the quadratic objective coefficients' matrix for the columns in a given range. To achieve maximum efficiency, `getmqobj` returns the lower triangular part of this matrix only.

Synopsis

```
problem.getmqobj(start, colind, objqcoef, maxcoefs, first, last)
```

Arguments

`start`	Array which will be filled with indices indicating the starting offsets in the `colind` and `objqcoef` arrays for each requested column. It must be length of at least `last-first+2`. Column `i` starts at position `start[i]` in the `colind` and `objqcoef` arrays, and has `start[i+1]-start[i]` elements in it. May be `None` if `maxcoefs` is `0`.
`colind`	Array which will be filled with at most `maxcoefs` columns of the nonzero elements in the lower triangular part of `Q`. May be `None` if `maxcoefs` is `0`.
`objqcoef`	Array which will be filled with at most `maxcoefs` nonzero element values. May be `None` if `maxcoefs` is `0`.
`maxcoefs`	The maximum number of elements to be returned (maxcoefs of the arrays).
`first`	First column in the range.
`last`	Last column in the range.

Further information

The objective function is of the form $c^T x + 0.5 x^T Q x$ where `Q` is positive semi-definite for minimization problems and negative semi-definite for maximization problems. If this is not the case the optimization algorithms may converge to a local optimum or may not converge at all. Note that only the upper or lower triangular part of the `Q` matrix is returned.

Related topics

`problem.chgMQObj`, `problem.chgQObj`, `problem.getQObj`, `problem.getMQObj`.

problem.getobjn

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getObjN` *instead.*
Returns the coefficients of a given objective function for the columns in a given range.

Synopsis

```
problem.getobjn(objidx, objcoef, first, last)
```

Arguments

`objidx`	Index of the objective function whose coefficients to return.
`objcoef`	Array of length `last-first+1` where the objective function coefficients are to be placed.
`first`	First column in the range.
`last`	Last column in the range.

Example

The following example retrieves the coefficients of the first five variables in the second objective function:

```
objcoef = []
p.getobjn(1, objcoef, 0, 4)
```

Related topics

`problem.getObj`, `problem.getObjN`.

problem.getnamelist

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getNameList` *instead.* Returns the names for the rows, columns, sets, piecewise linear constraints, general constraints or objectives in a given range. The names will be returned in a list of Python strings.

Synopsis

```
names = problem.getnamelist(type, first, last)
```

Arguments

`type`	`xpress.Namespaces.ROW` for row names; `xpress.Namespaces.COLUMN` for column names; `xpress.Namespaces.SET` for set names; `xpress.Namespaces.PWLCONS` for piecewise linear constraint names; `xpress.Namespaces.GENCONS` for general constraint names; `xpress.Namespaces.OBJECTIVE` for objective names.
`names`	A list containing all returned names.
`first`	First row, column or set in the range. If `None`, it is taken as zero.
`last`	Last row, column or set in the range. If `None`, it is taken as the penultimate element in the list defined by `type`.

Example

The following example retrieves and outputs the row and column names for the current problem.

```
cols = prob.attributes.cols
rows = prob.attributes.rows

rnames = prob.getnamelist(1, 0, rows - 1)
cnames = prob.getnamelist(2, 0, cols - 1)

for k,v in enumerate(rnames):
    print("Row {0:4d}: {1}", k, v)

for k,v in enumerate(cnames):
    print("Column {0:4d}: {1}", k, v)
```

problem.getobj

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getObj` *instead.*
Returns the objective function coefficients for the columns in a given range.

Synopsis

```
problem.getobj(objcoef, first, last)
```

Arguments

`objcoef`	Array of length `last-first+1` where the objective function coefficients are to be placed.
`first`	First column in the range.
`last`	Last column in the range.

Example

The following example retrieves the objective function coefficients of the first five variables of the current problem:

```
objcoef = []
p.getobj(objcoef, 0, 4)
```

Related topics

`problem.chgObj`, `problem.getObj`.

problem.getObjVal

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.attributes.objval` *instead.*
Returns the objective value of the solution found by the Optimizer.

Synopsis

```
o = problem.getObjVal()
```

Example

The following prints the objective value of an optimal solution after the `problem.optimize` function is called:

```
p.optimize()
print("optimal solution:", p.getObjVal())
```

Related topics

`problem.optimize`.

problem.getpivotorder

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getPivotOrder` *instead.*
Returns the pivot order of the basic variables.

Synopsis

```
problem.getpivotorder(pivotorder)
```

Argument

`pivotorder` Array where the pivot order will be returned.

Example

The following returns the pivot order of the variables into an array `pPivot`:

```
pivotorder = []
p.getpivotorder(pivotorder)
```

Further information

Row indices are in the range `0` to `ROWS-1`, whilst columns are in the range `ROWS+SPAREROWS` to `ROWS+SPAREROWS+COLS-1`.

Related topics

`problem.getPivots`, `problem.getPivotOrder`.

problem.getpivots

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getPivots` instead. Returns a list of potential leaving variables if a specified variable enters the basis. The return value is a tuple containing the objective function value that would result if `enter` entered the basis; and an integer where the actual number of potential leaving variables will be returned.

Synopsis

```
dobj, npiv = problem.getpivots(enter, outlist, x, maxpivots)
```

Arguments

`enter`	Index of the specified row or column to enter basis.
`outlist`	Array of length at least `maxpivots` to hold list of potential leaving variables. May be `None` if not required.
`x`	Array of length `problem.attributes.rows + problem.attributes.sparerows + problem.attributes.cols` to hold the values of all the variables that would result if `enter` entered the basis. May be `None` if not required.
`maxpivots`	Maximum number of potential leaving variables to return.

Example

The following retrieves a list of up to 5 potential leaving variables if variable 6 enters the basis:

```
outlist = []
x = []
obj, npiv = p.getpivots(2, outlist, x, 10)
```

Further information

1. If the variable `enter` enters the basis and the problem is degenerate then several basic variables are candidates for leaving the basis, and the number of potential candidates is returned enter `npiv`. A list of at most `maxpivots` of these candidates is returned enter `outlist` which must be at least `maxpivots` long. If variable `enter` were to be pivoted enter, then because the problem is degenerate, the resulting values of the objective function and all the variables do not depend on which of the candidates from `outlist` is chosen to leave the basis. The value of the objective is returned enter `dobj` and the values of the variables into `x`.
2. Row indices are enter the range `0` to `ROWS-1`, whilst columns are enter the range `ROWS+SPAREROWS` to `ROWS+SPAREROWS+COLS-1`.

Related topics

`problem.getPivotOrder`, `problem.getPivots`.

problem.getpresolvebasis

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getPresolveBasis` *instead.* Returns the current basis from memory into the user's data areas. If the problem is presolved, the presolved basis will be returned. Otherwise the original basis will be returned.

Synopsis

```
problem.getpresolvebasis(rstatus, cstatus)
```

Arguments

`rstatus` Array of length `problem.attributes.rows` to the basis status of the stack, surplus or artificial variable associated with each row. The status will be one of:

- `0` slack, surplus or artificial is non-basic at lower bound;
- `1` slack, surplus or artificial is basic;
- `2` slack or surplus is non-basic at upper bound.

May be `None` if not required.

`cstatus` Array of length `problem.attributes.cols` to hold the basis status of the columns in the constraint matrix. The status will be one of:

- `0` variable is non-basic at lower bound, or superbasic at zero if the variable has no lower bound;
- `1` variable is basic;
- `2` variable is at upper bound;
- `3` variable is super-basic.

May be `None` if not required.

Example

The following obtains and outputs basis information on a presolved problem prior to the tree search:

```
cs = []
p = xpress.problem()
p.read("global1", "")
p.mipoptimize()
p.getpresolvebasis(cstatus=cs)
```

Related topics

`problem.getBasis`, `problem.loadBasis`, `problem.loadPresolveBasis`, `problem.getPresolveBasis`.

problem.getpresolvemap

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getPresolveMap` *instead.*
Returns the mapping of the row and column numbers from the presolve problem back to the original problem.

Synopsis

```
problem.getpresolvemap(rowmap, colmap)
```

Arguments

`rowmap`	Array to store the row maps.
`colmap`	Array to store the column maps.

Example

The following reads in a (Mixed) Integer Programming problem and gets the mapping for the rows and columns back to the original problem following optimization of the linear relaxation. The elimination operations of the presolve are turned off so that a one-to-one mapping between the presolve problem and the original problem.

```
p.read("MyProb", "")
p.controls.presolveops = 255
p.controls.mipstopstage = xp.MIPStopStage.INITIALRELAXATION
p.mipoptimize()
p.setDefaultControl('mipstopstage')
rowmap = []
colmap = []
p.getpresolvemap(rowmap, colmap)
```

Further information

The presolved problem can contain rows or columns that do not map to anything in the original problem. An example of this are cuts created during the MIP solve and temporarily added to the presolved problem. It is also possible that the presolver will introduce new rows or columns. For any row or column that does not have a mapping to a row or column in the original problem, the corresponding entry in the returned `rowmap` and `colmap` arrays will be `-1`.

problem.getpresolvesol

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getPresolveSol` *instead.* Returns the solution for the presolved problem from memory.

Synopsis

```
problem.getpresolvesol(x, slack, duals, djs)
```

Arguments

`x`	Array to store the values of the primal variables. May be `None` if not required.
`slack`	Array to store the values of the slack variables. May be `None` if not required.
`duals`	Array to store the values of the dual variables. May be `None` if not required.
`djs`	Array to store the reduced cost for each variable. May be `None` if not required.

Example

The following reads in a (Mixed) Integer Programming problem and displays the solution to the presolved problem following optimization of the linear relaxation:

```
p.read("MyProb", "")
p.controls.mipstopstage = xp.MIPStopStage.INITIALRELAXATION
p.mipoptimize()
p.setDefaultControl('mipstopstage')
sol = []
p.getpresolvesol(x=sol)
print("presolved sol", sol)
```

Further information

1. If the problem has not been presolved, the solution in memory will be returned.
2. The solution to the original problem should be returned using the related function `problem.getLpSol`.
3. If called during a MIP callback the solution of the current node will be returned.
4. When an integer solution is found during tree search, it is always set up as a solution to the current node; therefore the integer solution is available as the current node solution and can be retrieved with `getlpsol` and `problem.getPresolveSol`.

problem.getprimalray

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getPrimalRay` *instead.* Retrieves a primal ray (primal unbounded direction) for the current problem, if the problem is found to be unbounded.

Synopsis

```
problem.getprimalray(ray)
```

Argument

`ray` Array of length `problem.attributes.cols` to hold the ray. May be `None` if not required.

Example

The following code tries to retrieve a primal ray:

```
if not p.hasprimalray():
    print("Could not retrieve a primal ray")
else:
    ray = []
    p.getprimalray(ray)
    print("primal ray:", ray)
```

Further information

1. It is possible to retrieve a primal ray only when, after solving an LP problem, the final status (`LPSTATUS`) is `xpress.LPStatus.UNBOUNDED`.
2. Primal rays are not post-solved. If the problem is in a presolved state, the primal ray that is returned will be for the presolved problem. If the problem was solved with presolve on and has been restored to the original state (the default behavior), this function will not be able to return a ray. To ensure that a primal ray can be obtained, it is recommended to solve a problem with presolve turned off (`PRESOLVE` = 0).

Related topics

`problem.getDualRay`, `problem.getPrimalRay`.

problem.getProbStatus

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.attributes.solvestatus` *and* `problem.attributes.solstatus` *instead.*
Returns the problem status before or after a call to `problem.optimize`.

Synopsis

```
s = problem.getProbStatus()
```

Example

```
p = xpress.problem()
p.read("example2", "")
p.optimize()
print("solution status code: ", p.getProbStatus(), " -->",
       p.getProbStatusString())
```

Further information

The returned number corresponds to the one of the problem status attributes described in the Xpress Optimizer reference manual: `problem.attributes.lpstatus` if the problem is an LP, `problem.attributes.mipstatus` if the problem is a MIP, or `problem.attributes.nlpstatus` if the problem is nonlinear.

Related topics

`problem.optimize`, `problem.getSolution`, `problem.getDuals`, `problem.getSlacks`, `problem.getRedCosts`, `problem.getProbStatusString`.

problem.getProbStatusString

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.attributes.solvestatus` *and* `problem.attributes.solstatus` *instead.*
Returns the string corresponding to the problem status before or after a call to `problem.optimize`.

Synopsis

```
s = problem.getProbStatusString()
```

Example

```
p = xpress.problem()
p.read("example2", "")
p.optimize()
print("solution status code: ", p.getProbStatus(), " -->",
       p.getProbStatusString())
```

Related topics

`problem.optimize`, `problem.getSolution`, `problem.getDuals`, `problem.getSlacks`, `problem.getRedCosts`, `problem.getProbStatus`.

problem.getpwlcons

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getPwlCons` instead.
Returns the piecewise linear constraints `y = f(x)` in a given range.

Synopsis

```
npoints = problem.getpwlcons(colind, resultant, start, xval, yval, maxpoints, first,
       last)
```

Arguments

`colind`	Integer array which will be filled with the indices of the input variables `x`. It must be of length at least `last-first+1`. May be `None` if not required.
`resultant`	Integer array which will be filled with the indices of the output variables `y`. It must be of length at least `last-first+1`. May be `None` if not required.
`start`	Integer array which will be filled with the start indices of the different constraints in the breakpoint arrays. It must be of length at least `last-first+1`. The `x`-values of the breakpoints of piecewise linear constraint `i < last` will be given in `xval[start[i]]` to `xval[start[i+1]]`. May be `None` if not required.
`xval`	Array of length `maxpoints`which will be filled with the `x`-values of the breakpoints. May be `None` if not required.
`yval`	Array of length `maxpoints`which will be filled with the `y`-values of the breakpoints. May be `None` if not required.
`maxpoints`	Maximum number of breakpoints to be retrieved.
`first`	First piecewise linear constraint in the range.
`last`	Last piecewise linear constraint in the range.
`npoints`	The returned number of breakpoints in the `xval` and `yval` arrays. If the number of breakpoints is greater than `maxpoints`, then only `maxpoints` elements will be returned.

Example

The following example retrieves all variables and breakpoints in the first two piecewise linear constraints:

```
colind, resultant, start, xval, yval = [], [], [], [], []
npoints = prob.getpwlcons(prob, colind, resultant, start, xval, yval, 1e9, 0, 1)
```

Further information

It is possible to obtain just the number of breakpoints in the range of piecewise linear constraints by calling this function with `maxpoints` set to `0`, in which case the required maxpoints for the breakpoint arrays will be returned in `npoints`.

Related topics

`problem.addPwlCons`, `problem.delPwlCons`, `xpress.pwl`, `problem.getPwlCons`.

problem.getqobj

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getQObj` *instead.*
Returns a single quadratic objective function coefficient corresponding to the variable pair `(objqcol1, objqcol2)` of the Hessian matrix.

Synopsis

```
objqcoef = problem.getqobj(objqcol1, objqcol2)
```

Arguments

`objqcol1`	Column index for the first variable in the quadratic term.
`objqcol2`	Column index for the second variable in the quadratic term.

Example

The following returns the coefficient of the x_0^2 term in the objective function, placing it in the variable `value`:

```
print("diagonal coeff of the Hessian:",
  [p.getqobj(i,i) for i in range(p.attributes.cols)])
```

Further information

For example, if the objective function has the term $[3x_1x_2 + 3x_2x_1]/2$ the value retrieved by `getqobj` is `3.0` and if the objective function has the term $[6x_1^2]/2$ the value retrieved by `getqobj` is `6.0`.

Related topics

`problem.getMQObj`, `problem.chgQObj`, `problem.chgMQObj`, `problem.getQObj`.

problem.getqrowcoeff

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getQRowCoeff` instead. Returns a single quadratic constraint coefficient corresponding to the variable pair (`rowqcol1`, `rowqcol2`) of the Hessian of a given constraint.

Synopsis

```
coeff = problem.getqrowcoeff (row, rowqcol1, rowqcol2)
```

Arguments

`row`	The quadratic row where the coefficient is to be looked up.
`rowqcol1`	Column index for the first variable in the quadratic term.
`rowqcol2`	Column index for the second variable in the quadratic term.

Example

The following returns the coefficient of the $dist^2$ term in the constraint cons1:

```
print("diagonal coeff of dist:", p.getqrowcoeff(cons1, dist, dist)
```

Further information

The coefficient returned corresponds to the Hessian of the constraint. That means the for constraint `x + [x^2 + 6 xy] <= 10` `getqrowcoeff` would return `1` as the coefficient of x^2 and `3` as the coefficient of `xy`.

Related topics

`problem.loadproblem`, `problem.addQMatrix`, `problem.chgQRowCoeff`, `problem.getQRowQMatrix`, `problem.getQRowQMatrixTriplets`, `problem.getQRows`, `problem.chgQObj`, `problem.chgMQObj`, `problem.getQObj`, `problem.getQRowCoeff`.

problem.getqrowqmatrix

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getQRowQMatrix` instead. Returns the nonzeros in a quadratic constraint coefficients matrix for the columns in a given range. To achieve maximum efficiency, `getqrowqmatrix` returns the lower triangular part of this matrix only.

Synopsis

```
problem.getqrowqmatrix(row, start, colind, rowqcoef, maxcoefs, first, last)
```

Arguments

`row`	Row (i.e., `xpress.constraint` object, index, or name) for which the quadratic coefficients are to be returned.
`start`	List to be filled with indices indicating the starting offsets in the `colind` and `dobjval` lists for each requested column. It must be length of at least `last-first+2`. Column `i` starts at position `start[i]` in the `colind` and `rowqcoef` arrays, and has `start[i+1]-start[i]` elements in it. May be `None` if maxcoefs is 0.
`colind`	Array of length `maxcoefs` which will be filled with the columns of the nonzero elements in the lower triangular part of Q. May be `None` if maxcoefs is 0.
`rowqcoef`	Array of length `maxcoefs` which will be filled with the nonzero element values. May be `None` if maxcoefs is 0.
`maxcoefs`	Maximum number of elements to be returned in `colind` and `rowqcoef`.
`first`	First column in the range.
`last`	Last column in the range.

Related topics

`problem.loadproblem`, `problem.getQRowCoeff`, `problem.addQMatrix`, `problem.chgQRowCoeff`, `problem.getQRowQMatrixTriplets`, `problem.getQRows`, `problem.chgQObj`, `problem.chgMQObj`, `problem.getQObj`, `problem.getQRowQMatrix`.

problem.getqrowqmatrixtriplets

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getQRowQMatrixTriplets` *instead.*

Returns the nonzeros in a quadratic constraint coefficients matrix as triplets (index pairs with coefficients). To achieve maximum efficiency, `getqrowqmatrixtriplets` returns the lower triangular part of this matrix only.

Synopsis

```
problem.getqrowqmatrixtriplets(row, rowqcol1, rowqcol2, rowqcoef)
```

Arguments

`row`	Row (i.e., `xpress.constraint` object, index, or name) for which the quadratic coefficients are to be returned.
`rowqcol1`	First index in the triplets. May be `None` if not required.
`rowqcol2`	Second index in the triplets. May be `None` if not required.
`rowqcoef`	Coefficients in the triplets. May be `None` if not required.

Further information

If a row index of `-1` is used, the function returns the quadratic coefficients for the objective function.

Related topics

`problem.loadproblem`, `problem.getQRowCoeff`, `problem.addQMatrix`, `problem.chgQRowCoeff`, `problem.getQRowQMatrix`, `problem.getQRows`, `problem.chgQObj`, `problem.chgMQObj`, `problem.getQObj`, `problem.getQRowQMatrixTriplets`.

problem.getqrows

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getQRows` *instead.*
Returns a list of row objects that have quadratic coefficients.

Synopsis

```
problem.getqrows(rowind)
```

Argument

`rowind`	Array to contain the indices of rows with quadratic coefficients in them. May be `None` if not required.

Related topics

`problem.loadproblem`, `problem.getQRowCoeff`, `problem.addQMatrix`, `problem.chgQRowCoeff`, `problem.getQRowQMatrix`, `problem.getQRowQMatrixTriplets`, `problem.chgQObj`, `problem.chgMQObj`, `problem.getQObj`, `problem.getQRows`.

problem.getRCost

Purpose

This subroutine is deprecated and will be removed in a future release. See `problem.getRedCosts`.

problem.getrhs

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getRHS` instead.
Returns the right hand side elements for the rows in a given range.

Synopsis

```
problem.getrhs(rhs, first, last)
```

Arguments

`rhs`	Array where the `(last - first + 1)` right hand side elements are to be placed.
`first`	First row in the range.
`last`	Last row in the range.

Example

The following example retrieves the right hand side values of the problem:

```
b = []
p.getrhs(b, 0, p.attributes.rows - 1)
```

Related topics

`problem.chgRHS`, `problem.chgRHSRange`, `problem.getRHSRange`, `problem.getRHS`.

problem.getrhsrange

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getRHSRange` *instead.*
Returns the right hand side range values for the rows in a given range.

Synopsis

```
problem.getrhsrange(range, first, last)
```

Arguments

`range`	Array of length `last-first+1` where the right hand side range values are to be placed.
`first`	First row in the range.
`last`	Last row in the range.

Related topics

`problem.chgRHS`, `problem.chgRHSRange`, `problem.getRHS`, `problem.getRHSRange`.

problem.getrowinfo

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpGetRowInfo` *instead.*
Get current row information.

Synopsis

```
info = problem.getrowinfo(infotype, rowindex)
```

Arguments

`infotype`	Type of information (see below)
`rowindex`	Row (i.e., `xpress.constraint` object, index, or name) whose information is to be handled
`info`	Information to be retrieved

Further information

If the data is not available, the type of the returned `info` is set to `None`.
The following constants are provided for row information handling:

`rowinfo_slack`	Get the current slack value of the row
`rowinfo_dual`	Get the current dual multiplier of the row
`rowinfo_numpenaltyerrors`	Get the number of times the penalty error vector has been active for the row
`rowinfo_maxpenaltyerror`	Get the maximum size of the penalty error vector activity for the row
`rowinfo_totalpenaltyerror`	Get the total size of the penalty error vector activity for the row
`rowinfo_currentpenaltyerror`	Get the size of the penalty error vector activity in the current iteration for the row
`rowinfo_currentpenaltyfactor`	Set the size of the penalty error factor for the current iteration for the row
`rowinfo_penaltycolumnplus`	Get the index of the positive penalty column for the row (+)
`rowinfo_penaltycolumnplusvalue`	Get the value of the positive penalty column for the row (+)
`rowinfo_penaltycolumnplusdj`	Get the reduced cost of the positive penalty column for the row (+)
`rowinfo_penaltycolumnminus`	Get the index of the negative penalty column for the row (-)
`rowinfo_penaltycolumnminusvalue`	Get the value of the negative penalty column for the row (-)
`rowinfo_penaltycolumnminusdj`	Get the reduced cost of the negative penalty column for the row (-)

problem.getrows

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getRows` instead.
Returns the nonzeros in the constraint matrix for the rows in a given range.

Synopsis

```
problem.getrows(start, colind, colcoef, maxcoefs, first, last)
```

Arguments

`start`	Array which will be filled with the indices indicating the starting offsets in the `colind` and `colcoef` arrays for each requested row. It must be of length at least `last-first+2`. Column `i` starts at position `start[i]` in the `colind` and `colcoef` arrays, and has `start[i+1]-start[i]` elements in it. May be `None` if not required.
`colind`	Arrays which will be filled with at most `maxcoefs` column of the nonzero elements for each row. May be `None` if not required.
`colcoef`	Array which will be filled with at most `maxcoefs` nonzero element values. May be `None` if not required.
`maxcoefs`	Maximum number of elements to be retrieved.
`first`	First row in the range.
`last`	Last row in the range.

Related topics

`problem.getCols`, `problem.getRowType`, `problem.getRows`.

problem.getrowstatus

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpGetRowStatus` *instead.*
Retrieve the status setting of a constraint

Synopsis

```
status = problem.getrowstatus(row)
```

Arguments

`row`	The index of the matrix row whose data is to be obtained.
`status`	The status settings.

Example

This recovers the status of the rows of the matrix of the current problem and reports those which are flagged as enforced constraints.

```
m = p.getintattrib('rows')
for i in range(m):
  status = p.getrowstatus(i)
  if(Status & 0x800) print("Row {0} is enforced".format(i))
```

Further information

See the section on bitmap settings of the XSLP reference manual for details on the possible information in `Status`.

Related topics

`problem.slpChgRowStatus`, `problem.slpGetRowStatus`

problem.getrowtype

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getRowType` *instead.*
Returns the row types for the rows in a given range.

Synopsis

```
problem.getrowtype(rowtype, first, last)
```

Arguments

`rowtype`	Character array of length `last-first+1` characters where the row types will be returned: `N` indicates a free constraint; `L` indicates a $\leqslant$ constraint; `E` indicates an = constraint; `G` indicates a $\geqslant$ constraint; `R` indicates a range constraint.
`first`	First row in the range.
`last`	Last row in the range.

Example

The following example retrieves the type of the first four rows of the problem into an array `qrt`:

```
qrt = []
p.getrowtype(qrt, 0, 3)
```

Related topics

`problem.chgRowType`, `problem.getRows`, `problem.getRowType`.

problem.getrowwt

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpGetRowWT` instead.
Get the initial penalty error weight for a row

Synopsis

```
value = problem.getrowwt(row)
```

Arguments

`row`	The row (i.e., `xpress.constraint` object, index, or name) whose weight is to be retrieved.
`value`	The value of the weight.

Example

The following example gets the initial weight of row number 2.

```
value = p.getrowwt(2)
```

Further information

The initial row weight is used only when the augmented structure is created. After that, the current weighting can be accessed using `problem.slpGetRowInfo`.

Related topics

`problem.slpChgRowWt`, `problem.slpGetRowInfo`, `problem.slpGetRowWT`

problem.getscaledinfeas

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getScaledInfeas` *instead.* Returns a list of scaled infeasible primal and dual variables for the original problem. If the problem is currently presolved, it is postsolved before the function returns.

Synopsis

```
problem.getscaledinfeas(x, slack, duals, djs)
```

Arguments

`x`	Array to store the primal infeasible variables. May be `None` if not required.
`slack`	Array to store the primal infeasible rows. May be `None` if not required.
`duals`	Array to store the dual infeasible rows. May be `None` if not required.
`djs`	Array to store the dual infeasible variables. May be `None` if not required.

Example

```
x = []
slack = []
duals = []
djs = []
p.getscaledinfeas(x, slack, duals, djs)
```

Related topics

`problem.getInfeas`, `problem.getIISData`, `problem.IISAll`, `problem.clearIIS`, `problem.firstIIS`, `problem.IISIsolations`, `problem.nextIIS`, `problem.IISStatus`, `problem.writeIIS`, `problem.getScaledInfeas`.

problem.getSlack

Purpose

This subroutine is deprecated and will be removed in a future release. See `problem.getSlacks`.

problem.getslpsol

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getSolution` *and related functions instead.*
Obtain the solution values for the most recent SLP iteration

Synopsis

```
problem.getslpsol(x, slack, duals, djs)
```

Arguments

`x`	Array of length `problem.attributes.xslp_originalcols` to hold the values of the primal variables. May be `None` if not required.
`slack`	Array of length `problem.attributes.xslp_originalrows` to hold the values of the slack variables. May be `None` if not required.
`duals`	Array of length `problem.attributes.xslp_originalrows` to hold the values of the dual variables. May be `None` if not required.
`djs`	Array of length `problem.attributes.xslp_originalcols` to hold the recuded costs of the primal variables. May be `None` if not required.

Example

The following code fragment recovers the values and reduced costs of the primal variables from the most recent SLP iteration:

```
ncol = p.getintattrib(prob,xpress.xslp_originalcols)
val  = []
djs  = []
p.getslpsol(val,None,None,djs)
```

Further information

`getslpsol` can be called at any time after an SLP iteration has completed, and will return the same values even if the problem is subsequently changed. `getslpsol` returns values for the columns and rows originally in the problem and not for any augmentation rows or columns. To access the values of any augmentation columns or rows, use `getlpsol`; accessing the augmented solution is only recommended if `xslp_presolvelevel` indicates that the problem dimensions should not be changed in presolve.

Related topics

`problem.getSolution`.

problem.gettolset

Purpose

This subroutine is deprecated and will be removed in a future release.
Retrieve the values of a set of convergence tolerances for an SLP problem

Synopsis

```
status = problem.gettolset(tolset, tols)
```

Arguments

`tolset`	The index of the tolerance set.
`status`	The bit-map of status settings.
`tols`	Array of 9 double-precision values to hold the tolerances. May be `None` if not required.

Example

The following example retrieves the values for tolerance set 3 and prints those which are set:

```
tols = []
status = p.gettolset(3, tols)
for i in range(9):
  if status &(1<<i):
    print("Tolerance {0} = {1}".format(i,tols[i]))
```

Further information

If `tols` is `None`, then the corresponding information will not be returned.

If `tols` is not `None`, then a set of 9 values will always be returned. `status` indicates which of these values is active as follows. Bit `n` of `status` is set if `tols[n]` is active, where `n` is:

Entry / Bit	Tolerance	XSLP constant	XSLP bit constant
0	Closure tolerance (TC)	`xslp_TOLSET_TC`	`xslp_TOLSETBIT_TC`
1	Absolute delta tolerance (TA)	`xslp_TOLSET_TA`	`xslp_TOLSETBIT_TA`
2	Relative delta tolerance (RA)	`xslp_TOLSET_RA`	`xslp_TOLSETBIT_RA`
3	Absolute coefficient tolerance (TM)	`xslp_TOLSET_TM`	`xslp_TOLSETBIT_TM`
4	Relative coefficient tolerance (RM)	`xslp_TOLSET_RM`	`xslp_TOLSETBIT_RM`
5	Absolute impact tolerance (TI)	`xslp_TOLSET_TI`	`xslp_TOLSETBIT_TI`
6	Relative impact tolerance (RI)	`xslp_TOLSET_RI`	`xslp_TOLSETBIT_RI`
7	Absolute slack tolerance (TS)	`xslp_TOLSET_TS`	`xslp_TOLSETBIT_TS`
8	Relative slack tolerance (RS)	`xslp_TOLSET_RS`	`xslp_TOLSETBIT_RS`

The xslp_TOLSET constants can be used to access the corresponding entry in the value arrays, while the xslp_TOLSETBIT constants are used to set or retrieve which tolerance values are used for a given SLP variable.

Related topics

Related topics

`problem.addtolsets`, `problem.chgtolset`, `problem.deltolsets`, `problem.loadtolsets`

problem.getub

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getUB` *instead.*
Returns the upper bounds on the columns in a given range.

Synopsis

```
problem.getub(ub, first, last)
```

Arguments

`ub`	Array where the `last - first + 1` upper bounds are to be placed.
`first`	First column in the range.
`last`	Last column in the range.

Related topics

`problem.chgBounds`, `problem.getLB`, `problem.getUB`.

problem.getunbvec

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getUnbVec` *instead.*
Returns the index vector which causes the primal simplex or dual simplex algorithm to determine that a problem is primal or dual unbounded respectively.

Synopsis

```
junb = problem.getunbvec()
```

Further information

When solving using the dual simplex method, if the problem is primal infeasible then `getunbvec` returns the pivot row where dual unboundedness was detected. Also note that when solving using the dual simplex method, if the problem is primal unbounded then `getunbvec` returns -1 since the problem is dual infeasible and not dual unbounded.

Related topics

`problem.getInfeas`, `problem.lpOptimize`, `problem.getUnbVec`.

problem.getvar

Purpose

This subroutine is deprecated and will be removed in a future release.
Retrieve information about an SLP variable

Synopsis

```
(detrow, initstepbound, stepbound, penalty, damp, initial, value, tolset, history,
       converged, vartype, delta, penaltydelta, updaterow, oldvalue) =
       problem.getvar(col)
```

Arguments

`col`	The column (i.e., `xpress.var` object, index, or name).
`detrow`	An integer to receive the index of the determining row. May be `None` if not required.
`initstepbound`	A double precision variable to receive the value of the initial step bound of the variable. May be `None` if not required.
`stepbound`	A double precision variable to receive the value of the current step bound of the variable. May be `None` if not required.
`penalty`	A double precision variable to receive the value of the penalty delta weighting of the variable. May be `None` if not required.
`damp`	A double precision variable to receive the value of the current damping factor of the variable. May be `None` if not required.
`initial`	A double precision variable to receive the value of the initial value of the variable. May be `None` if not required.
`value`	A double precision variable to receive the current activity of the variable. May be `None` if not required.
`tolset`	An integer to receive the index of the tolerance set of the variable. May be `None` if not required.
`history`	An integer to receive the SLP history of the variable. May be `None` if not required.
`converged`	An integer to receive the convergence status of the variable as defined in the "Convergence Criteria" section (The returned value will match the numbering of the tolerances). May be `None` if not required.
`vartype`	An integer to receive the status settings (a bitmap defining the existence of certain properties for this variable). The following bits are defined: `Bit 1:` Variable has a delta vector `Bit 2:` Variable has an initial value `Bit 14:` Variable is the reserved "=" column Other bits are reserved for internal use. May be `None` if not required.
`delta`	An integer to receive the index of the delta vector for the variable. May be `None` if not required.
`penaltydelta`	An integer to receive the index of the first penalty delta vector for the variable. The second penalty delta immediately follows the first. May be `None` if not required.
`updaterow`	An integer to receive the index of the update row for the variable. May be `None` if not required.
`oldvalue`	A double precision variable to receive the value of the variable at the previous SLP iteration. May be `None` if not required.

Example

The following example retrieves the current value, convergence history and status for column 3.

```
(a,b,c,d,e,value,g,history,converged,j,k,i,h,k,l) = p.getvar(3)
```

Further information

If `col` refers to a column which is not an SLP variable, then all the return values will indicate that there is no corresponding data.

`detrow` will be set to -1 if there is no determining row.

`delta`, `penaltydelta` and `updaterow` will be set to -1 if there is no corresponding item.

Related topics

`problem.addvars`, `problem.chgvar`, `problem.delvars`, `problem.loadvars`

problem.hasdualray

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getDualRay` *instead.* Returns true if a dual ray (dual unbounded direction) exists for the current problem, if the problem is found to be infeasible.

Synopsis

```
v = problem.hasdualray()
```

Related topics

`problem.getDualRay`.

problem.hasprimalray

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getPrimalRay` *instead.* Returns true if a primal ray (primal unbounded direction) exists for the current problem, if the problem is found to be unbounded.

Synopsis

```
v = problem.hasprimalray()
```

Related topics

`problem.getPrimalRay`.

problem.iisall

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.IISAll` *instead.*
Performs an automated search for independent Irreducible Infeasible Sets (IIS) in an infeasible problem.

Synopsis

```
problem.iisall()
```

Example

This example searches for IISs and then questions the problem attribute `NUMIIS` to determine how many were found:

```
p.iisall()
print("The problem has {0} IISs".format(p.attributes.numiis))
```

Further information

1. A model may have several infeasibilities. Repairing a single IIS may not make the model feasible. For this reason the Optimizer can find an IIS for each of the infeasibilities in a model. If the control `MAXIIS` is set to a positive integer value then the `problem.IISAll` function will stop if `MAXIIS` IISs have been found. By default the control `MAXIIS` is set to `-1`, in which case an IIS is found for each of the infeasibilities in the model.
2. The problem attribute `NUMIIS` allows the user to recover the number of IISs found in a particular search. Alternatively, the `problem.IISStatus` function may be used to retrieve the number of IISs found by the `problem.firstIIS`, `problem.nextIIS`, or `problem.IISAll` functions.

Related topics

`problem.getIISData`, `problem.clearIIS`, `problem.firstIIS`, `problem.IISIsolations`, `problem.nextIIS`, `problem.IISStatus`, `problem.writeIIS`, `problem.IISAll`.

problem.iisclear

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.clearIIS` *instead.*
Resets the search for Irreducible Infeasible Sets (IIS).

Synopsis

```
problem.iisclear()
```

Further information

1. The information stored internally about the IISs identified by `problem.firstIIS`, `problem.nextIIS` or `problem.IISAll` are cleared. Functions `problem.getIISData`, `problem.IISStatus`, `problem.writeIIS` and `problem.IISIsolations` cannot be called until the IIS identification procedure is started again.
2. This function is automatically called by `problem.firstIIS` and `problem.IISAll`.

Related topics

`problem.getIISData`, `problem.IISAll`, `problem.firstIIS`, `problem.IISIsolations`, `problem.nextIIS`, `problem.IISStatus`, `problem.writeIIS`, `problem.clearIIS`.

problem.iisfirst

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.firstIIS` *instead.*
Initiates a search for an Irreducible Infeasible Set (IIS) in an infeasible problem. The returned value can be 0 for success, 1 if the problem is feasible, or 2 in case of error.

Synopsis

```
status_code = problem.iisfirst(mode)
```

Argument

`mode`	The IIS search mode:
`0`	stops after finding the initial infeasible subproblem;
`1`	find an IIS, emphasizing simplicity of the IIS;
`2`	find an IIS, emphasizing a quick result.

Example

This looks for the first IIS.

```
p.iisfirst(1)
```

Further information

1. A model may have several infeasibilities. Repairing a single IIS may not make the model feasible. For this reason the Optimizer can find an IIS for each of the infeasibilities in a model. For the generation of several independent IISs use functions `problem.nextIIS` or `problem.IISAll`.
2. IIS sensitivity filter: after an optimal but infeasible first phase primal simplex, it is possible to identify a subproblem containing all the infeasibilities (corresponding to the given basis) to reduce the size of the IIS working problem dramatically, i.e., rows with zero duals (thus with artificials of zero reduced cost) and columns that have zero reduced costs may be deleted. Moreover, for rows and columns with nonzero costs, the sign of the cost is used to relax equality rows either to less than or greater than equal rows, and to drop either possible upper or lower bounds on columns.
3. Initial infeasible subproblem: The subproblem identified after the sensitivity filter is referred to as initial infeasible subproblem. Its size is crucial to the running time of the deletion filter and it contains all the infeasibilities of the first phase simplex, thus if the corresponding rows and bounds are removed the problem becomes feasible.
4. `problem.firstIIS` performs the initial sensitivity analysis on rows and columns to reduce the problem size, and sets up the initial infeasible subproblem. This subproblem significantly speeds up the generation of IISs, however in itself it may serve as an approximation of an IIS, since its identification typically takes only a fraction of time compared to the identification of an IIS.
5. The IIS approximation and the IISs generated so far are always available.

Related topics

`problem.getIISData`, `problem.IISAll`, `problem.clearIIS`, `problem.IISIsolations`, `problem.nextIIS`, `problem.IISStatus`, `problem.writeIIS`, `problem.firstIIS`.

problem.iisisolations

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.IISIsolations` *instead.*
Performs the isolation identification procedure for an Irreducible Infeasible Set (IIS).

Synopsis

```
problem.iisisolations(iis)
```

Argument

`iis`	The number of the IIS identified by either `problem.firstIIS`, `problem.nextIIS`, or `problem.IISAll` in which the isolations should be identified.

Example

This example finds the first IIS and searches for the isolations in that IIS.

```
if p.iisfirst(1) == 0:
    iisisolations(1)
```

Further information

1. An IIS isolation is a special constraint or bound in an IIS. Removing an IIS isolation constraint or bound will remove all infeasibilities in the IIS without increasing the infeasibilities in any row or column outside the IIS, thus in any other IISs. The IIS isolations thus indicate the likely cause of each independent infeasibility and give an indication of which constraint or bound to drop or modify. It is not always possible to find IIS isolations.
2. Generally, one should first look for rows or columns in the IIS which are both in isolation, and have a high dual multiplier relative to the others.
3. The `iis` parameter cannot be zero: the concept of isolations is meaningless for the initial infeasible subproblem.

Related topics

`problem.getIISData`, `problem.IISAll`, `problem.clearIIS`, `problem.firstIIS`, `problem.nextIIS`, `problem.IISStatus`, `problem.writeIIS`, `problem.IISIsolations`.

problem.iisnext

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.nextIIS` instead.
Continues the search for further Irreducible Infeasible Sets (IIS), or calls `problem.firstIIS` if no IIS has been identified yet. The returned value is 0 in case of success; 1 if no more IIS could be found, or problem is feasible if no `problem.firstIIS` call preceded; or 2 in case of an error.

Synopsis

```
status_code = problem.iisnext()
```

Example

This looks for a further IIS.

```
while p.iisnext() == 0:
   [...] # do something with the iis
```

Further information

1. A model may have several infeasibilities. Repairing a single IIS may not make the model feasible. For this reason the Optimizer attempts to find an IIS for each of the infeasibilities in a model. Call the `problem.nextIIS` function repeatedly, or use the `problem.IISAll` function to retrieve all IIS at once.
2. This function is not affected by the control `MAXIIS`.
3. If the problem has been modified since the last call to `problem.firstIIS` or `problem.nextIIS`, the generation process has to be started from scratch.

Related topics

`problem.getIISData`, `problem.IISAll`, `problem.clearIIS`, `problem.firstIIS`, `problem.IISIsolations`, `problem.IISStatus`, `problem.writeIIS`, `problem.nextIIS`.

problem.iisstatus

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.IISStatus` *instead.*
Returns statistics on the Irreducible Infeasible Sets (IIS) found so far by `problem.firstIIS`, `problem.nextIIS`, or `problem.IISAll`. The returned value is the number of IISs found so far.

Synopsis

```
iiscount = problem.iisstatus(nrows, ncols, suminfeas, numinfeas)
```

Arguments

`nrows`	Number of rows in the IISs.
`ncols`	Number of bounds in the IISs.
`suminfeas`	The sum of infeasibilities in the IISs after the first phase simplex.
`numinfeas`	The number of infeasible variables in the IISs after the first phase simplex.

Example

This example first retrieves the number of IISs found so far, and then retrieves their main properties. Note that the arrays have size `count+1`, since the first index is reserved for the initial infeasible subset.

```
rs = []
cs = []
ninf = []
p.iisstatus(rs, cs, numinfeas=ninf) # suminf is not of interest
```

Further information

1. The arrays are `0` based, index 0 corresponding to the initial infeasible subproblem.
2. The arrays may be `None` if not required.
3. For the initial infeasible problem (at position 0) the subproblem size is returned (which may be different from the number of bounds), while for the IISs the number of bounds is returned (usually much smaller than the number of columns in the IIS).
4. Note that the values in `suminfeas` and `numinfeas` heavily depend on the actual basis where the simplex has stopped.
5. `iiscount` is set to `-1` if the search for IISs has not yet started.

Related topics

`problem.getIISData`, `problem.IISAll`, `problem.clearIIS`, `problem.firstIIS`, `problem.IISIsolations`, `problem.nextIIS`, `problem.writeIIS`, `problem.IISStatus`.

problem.iiswrite

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.writeIIS` *instead.*
Writes an LP/MPS/CSV file containing a given Irreducible Infeasible Set (IIS). If 0 is passed as the IIS number parameter, the initial infeasible subproblem is written.

Synopsis

```
problem.iiswrite(iis, filename, filetype, flags)
```

Arguments

`iis`	The ordinal number of the IIS to be written.
`filename`	The name of the file to be created.
`filetype`	Type of file to be created:
`0`	creates an lp/mps file containing the IIS as a linear programming problem;
`1`	creates a comma separated (csv) file containing the description and supplementary information on the given IIS.
`flags`	Flags passed to the `problem.writeProb` function.

Example

This writes the first IIS (if one exists and is already found) as an lp file.

```
p.iiswrite(1, "iis.lp", 0, "l")
```

Further information

1. Please note that there are problems on the boundary of being infeasible or not. For such problems, feasibility or infeasibility often depends on tolerances or even on scaling. This phenomenon makes it possible that after writing an IIS out as an LP file and reading it back, it may report feasibility. As a first check it is advised to consider the following options:
    1. save the IIS using MPS hexadecimal format to eliminate rounding errors associated with conversion between internal and decimal representation.
    2. turn presolve off since the nature of an IIS makes it necessary that during their identification the presolve is turned off.
    3. use the primal simplex method to solve the problem.
2. Note that the original sense of the original objective function plays no role in an IIS.
3. Even though an attempt is made to identify the most infeasible IISs first by the `problem.firstIIS`, `problem.nextIIS`, and `problem.IISAll` functions, it is also possible that an IIS becomes just infeasible in problems that are otherwise highly infeasible. In such cases, it is advised to try to deal with the more stable IISs first, and consider to use the infeasibility breaker tool if only slight infeasibilities remain.
4. The LP or MPS files created by `problem.writeIIS` corresponding to an IIS contain no objective function, since infeasibility is independent from the objective.

Related topics

`problem.getIISData`, `problem.IISAll`, `problem.clearIIS`, `problem.firstIIS`, `problem.IISIsolations`, `problem.nextIIS`, `problem.IISStatus`, `problem.writeIIS`.

problem.loadbasis

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.loadBasis` *instead.*
Loads a basis as specified by the user.

Synopsis

```
problem.loadbasis(rowstat, colstat)
```

Arguments

`rowstat` Array of length `problem.attributes.rows` containing the basis status of the slack, surplus or artificial variable associated with each row. The status must be one of:

- `0` slack, surplus or artificial is non-basic at lower bound;
- `1` slack, surplus or artificial is basic;
- `2` slack or surplus is non-basic at upper bound.
- `3` slack or surplus is super-basic.

`colstat` Array of length `problem.attributes.cols` containing the basis status of each of the columns in the constraint matrix. The status must be one of:

- `0` variable is non-basic at lower bound or superbasic at zero if the variable has no lower bound;
- `1` variable is basic;
- `2` variable is at upper bound;
- `3` variable is super-basic.

Example

This example loads a problem and then reloads a (previously optimized) basis from a similar problem to speed up the optimization:

```
p.read("problem", "")
p.loadbasis(rstatus, cstatus)
p.lpoptimize("")
```

Further information

If the problem has been altered since saving an advanced basis, one can alter the basis as follows before loading it:

- Make new variables non-basic at their lower bound (`cstatus[icol]=0`), unless a variable has an infinite lower bound and a finite upper bound, in which case make the variable non-basic at its upper bound (`cstatus[icol]=2`);
- Make new constraints basic (`rstatus[jrow]=1`);
- Try not to delete basic variables, or non-basic constraints.

Related topics

`problem.getBasis`, `problem.getPresolveBasis`, `problem.loadPresolveBasis`, `problem.loadBasis`.

problem.loadbranchdirs

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.loadBranchDirs` *instead.* Loads directives into the current problem to specify which MIP entities the Optimizer should continue to branch on when a node solution is integer feasible.

Synopsis

```
problem.loadbranchdirs(colind, dir)
```

Arguments

`colind`	Array containing the column numbers. A negative value indicates a set number (the first set being -1, the second -2, and so on).
`dir`	Array containing either 0 or 1 for the entities given in `colind`. Entities for which dir is set to 1 will be branched on until fixed before a integer feasible solution is returned. If `dir` is `None`, the branching directive will be set for all entities in `colind`.

Related topics

`problem.loadDirs`, `problem.readDirs`, `problem.loadBranchDirs`.

problem.loadcoefs

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpLoadCoefs` *instead.*
Load non-linear coefficients into the SLP problem

Synopsis

```
problem.loadcoefs(rowindex, colindex, factor, fstart, parsed, type, value)
```

Arguments

`rowindex`	Integer array holding index of row for the coefficient.
`colindex`	Integer array holding index of column for the coefficient.
`factor`	Double array holding factor by which formula is scaled. If this is `None`, then a value of 1.0 will be used.
`fstart`	Integer array holding the start position in the arrays `Type` and `Value` of the formula for the coefficients. `fstart[nSLPCoef]` should be set to the next position after the end of the last formula.
`parsed`	Boolean indicating whether the token arrays are formatted as internal unparsed (`parsed=False`) or internal parsed reverse Polish (`parsed=False`).
`type`	Array of token types providing the formula for each coefficient.
`value`	Array of values corresponding to the types in `Type`.

Example

Assume that the rows and columns of `Prob` are named `Row1`, `Row2` ..., `Col1`, `Col2` ... The following example loads coefficients representing:
`Col2 * Col3 + Col6 * Col2^2` into `Row1` and
`Col2 ^ 2` into `Row3`.

```
rowindex = [Row1,Row1,Row3]
colindex = [Col2,Col6,Col2]

formulastart = []
type = []
value = []

formulastart.append(len(type))
type.append(xp.constants.TOK_COL); value.append(3)
type.append(xp.constants.TOK_EOF); value.append(0)

formulastart.append(len(type))
type.append(xp.constants.TOK_COL); value.append(2)
type.append(xp.constants.TOK_COL); value.append(2)
type.append(xp.constants.TOK_OP);  value.append(xp.constants.OP_MULTIPLY)
type.append(xp.constants.TOK_EOF); value.append(0)

formulastart.append(len(type))
type.append(xp.constants.TOK_COL); value.append(2)
type.append(xp.constants.TOK_EOF); value.append(0)

formulastart.append(len(type))

p.loadcoefs(rowindex, colindex, None, formulastart, True, type, value)
```

The first coefficient in `Row1` is in `Col2` and has the formula `Col3`, so it represents `Col2 * Col3`.

The second coefficient in `Row1` is in `Col6` and has the formula `Col2 * Col2` so it represents `Col6 * Col2^2`. The formulae are described as *parsed* (`parsed=True`), so the formula is written as
`Col2 Col2 *`
rather than the unparsed form
`Col2 * Col2`

The last coefficient, in `Row3`, is in `Col2` and has the formula `Col2`, so it represents `Col2 * Col2`.

Further information

The j^{th} coefficient is made up of two parts: `Factor` and `Formula`. `Factor` is a constant multiplier, which can be provided in the `Factor` array. If Xpress Nonlinear can identify a constant factor in `Formula`, then it will use that as well, to minimize the size of the formula which has to be calculated. `Formula` is made up of a list of tokens in `Type` and `Value` starting at `fstart[j]`. The tokens follow the rules for parsed or unparsed formulae as indicated by the setting of `parsed`. The formula must be terminated with an `xslp_op_eof` token. If several coefficients share the same formula, they can have the same value in `fstart`. For possible token types and values see the chapter on Formula Parsing in the SLP reference manual.

The `loadcoefs` function loads items into the SLP problem. Any existing items of the same type are deleted first. The corresponding `addcoefs` function adds or replace items leaving other items of the same type unchanged.

Related topics

`problem.slpAddCoefs`, `problem.slpChgCoef`, `problem.slpChgCoefStr`, `problem.slpGetCoefStr`,`problem.slpLoadCoefs`

problem.loadcuts

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.loadCuts` *instead.*
Loads cuts from the cut pool into the matrix. Without calling `loadcuts` the cuts will remain in the cut pool but will not be active at the node. Cuts loaded at a node remain active at all descendant nodes unless they are deleted using `problem.delCuts`.

Synopsis

```
problem.loadcuts(coltype, interp, cutind)
```

Arguments

`cuttype`	Cut type.
`interp`	The way in which the cut type is interpreted: `-1` load all cuts; `1` treat cut types as numbers; `2` treat cut types as bit maps - load cut if any bit matches any bit set in `coltype`; `3` treat cut types as bit maps - `0` load cut if all bits match those set in `coltype`.
`cutind`	Array containing the cuts to be loaded into the matrix.

Further information

This function should be called only from within callback functions set by either `problem.addOptnodeCallback`, `problem.addPrenodeCallback`, `problem.addNodeLPSolvedCallback` or `problem.addPreIntsolCallback`.

Related topics

`problem.addCuts`, `problem.delCPCuts`, `problem.delCuts`, `problem.getCPCutList`, `problem.loadCuts`, Section "Working with the cut manager" of the Xpress Optimizer reference manual.

problem.loaddelayedrows

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.loadDelayedRows` *instead.* Specifies that a set of rows in the problem will be treated as delayed rows during a tree search. These are rows that must be satisfied for any integer solution, but will not be loaded into the active set of constraints until required.

Synopsis

```
problem.loaddelayedrows(rowind)
```

Argument

`rowind` An array of rows (i.e., `xpress.constraint` objects, indices, or names) to treat as delayed rows.

Example

This sets the first six matrix rows as delayed rows in the MIP problem `prob`.

```
p.loaddelayedrows([0,1,2,3,4,5])
p.mipoptimize("")
```

Further information

Delayed rows must be set up before solving the problem. Any delayed rows will be removed from the problem after presolve and added to a special pool. A delayed row will be added back into the active matrix only when such a row is violated by an integer solution found by the Optimizer.

Related topics

`problem.loadModelCuts`, `problem.loadDelayedRows`.

problem.loaddfs

Purpose

This subroutine is deprecated and will be removed in a future release.
Load a set of distribution factors

Synopsis

```
problem.loaddfs(colindex, rowindex, value)
```

Arguments

`colindex`	Array of columns whose distribution factor is to be changed.
`rowindex`	Array of rows where each distribution factor applies.
`value`	Array of the new values of the distribution factors.

Example

The following example loads distribution factors as follows:
column 282 in row 134 = 0.1
column 282 in row 136 = 0.15
column 285 in row 133 = 1.0.
Any other first-order derivative placeholders are set to `xslp_DELTA_Z`.

```
colindex = [282, 282, 285]
rowindex = [134, 136, 133]
value    = [0.1, 0.15,  1]
p.loaddfs(colindex, rowindex, value)
```

Further information

The *distribution factor* of a column in a row is the matrix coefficient of the corresponding delta vector in the row. Distribution factors are used in conventional recursion models, and are essentially normalized first-order derivatives. Xpress SLP can accept distribution factors instead of initial values, provided that the values of the variables involved can all be calculated after optimization using determining rows, or by a callback.

The `adddfs` functions load additional items into the SLP problem. The corresponding `loaddfs` functions delete any existing items first.

Related topics

`problem.adddfs`, `problem.chgdf`, `problem.getdf`

problem.loaddirs

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.loadDirs` *instead.*
Loads directives into the problem.

Synopsis

```
problem.loaddirs(colind, priority, dir, uppseudo, downpseudo)
```

Arguments

`colind`	Array containing the column numbers. A negative value indicates a set number (the first set being `-1`, the second `-2`, and so on).
`priority`	Array containing the priorities for the columns or sets. Priorities must be between 0 and 1000. May be `None` if not required.
`dir`	Character array specifying the branching direction for each column or set: `U` the entity is to be forced up; `D` the entity is to be forced down; `N` not specified. May be `None` if not required.
`uppseudo`	Array containing the up pseudo costs for the columns or sets. May be `None` if not required.
`downpseudo`	Array containing the down pseudo costs for the columns or sets. May be `None` if not required.

Related topics

`problem.getDirs`, `problem.loadPresolveDirs`, `problem.readDirs`, `problem.loadDirs`.

problem.loadlpsol

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.loadLPSol` *instead.*
Loads an LP solution for the problem into the Optimizer. The returned status is either 0 if the solution is loaded or 1 if the solution is not loaded because the problem is in presolved status.

Synopsis

```
status = problem.loadlpsol(x, slack, duals, djs)
```

Arguments

`x`	Optional: Array of length `problem.attributes.cols` (for the original problem and not the presolve problem) containing the values of the variables.
`slack`	Optional: double array of length `problem.attributes.rows` containing the values of slack variables.
`duals`	Optional: double array of length `problem.attributes.rows` containing the values of duals variables.
`djs`	Optional: double array of length `problem.attributes.cols` containing the values of reduced costs.

Example

This example loads a problem and loads a solution for the problem.

```
p.read("problem", "")
status = p.loadlpsol(x, None, duals, None)
```

Further information

1. At least one of variables `x` and duals variables `duals` must be provided.
2. When variables `x` is `None`, the variables will be set to their bounds.
3. When slack variables `slack` is `None`, it will be computed from variables `x`. If slacks are provided, variables cannot be omitted.
4. When duals variables `duals` is `None`, both duals variables and reduced costs will be set to zero.
5. When reduced costs `djs` is `None`, it will be computed from duals variables `duals`. If reduced costs are provided, duals variables cannot be omitted.

Related topics

`problem.getSolution`, `problem.loadLPSol`.

problem.loadmipsol

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.loadMipSol` *instead.*
Loads a MIP solution for the problem into the Optimizer. The returned status is one of the following values:

- -1: Solution rejected because an error occurred;
- 0: Solution accepted. When loading a solution before a MIP solve, the solution is always accepted. See Further Information below.
- 1: Solution rejected because it is infeasible;
- 2: Solution rejected because it is cut off;
- 3: Solution rejected because the LP reoptimization was interrupted.

Synopsis

```
status = problem.loadmipsol(x)
```

Argument

`x`	Array of length `problem.attributes.cols` (for the original problem and not the presolve problem) containing the values of the variables.

Example

This example loads a problem and then loads a solution found previously for the problem to help speed up the MIP search:

```
p.read("problem", "")
status = p.loadmipsol(x)
p.mipoptimize("")
```

Further information

1. When a solution is loaded before a MIP solve, the solution is simply placed in temporary storage until the MIP solve is started. Only after the MIP solve has commenced and any presolve has been applied, will the loaded solution be checked and possibly accepted as a new incumbent integer solution. There are no checks performed on the solution before the MIP solve and the returned status in `problem.loadMipSol` will always be `0` for accepted.
2. Solutions can be loaded during a MIP solve using the optnode callback function. Any solution loaded this way is immediately checked and the returned status will be one of the values `0` through `3`.
3. Loaded solution values will automatically be adjusted to fit within the current problem bounds.

Related topics

`problem.getSolution`, `problem.addOptnodeCallback`, `problem.loadMipSol`.

problem.loadmodelcuts

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.loadModelCuts` *instead.*
Specifies that a set of rows in the problem will be treated as model cuts.

Synopsis

```
problem.loadmodelcuts(rowind)
```

Argument

`rowind` An array of rows (i.e., `xpress.constraint` objects, indices, or names) to be treated as cuts.

Example

This sets the first six matrix rows as model cuts in the MIP problem `myprob`.

```
p.loadmodelcuts([0,1,2,3,4,5])
p.mipoptimize("")
```

Further information

1. During presolve the model cuts are removed from the problem and added to an internal cut pool. During the tree search, the Optimizer will regularly check this cut pool for any violated model cuts and add those that cuts off a node LP solution.
2. The model cuts must be ”true” model cuts, in the sense that they are redundant at the optimal MIP solution. The Optimizer does not guarantee to add all violated model cuts, so they must not be required to define the optimal MIP solution.

problem.loadpresolvebasis

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.loadPresolveBasis` *instead.*
Loads a presolved basis from the user's areas.

Synopsis

```
problem.loadpresolvebasis(rowstat, colstat)
```

Arguments

`rowstat` Array containing the basis status of the slack, surplus or artificial variable associated with each row. The status must be one of:

- 0 slack, surplus or artificial is non-basic at lower bound;
- 1 slack, surplus or artificial is basic;
- 2 slack or surplus is non-basic at upper bound.

`colstat` Array containing the basis status of each of the columns in the matrix. The status must be one of:

- 0 variable is non-basic at lower bound or superbasic at zero if the variable has no lower bound;
- 1 variable is basic;
- 2 variable is at upper bound;
- 3 variable is super-basic.

Example

The following example saves the presolved basis for one problem, loading it into another:

```
p1 = xpress.problem()
p2 = xpress.problem()

p1.read("myprob", "")
p1.controls.mipstopstage = xp.MIPStopStage.INITIALRELAXATION
p1.mipoptimize()
p1.setDefaultControl('mipstopstage')
rs = []
cs = []
p1.getpresolvebasis(rs, cs)

p2.read("myprob2", "")
p2.controls.mipstopstage = xp.MIPStopStage.INITIALRELAXATION
p2.mipoptimize()
p2.setDefaultControl('mipstopstage')
p2.loadpresolvebasis(rs, cs)
```

Related topics

`problem.getBasis`, `problem.getPresolveBasis`, `problem.loadBasis`, `problem.loadPresolveBasis`.

problem.loadpresolvedirs

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.loadPresolveDirs` *instead.*
Loads directives into the presolved matrix.

Synopsis

```
problem.loadpresolvedirs(colind, priority, dir, uppseudo, downpseudo)
```

Arguments

`colind`	Array containing the column numbers. A negative value indicates a set number (`-1` being the first set, `-2` the second, and so on).
`priority`	Array containing the priorities for the columns or sets. May be `None` if not required.
`dir`	Character array specifying the branching direction for each column or set: `U` the entity is to be forced up; `D` the entity is to be forced down; `N` not specified. May be `None` if not required.
`uppseudo`	Array containing the up pseudo costs for the columns or sets. May be `None` if not required.
`downpseudo`	Array containing the down pseudo costs for the columns or sets. May be `None` if not required.

Example

The following loads priority directives for column `0` in the problem:

```
p.controls.mipstopstage = xp.MIPStopStage.INITIALRELAXATION
p.mipoptimize()
p.setDefaultControl('mipstopstage')
p.loadpresolvedirs([0], [1], None, None, None)
p.mipoptimize("")
```

Related topics

`problem.getDirs`, `problem.loadDirs`, `problem.loadPresolveDirs`.

problem.loadproblem

Purpose

This subroutine is deprecated and will be removed in a future release.
Load an optimization problem, possibly with quadratic objective and/or constraints, and integer variables.

Synopsis

```
problem.loadproblem(probname, rowtype, rhs, rng, objcoef, start, collen, rowind,
       rowcoef, lb, ub, objqcol1, objqcol2, objqcoef, qrowind, nrowqcoefs, rowqcol1,
       rowqcol2, rowqcoef, coltype, entind, limit, settype, setstart, setind,
       refval, colnames, rownames, unlinked=False)
```

Arguments

`probname` A string of up to 200 characters containing the problem name.

`rowtype` Character array containing the row types:

- `L` indicates a <= constraint;
- `E` indicates an = constraint;
- `G` indicates a >= constraint;
- `R` indicates a range constraint;
- `N` indicates a nonbinding constraint.

`rhs` Array containing the right hand side coefficients of the rows. The right hand side value for a range row gives the **upper** bound on the row.

`rng` Array containing the range values for range rows. Values for all other rows will be ignored. May be `None` if there are no ranged constraints. The lower bound on a range row is the right hand side value minus the range value. The sign of the range value is ignored - the absolute value is used in all cases.

`objcoef` Array containing the objective function coefficients.

`start` Array containing the offsets in the `rowind` and `rowcoef` arrays of the start of the elements for each column. This array is of length equal to the number `ncol` of added variables or, if `collen` is `None`, `ncol+1`. If `collen` is `None` the extra entry of `start`, `start[ncol]`, contains the position in the `rowind` and `rowcoef` arrays at which an extra column would start, if it were present.

`collen` Array containing the number of nonzero elements in each column. May be `None` if all elements are contiguous and `start[ncol]` contains the offset where the elements for column `ncol+1` would start. This array is not required if the nonzero coefficients in the `rowind` and `rowcoef` arrays are continuous, and the `start` array has `ncol+1` entries as described above. It may be `None` if not required.

`rowind` Array containing the row indices for the nonzero elements in each column. If the indices are input contiguously, with the columns in ascending order, the length of the `rowind` is `start[ncol-1]+collen[ncol-1]` or, if `collen` is `None`, `start[ncol]`.

`rowcoef` Array containing the nonzero element values; length as for `rowind`.

`lb` Array containing the lower bounds on the columns. Use `-xpress.infinity` to represent a lower bound of minus infinity.

`ub` Array containing the upper bounds on the columns. Use `xpress.infinity` to represent an upper bound of plus infinity.

`objqcol1` (optional) Array with the first variable in each quadratic term.

`objqcol2` (optional) Array with the second variable in each quadratic term.

`objqcoef` (optional) Array with the quadratic coefficients.

`qrowind` (optional) Integer containing the indices of rows with quadratic matrices in them. Note that the rows are expected to be defined in `rowtype` as type `L`.

`nrowqcoefs` (optional) Array containing the number of nonzeros in each quadratic constraint matrix.

`rowqcol1` (optional) Array with a number of elements equal to the sum of the elements in `nrowqcoefs` (i.e., the total number of quadratic matrix elements in all the constraints). It contains the first column indices of the quadratic matrices. Indices for the first matrix are listed from `0` to `nrowqcoefs[0]-1`, for the second matrix from `nrowqcoefs[0]` to `nrowqcoefs[0]+nrowqcoefs[1]-1`, etc.

`rowqcol2` (optional) Array containing the second index for the quadratic constraint matrices.

`rowqcoef` (optional) Array containing the coefficients for the quadratic constraint matrices.

`coltype` Character array containing the entity types:
- `B` binary variables;
- `I` integer variables;
- `P` partial integer variables;
- `S` semi-continuous variables;
- `R` semi-continuous integer variables.

`entind` (optional) Array containing the variables of the MIP entities.

`limit` (optional) Array containing the integer limits for the partial integer variables and lower bounds for semi-continuous and semi-continuous integer variables (any entries in the positions corresponding to binary and integer variables will be ignored). May be `None` if not required.

`settype` (optional) Character array of length equal to the number of sets specified, `problem.attributes.nsets`, and specifies the set types:
- `1` SOS1 type sets;
- `2` SOS2 type sets.

May be `None` if not required.

`setstart` (optional) Array containing the offsets in the `setind` and `refval` arrays indicating the start of the sets. This array is of length `nsets+1`, the last member containing the offset where set `nsets+1` would start. May be `None` if not required.

`setind` (optional) Array of length `setstart[nsets]-1` containing the columns in each set. May be `None` if not required.

`refval` (optional) Array of length `setstart[nsets]-1` containing the reference row entries for each member of the sets. May be `None` if not required.

`colnames` (optional) Array of containing the column names for all variables added.

`rownames` (optional) Array of containing the row names for all constraints added.

`unlinked` (optional) If `True`, unlinked variables and constraints will be created (deprecated).

Further information

1. The objective function is of the form $c^T x + 1/2\ x^T Q x$ where `Q` is positive semi-definite for minimization problems and negative semi-definite for maximization problems. If this is not the case the optimization algorithms may converge to a local optimum or may not converge at all. Note that only the upper or lower triangular part of the `Q` matrix is specified.
2. All `Q` matrices in the constraints must be positive semi-definite. Note that only the upper or lower triangular part of the `Q` matrix is specified for constraints as well.
3. If indices are specified, both row and column indices are from `0` to `rows-1` and `0` to `cols-1` respectively.
4. Semi-continuous lower bounds are taken from the dlim array. If this is `None` then they are given a default value of `1.0`. If a semi-continuous variable has a positive lower bound then this will be used as the semi-continuous lower bound and the lower bound on the variable will be set to zero.

Related topics

`problem.readProb`, `problem.loadLP`, `problem.loadMIP`, `problem.loadMIQP`, `problem.loadQP`, `problem.loadMIQCQP`, `problem.loadQCQP`.

problem.loadsecurevecs

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.loadSecureVecs` *instead.* Allows the user to mark rows and columns in order to prevent the presolve removing these rows and columns from the problem.

Synopsis

```
problem.loadsecurevecs(rowind, colind)
```

Arguments

`rowind`	Array containing the rows to be marked. May be `None` if not required.
`colind`	Array containing the columns to be marked. May be `None` if not required.

Example

This sets the first six rows and the first four columns to not be removed during presolve.

```
p.read("myprob", "")
p.loadsecurevecs(rowind=[0,1,2,3,4,5], colind=[0,1,2,3])
p.mipoptimize("")
```

problem.loadtolsets

Purpose

This subroutine is deprecated and will be removed in a future release.
Load sets of standard tolerance values into an SLP problem

Synopsis

```
problem.loadtolsets(slptol)
```

Argument

`tol` Array of 9h items containing the 9 tolerance values for each set in order.

Example

The following example creates two tolerance sets: the first has values of 0.005 for all tolerances; the second has values of 0.001 for relative tolerances (numbers 2,4,6,8), values of 0.01 for absolute tolerances (numbers 1,3,5,7) and zero for the closure tolerance (number 0).

```
tol = 9*[0.005]+[0]+[0.01,0.001]*4
p.loadtolsets(tol)
```

Further information

A tolerance set is an array of 9 values containing the following tolerances:

Entry / Bit	Tolerance	XSLP constant	XSLP bit constant
0	Closure tolerance (TC)	`xslp_TOLSET_TC`	`xslp_TOLSETBIT_TC`
1	Absolute delta tolerance (TA)	`xslp_TOLSET_TA`	`xslp_TOLSETBIT_TA`
2	Relative delta tolerance (RA)	`xslp_TOLSET_RA`	`xslp_TOLSETBIT_RA`
3	Absolute coefficient tolerance (TM)	`xslp_TOLSET_TM`	`xslp_TOLSETBIT_TM`
4	Relative coefficient tolerance (RM)	`xslp_TOLSET_RM`	`xslp_TOLSETBIT_RM`
5	Absolute impact tolerance (TI)	`xslp_TOLSET_TI`	`xslp_TOLSETBIT_TI`
6	Relative impact tolerance (RI)	`xslp_TOLSET_RI`	`xslp_TOLSETBIT_RI`
7	Absolute slack tolerance (TS)	`xslp_TOLSET_TS`	`xslp_TOLSETBIT_TS`
8	Relative slack tolerance (RS)	`xslp_TOLSET_RS`	`xslp_TOLSETBIT_RS`

The xslp_TOLSET constants can be used to access the corresponding entry in the value arrays, while the xslp_TOLSETBIT constants are used to set or retrieve which tolerance values are used for a given SLP variable.

Once created, a tolerance set can be used to set the tolerances for any SLP variable. If a tolerance value is zero, then the default tolerance will be used instead. To force the use of a tolerance, use the `problem.chgtolset` function and set the `Status` variable appropriately.

See the section "Convergence Criteria" in the SLP reference manual for a fuller description of tolerances and their uses. The `loadtolsets` functions load items into the SLP problem. Any existing items of the same type are deleted first. The corresponding `addtolsets` functions add or replace items leaving other items of the same type unchanged.

Related topics

`problem.addtolsets`, `problem.deltolsets`, `problem.chgtolset`, `problem.gettolset`

problem.loadvars

Purpose

This subroutine is deprecated and will be removed in a future release.
Load SLP variables defined as matrix columns into an SLP problem

Synopsis

```
problem.loadvars (colindex, vartype, detrow, seqnum, tolindex, initvalue, stepbound)
```

Arguments

`colindex` Integer array holding the index of the matrix column corresponding to each SLP variable.

`vartype` Bitmap giving information about the SLP variable as follows (note that Bit numbering begins at zero):
`Bit 1` Variable has a delta vector;
`Bit 2` Variable has an initial value;
`Bit 14` Variable is the reserved "=" column;
May be `None` if not required.

`detrow` Integer array holding the index of the determining row for each SLP variable (a negative value means there is no determining row)
May be `None` if not required.

`seqnum` Integer array holding the index sequence number for cascading for each SLP variable (a zero value means there is no pre-defined order for this variable)
May be `None` if not required.

`tolindex` Integer array holding the index of the tolerance set for each SLP variable (a zero value means the default tolerances are used)
May be `None` if not required.

`initvalue` Double array holding the initial value for each SLP variable (use the `VarType` bit map to indicate if a value is being provided)
May be `None` if not required.

`stepbound` Double array holding the initial step bound size for each SLP variable (a zero value means that no initial step bound size has been specified). If a value of `xpress.infinity` is used for a value in `StepBound`, the delta will never have step bounds applied, and will almost always be regarded as converged.
May be `None` if not required.

Example

The following example loads two SLP variables into the problem. They correspond to columns 23 and 25 of the underlying LP problem. Column 25 has an initial value of 1.42; column 23 has no specific initial value

```
colindex  = [23,25]
vartype   = [0,4]
initvalue = [0,1.42]

p.loadvars(colindex, vartype, None, None, None, initvalue, None)
```

`InitValue` is not set for the first variable, because it is not used (`VarType` = 0). Bit 1 of `VarType` is set for the second variable to indicate that the initial value has been set. The arrays for determining rows, sequence numbers, tolerance sets and step bounds are not used at all, and so have been passed to the function as `None`.

Further information

The `loadvars` functions load items into the SLP problem. Any existing items of the same type are deleted first. The corresponding `addvars` functions add or replace items leaving other items of the same type unchanged.

Related topics

`problem.addvars`, `problem.chgvar`, `problem.delvars`, `problem.getvar`

problem.lpoptimize

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.lpOptimize` *instead.* Begins a search for the optimal continuous (LP) solution. The direction of optimization is given by `OBJSENSE`. The status of the problem when the function completes can be checked using `LPSTATUS`. Any MIP entities in the problem will be ignored.

Synopsis

```
problem.lpoptimize(flags)
```

Argument

`flags` (optional) Flags to pass to `lpoptimize`. The default is `""` or `None`, in which case the algorithm used is determined by the `DEFAULTALG` control. If the argument includes:

- `b` the model will be solved using the Newton barrier method;
- `p` the model will be solved using the primal simplex algorithm;
- `d` the model will be solved using the dual simplex algorithm;
- `n` (lower case `N`), the network part of the model will be identified and solved using the network simplex algorithm;

Further information

1. The algorithm used to optimize is determined by the `DEFAULTALG` control if no flags are provided. By default, the dual simplex is used for linear problems and the barrier is used for non-linear problems.
2. The `d` and `p` flags can be used with the `n` flag to complete the solution of the model with either the dual or primal algorithms once the network algorithm has solved the network part of the model.
3. The `b` flag cannot be used with the `n` flag.

Related topics

`problem.mipOptimize`, `problem.lpOptimize`, Chapter 4 of the Xpress Optimizer reference manual.

problem.mipoptimize

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.mipOptimize` *instead.* Begins a tree search for the optimal MIP solution. The direction of optimization is given by `OBJSENSE`. The status of the problem when the function completes can be checked using `MIPSTATUS`.

Synopsis

```
problem.mipoptimize(flags)
```

Argument

`flags` (optional) Flags to pass to `problem.mipOptimize`, which specifies how to solve the initial continuous problem where the MIP entities are relaxed. If the argument includes:

- `b` the initial continuous relaxation will be solved using the Newton barrier method;
- `p` the initial continuous relaxation will be solved using the primal simplex algorithm;
- `d` the initial continuous relaxation will be solved using the dual simplex algorithm;
- `n` the network part of the initial continuous relaxation will be identified and solved using the network simplex algorithm;
- `l` (deprecated) stop after having solved the initial continous relaxation. This flag is deprecated, use the `MIPSTOPSTAGE` control instead.

Further information

1. If the `l` flag is used, the Optimizer will stop immediately after solving the initial continuous relaxation. The status of the continuous solve can be checked with `LPSTATUS` and standard LP results are available, such as the objective value (`LPOBJVAL`) and solution (use `problem.getLpSol`), depending on `LPSTATUS`.
2. It is possible for the Optimizer to find integer solutions before solving the initial continuous relaxation, either through heuristics or by having the user load an initial integer solution. This can potentially result in the tree search finishing before solving the continuous relaxation to optimality.
3. If the function returns without having completed the search for an optimal solution, the search can be resumed from where it stopped by calling `problem.mipOptimize` again.
4. The algorithm used to reoptimize the continuous relaxations during the tree search is given by `DEFAULTALG`. The default is to use the dual simplex algorithm.

Related topics

`problem.mipOptimize`.

problem.msaddcustompreset

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.msAddCustomPreset` *instead.*
A combined version of `problem.msAddJob` and `problem.msAddPreset`. The preset described is loaded, topped up with the specific settings supplied

Synopsis

```
problem.msaddcustompreset(description, preset, count, ivcols, ivvalues, control,
       job_object)
```

Arguments

`description`	Text description of the job. Used for messaging, may be `None` if not required.
`preset`	Which preset to load.
`count`	Maximum number of jobs to be added to the multistart pool.
`ivcols`	Indices of the variables for which to set an initial value. May be `None` if nIVs is zero.
`ivvalues`	Initial values for the variables for which to set an initial value. May be `None` if nIVs is zero.
`control`	Python dictionary with control strings as keys and numbers as values. Note that only numerical controls are allowed.
`job_object`	Job-specific user context object to be passed to the multistart callbacks.

Further information

This function allows for repeatedly calling the same multistart preset (e.g. initial values) using different basic controls.

Related topics

`problem.msAddPreset`, `problem.msAddJob`, `problem.msClear`, `problem.msAddCustomPreset`

problem.msaddjob

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.msAddJob` *instead.*
Adds a multistart job to the multistart pool

Synopsis

```
problem.msaddjob(description, ivcols, ivvalues, control, job_object)
```

Arguments

`description`	Text description of the job. Used for messaging, may be `None` if not required.
`ivcols`	Indices of the variables for which to set an initial value. May be `None` if nIVs is zero.
`ivvalues`	Initial values for the variables for which to set an initial value. May be `None` if nIVs is zero.
`control`	Python dictionary with control strings as keys and numbers as values. Note that only numerical controls are allowed.
`job_object`	Job-specific user context object to be passed to the multistart callbacks.

Further information

Adds a mutistart job, applying the specified initial point and option combinations on top of the base problem, i.e., the options and initial values specified to the function is applied on top of the existing settigns.

This function allows for loading empty template jobs, that can then be identified using the pJobObject variable.

Related topics

`problem.msAddPreset`, `problem.msAddCustomPreset`, `problem.msClear`, `problem.msAddJob`

problem.msaddpreset

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.msAddPreset` *instead.*
Loads a preset of jobs into the multistart job pool.

Synopsis

```
problem.msaddpreset(description, preset, maxjobs, data)
```

Arguments

`description`	Text description of the preset. Used for messaging, may be `None` if not required.
`preset`	Which preset to load.
`maxjobs`	Maximum number of jobs to be added to the multistart pool.
`data`	Job-specific user context object to be passed to the multistart callbacks.

Further information

The following presets are defined:

`msset_initialvalues`: generate `count` number of random base points.

`msset_solvers`: load all solvers.

`msset_slp_basic`: load the most typical SLP tuning settings. A maximum of `count` jobs are loaded.

`msset_slp_extended`: load a comprehensive set of SLP tuning settings. A maximum of `count` jobs are loaded.

`msset_knitro_basic`: load the most typical Knitro tuning settings. A maximum of `count` jobs are loaded.

`msset_knitro_extended`: load a comprehensive set of Knitro tuning settings. A maximum of `count` jobs are loaded.

`msset_initialfiltered`: generate `count` number of random base points, filtered by a merit function centred on initial feasibility.

See `xslp_MSMAXBOUNDRANGE` for controlling the range in which initial values are generated.

Related topics

`problem.msAddJob`, `problem.msAddCustomPreset`, `problem.msClear`, `problem.msAddPreset`

problem.msclear

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.msClear` *instead.*
Removes all scheduled jobs from the multistart job pool

Synopsis

```
problem.msclear()
```

Related topics

`problem.msAddJob`, `problem.msAddPreset`, `problem.msAddCustomPreset`, `problem.msClear`

problem.name

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.getProbName` *instead.*
Returns the name of the problem as a Python string.

Synopsis

```
brian = problem.name()
```

Related topics

`problem.setProbName`, `problem.getProbName`.

problem.nlpchgformula

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.nlpChgFormula` *instead.*
Add or replace a single matrix formula using a parsed or unparsed formula

Synopsis

```
problem.nlpchgformula(row, parsed, type, value)
```

Arguments

`row`	The row (i.e., `xpress.constraint` object, index, or name) for the formula.
`parsed`	Boolean indicating whether the token arrays are formatted as internal unparsed (`parsed=False`) or internal parsed reverse Polish (`parsed=True`).
`type`	Array of token types providing the formula for each coefficient.
`value`	Array of values corresponding to the types in `Type`.

Example

Assuming that the columns of the matrix are named `Col1`, `Col2`, etc, the following example puts the formula `sin(Col1)` into the coefficient in row 1.

```
iCol1 = problem.getIndex(Col1)
    type =  [xp.constants.TOK_IFUN, xp.constants.TOK_COL, xp.constants.TOK_RB, xp.constants.TO
    value = [xp.constants.IFUN_SIN, iCol1,      0,          0]
    problem.nlpchgformula(1, False, type, value)
```

The formula is written in unparsed form (`parsed=False`) and so it is provided as tokens in the same order as they would appear if the formula were written in character form.

Further information

If the row already has a nonlinear expression in it, it will be changed into the new formula. If it does not exist, it will be added to the problem.

Related topics

`problem.nlpGetFormulaStr`, `problem.nlpChgFormulaStr`, `problem.nlpGetFormula`, `problem.nlpChgFormula`.

problem.nlpchgformulastr

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.nlpChgFormulaStr` *instead.* Add or replace a single matrix formula using a character string for the formula.

Synopsis

```
problem.nlpchgformulastr(row, formula)
```

Arguments

`row`	The row (i.e., `xpress.constraint` object, index, or name) for the formula.
`formula`	Character string holding the formula with the tokens separated by spaces.

Example

Assuming that the columns of the matrix are named `Col1`, `Col2`, etc, the following example puts the formula `sin(Col1)` into row 1.

```
problem.nlpchgformulastr(prob, 1, "sin ( Col1 )");
```

Note that all the tokens in the formula (including mathematical operators and separators) are separated by one or more spaces.

Further information

If the coefficient already exists as a constant or formula, it will be changed into the new coefficient. If it does not exist, it will be added to the problem.

This function can only be used if all the operands in the formula can be correctly identified as constants, existing columns, character variables or functions. Therefore, if a formula refers to a new column, that new item must be added to the Xpress NonLinear problem first.

Related topics

`problem.nlpGetFormulaStr`, `problem.nlpChgFormula`, `problem.nlpGetFormula`, `problem.nlpChgFormulaStr`.

problem.nlpgetformula

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.nlpGetFormula` instead.
Retrieve a single matrix formula split into tokens

Synopsis

```
ntypes, type, value = problem.nlpgetformula(row, parsed, type, value)
```

Arguments

`row`	The row (i.e., `xpress.constraint` object, index, or name) for the formula.
`parsed`	Whether the formula of the row is to be returned in internal unparsed format (`parsed=False`) or parsed (reverse Polish) format (`parsed=True`).
`type`	(optional) A list to populate with the token types for the formula.
`value`	(optional) A list to populate with the values corresponding to `type`.

Return value

`ntypes`	The number of tokens available.
`type`	The token types for the formula.
`value`	The values corresponding to `type`.

Example

The following example retrieves the formula tokens for row 0 in parsed format:

```
_, type, value = p.nlpgetformula(0, True)
```

Further information

If you do not need the value of `type` or `value`, you can pass `None` for these arguments: no arrays will be allocated and `None` will be returned in their place.

Related topics

`problem.nlpGetFormulaStr`, `problem.nlpChgFormulaStr`, `problem.nlpChgFormula`, `problem.nlpGetFormula`.

problem.slpgetcoefformula

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpGetCoefFormula` instead.
Retrieve a single matrix formula split into tokens

Synopsis

```
ntypes, factor, type, value = problem.slpgetcoefformula(row, col, parsed, type,
       value)
```

Arguments

`row`	The row (i.e., `xpress.constraint` object, index, or name) for the formula.
`col`	The column (i.e., `xpress.variable` object, index, or name) for the formula.
`parsed`	Whether the formula is to be returned in internal unparsed format (`parsed=False`) or parsed (reverse Polish) format (`parsed=True`).
`type`	(optional) A list to populate with the token types for the formula.
`value`	(optional) A list to populate with the values corresponding to `type`.

Return value

`factor`	The value of the constant factor multiplying the formula in the coefficient.
`ntypes`	The number of tokens available.
`type`	The token types for the formula.
`value`	The values corresponding to `type`.

Example

The following example displays the formula for the coefficient of column 3 in row 2 in unparsed form:

```
_, _, type, value = p.slpgetcoefformula(2, 3, False)
```

Further information

If you do not need the value of `type` or `value`, you can pass `None` for these arguments: no arrays will be allocated and `None` will be returned in their place.

Related topics

`problem.slpChgCoef`, `problem.nlpGetFormulaStr`, `problem.nlpChgFormulaStr`, `problem.nlpChgFormula`, `problem.nlpGetFormula`, `problem.slpGetCoefFormula`.

problem.nlpgetformulastr

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.nlpGetFormulaStr` *instead.*
Retrieve a single matrix formula in a character string

Synopsis

```
formula = problem.nlpgetformulastr(row)
```

Argument

`row` The row (i.e., `xpress.constraint` object, index, or name) for the formula.

Example

The following example prints the formula for row 0:

```
print(p.nlpgetformulastr(0))
```

problem.nlpoptimize

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.nlpOptimize` *instead.*
Solves an SLP problem

Synopsis

```
problem.nlpoptimize(flags)
```

Argument

`flags` Flags affecting the solve. See the SLP reference manual for their meaning

problem.nlpsetinitval

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.nlpSetInitVal` *instead.*
Set the initial value of a nonlinear variable

Synopsis

```
problem.nlpsetinitval(colind, initial)
```

Arguments

`colind`	Array containing the columns (i.e., `xpress.var` objects, indices, or names) for which to set the initial value
`initial`	Array containing the initial values to set

Example

The following example sets an initial value for the variable `x`:

```
problem.nlpsetinitval([x], [9])
```

problem.objsa

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.objSA` *instead.*
Returns upper and lower sensitivity ranges for specified objective function coefficients. If the objective coefficients are varied within these ranges the current basis remains optimal and the reduced costs remain valid.

Synopsis

```
problem.objsa(colind, lower, upper)
```

Arguments

`colind`	Array containing the columns (i.e., `xpress.var` objects, indices, or names) whose objective function coefficients sensitivity ranges are required.
`lower`	Array of the same size as `mindex` where the objective function lower range values are to be returned.
`upper`	Array of the same size as `mindex` where the objective function upper range values are to be returned.

Example

Here we obtain the objective function ranges for the three columns: `2`, `6` and `8`:

```
l = []
u = []
p.objsa([2,8,6], l, u)
```

After which `l` and `u` contain:

```
l = [5, 3.8, 5.7]
u = [7, 5.2, 1e+20]
```

Meaning that the current basis remains optimal when $5.0 \leqslant C_2 \leqslant 7.0$, $3.8 \leqslant C_8 \leqslant 5.2$ and $5.7 \leqslant C_6$, C_i being the objective coefficient of column `i`.

Further information

`objsa` can only be called when an optimal solution to the current LP has been found. It cannot be used when the problem is MIP presolved.

Related topics

`problem.bndSA`, `problem.rhsSA`, `problem.objSA`.

problem.postsolve

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.postSolve` *instead.*
Postsolve the current problem when it is in a presolved state.

Synopsis

```
problem.postsolve()
```

Further information

A problem is left in a presolved state whenever a LP or MIP optimization does not complete. In these cases `postsolve` can be called to get the problem back into its original state.

Related topics

`problem.lpOptimize`, `problem.mipOptimize`, `problem.postSolve`.

problem.presolve

Purpose

This subroutine is deprecated and will be removed in a future release.
Perform a nonlinear presolve on the problem

Synopsis

```
problem.presolve()
```

Example

The following example reads a problem from file, sets the presolve control, presolves the problem and then maximizes it.

```
p.readprob("Matrix", "")
p.controls.xslp_presolve = 1
p.presolve()
p.optimize("")
```

Further information

If bit 1 of `xslp_presolve` is not set, no nonlinear presolve will be performed. Otherwise, the presolve will be performed in accordance with the bit settings. `problem.presolve` is called automatically by `problem.slpConstruct`, so there is no need to call it explicitly unless there is a requirement to interrupt the process between presolve and optimization. `problem.presolve` must be called before `problem.slpConstruct` or any of the SLP optimization procedures..

Related topics

`xslp_presolve`

problem.presolverow

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.presolveRow` *instead.* Presolves a row formulated in terms of the original variables such that it can be added to a presolved problem. Returns a tuple of two elements containing, respectively, the presolved right-hand side and the status of the presolved row:

- -3: Failed to presolve the row due to presolve dual reductions;
- -2: Failed to presolve the row due to presolve duplicate column reductions;
- -1: Failed to presolve the row due to an error. Check the Optimizer error code for the cause;
- 0: The row was successfully presolved;
- 1: The row was presolved, but may be relaxed.

Synopsis

```
drhsp, status = problem.presolverow(rowtype, origcolind, origrowcoef, origrhs,
       maxcoefs, colind, rowcoef)
```

Arguments

`rowtype`	The type of the row: `L` indicates a $\leqslant$ row; `G` indicates a $\geqslant$ row.
`origcolind`	Array containing the columns (i.e., `xpress.var` objects, indices, or names) of the row to presolve.
`origrowcoef`	Array containing the nonzero coefficients of the row to presolve.
`origrhs`	The right-hand side constant of the row to presolve.
`maxcoefs`	Maximum number of elements to return in the `colind` and `rowcoef` arrays.
`colind`	Array which will be filled with the columns of the presolved row.
`rowcoef`	Array which will be filled with the coefficients of the presolved row.

Example

Adding the row $2x_1 + x_2 \leqslant 1$ to our presolved problem can be done as follows:

```
presind = []
prescoe = []
prhs, status = p.presolverow('L', [1,2], [2,1], 1.0,
   p.attributes.cols, presind, prescoe)
```

Further information

There are certain presolve operations that can prevent a row from being presolved exactly. If the row contains a coefficient for a column that was eliminated due to duplicate column reductions or singleton column reductions, the row might have to be relaxed to remain valid for the presolved problem. The relaxation will be done automatically by the `problem.presolveRow` function, but a return status of `+1` will be returned. If it is not possible to relax the row, a status of `−2` will be returned instead. Likewise, it is possible that certain dual reductions prevents the row from being presolved. In such a case a status of `−3` will be returned instead.

If `problem.presolveRow` is used for presolving e.g. branching bounds or constraints, then dual reductions and duplicate column reductions should be disabled, by clearing the corresponding bits of `PRESOLVEOPS`. By clearing these bits, the default value for `PRESOLVEOPS` changes to 471.

If the user knows in advance which columns will have nonzero coefficients in rows that will be presolved, it is possible to protect these individual columns through the `problem.loadSecureVecs` function. This way the Optimizer is left free to apply all possible reductions to the remaining columns.

Related topics

`problem.addCuts`, `problem.loadSecureVecs`, `problem.storeCuts`, `problem.presolveRow`.

problem.printmemory

Purpose

This subroutine is deprecated and will be removed in a future release.
Print the dimensions and memory allocations for a problem

Synopsis

```
problem.printmemory()
```

Example

The following example loads a problem from file and then prints the dimensions of the arrays.

```
p.readprob("Matrix1", "")
p.printmemory()
```

The output is similar to the following:

```
Arrays
and dimensions: Array Item Used Max Allocated Memory Size Items Items
Memory Control MemList 28 103 129 4K String 1 8779 13107 13K
xslp_MEM_STRING Xv 16 2 1000 16K xslp_MEM_XV Xvitem 48 11 1000 47K
xslp_MEM_XVITEM ....
```

Further information

`printmemory` lists the current sizes and amounts used of the variable arrays in the current problem. For each array, the size of each item, the number used and the number allocated are shown, together with the size of memory allocated and, where appropriate, the name of the memory control variable to set the array size. Loading and execution of some problems can be speeded up by setting the memory controls immediately after the problem is created. If an array has to be moved to re-allocate it with a larger size, there may be insufficient memory to hold both the old and new versions; pre-setting the memory controls reduces the number of such re-allocations which take place and may allow larger problems to be solved.

problem.printevalinfo

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.nlpPrintEvalInfo` *instead.*
Print a summary of any evaluation errors that may have occurred during solving a problem

Synopsis

```
problem.printevalinfo()
```

Related topics

`problem.setcbcoefevalerror`, `problem.nlpPrintEvalInfo`

problem.read

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.readProb` *instead.*
Read an optimization problem into a Python problem object created prior to the call. All formats allowed by the Xpress Optimizer C API are allowed.

Synopsis

```
problem.read(filename, flags, unlinked=False)
```

Arguments

`filename` A string of up to 200 characters with the name of the file to be read.

`flags` (optional) Flags to pass to `read`:

- `l` only the `.lp` version of the file is searched.
- `z` read the input file in compressed `.gz` format.

`unlinked` (optional) If `True`, unlinked variables and constraints will be created (deprecated).

Example

Read problem `problem1.lp` and output an optimal solution:

```
p.read("problem1", "l")
p.optimize()
print("solution of problem1.lp:", p.getSolution())
```

Related topics

`problem.writeProb`.

problem.readbasis

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.readBasis` *instead.*
Instructs the Optimizer to read in a previously saved basis from a file.

Synopsis

```
problem.readbasis(filename, flags)
```

Arguments

`filename` A string of up to 200 characters containing the file name from which the basis is to be read. If omitted, the default *problem_name* is used with a `.bss` extension.

`flags` (optional) Flags to pass to `readbasis`:

- `i` output the internal presolved basis.
- `t` input a compact advanced form of the basis;

Example

If an advanced basis is available for the current problem the Optimizer input might be:

```
p.read("filename", "")
p.readbasis("", "")
p.mipoptimize("")
```

This reads in a matrix file, inputs an advanced starting basis and maximizes the MIP.

Further information

1. The only check done when reading compact basis is that the number of rows and columns in the basis agrees with the current number of rows and columns.
2. `readbasis` will read the basis for the original problem even if the problem has been presolved. The Optimizer will read the basis, checking that it is valid, and will display error messages if it detects inconsistencies.

Related topics

`problem.loadBasis`, `problem.writeBasis`, `problem.readBasis`.

problem.readbinsol

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.readBinSol` *instead.*
Reads a solution from a binary solution file.

Synopsis

```
problem.readbinsol(filename, flags)
```

Arguments

`filename` A string of up to 200 characters containing the file name from which the solution is to be read. If omitted, the default *problem_name* is used with a `.sol` extension.

`flags` (optional) Flags to pass to `readbinsol`:

`m` load the solution as a solution for the MIP.

Example

A previously saved solution can be loaded into memory and a print file created from it with the following function calls:

```
p.read("myprob", "")
p.readbinsol("", "")
p.writeprtsol("", "")
```

Related topics

`problem.getSolution`, `problem.writeBinSol`, `problem.writeSol`, `problem.writePrtSol`, `problem.readBinSol`.

problem.readdirs

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.readDirs` *instead.*
Reads a directives file to help direct the tree search.

Synopsis

```
problem.readdirs(filename)
```

Argument

`filename`	A string of up to 200 characters containing the file name from which the directives are to be read. If omitted (or `None`), the default *problem_name* is used with a `.dir` extension.

Example

The following example reads in directives from the file `dirfile.dir` for use with the problem, `prob2`:

```
p.read("prob2","")
p.readdirs("dirfile")
p.mipoptimize("")
```

Further information

1. Directives cannot be read in after a model has been presolved, so unless presolve has been disabled by setting `PRESOLVE` to `0`, this function must be called before `problem.mipOptimize`.
2. Directives can be given relating to priorities, forced branching directions, pseudo costs and model cuts. There is a priority value associated with each MIP entity. The *lower* the number, the *more* likely the entity is to be selected for branching; the *higher*, the *less* likely. By default, all MIP entities have a priority value of `500` which can be altered with a priority entry in the directives file. In general, it is advantageous for the entity's priority to reflect its relative importance in the model. Priority entries with values in excess of `1000` are illegal and are ignored. A full description of the directives file format may be found in the Xpress Optimizer reference manual.
3. By default, `problem.mipOptimize` will explore the branch expected to yield the best integer solution from each node, irrespective of whether this forces the MIP entity up or down. This can be overridden with an `UP` or `DN` entry in the directives file, which forces `mipoptimize` to branch up first or down first on the specified entity.
4. Pseudo-costs are estimates of the unit cost of forcing an entity up or down. By default `mipoptimize` uses dual information to calculate estimates of the unit up and down costs and these are added to the default pseudo costs which are set to the `PSEUDOCOST` control. The default pseudo costs can be overridden by a `PU` or `PD` entry in the directives file.
5. If model cuts are used, then the specified constraints are removed from the problem and added to the Optimizer cut pool, and only put back in the problem when they are violated by an LP solution at one of the nodes in the tree search.
6. If creating a directives file by hand, wild cards can be used to specify several vectors at once, for example `PR x1* 2` will give all MIP entities whose names start with `x1` a priority of `2`.

Related topics

`problem.loadDirs`, `problem.readDirs`.

problem.readslxsol

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.readSlxSol` *instead.*
Reads an ASCII solution file (`.slx`) created by the `problem.writeSlxSol` function.

Synopsis

```
problem.readslxsol(filename, flags)
```

Arguments

`filename` A string of up to 200 characters containing the file name to which the solution is to be read. If omitted, the default *problem_name* is used with a `.slx` extension.

`flags` (optional) Flags to pass to `writeslxsol`:

- `l` read the solution as an LP solution in case of a MIP problem;
- `m` read the solution as a solution for the MIP problem;
- `a` reads multiple MIP solutions from the `.slx` file and adds them to the MIP problem.

Example

```
p.readslxsol("lpsolution", "")
```

This loads the solution to the MIP problem if the problem contains MIP entities, or otherwise loads it as an LP (barrier in case of quadratic problems) solution into the problem.

Further information

1. When `readslxsol` is called before a MIP solve, the loaded solutions will not be checked before calling `problem.mipOptimize`. By default, only the last MIP solution read from the `.slx` file will be stored. Use the `a` flag to store all MIP solutions read from the file.
2. When using the `a` flag, read solutions will be queued similarly to the user of the `problem.addMipSol` function. Each name string given by the `NAME` field in the `.slx` file will be associated with the corresponding solution. Any registered `usersolnotify` callback will be fired when the solution has been checked, and will include the read name string as one of its arguments.
3. Refer to the Appendix of the Xpress Optimizer reference manual on Log and File Formats for a description of the ASCII Solution (.slx) file format.

Related topics

`problem.readBinSol`, `problem.writeSlxSol`, `problem.writeBinSol`,`problem.addMipSol`, `problem.addUserSolNotifyCallback`, `problem.readSlxSol`.

problem.reinitialize

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpReInitialize` *instead.*
Reset the SLP problem to match a just augmented system

Synopsis

```
problem.reinitialize()
```

Further information

Can be used to rerun the SLP optimization process with updated parameters, penalties or initial values, but unchanged augmentation.

Related topics

`problem.slpUnConstruct`, `problem.nlpCurrentIV`, `problem.slpReInitialize`.

problem.removecbbariteration

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removeBarIterationCallback` *instead.*
Removes a barrier iteration callback function previously added by `addcbbariteration`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbbariteration(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all bariteration callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all barrier iteration callbacks with the function `callback` will be removed.

Related topics

`problem.addBarIterationCallback`, `problem.removeBarIterationCallback`.

problem.removecbbarlog

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removeBarlogCallback` *instead.*
Removes a newton barrier log callback function previously added by `addcbbarlog`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbbarlog(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all barrier log callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all barrier log callbacks with the function `callback` will be removed.

Related topics

`problem.addBarlogCallback`, `problem.removeBarlogCallback`.

problem.removecbchecktime

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removeCheckTimeCallback` *instead.*
Removes a callback function previously added by `problem.addCheckTimeCallback`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbchecktime(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None`, then all checktime callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all checktime callbacks with the function pointer `callback` will be removed.

Related topics

`problem.addCheckTimeCallback`, `problem.removeCheckTimeCallback`

problem.removecbchgbranchobject

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removeChangeBranchObjectCallback` *instead.*
Removes a callback function previously added by `addcbchgbranchobject`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbchgbranchobject(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all branch object callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If None, the object value will not be checked and all branch object callbacks with the function `callback` will be removed.

Related topics

`problem.addChangeBranchObjectCallback`, `problem.removeChangeBranchObjectCallback`.

problem.removecbcutlog

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removeCutlogCallback` *instead.*

Removes a cut log callback function previously added by `addcbcutlog`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbcutlog(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all cut log callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all cut log callbacks with the function `callback` will be removed.

Related topics

`problem.addCutlogCallback`, `problem.removeCutlogCallback`.

problem.removecbcutround

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removeCutRoundCallback` *instead.*
Removes a cut round callback function previously added by `addcbcutround`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbcutround(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all cut log callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all cut log callbacks with the function `callback` will be removed.

Related topics

`problem.addCutRoundCallback`, `problem.removeCutRoundCallback`.

problem.removecbdestroymt

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removeMipThreadDestroyCallback` *instead.*
Removes a child thread destruction callback function previously added by `addcbdestroymt`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbdestroymt(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all thread destruction callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all thread destruction callbacks with the function `callback` will be removed.

Related topics

`problem.addMipThreadDestroyCallback`, `problem.removeMipThreadDestroyCallback`.

problem.removecbgapnotify

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removeGapNotifyCallback` instead.
Removes a callback function previously added by `problem.addGapNotifyCallback`. The specified callback function will no longer be removed after it has been returned.

Synopsis

```
problem.removecbgapnotify(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all `gapnotify` callback functions added with the given user-defined value will be removed.
`data`	The user-defined object that the callback was added with. If `None` then the object will not be checked and all the `gapnotify` callbacks with the function `callback` will be removed.

Related topics

`problem.addGapNotifyCallback`, `problem.removeGapNotifyCallback`.

problem.removecbmiplog

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removeMiplogCallback` instead.
Removes a MIP log callback function previously added by `addcbmiplog`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbmiplog(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all MIP log callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all MIP log callbacks with the function `callback` will be removed.

Example

The following code sets and removes a callback function:

```
prob.controls.miplog = 3
prob.addcbmiplog(mipLog, None, 0)
prob.mipoptimize("")
prob.removecbmiplog(mipLog, None)
```

Related topics

`problem.addMiplogCallback`, `problem.removeMiplogCallback`.

problem.removecbinfnode

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removeInfnodeCallback` *instead.*

Removes a user infeasible node callback function previously added by `addcbinfnode`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbinfnode(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all user infeasible node callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all user infeasible node callbacks with the function `callback` will be removed.

Related topics

`problem.addInfnodeCallback`, `problem.removeInfnodeCallback`.

problem.removecbintsol

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removeIntsolCallback` *instead.*

Removes an integer solution callback function previously added by `addcbintsol`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbintsol(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all integer solution callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all integer solution callbacks with the function `callback` will be removed.

Related topics

`problem.addIntsolCallback`, `problem.removeIntsolCallback`.

problem.removecblplog

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removeLplogCallback` instead.
Removes a simplex log callback function previously added by `addcblplog`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecblplog(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all lplog callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all lplog callbacks with the function `callback` will be removed.

Example

The following code sets and removes a callback function:

```
prob.controls.lplog = 10
prob.addcblplog(lpLog, None, 0)
prob.readprob("problem", "")
prob.lpoptimize("")
prob.removecblplog(lpLog, None)
```

Related topics

`problem.addLplogCallback`, `problem.removeLplogCallback`.

problem.removecbmessage

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removeMessageCallback` *instead.*

Removes a message callback function previously added by `addcbmessage`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbmessage(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all message callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all message callbacks with the function `callback` will be removed.

Further information

By default, Optimizer messages are also printed to `stdout`. This behaviour is unaffected by user message callbacks. It can be disabled by calling `problem.setOutputEnabled` or `xpress.setOutputEnabled` with argument `False`.

Related topics

`problem.addMessageCallback`, `problem.setOutputEnabled`, `xpress.setOutputEnabled`, `problem.removeMessageCallback`.

problem.removecbmipthread

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removeMipThreadCallback` *instead.*
Removes a callback function previously added by `addcbmipthread`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbmipthread(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all variable branching callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all variable branching callbacks with the function `callback` will be removed.

Related topics

`problem.addMipThreadCallback`, `problem.removeMipThreadCallback`.

problem.removecbnewnode

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removeNewnodeCallback` *instead.*

Removes a new-node callback function previously added by `addcbnewnode`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbnewnode(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all separation callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all separation callbacks with the function `callback` will be removed.

Related topics

`problem.addNewnodeCallback`, `problem.removeNewnodeCallback`.

problem.removecbnodecutoff

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removeNodecutoffCallback` *instead.*
Removes a node-cutoff callback function previously added by `addcbnodecutoff`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbnodecutoff(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all node-cutoff callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all node-cutoff callbacks with the function `callback` will be removed.

Related topics

`problem.addNodecutoffCallback`, `problem.removeNodecutoffCallback`.

problem.removecbnodelpsolved

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removeNodeLPSolvedCallback` *instead.*
Removes a node lp solved callback function previously added by `addcbnodelpsolved`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbnodelpsolved(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all lp solved callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all lp solved callbacks with the function `callback` will be removed.

Related topics

`problem.addNodeLPSolvedCallback`, `problem.removeNodeLPSolvedCallback`.

problem.removecboptnode

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removeOptnodeCallback` *instead.*
Removes a node-optimal callback function previously added by `addcboptnode`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecboptnode(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all node-optimal callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all node-optimal callbacks with the function `callback` will be removed.

Related topics

`problem.addOptnodeCallback`, `problem.removeOptnodeCallback`.

problem.removecbpreintsol

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removePreIntsolCallback` *instead.*
Removes a pre-integer solution callback function previously added by `addcbpreintsol`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbpreintsol(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all user infeasible node callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all user infeasible node callbacks with the function `callback` will be removed.

Related topics

`problem.addPreIntsolCallback`, `problem.removePreIntsolCallback`.

problem.removecbprenode

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removePrenodeCallback` *instead.*

Removes a preprocess node callback function previously added by `addcbprenode`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbprenode(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all preprocess node callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all preprocess node callbacks with the function `callback` will be removed.

Related topics

`problem.addPrenodeCallback`, `problem.removePrenodeCallback`.

problem.removecbusersolnotify

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removeUserSolNotifyCallback` *instead.*
Removes a user solution notification callback previously added by `problem.addUserSolNotifyCallback`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbusersolnotify(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all user solution notification callback functions added with the given user defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all integer solution callbacks with the function `callback` will be removed.

Related topics

`problem.addUserSolNotifyCallback`, `problem.removeUserSolNotifyCallback`.

problem.removecbbeforeobjective

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removeBeforeObjectiveCallback` *instead.*
Removes a user before objective callback function previously added by `addcbbeforeobjective`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbbeforeobjective(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all user before objective callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all user before objective callbacks with the function `callback` will be removed.

Related topics

`problem.addBeforeObjectiveCallback`, `problem.removeBeforeObjectiveCallback`.

problem.removecbafterobjective

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removeAfterObjectiveCallback` *instead.*
Removes a user after objective callback function previously added by `addcbafterobjective`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbafterobjective(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all user after objective callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all user after objective callbacks with the function `callback` will be removed.

Related topics

`problem.addAfterObjectiveCallback`, `problem.removeAfterObjectiveCallback`.

problem.removecbpresolve

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.removePresolveCallback` *instead.*

Removes a presolve callback function previously added by `addcbpresolve`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbpresolve(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all presolve callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all presolve callbacks with the function `callback` will be removed.

Related topics

`problem.addPresolveCallback`, `problem.removePresolveCallback`.

problem.repairinfeas

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.repairInfeas` *instead.*
Provides a simplified interface for `problem.repairWeightedInfeas`. The returned value is as follows:

- 0: relaxed optimum found;
- 1: relaxed problem is infeasible;
- 2: relaxed problem is unbounded;
- 3: solution of the relaxed problem regarding the original objective is nonoptimal;
- 4: error (when return code is nonzero);
- 5: numerical instability;
- 6: analysis of an infeasible relaxation was performed, but the relaxation is feasible.

Synopsis

```
status_code = problem.repairinfeas(penalty, phase2, flags, lepref, gepref, lbpref,
       ubpref, delta)
```

Arguments

`penalty` The type of penalties created from the preferences:
- `c` each penalty is the reciprocal of the preference (default);
- `s` the penalties are placed in the scaled problem.

`phase2` Controls the second phase of optimization:
- `o` use the objective sense of the original problem (default);
- `x` maximize the relaxed problem using the original objective;
- `f` skip optimization regarding the original objective;
- `n` minimize the relaxed problem using the original objective;
- `i` if the relaxation is infeasible, generate an irreducible infeasible subset for the analys of the problem;
- `a` if the relaxation is infeasible, generate all irreducible infeasible subsets for the analys of the problem.

`flags` Specifies if the tree search should be done:
- `g` do the tree search (default);
- `l` solve as a linear model ignoring the discreteness of variables.

`lepref` Preference for relaxing the less or equal side of row.

`gepref` Preference for relaxing the greater or equal side of a row.

`lbpref` Preferences for relaxing lower bounds.

`ubpref` Preferences for relaxing upper bounds.

`delta` The relaxation multiplier in the second phase -1. A positive value means a relative relaxation by multiplying the first phase objective with (`delta-1`), while a negative value means an absolute relaxation, by adding `abs(delta)` to the first phase objective.

Further information

1. A row or bound is relaxed by introducing a new nonnegative variable that will contain the infeasibility of the row or bound. Suppose for example that row $a^T x = b$ is relaxed from below. Then a new variable (infeasibility breaker) s>=0 is added to the row, which becomes $a^T x + s = b$. Observe that $a^T x$ may now take smaller values than b. To minimize such violations, the weighted sum of these new variables is minimized.
2. A preference of 0 results in the row or bound not being relaxed.
3. A negative preference indicates that a quadratic penalty cost should be applied. This can specified on a per constraint side or bound basis.
4. Note that the set of preferences are scaling independent.
5. If a feasible solution is identified for the relaxed problem, with a sum of violations p, then the sum of violations is restricted to be no greater than `(1+delta)`p, and the problem is optimized with respect to the original objective function. A nonzero delta increases the freedom of the original problem.
6. Note that on some problems, slight modifications of delta may affect the value of the original objective drastically.
7. Note that because of their special associated modeling properties, binary and semi-continuous variables are not relaxed.
8. The default algorithm for the first phase is the simplex algorithm, since the primal problem can be efficiently warm started in case of the extended problem. These may be altered by setting the value of control `DEFAULTALG`.
9. If `penalty` is set such that each penalty is the reciprocal of the preference, the following rules are applied while introducing the auxiliary variables:

Preference	Affects	Relaxation	Cost if pref.>0	Cost if pref.<0
`lepref`	`= rows`	$a^T x$ `- aux_var = b`	`1/lepref*aux_var`	`1/lepref*aux_var`^{2}
`lepref`	`<= rows`	$a^T x$ `- aux_var <= b`	`1/lepref*aux_var`	`1/lepref*aux_var`^{2}
`gepref`	`= rows`	$a^T x$ `+ aux_var = b`	`1/gepref*aux_var`	`1/gepref*aux_var`^{2}
`gepref`	`>= rows`	$a^T x$ `+ aux_var >= b`	`1/gepref*aux_var`	`1/gepref*aux_var`^{2}
`ubpref`	`upper bounds`	x_i `- aux_var <= u`	`1/ubpref*aux_var`	`1/ubpref*aux_var`^{2}
`lbpref`	`lower bounds`	x_i `+ aux_var >= l`	`1/lbpref*aux_var`	`1/lbpref*aux_var`^{2}

10. If an irreducible infeasible set (IIS) has been identified, the generated IIS(s) are accesible through the IIS retrieval functions, see `NUMIIS` and `problem.getIISData`.

Related topics

`problem.repairWeightedInfeas`, `problem.repairInfeas`.

problem.repairweightedinfeas

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.repairWeightedInfeas` *instead.*

By relaxing a set of selected constraints and bounds of an infeasible problem, it attempts to identify a 'solution' that violates the selected set of constraints and bounds minimally, while satisfying all other constraints and bounds. Among such solution candidates, it selects one that is optimal regarding to the original objective function. Similar to `repairinfeas`, the returned value is as follows:

- 1: relaxed problem is infeasible;
- 2: relaxed problem is unbounded;
- 3: solution of the relaxed problem regarding the original objective is nonoptimal;
- 4: error (when return code is nonzero);
- 5: numerical instability;
- 6: analysis of an infeasible relaxation was performed, but the relaxation is feasible.

Synopsis

```
status_code = problem.repairweightedinfeas(lepref, gepref, lbpref, ubpref, phase2,
       delta, flags)
```

Arguments

`lepref`	Array of size `ROWS` containing the preferences for relaxing the less or equal side of row.
`gepref`	Array of size `ROWS` containing the preferences for relaxing the greater or equal side of a row.
`lbpref`	Array of size `COLS` containing the preferences for relaxing lower bounds.
`ubpref`	Array of size `COLS` containing preferences for relaxing upper bounds.
`phase2`	Controls the second phase of optimization: `o` use the objective sense of the original problem (default); `x` maximize the relaxed problem using the original objective; `f` skip optimization regarding the original objective; `n` minimize the relaxed problem using the original objective; `i` if the relaxation is infeasible, generate an irreducible infeasible subset for the analys of the problem; `a` if the relaxation is infeasible, generate all irreducible infeasible subsets for the analys of the problem.
`delta`	The relaxation multiplier in the second phase -1.
`flags`	Specifies flags to be passed to the Optimizer.

Further information

1. A row or bound is relaxed by introducing a new nonnegative variable that will contain the infeasibility of the row or bound. Suppose for example that row $a^T x = b$ is relaxed from below. Then a new variable ('infeasibility breaker') s>=0 is added to the row, which becomes $a^T x + s = b$. Observe that $a^T x$ may now take smaller values than b. To minimize such violations, the weighted sum of these new variables is minimized.
2. A preference of 0 results in the row or bound not being relaxed. The higher the preference, the more willing the modeller is to relax a given row or bound.
3. The weight of each infeasibility breaker in the objective minimizing the violations is 1/p, where `p` is the preference associated with the infeasibility breaker. Thus the higher the preference is, the lower a penalty is associated with the infeasibility breaker while minimizing the violations.
4. If a feasible solution is identified for the relaxed problem, with a sum of violations `p`, then the sum of violations is restricted to be no greater than `(1+delta)p`, and the problem is optimized with respect to the original objective function. A nonzero delta increases the freedom of the original problem.
5. Note that on some problems, slight modifications of delta may affect the value of the original objective drastically.
6. Note that because of their special associated modeling properties, binary and semi-continuous variables are not relaxed.
7. If `pflags` is set such that each penalty is the reciprocal of the preference, the following rules are applied while introducing the auxiliary variables:

Pref. array	Affects	Relaxation	Cost if pref.>0	Cost if pref.<0
`lepref`	`= rows`	$a^T x$ `- aux_var = b`	`1/lrp*aux_var`	`1/lrp*aux_var`^{2}
`lepref`	`<= rows`	$a^T x$ `- aux_var <= b`	`1/lrp*aux_var`	`1/lrp*aux_var`^{2}
`gepref`	`= rows`	$a^T x$ `+ aux_var = b`	`1/grp*aux_var`	`1/grp*aux_var`^{2}
`gepref`	`>= rows`	$a^T x$ `+ aux_var >= b`	`1/grp*aux_var`	`1/grp*aux_var`^{2}
`ubpref`	`upper bounds`	x_i `- aux_var <= u`	`1/ubp*aux_var`	`1/ubp*aux_var`^{2}
`lbpref`	`lower bounds`	x_i `+ aux_var >= l`	`1/lbp*aux_var`	`1/lbp*aux_var`^{2}

8. If an irreducible infeasible set (IIS) has been identified, the generated IIS(s) are accesible through the IIS retrieval functions, see `NUMIIS` and `problem.getIISData`.

Related topics

`problem.repairInfeas`, `problem.repairWeightedInfeasBounds`, `problem.repairWeightedInfeas`.

problem.repairweightedinfeasbounds

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.repairWeightedInfeasBounds` *instead.*
An extended version of `problem.repairWeightedInfeas` that allows for bounding the level of relaxation allowed. The returned value is the same as `repairweightedinfeas`.

Synopsis

```
status = problem.repairweightedinfeasbounds(lepref, gepref, lbpref, ubpref, lerelax,
       gerelax, lbrelax, ubrelax, phase2, delta, flags)
```

Arguments

`lepref`	Array of size `ROWS` containing the preferences for relaxing the less or equal side of row.
`gepref`	Array of size `ROWS` containing the preferences for relaxing the greater or equal side of a row.
`lbpref`	Array of size `COLS` containing the preferences for relaxing lower bounds.
`ubpref`	Array of size `COLS` containing preferences for relaxing upper bounds.
`lerelax`	Array of size `ROWS` containing the upper bounds on the amount the less or equal side of a row can be relaxed.
`gerelax`	Array of size `ROWS` containing the upper bounds on the amount the greater or equal side of a row can be relaxed.
`lbrelax`	Array of size `COLS` containing the upper bounds on the amount the lower bounds can be relaxed.
`ubrelax`	Array of size `COLS` containing the upper bounds on the amount the upper bounds can be relaxed.
`phase2`	Controls the second phase of optimization: `o` use the objective sense of the original problem (default); `x` maximize the relaxed problem using the original objective; `f` skip optimization regarding the original objective; `n` minimize the relaxed problem using the original objective; `i` if the relaxation is infeasible, generate an irreducible infeasible subset for the analys of the problem; `a` if the relaxation is infeasible, generate all irreducible infeasible subsets for the analys of the problem.
`delta`	The relaxation multiplier in the second phase -1.
`flags`	Specifies flags to be passed to the Optimizer.

Further information

1. A row or bound is relaxed by introducing a new nonnegative variable that will contain the infeasibility of the row or bound. Suppose for example that row $a^T x = b$ is relaxed from below. Then a new variable ('infeasibility breaker') s>=0 is added to the row, which becomes $a^T x + s = b$. Observe that $a^T x$ may now take smaller values than b. To minimize such violations, the weighted sum of these new variables is minimized.
2. A preference of 0 results in the row or bound not being relaxed. The higher the preference, the more willing the modeller is to relax a given row or bound.
3. A negative preference indicates that a quadratic penalty cost should be applied. This can specified on a per constraint side or bound basis.
4. If a feasible solution is identified for the relaxed problem, with a sum of violations `p`, then the sum of violations is restricted to be no greater than `(1+delta)p`, and the problem is optimized with respect to the original objective function. A nonzero delta increases the freedom of the original problem.
5. Note that on some problems, slight modifications of delta may affect the value of the original objective drastically.
6. Note that because of their special associated modeling properties, binary and semi-continuous variables are not relaxed.
7. Given any row `j` with preferences `lrp=lepref[j]` and `grp=gepref[j]`, or variable `i` with bound preferences `ubp=ubpref[i]` and `lbp=lbpref[i]`, the following rules are applied while introducing the auxiliary variables:

Preference	Affects	Relaxation	Cost if pref.>0	Cost if pref.<0
`lrp`	`= rows`	$a^T x$ `- aux_var = b`	`1/lrp*aux_var`	`1/lrp*aux_var`^{2}
`lrp`	`<= rows`	$a^T x$ `- aux_var <= b`	`1/lrp*aux_var`	`1/lrp*aux_var`^{2}
`grp`	`= rows`	$a^T x$ `+ aux_var = b`	`1/grp*aux_var`	`1/grp*aux_var`^{2}
`grp`	`>= rows`	$a^T x$ `+ aux_var >= b`	`1/grp*aux_var`	`1/grp*aux_var`^{2}
`ubp`	`upper bounds`	x_i `- aux_var <= u`	`1/ubp*aux_var`	`1/ubp*aux_var`^{2}
`lbp`	`lower bounds`	x_i `+ aux_var >= l`	`1/lbp*aux_var`	`1/lbp*aux_var`^{2}

8. Only positive bounds are applied; a zero or negative bound is ignored and the amount of relaxation allowed for the corresponding row or bound is not limited. The effect of a zero bound on a row or bound would be equivalent with not relaxing it, and can be achieved by setting its preference array value to zero instead, or not including it in the preference arrays.
9. If an irreducible infeasible set (IIS) has been identified, the generated IIS(s) are accesible through the IIS retrieval functions, see `NUMIIS` and `problem.getIISData`.

Related topics

`problem.repairInfeas`, `problem.repairWeightedInfeasBounds`.

problem.rhssa

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.rhsSA` *instead.*
Returns upper and lower sensitivity ranges for specified right hand side (RHS) function coefficients. If the RHS coefficients are varied within these ranges the current basis remains optimal and the reduced costs remain valid.

Synopsis

```
problem.rhssa(rowind, lower, upper)
```

Arguments

`rowind`	Array containing the rows (i.e., `xpress.constraint` objects, indices, or names) whose RHS coefficients sensitivity ranges are required.
`lower`	Array where the RHS lower range values are to be returned.
`upper`	Array where the RHS upper range values are to be returned.

Example

Here we obtain the RHS function ranges for the three columns: `2`, `6` and `8`:

```
l = []
u = []
p.rhssa([2,8,6], l, u)
```

After which lower and upper contain:

```
l = [5, 3.8, 5.7]
u = [7, 5.2, 1e+20]
```

Meaning that the current basis remains optimal when $5.0 \leqslant \text{rhs}_2$, $3.8 \leqslant \text{rhs}_8 \leqslant 5.2$ and $5.7 \leqslant \text{rhs}_6$, rhs_i being the RHS coefficient of row `i`.

Further information

`rhssa` can only be called when an optimal solution to the current LP has been found. It cannot be used when the problem is MIP presolved.

Related topics

`problem.bndSA`, `problem.rhsSA`, `problem.objSA`.

problem.scaling

Purpose

This subroutine is deprecated and will be removed in a future release.
Analyze the current matrix for largest/smallest coefficients and ratios

Synopsis

```
problem.scaling()
```

Example

The following example analyzes the matrix

```
p.scaling()
```

Further information

The current matrix (including augmentation if it has been carried out) is scanned for the absolute and relative sizes of elements. The following information is reported:

- Largest and smallest elements in the matrix;
- Counts of the ranges of row ratios in powers of 10 (e.g. number of rows with ratio between 10 and 100);
- List of the rows (with largest and smallest elements) which appear in the highest range;
- Counts of the ranges of column ratios in powers of 10;
- List of the columns (with largest and smallest elements) which appear in the highest range;
- Element ranges in powers of 10.

Where any of the reported items (largest or smallest element in the matrix or any reported row or column element) is in a penalty error vector, the results are repeated, excluding all penalty error vectors.

problem.setcurrentiv

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.nlpCurrentIV` *instead.*
Transfer the current solution to initial values

Synopsis

```
problem.setcurrentiv()
```

Further information

Provides a way to set the current iterates solution as initial values, make changes to parameters or to the underlying nonlinear problem and then rerun the SLP optimization process.

Related topics

`problem.slpReInitialize`, `problem.slpUnConstruct`, `problem.nlpCurrentIV`

problem.setdefaultcontrol

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.setDefaultControl` *instead.*
Sets one control to its default value. Must be called before the problem is read or loaded by `problem.readProb` and `problem.loadproblem`.

Synopsis

```
problem.setdefaultcontrol(control)
```

Argument

`control` Name of the control to be set to default.

Example

The following turns off presolve to solve a problem, before resetting the control defaults, reading it and solving it again:

```
p.controls.presolve = 0
p.mipoptimize("")
p.writeprtsol()
p.setdefaultcontrol('presolve')
p.read()
p.mipoptimize("")
```

Related topics

`xpress.setDefaultControl`, `xpress.setDefaults`, `problem.setDefaultControl`.

problem.setdefaults

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.setDefaults` *instead.*
Sets all controls to their default values. It must be called before the problem is read with `problem.readProb` or loaded with `problem.loadproblem`.

Synopsis

```
problem.setdefaults()
```

Example

The following turns off presolve to solve a problem, before resetting the control defaults, reading it and solving it again:

```
p.controls.presolve = 0
p.mipoptimize("")
p.writeprtsol()
p.setdefaults()
p.read()
p.mipoptimize("")
```

Related topics

`xpress.setDefaultControl`, `xpress.setDefaults`, `problem.setDefaults`.

problem.setindicators

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.setIndicators` *instead.* Specifies that a set of rows in the problem will be treated as indicator constraints during a tree search. An indicator constraint is made of a `condition` and a `linear inequality`. The `condition` is of the type "`bin = value`", where `bin` is a binary variable and `value` is either 0 or 1. The `linear inequality` is any linear row in the problem with type <= (`L`) or >= (`G`). During tree search, a row configured as an indicator constraint is enforced only when condition holds, that is only if the indicator variable `bin` has the specified value.

Synopsis

```
problem.setindicators(rowind, colind, complement)
```

Arguments

`rowind` Array containing the rows (i.e., `xpress.constraint` objects, indices, or names) that define the linear inequality part for the indicator constraints.

`colind` Array containing the columns (i.e., `xpress.var` objects, indices, or names) of the indicator variables.

`complement` Array with the complement flags:

- `0` not an indicator constraint (in this case the corresponding entry in the `colind` array is ignored);
- `1` for indicator constraints with condition "`bin = 1`";
- `-1` for indicator constraints with condition "`bin = 0`";

Example

This sets the first two matrix rows as indicator rows in the MIP problem prob; the first row controlled by condition `x4=1` and the second row controlled by condition `x5=0` (assuming `x4` and `x5` correspond to columns indices 4 and 5).

```
p.setindicators([0,1],[4,5],[1,-1])
p.mipoptimize("")
```

Further information

Indicator rows must be set up before solving the problem. Any indicator row will be removed from the problem after presolve and added to a special pool. An indicator row will be added back into the active matrix only when its associated condition holds. An indicator variable can be used in multiple indicator rows and can also appear in normal rows and in the objective function.

Related topics

`problem.getIndicators`, `problem.setIndicators`.

problem.setlogfile

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.setLogFile` *instead.*
This directs all Optimizer output to a log file.

Synopsis

```
problem.setlogfile(filename)
```

Argument

`filename` The name of the file to which all output will be directed. If set to `None`, redirection of the output will stop and all screen output will be turned back on (except for DLL users where screen output is always turned off).

Example

The following directs output to the file `logfile.log`:

```
p = xpress.problem()
p.setlogfile("logfile.log")
```

Further information

1. It is recommended that a log file be set up for each problem being worked on, since it provides a means for obtaining any errors or warnings output by the Optimizer during the solution process.
2. If output is redirected with `setlogfile` all screen output will be turned off.
3. Alternatively, an output callback can be defined using `problem.addMessageCallback`, which will be called every time a line of text is output. Defining a user output callback will turn all screen output off. To discard all output messages the `OUTPUTLOG` integer control can be set to `0`.

Related topics

`problem.addMessageCallback`, `problem.setLogFile`.

problem.setmessagestatus

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.setMessageStatus` *instead.*
Manages suppression of messages.

Synopsis

```
problem.setmessagestatus(msgcode, status)
```

Arguments

`msgcode`	The id number of the message. Refer to the Section 9 of the Xpress Optimizer reference manual for a list of possible message numbers.
`status`	Nonzero if the message is not suppressed; `0` otherwise.

Example

Attempting to optimize a problem that has no matrix loaded gives error 91. The following code uses `setmessagestatus` to suppress the error message:

```
p = xpress.problem()
p.setmessagestatus(91, 0)
p.lpoptimize("")
```

Further information

If a message is suppressed globally then the message can only be enabled for any problem once the global suppression is removed with a call to `setmessagestatus` with `prob` passed as `None`.

Related topics

`problem.getMessageStatus`, `problem.setMessageStatus`.

problem.setprobname

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.setProbName` *instead.*
Sets the current default problem name.

Synopsis

```
problem.setprobname(probname)
```

Argument

`probname` A string of up to `MAXPROBNAMELENGTH` characters containing the problem name.

Related topics

`problem.readProb`, `problem.getProbName`, `problem.setProbName`, `MAXPROBNAMELENGTH`.

problem.slpgetcoefstr

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpGetCoefStr` instead.
Retrieve a single matrix coefficient as a formula in a character string

Synopsis

```
formula, factor = problem.slpgetcoefstr(row, col)
```

Arguments

`row`	The row (i.e., `xpress.constraint` object, index, or name) for the coefficient.
`col`	The column (i.e., `xpress.var` object, index, or name) for the coefficient.

Return value

`formula`	The formula string.
`factor`	The constant factor multiplying the formula in the coefficient.

Example

The following example retrieves the formula for the coefficient of column 3 in in row 2:

```
formula, _ = problem.slpgetformulastr(2, 3)
  print("Formula is {}".format(formula))
```

Further information

If the requested coefficient is constant, then `factor` will be set to 1.0 and the value will be formatted in `formula`.

Related topics

`problem.slpChgCoefStr`, `problem.slpChgCoef`, `problem.slpGetCoefStr`

problem.storecuts

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.storeCuts` *instead.*
Stores cuts into the cut pool, but does not apply them to the current node. These cuts must be explicitly loaded into the matrix using `problem.loadCuts` before they become active.

Synopsis

```
problem.storecuts(nodups, cuttype, rowtype, rhs, start, cutind, colind, cutcoef)
```

Arguments

`nodups`	`0`	do not exclude duplicates from the cut pool;
	`1`	duplicates are to be excluded from the cut pool;
	`2`	duplicates are to be excluded from the cut pool, ignoring cut type.
`cuttype`	Array containing the cut types. The cut types can be any integer and are used to identify the cuts.	
`rowtype`	Character array containing the row types:	
	`L`	indicates a $\leqslant$ row;
	`E`	indicates an = row;
	`G`	indicates a $\geqslant$ row.
`rhs`	Array containing the right hand side elements for the cuts.	
`start`	Array containing offsets into the `colind` and `cutcoef` arrays indicating the start of each cut. This array is of length `ncuts+1` where `ncuts` is the length of `rhs`, with the last element `start[ncuts]` being where cut `ncuts+1` would start.	
`cutind`	Array where the cuts will be returned.	
`colind`	Array containing the columns in the cuts.	
`cutcoef`	Array containing the matrix values for the cuts.	

Further information

1. `storecuts` can be used to eliminate duplicate cuts. If the `nodups` parameter is set to `1`, the cut pool will be checked for duplicate cuts with a cut type identical to the cuts being added. If a duplicate cut is found the new cut will only be added if its right hand side value makes the cut stronger. If the cut in the pool is weaker than the added cut it will be removed unless it has been applied to an active node of the tree. If `nodups` is set to `2` the same test is carried out on all cuts, ignoring the cut type.
2. `storecuts` returns a list of the cuts added to the cut pool in the `cutind` array. If the cut is not added to the cut pool because a stronger cut exits a None will be returned. The `cutind` array can be passed directly to `problem.loadCuts` to load the most recently stored cuts into the matrix.
3. The columns and elements of the cuts must be stored contiguously in the `colind` and `cutcoef` arrays passed to `storecuts`. The starting point of each cut must be stored in the `start` array. To determine the length of the final cut the `start` array must be of length `ncuts+1` with the last element of this array containing where the cut `ncuts+1` would start.

Related topics

`problem.loadCuts`, `problem.storeCuts`, Section "Working with the cut manager" of the Xpress Optimizer reference manual.

problem.strongbranch

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.strongBranch` *instead.* Performs strong branching iterations on all specified bound changes. For each candidate bound change, `strongbranch` performs dual simplex iterations starting from the current optimal solution of the base LP, and returns both the status and objective value reached after these iterations.

Synopsis

```
problem.strongbranch(colind, bndtype, bndval, iterlim, objval, status)
```

Arguments

`colind`	Array containing the columns (i.e., `xpress.var` objects, indices, or names) on which the bounds will change.
`bndtype`	Character array indicating the type of bound to change: `U` indicates change the upper bound; `L` indicates change the lower bound; `B` indicates change both bounds, i.e., fix the column.
`bndval`	Array giving the new bound values.
`iterlim`	Maximum number of LP iterations to perform for each bound change.
`objval`	Objective value of each LP after performing the strong branching iterations.
`status`	Status of each LP after performing the strong branching iterations, as detailed for the `LPSTATUS` attribute.

Example

Suppose that the current LP relaxation has two integer columns (columns `0` and `1` which are fractionals at 0.3 and 1.5, respectively, and we want to perform strong branching in order to choose which to branch on. This could be done in the following way:

```
objval = []
status = []
p.strongbranch([0,0,1,0], ['','','',''], [1,0,2,1],
               1000, objval, status)
```

Further information

Prior to calling strongbranch, the current LP problem must have been solved to optimality and an optimal basis must be available.

problem.strongbranchcb

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.strongBranchCB` *instead.* Performs strong branching iterations on all specified bound changes. For each candidate bound change, `strongbranchcb` performs dual simplex iterations starting from the current optimal solution of the base LP, and returns both the status and objective value reached after these iterations.

Synopsis

```
problem.strongbranchcb(colind, bndtype, bndval, iterlim, objval, status, callback,
       data)
ret = callback(my_prob, my_object, ibnd)
```

Arguments

`colind`	Array containing the columns (i.e., `xpress.var` objects, indices, or names) on which the bounds will change.
`bndtype`	Character array indicating the type of bound to change: `U` indicates change the upper bound; `L` indicates change the lower bound; `B` indicates change both bounds, i.e., fix the column.
`bndval`	Array giving the new bound values.
`iterlim`	Maximum number of LP iterations to perform for each bound change.
`objval`	Objective value of each LP after performing the strong branching iterations.
`status`	Status of each LP after performing the strong branching iterations, as detailed for the `LPSTATUS` attribute.
`callback`	Function to be called after each strong branch has been reoptimized.
`data`	The user-defined object passed as `my_object` to `callback`.
`ibnd`	The index of bound for which `callback` is called.

Further information

Prior to calling `strongbranchcb`, the current LP problem must have been solved to optimality and an optimal basis must be available.

`strongbranchcb` is an extension to `problem.strongBranch`. If identical input arguments are provided both will return identical results, the difference being that for the case of `PRSstrongbranchcb` the `sbnodecb` function is called at the end of each LP reoptimization. For each branch optimized, the LP can be interrogated: the LP status of the branch is available through checking `LPSTATUS`, and the objective function value is available through `LPOBJVAL`. It is possible to access the full current LP solution by using `problem.getCallbackSolution`.

problem.tuneprobsetfile

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.tuneProbSetFile` *instead.* Begins a tuner session for a set of problems. The tuner will solve the problems multiple times while evaluating a list of control settings and promising combinations of them. When finished, the tuner will select and set the best control setting on the problems.

Synopsis

```
problem.tuneprobsetfile(setfile, ifmip, sense)
```

Arguments

`setfile`	A plain text file which contains a list of problem files in MPS or LP format.	
`ifmip`	`-1`	to automatically determine whether to solve the problem set as LP or MIP (the default);
	`0`	to force the tuner to tune the problem set as LP;
	`1`	to force the tuner to tune the problem set as MIP.
`sense`	`0`	to automatically determine the sense of each problem (the default);
	`1`	to force the tuner to minimize each problem;
	`-1`	to force the tuner to maximize each problem.

Example

```
problem.tuneprobsetfile("problem.set")
```

This tunes the problems contained in `problem.set`. The problem types and objective senses are automatically determined.

Further information

Please refer to the Xpress Optimizer reference manual for a detailed guide of how to use the tuner.

Related topics

`problem.tune`, `problem.tunerReadMethod`, `problem.tunerWriteMethod`, `problem.tuneProbSetFile`.

problem.tunerreadmethod

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.tunerReadMethod` *instead.*
Load a user defined tuner method from the given file.

Synopsis

```
problem.tunerreadmethod(methodfile)
```

Argument

`methodfile`	The method file name, from which the tuner can load a user-defined tuner method.

Example

```
p.tunerreadmethod('method.xtm')
```

This loads the tuner method from the `method.xtm` file.

Further information

Please refer to the Xpress Optimizer reference manual for more information about the tuner method and for the format of the tuner method file.

Related topics

`problem.tunerWriteMethod`, `problem.tune`, `problem.tuneProbSetFile`, `problem.tunerReadMethod`.

problem.tunerwritemethod

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.tunerWriteMethod` *instead.*
Writes the current tuner method to a given file or prints it to the console.

Synopsis

```
problem.tunerwritemethod(methodfile)
```

Argument

`methodfile`	The file name to which the tuner will write the current tuner method. If the input is `stdout` or `STDOUT`, then the tuner will print the method to the console instead.

Example 1 (Library)

```
p.tunerwritemethod('method.xtm')
```

This writes the tuner method to the file `method.xtm`.

Example 2 (Library)

```
p.tunerwritemethod('stdout')
```

This prints the tuner method to the console.

Further information

Please refer to the Xpress Optimizer reference manual for more information about the tuner method and for the format of the tuner method file.

Related topics

`problem.tunerReadMethod`, `problem.tune`, `problem.tuneProbSetFile`, `problem.tunerWriteMethod`.

problem.unconstruct

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpUnConstruct` *instead.*
Reset the SLP problem and removes the augmentation structures

Synopsis

```
problem.unconstruct()
```

Further information

Can be used to rerun the SLP optimization process with changed parameters or underlying lienar / nonlienar strcutures.

Related topics

`problem.slpReInitialize`, `problem.nlpCurrentIV`, `problem.slpUnConstruct`.

problem.updatelinearization

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.slpUpdateLinearization` *instead.*

Updates the current linearization

Synopsis

```
problem.updatelinearization()
```

Further information

Updates the augmented probem (the linearization) to match the current base point. The base point is the current SLP solution. The values of the SLP variables can be changed using `problem.chgvar`.

The linearization must be present, and this function can only be called after the problem has been augmented by `problem.slpConstruct`.

Related topics

`problem.slpConstruct`, `problem.slpUpdateLinearization`

problem.validate

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.nlpValidate` *instead.*
Validate the feasibility of constraints in a converged solution

Synopsis

```
problem.validate()
```

Example

The following example sets the validation tolerance parameters, validates the converged solution and retrieves the validation indices.

```
p.controls.xslp_validationtol_a = 0.001
p.controls.xslp_validationtol_r = 0.001
p.validate()
indexA = p.attributes.xslp_validationindex_a
indexR = p.attributes.xslp_validationindex_r
```

Further information

`validate` checks the feasibility of a converged solution against relative and absolute tolerances for each constraint. The left hand side and the right hand side of the constraint are calculated using the converged solution values. If the calculated values imply that the constraint is infeasible, then the difference (D) is tested against the absolute and relative validation tolerances.
If $\mathrm{D} < \mathrm{XSLP_VALIDATIONTOL_A}$
then the constraint is within the absolute validation tolerance. The total positive (TPos) and negative contributions (TNeg) to the left hand side are also calculated.
If $\mathrm{D} < MAX(\mathrm{ABS(TPos)}, \mathrm{ABS(TNeg)}) * \mathrm{XSLP_VALIDATIONTOL_R}$
then the constraint is within the relative validation tolerance. For each constraint which is outside both the absolute and relative validation tolerances, validation factors are calculated which are the factors by which the infeasibility exceeds the corresponding validation tolerance; the smallest factor is printed in the validation report.
The validation index `xslp_validationindex_a` is the largest absolute validation factor multiplied by the absolute validation tolerance; the validation index `xslp_validationindex_r` is the largest relative validation factor multiplied by the relative validation tolerance.

Related topics

, `problem.nlpValidate` `xslp_validationindex_A`, `xslp_validationindex_R`, `xslp_validationtol_A`, `xslp_validationtol_R`

problem.validatekkt

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.nlpValidateKKT` *instead.* Validates the first order optimality conditions also known as the Karush-Kuhn-Tucker (KKT) conditions versus the currect solution

Synopsis

```
problem.validatekkt(mode, respectbasis, updatemult, violtarget)
```

Arguments

`mode` The calculation mode can be:

- `0` recalculate the reduced costs at the current solution using the current dual solution.
- `1` minimize the sum of KKT violations by adjusting the dual solution.
- `2` perform both.

`respectbasis` The following ways are defined to assess if a constraint is active:

- `0` evaluate the recalculated slack activity versus `xslp_ECFTOL_R`.
- `1` use the basis status of the slack in the linearized problem if available.
- `2` use both.

`updatemult` The calculated values can be:

- `0` only used to calculate the `xslp_validationindex_k` measure.
- `1` used to update the current dual solution and reduced costs.

`violtarget` When calculating the best KKT multipliers, it is possible to enforce an even distribution of reduced costs violations by enforcing a bound on them.

Further information

The bounds enforced by `violtarget` are automatically relaxed if the desired accuracy cannot be achieved.

problem.validaterow

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.nlpValidateRow` *instead.*
Prints an extensive analysis on a given constraint of the SLP problem

Synopsis

```
problem.validate(row)
```

Argument

`row` The row (i.e., `xpress.constraint` object, index, or name) to be analyzed.

Further information

The analysis will include the readable format of the original constraint and the augmented constraint. For infeasible constraints, the absolute and relative infeasibility is calculated. Variables in the constraints are listed including their value in the solution of the last linearization, the internal value (e.g. cascaded), reduced cost, step bound and convergence status. Scaling analysis is also provided.

problem.validatevector

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.nlpValidateVector` *instead.*

Validate the feasibility of constraints for a given solution

Synopsis

```
(suminf, sumscaledinf, obj) = problem.validate(solution)
```

Arguments

`solution`	A vector of length `xpress.attributes.cols` containing the solution vector to be checked.
`suminf`	The sum of infeasibilities.
`sumscaledinf`	The sum of scaled (relative) infeasibilities.
`obj`	The net objective.

Further information

`validatevector` works the same way as `problem.nlpValidate`, and will update `xslp_validationindex_a` and `xslp_validationindex_r`.

Related topics

, `problem.nlpValidateVector` `Xslp_Validationindex_a`, `xslp_validationindex_r`, `xslp_validationtol_a`, `xslp_validationtol_r`

problem.write

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.writeProb` *instead.*
Writes the current problem to an MPS or LP file.

Synopsis

```
problem.write(filename, flags)
```

Arguments

`filename` A string of up to 200 characters to contain the file name to which the problem is to be written. If omitted, the default *problem_name* is used with a `.mps` extension, unless the `l` flag is used in which case the extension is `.lp`.

`flags` (optional) Flags, which can be one or more of the following:

- `h` single precision of numerical values;
- `o` one element per line;
- `n` scaled;
- `s` scrambled vector names;
- `l` output in LP format;
- `x` output MPS file in hexadecimal format.
- `p` obsolete flag (now default behavior).

Example

The following example outputs the current problem in full precision, LP format with scrambled vector names to the file *problem_name*`.lp`.

```
p.write("", "lps")
```

Further information

1. If `problem.loadproblem` is used to obtain a problem then there is no association between the objective function and the `N` rows in the problem and so a separate `N` row (called `__OBJ___`) is created upon a `write`. Also, if after a call to `read` either the objective row or the `N` row in the problem corresponding to the objective row are changed, the association between the two is lost and the `__OBJ___` row is created with an `write`. To remove the objective row from the problem when doing a `read`, set `keepnrows` to `-1` before `read`.
2. The hexadecimal format is useful for saving the exact internal precision of the problem.
3. **Warning:** If `problem.readProb` is used to input a problem, then the input file will be overwritten by `write` if a new filename is not specified.

Related topics

`problem.readProb`.

problem.writebasis

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.writeBasis` *instead.*
Writes the current basis to a file for later input into the Optimizer.

Synopsis

```
problem.writebasis (filename, flags)
```

Arguments

`filename` A string of up to 200 characters containing the file name from which the basis is to be written. If omitted, the default *problem_name* is used with a `.bss` extension.

`flags` (optional) Flags to pass to `writebasis`:

`i`	output the internal presolved basis.
`t`	output a compact advanced form of the basis.
`n`	output basis file containing current solution values.
`h`	output values in single precision.
`x`	output values in hexadecimal format.
`p`	obsolete flag (now default behavior).

Example

After an LP has been solved it may be desirable to save the basis for future input as an advanced starting point for other similar problems. This may save significant amounts of time if the LP is complex. For example:

```
p.read("myprob", "")
p.lpoptimize("")
p.writebasis("", "")
```

This reads in a problem file, maximizes the LP and saves the basis. Loading a basis for a MIP problem can disable some MIP presolve operations which can result in a large increase in solution times so it is generally not recommended.

Further information

1. The `t` flag is only useful for later input to a similar problem using the `t` flag with `problem.readBasis`.
2. If the Newton barrier algorithm has been used for optimization then crossover must have been performed before there is a valid basis. This basis can then only be used for restarting the simplex (primal or dual) algorithm.
3. `writebasis` will output the basis for the original problem even if the problem has been presolved.

Related topics

`problem.getBasis`, `problem.readBasis`, `problem.writeBasis`.

problem.writebinsol

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.writeBinSol` *instead.*
Writes the current MIP or LP solution to a binary solution file for later input into the Optimizer.

Synopsis

```
problem.writebinsol(filename, flags)
```

Arguments

`filename` A string of up to 200 characters containing the file name to which the solution is to be written. If omitted, the default *problem_name* is used with a `.sol` extension.

`flags` (optional) Flags to pass to `writebinsol`:
`x` output the LP solution.

Example

After an LP has been solved or a MIP solution has been found the solution can be saved to file. If a MIP solution exists it will be written to file unless the `x` flag is passed to `writebinsol` in which case the LP solution will be written.

```
p.read("myprob", "")
p.mipoptimize("")
p.writebinsol("", "")
```

Related topics

`problem.getSolution`, `problem.readBinSol`, `problem.writeSol`, `problem.writePrtSol`, `problem.writeBinSol`.

problem.writedirs

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.writeDirs` *instead.*
Writes the tree search directives from the current problem to a directives file.

Synopsis

```
problem.writedirs(filename)
```

Argument

`filename`	A string of up to 200 characters containing the file name to which the directives should be written. If omitted (or `None`), the default *problem_name* is used with a `.dir` extension.

Further information

If the problem has been presolved, only the directives for columns in the presolved problem will be written to file.

Related topics

`problem.loadDirs`, `problem.writeDirs`.

problem.writeprtsol

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.writePrtSol` *instead.*
Writes the current solution to a fixed format ASCII file, *problem_name* `.prt`.

Synopsis

```
problem.writeprtsol(filename, flags)
```

Arguments

`filename` A string of up to 200 characters containing the file name to which the solution is to be written. If omitted, the default *problem_name* will be used. The extension `.prt` will be appended.

`flags` (optional) Flags for `writeprtsol` are:

`x` write the LP solution instead of the current MIP solution.

Example

This example shows the standard use of this function, outputting the solution to file immediately following optimization:

```
p.read("myprob", "")
p.lpoptimize("")
p.writeprtsol("", "")
```

Further information

1. The fixed width ASCII format created by this function is not as readily useful as that produced by `problem.writeSol`. The main purpose of `writeprtsol` is to create a file that can be sent directly to a printer. The format of this fixed format ASCII file is described in the Xpress Optimizer reference manual.
2. To create a prt file for a previously saved solution, the solution must first be loaded with the `problem.readBinSol` function.

Related topics

`problem.getSolution`, `problem.readBinSol`, `problem.writeBinSol`, `problem.writeSol`, `problem.writePrtSol`.

problem.writeslxsol

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.writeSlxSol` *instead.* Creates an ASCII solution file (`.slx`) using a similar format to MPS files. These files can be read back into the Optimizer using the `problem.readSlxSol` function.

Synopsis

```
problem.writeslxsol(filename, flags)
```

Arguments

`filename` A string of up to 200 characters containing the file name to which the solution is to be written. If omitted, the default *problem_name* is used with a `.slx` extension.

`flags` (optional) Flags to pass to `writeslxsol`:

`l`	write the LP solution in case of a MIP problem;
`m`	write the MIP solution;
`p`	use full precision for numerical values;
`x`	use hexadecimal format to write values;
`d`	LP solution only: including dual variables;
`s`	LP solution only: including slack variables;
`r`	LP solution only: including reduced cost.

Example

```
p.writeslxsol("lpsolution", "")
```

This saves the MIP solution if the problem contains MIP entities, or otherwise saves the LP(barrier in case of quadratic problems) solution of the problem.

Related topics

`problem.readSlxSol`, `problem.writePrtSol`, `problem.writeBinSol`, `problem.readBinSol`, `problem.writeSlxSol`.

problem.writesol

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.writeSol` *instead.*
Writes the current solution to a CSV format ASCII file, *problem_name* `.asc`(and `.hdr`).

Synopsis

```
problem.writesol(filename, flags)
```

Arguments

`filename` A string of up to 200 characters containing the file name to which the solution is to be written. If omitted, the default *problem_name* will be used. The extensions `.hdr` and `.asc` will be appended.

`flags` (optional) Flags to control which optional fields are output:

- `s` sequence number;
- `n` name;
- `t` type;
- `b` basis status;
- `a` activity;
- `c` cost (columns), slack (rows);
- `l` lower bound;
- `u` upper bound;
- `d` dj (column; reduced costs), dual value (rows; shadow prices);
- `r` right hand side (rows).

If no flags are specified, all fields are output.
Additional flags:

- `e` outputs every MIP solution saved;
- `p` outputs in full precision;
- `q` only outputs vectors with nonzero optimum value;
- `x` output the current LP solution instead of the MIP solution.

Example

In this example the basis status is output (along with the sequence number) following optimization:

```
p.read("prob1", "")
p.lpoptimize("")
p.writesol("", "sb")
```

Further information

1. The function produces two readable files: `filename.hdr` (the solution header file) and `filename.asc` (the CSV foramt solution file). The header file contains summary information, all in one line. The ASCII file contains one line of information for each row and column in the problem. Any fields appearing in the `.asc` file will be in the order the flags are described above. The order that the flags are specified by the user is irrelevant.
2. Additionally, the mask control `OUTPUTMASK` may be used to control which names are reported to the ASCII file. Only vectors whose names match `OUTPUTMASK` are output. `OUTPUTMASK` is set by default to "`????????`", so that all vectors are output.

Related topics

`problem.getSolution`, `problem.writePrtSol`, `problem.writeSol`.

branchobj.addbounds

Purpose

This subroutine is deprecated and will be removed in a future release. Use `branchobj.addBounds` *instead.*
Adds new bounds to a branch of a user branching object.

Synopsis

```
branchobj.addbounds(branch, bndtype, colind, bndval)
```

Arguments

`branch`	The number of the branch to add the new bounds for. This branch must already have been created using `branchobj.addBranches`. Branches are indexed starting from zero.
`bndtype`	Character array indicating the type of bounds to add: `L` Lower bound. `U` Upper bound.
`colind`	Array containing the columns for the new bounds.
`bndval`	Array giving the bound values.

branchobj.addbranches

Purpose

This subroutine is deprecated and will be removed in a future release. Use `branchobj.addBranches` *instead.*
Adds new, empty branches to a user defined branching object.

Synopsis

```
branchobj.addbranches(nbranches)
```

Argument

`nbranches` Number of new branches to create.

branchobj.addcuts

Purpose

This subroutine is deprecated and will be removed in a future release. Use `branchobj.addCuts` *instead.*
Adds stored user cuts as new constraints to a branch of a user branching object.

Synopsis

```
branchobj.addcuts(branch, cutind)
```

Arguments

`branch`	The number of the branch to add the cuts for. This branch must already have been created using `branchobj.addBranches`. Branches are indexed starting from zero.
`cutind`	Array containing the user cuts that should be added to the branch.

Related topics

`branchobj.addRows`, `branchobj.addCuts`.

branchobj.addrows

Purpose

This subroutine is deprecated and will be removed in a future release. Use `branchobj.addRows` *instead.*
Adds new constraints to a branch of a user branching object.

Synopsis

```
branchobj.addrows(branch, rowtype, rhs, start, colind, rowcoef)
```

Arguments

`branch` The number of the branch to add the new constraints for. This branch must already have been created using `branchobj.addBranches`. Branches are indexed starting from zero.

`rowtype` Character array indicating the type of constraints to add:
- `L` Less than type.
- `G` Greater than type.
- `E` Equality type.

`rhs` Array containing the right-hand side values.

`start` Array containing the offsets in the `colind` and `rowcoef` arrays of the start of the nonzero coefficients for each new constraint. The array must contain one more element than the number of rows to add, with the last element being the total number of coefficients across all rows.

`colind` Array containing the columns for the nonzero coefficients.

`rowcoef` Array containing the nonzero coefficient values.

Example

The following function will create a branching object that branches on constraints $x_1 + x_2 \geqslant 1$ or $x_1 + x_2 \leqslant 0$, where `x1` and `x2` are instances of `xpress.var`:

```
def CreateConstraintBranch(prob, x1, x2):

    # Create the new object with two empty branches.
    bo = xpress.branchobj(prob, isoriginal=True)
    bo.addbranches(2)

    # Add the constraints to the branching object
    # x1 + x2 >= 1
    # x1 + x2 <= 0
    bo.addrows(0, ['G'], [1.0], [0], [x1, x2], [1.0, 1.0])
    bo.addrows(1, ['L'], [0.0], [0], [x1, x2], [1.0, 1.0])

    # Set a low priority value so our branch object is picked up
    # before the default branch candidates.
    bo.setpriority(100)

    return bo
```

branchobj.getbounds

Purpose

This subroutine is deprecated and will be removed in a future release. Use `branchobj.getBounds` *instead.* Returns the bounds for a branch of a user branching object. The returned value is the actual number of bounds returned in the output arrays.

Synopsis

```
branchobj.getbounds(branch, maxbounds, bndtype, colind, bndval)
```

Arguments

`branch`	The number of the branch to get the bounds for.
`maxbounds`	Maximum number of bounds to return.
`bndtype`	Character array of length `maxbounds` where the types of bounds will be returned: `L` Lower bound. `U` Upper bound.
`colind`	Array of length `maxbounds` where the columns will be returned.
`bndval`	Array of length `maxbounds` where the bound values will be returned.

Related topics

`branchobj.addbounds`.

branchobj.getbranches

Purpose

This subroutine is deprecated and will be removed in a future release. Use `branchobj.getBranches` *instead.*
Returns the number of branches of a branching object.

Synopsis

```
branchobj.getbranches()
```

Related topics

`branchobj.addBranches`.

branchobj.getid

Purpose

This subroutine is deprecated and will be removed in a future release. Use `branchobj.getID` *instead.*
Returns the unique identifier assigned to a branching object.

Synopsis

```
branchobj.getid()
```

Further information

1. Branching objects associated with existing column entities (binaries, integers, semi–continuous and partial integers), are given an identifier from 1 to `MIPENTS`.
2. Branching objects associated with existing Special Ordered Sets are given an identifier from `MIPENTS+1` to `MIPENTS+SETS`.
3. User created branching objects will always have a negative identifier.

branchobj.getlasterror

Purpose

This subroutine is deprecated and will be removed in a future release.
Returns the last error encountered during a call to the given branch object.

Synopsis

```
(id,msg) = branchobj.getlasterror()
```

Arguments

`id`	Error code.
`msg`	A string with the last error message relating to the branching object will be returned.

Example

The following shows how this function might be used in error checking:

```
obranch = xpress.branchobj()

try:
    obranch.setpreferredbranch(3)
except:
   (i,m) = obranch.getlasterror()
   print("ERROR when setting preferred branch:", m)
```

branchobj.getrows

Purpose

This subroutine is deprecated and will be removed in a future release. Use `branchobj.getRows` *instead.*
Returns the constraints for a branch of a user branching object.

Synopsis

```
branchobj.getrows(branch, maxrows, maxcoefs, rowtype, rhs, start, colind, rowcoef)
```

Arguments

`branch`	The number of the branch to get the constraints from.
`maxrows`	Maximum number of rows to return.
`maxcoefs`	Maximum number of non zero coefficients to return.
`rowtype`	Character array of length `maxrows` where the types of the rows will be returned: `L` Less than type. `G` Greater than type. `E` Equality type.
`rhs`	Array of length `maxrows` where the right hand side values will be returned.
`start`	Array of length `maxrows` which will be filled with the offsets of the `colind` and `rowcoef` arrays of the start of the non zero coefficients in the returned constraints.
`colind`	Array of length `maxcoefs` which will be filled with the column indices for the non zero coefficients.
`rowcoef`	Array of length `maxcoefs` which will be filled with the non zero coefficient values.

Related topics

`branchobj.addRows`, `branchobj.getRows`.

branchobj.setpreferredbranch

Purpose

This subroutine is deprecated and will be removed in a future release. Use `branchobj.setPreferredBranch` *instead.*

Specifies which of the child nodes corresponding to the branches of the object should be explored first.

Synopsis

```
branchobj.setpreferredbranch(branch)
```

Argument

`branch`	The number of the branch to mark as preferred.

branchobj.setpriority

Purpose

This subroutine is deprecated and will be removed in a future release. Use `branchobj.setPriority` *instead.*
Sets the priority value of a user branching object.

Synopsis

```
branchobj.setpriority(priority)
```

Argument

`priority`	The new priority value to assign to the branching object, which must be a number from 0 to 1000. User branching objects are created with a default priority value of 500.

Further information

1. A candidate branching object with lowest priority number will always be selected for branching before an object with a higher number.
2. Priority values must be an integer from 0 to 1000. User branching objects and MIP entities are by default assigned a priority value of 500. Special branching objects, such as those arising from structural branches or split disjunctions are assigned a priority value of 400.

xpress.addcbmsghandler

Purpose

This subroutine is deprecated and will be removed in a future release.
Declares an output callback function in the global environment, called every time a line of message text is output by any object in the library. This callback function will be called in addition to any output callbacks already added by `xpress.addcbmsghandler`.

Synopsis

```
xpress.addcbmsghandler(msghandler, data, priority)
ret = f_msghandler(vObject, vUserContext, vSystemThreadId, sMsg, iMsgType,
       iMsgNumber)
```

Arguments

`msghandler` The callback function which takes six arguments, `vObject`, `vUserContext`, `vSystemThreadId`, `sMsg`, `iMsgType` and `iMsgNumber`. Use `None` to cancel a callback function.

`vObject` The object sending the message.

`vUserContext` The user-defined object passed to the callback function.

`vSystemThreadId` The system id of the thread sending the message cast to a `void *`.

`sMsg` A string containing the message, which may simply be a new line. When the callback is called for the first time `sMsg` will be empty.

`iMsgType` Indicates the type of output message:

- `1` information messages;
- `2` (not used);
- `3` warning messages;
- `4` error messages.

When the callback is called for the first time `iMsgType` will be a negative value.

`iMsgNumber` The number associated with the message. If the message is an error or a warning then you can look up the number in the section Optimizer Error and Warning Messages for advice on what it means and how to resolve the associated issue.

`data` A user-defined object to be passed to the callback function.

`priority` An integer that determines the order in which multiple message handler callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Related topics

`xpress.removecbmsghandler`.

xpress.featurequery

Purpose

This subroutine is deprecated and will be removed in a future release. Use `xpress.featureQuery` *instead.*
Returns `True` if the provided feature is available in the current license used by the optimizer, `False` otherwise.

Synopsis

```
xpress.featurequery(feature)
```

Argument

`feature` The feature string to be checked in the license.

Related topics

`xpress.featureQuery`.

xpress.getbanner

Purpose

This subroutine is deprecated and will be removed in a future release. Use `xpress.getBanner` *instead.*
Returns the banner and copyright message.

Synopsis

```
i = xpress.getbanner()
```

Example

```
print(xpress.getbanner())
```

Related topics

`xpress.getBanner`.

xpress.getcomputeallowed

Purpose

This subroutine is deprecated and will be removed in a future release. Use `xpress.getComputeAllowed` *instead.*
Queries whether the current application is allowed to use the Insight Compute interface.

Synopsis

```
isComputeAllowed = xpress.getcomputeallowed()
```

Return value

Whether to allow use of Insight Compute, will be one of the following values:

1 Always allow solves to be sent to Compute.

0 Never allow solves to be sent to Compute.

-1 Allow solves to be sent to Compute only from non-OEM applications.

Related topics

`xpress.getComputeAllowed`.

xpress.getcheckedmode

Purpose

This subroutine is deprecated and will be removed in a future release. Use `xpress.getCheckedMode` *instead.* Returns whether checking and validation of all Optimizer function calls is enabled for the current process. Checking and validation is enabled by default but can be disabled by `xpress.setCheckedMode`.

Synopsis

```
i = xpress.getcheckedmode()
```

Related topics

`xpress.setCheckedMode`, `xpress.getCheckedMode`.

xpress.getdaysleft

Purpose

This subroutine is deprecated and will be removed in a future release. Use `xpress.getDaysLeft` *instead.*
Returns the number of days left until an evaluation license expires.

Synopsis

```
d = xpress.getdaysleft()
```

Example

The following calls `getdaysleft` to print information about the license:

```
try:
    ndays = xpress.getdaysleft()
except RuntimeError:
    print("Not an evaluation license")
else
    print("Evaluation license expires in {0} days".format(ndays))
```

Further information

This function can only be used with evaluation licenses, and, if called when a normal license is in use, it returns an error. The expiry information for evaluation licenses is also included in the Optimizer banner message.

Related topics

`xpress.getDaysLeft`.

xpress.getlasterror

Purpose

This subroutine is deprecated and will be removed in a future release.
Returns the last error encountered during a call to the Xpress global environment.

Synopsis

```
(i,s) = xpress.getlasterror()
```

Arguments

`i`	Error code
`s`	Error message relating to the global environment will be returned.

Example

```
import xpress as xp
# last error referring to the global environment
print(xp.getlasterror())
```

xpress.getlicerrmsg

Purpose

This subroutine is deprecated and will be removed in a future release.
Returns the error message string describing the last licensing error, if any occurred.

Synopsis

```
m = xpress.getlicerrmsg()
```

Example

The following calls `getlicerrmsg` to find out why the Xpress Python module failed to initialize:

```
try:
   xpress.init()
except Exception:
   print(xpress.getlicerrmsg())
else:
   print("all good")
```

xpress.getversion

Purpose

This subroutine is deprecated and will be removed in a future release. Use `xpress.getVersion` *instead.*
Returns the full Optimizer version number as a string of the form 15.10.03, where 15 is the major release, 10 is the minor release, and 03 is the build number.

Synopsis

```
v = xpress.getversion()
```

Example

```
print("Using Xpress Optimizer version", xpress.getversion())
```

Related topics

`xpress.getVersion`.

xpress.getversionnumbers

Purpose

This subroutine is deprecated and will be removed in a future release. Use `xpress.getVersionNumbers` *instead.*
Returns the Optimizer version numbers split into major, minor, and build number.

Synopsis

```
major, minor, build = xpress.getversionnumbers()
```

Example

```
print("Using Xpress Optimizer version %d.%02d.%02d" % xpress.getversionnumbers())
```

Related topics

`xpress.getVersionNumbers`.

xpress.is_equal

Purpose

This subroutine is deprecated and will be removed in a future release. Deprecated alias for `xpress.isEqual`.

xpress.removecbmsghandler

Purpose

This subroutine is deprecated and will be removed in a future release.
Removes a message callback function previously added by `xpress.addcbmsghandler`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
xpress.removecbmsghandler(msghandler, data)
```

Arguments

`msghandler`	The callback function to remove. If `None` then all message callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all message callbacks with the function `msghandler` will be removed.

Related topics

`xpress.addcbmsghandler`.

xpress.setarchconsistency

Purpose

This subroutine is deprecated and will be removed in a future release. Use `xpress.setArchConsistency` *instead.*
Sets whether to force the same execution path on various CPU architecture extensions, in particular (pre-)AVX and AVX2.

Synopsis

```
xpress.setarchconsistency(ifArchConsistent=False)
```

Argument

`ifArchConsistent`	Whether to force the same execution path: `False` Do not force the same execution path (default behavior); `True` Force the same execution path.

Further information

Note that using this general environment API function is different from setting the `xpress.controls.cpuplatform` control. Setting this control selects a vectorization instruction set for the barrier method.

Related topics

`xpress.getComputeAllowed`, `xpress.setArchConsistency`.

xpress.setcomputeallowed

Purpose

This subroutine is deprecated and will be removed in a future release. Use `xpress.setComputeAllowed` *instead.*
Set whether the current application is allowed to use the Insight Compute interface.

Synopsis

```
xpress.setcomputeallowed(isComputeAllowed)
```

Argument

`isComputeAllowed` Whether to allow use of Insight Compute, must be one of the following values:

- `1` Always allow solves to be sent to Compute.
- `0` Never allow solves to be sent to Compute.
- `-1` Allow solves to be sent to Compute only from non-OEM applications.

Further information

1. This function controls whether this process would be allowed to use the Insight Compute Interface if the user tries to enable it.
2. If the user enables the Insight Compute Interface but the value specified through this function does not allow the Insight Compute Interface to be used, any solves will terminate with an immediate error. This function can be used to prevent solves from being sent to Insight Compute but cannot be used to force solves to be performed locally. The purpose of this function is to allow an application to prevent the optimization model being sent to the Insight Compute Interface.

Related topics

`xpress.setComputeAllowed`.

xpress.setcheckedmode

Purpose

This subroutine is deprecated and will be removed in a future release. Use `xpress.setCheckedMode` *instead.* Disable/enable some of the checking and validation of function calls and function call parameters for calls to the Xpress Optimizer API. This checking is relatively lightweight but disabling it can improve performance in cases where non-intensive Xpress Optimizer functions are called repeatedly in a short space of time.

Please note: after disabling checking and validation for function calls, invalid usage of Xpress Optimizer functions may not be detected and may cause the Xpress Optimizer process to behave unexpectedly or crash. It is not recommended to disable function call checking and validation during application development.

Synopsis

```
xpress.setcheckedmode(checked_mode)
```

Argument

`checked_mode`	Pass as False or 0 to disable much of the validation for all Xpress function calls from the current process. Pass True or 1 to re-enable validation. By default, validation is enabled.

Related topics

`xpress.getCheckedMode`, `xpress.setCheckedMode`.

xpress.setdefaultcontrol

Purpose

This subroutine is deprecated and will be removed in a future release. Deprecated alias for `xpress.setDefaultControl`.

xpress.setdefaults

Purpose

This subroutine is deprecated and will be removed in a future release. Deprecated alias for `xpress.setDefaults`.

xpress.vars

Purpose

This subroutine is deprecated and will be removed in a future release. Use `problem.addVariables` *to add linked variables to the problem instead.*
Creates a dictionary or array of variables. Similar to the creation of a single variable with `xpress.var`, `vars` allows for using one or more index sets, specified as sets, lists, `range` objects, or any iterable object. Specifying a number `k` as an argument is equivalent to `range(k)` but can be used to create a multi-dimensional `xpress.ndarray` of variables, and allows for more efficient creation. The result is otherwise a Python dictionary of variables, whose keys are tuple of indices. A collection of variables `x` that is created with `vars` can be indexed, for instance, as `x[i,j]` where `i` and `j` are indices in the lists provided.

Synopsis

```
x = xpress.vars(*indices, name="x", lb=0, ub=xpress.infinity, threshold=1,
       vartype=xpress.continuous)
```

Arguments

`indices`	One or more lists, sets, ranges, or iterable objects to be combined; in alternative, one can specify one or more numbers `k` to signify the range `0..k-1`. Using only numbers as argument will yield an `xpress.ndarray` with the dimensions as specified by the arguments themselves.
`name`	Prefix to be added to the name of each variable; see notes for more information.
`lb`	Lower bound for all variables.
`ub`	Upper bound for all variables.
`threshold`	Threshold for all variables; only used if the variables are partially integer.
`vartype`	Type of all variables, similar to the definition of single variables.

Example

The following creates a dictionary containing 6 variables whose indices vary in the set `{(0,'a'),(0,'b'),(0,'c'),(1,'a'),(1,'b'),(1,'c')}`:

```
x = xpress.vars([0,1],['a','b','c'])
```

The following creates a dictionary containing 6 variables whose indices vary in the set `{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)}`:

```
x = xpress.vars(2,3)
```

The code below creates a dictionary containing 5 integer variables with names `'y(a)', 'y(b)', 'y(c)', 'y(d)', 'y(e)'` and creates a constraint to bound their sum:

```
x = xpress.vars(['a','b','c','d','e'],
name='y', vartype=xpress.integer)
con1 = xpress.Sum(x) <= 4
```

The code below creates a dictionary whose keys range from 0 to 4:

```
x = xpress.vars(range(5),
name='y', vartype=xpress.integer)
con1 = xpress.Sum(x) <= 4
```

The following example creates an `xpress.ndarray` of dimensions `3,7,4` without assigning names to the variables:

```
x = xpress.vars(3,7,4, name="", lb=-1, ub=1)
```

Note that specifying anything other than a number yields a dictionary rather than an `xpress.ndarray`. Finally, the following creates a variable indexed by the set defined right before:

```
S = set()
S.add('john')
S.add('cleese')
x = xpress.vars(S, name='y', vartype=xpress.integer)
```

Further information

1. The name of each variable is created by concatenating its indices together. If the `name` argument is given as a non-empty string, this will be prepended to the name of each variable. If the `name` argument is given as an empty string, no names will be assigned to the variables. This option can be used to create large arrays of variables more quickly, since it will not be necessary to calculate a name for each variable.
2. All lists must contain non-repeated elements to avoid having variables with equal names. If a list in the argument is, for instance, `['a','b','a']`, an error is returned.
3. Unlinked variables are not tied to a problem but may exist globally in a Python program. In order for them to be included into a problem, they have to be explicitly added to that problem using `problem.addVariable`. Unlinked variables are deprecated.

Related topics

`problem.addVariables`, `problem.addVariable`, `xpress.var`.

Appendix A

Contacting FICO

FICO provides clients with support and services for all our products.

FICO Customer Support

FICO Customer Support offers technical support and services ranging from self-help tools to direct assistance with a FICO technical support engineer. Support is available to all clients who have an active maintenance contract.

The FICO Customer Self-Service Portal (support.fico.com) is a secure web portal that allows users to open, review, and update their support cases; manage their organization's portal users; find solutions to common problems in the FICO Knowledge Base; and view the availability of their cloud applications 24 hours a day, 7 days a week.

You can find support contact information and a link to the FICO Customer Self-Service Portal (online support) on the Product Support home page (www.fico.com/en/product-support).

Please include *'Xpress'* in the subject line of your support queries.

Documentation

FICO continually looks for new ways to improve and enhance the value of the products and services we provide.

If you have comments or suggestions regarding how we can improve this documentation, let us know by sending your suggestions to techpubs@fico.com. Please include your contact information (name, company, email address, and optionally, your phone number) so we may reach you if we have questions.

FICO Learning

FICO Learning is the principal provider of product training for our clients and partners. FICO Learning offers instructor-led classroom courses, web-based training, seminars, and training tools for both new user enablement and ongoing performance support.

For additional information, visit the FICO Learning home page at www.fico.com/en/product-training or email producteducation@fico.com.

Sales and maintenance

If you need information on other Xpress Optimization products, or you need to discuss maintenance contracts or other sales-related items, contact FICO by:

- Phone: +1 (408) 535-1500 or +44 207 940 8718
- Web: www.fico.com/optimization and use the available contact forms

About FICO

FICO (NYSE:FICO) is a leading analytics software company, helping businesses in 90+ countries make better decisions that drive higher levels of growth, profitability, and customer satisfaction. Learn more at www.fico.com or contact us at www.fico.com/en/contact-us.

Index

B

branchobj.addBounds, 518
branchobj.addbounds, 913
branchobj.addBranches, 514
branchobj.addbranches, 914
branchobj.addCuts, 522
branchobj.addcuts, 915
branchobj.addRows, 520
branchobj.addrows, 916
branchobj.getBounds, 519
branchobj.getbounds, 917
branchobj.getBranches, 515
branchobj.getbranches, 918
branchobj.getID, 523
branchobj.getid, 919
branchobj.getlasterror, 920
branchobj.getRows, 521
branchobj.getrows, 921
branchobj.setPreferredBranch, 517
branchobj.setpreferredbranch, 922
branchobj.setPriority, 516
branchobj.setpriority, 923
branchobj.store, 513
branchobj.validate, 524

C

constraint.getBody, 509
constraint.getDual, 511
constraint.getSlack, 510

O

object.extractLinear, 114
object.extractQuadratic, 115

P

problem.addAfterObjectiveCallback, 438
problem.addBarIterationCallback, 420
problem.addBarlogCallback, 399
problem.addBeforeObjectiveCallback, 436
problem.addcbafterobjective, 638
problem.addcbbariteration, 614
problem.addcbbarlog, 616
problem.addcbbeforeobjective, 637
problem.addcbchecktime, 617
problem.addcbchgbranchobject, 618
problem.addcbcutlog, 619
problem.addcbcutround, 620
problem.addcbdestroymt, 621
problem.addcbgapnotify, 622
problem.addcbinfnode, 625
problem.addcbintsol, 626
problem.addcblplog, 627
problem.addcbmessage, 628
problem.addcbmiplog, 624
problem.addcbmipthread, 629
problem.addcbnewnode, 630
problem.addcbnodecutoff, 631
problem.addcbnodelpsolved, 632
problem.addcboptnode, 633
problem.addcbpreintsol, 634
problem.addcbprenode, 635
problem.addcbpresolve, 639
problem.addcbusersolnotify, 636
problem.addChangeBranchObjectCallback, 426
problem.addCheckTimeCallback, 440
problem.addcoefs, 640
problem.addCols, 221
problem.addcols, 642
problem.addComputeRestartCallback, 434
problem.addConstraint, 474
problem.addCutlogCallback, 397
problem.addCutRoundCallback, 442
problem.addCuts, 297
problem.addcuts, 643
problem.adddfs, 644
problem.addGapNotifyCallback, 428
problem.addGenCons, 185
problem.addgencons, 645
problem.addIndicator, 475
problem.addInfnodeCallback, 405
problem.addIntsolCallback, 409
problem.addLplogCallback, 393
problem.addManagedCuts, 296
problem.addMessageCallback, 414
problem.addMiplogCallback, 395
problem.addMipSol, 327
problem.addmipsol, 646
problem.addMipThreadCallback, 416
problem.addMipThreadDestroyCallback, 418
problem.addMsJobEndCallback, 466
problem.addMsJobStartCallback, 464
problem.addMsWinnerCallback, 468
problem.addNames, 178
problem.addnames, 647
problem.addNewnodeCallback, 424
problem.addNlpCoefEvalErrorCallback, 470
problem.addNodecutoffCallback, 407
problem.addNodeLPSolvedCallback, 432
problem.addObj, 256
problem.addobj, 648
problem.addObjective, 476
problem.addOptnodeCallback, 401
problem.addPreIntsolCallback, 411
problem.addPrenodeCallback, 403

problem.addPresolveCallback, 422
problem.addPwlCons, 183
problem.addpwlcons, 649
problem.addQMatrix, 249
problem.addqmatrix, 650
problem.addRows, 219
problem.addrows, 651
problem.addsetnames, 652
problem.addSets, 227
problem.addSlpCascadeEndCallback, 444
problem.addSlpCascadeStartCallback, 446
problem.addSlpCascadeVarCallback, 448
problem.addSlpCascadeVarFailCallback, 450
problem.addSlpConstructCallback, 452
problem.addSlpDrColCallback, 462
problem.addSlpIntSolCallback, 454
problem.addSlpIterEndCallback, 456
problem.addSlpIterStartCallback, 458
problem.addSlpIterVarCallback, 460
problem.addSlpPreUpdateLinearizationCallback, 472
problem.addSOS, 477
problem.addtolsets, 653
problem.addUserSolNotifyCallback, 430
problem.addVariable, 478
problem.addVariables, 479
problem.addvars, 654
problem.basisStability, 245
problem.basisstability, 655
problem.bndSA, 247
problem.bndsa, 656
problem.BTran, 270
problem.btran, 657
problem.calcObjective, 313
problem.calcobjective, 659
problem.calcObjN, 314
problem.calcobjn, 658
problem.calcReducedCosts, 312
problem.calcreducedcosts, 660
problem.calcSlacks, 311
problem.calcslacks, 661
problem.calcSolInfo, 315
problem.calcsolinfo, 662
problem.cascade, 663
problem.cascadeorder, 664
problem.chgBounds, 324
problem.chgbounds, 665
problem.chgcascadenlimit, 668
problem.chgccoef, 692
problem.chgCoef, 337
problem.chgcoef, 666
problem.chgColType, 223
problem.chgcoltype, 667
problem.chgdeltatype, 693
problem.chgdf, 694
problem.chgGlbLimit, 234
problem.chgglblimit, 695
problem.chgMCoef, 338
problem.chgmcoef, 696
problem.chgMQObj, 339
problem.chgmqobj, 698
problem.chgnlcoef, 699
problem.chgObj, 336
problem.chgobj, 701
problem.chgObjN, 257
problem.chgobjn, 697
problem.chgObjSense, 233
problem.chgobjsense, 702
problem.chgQObj, 340
problem.chgqobj, 703
problem.chgQRowCoeff, 322
problem.chgqrowcoeff, 704
problem.chgRHS, 341
problem.chgrhs, 705
problem.chgRHSRange, 342
problem.chgrhsrange, 706
problem.chgrowstatus, 707
problem.chgRowType, 343
problem.chgrowtype, 708
problem.chgrowwt, 709
problem.chgtolset, 710
problem.chgvar, 711
problem.clearIIS, 207
problem.clearRowFlags, 284
problem.construct, 712
problem.copy, 481
problem.copycallbacks, 482
problem.copyControls, 259
problem.copycontrols, 713
problem.crossoverLpSol, 201
problem.crossoverlpsol, 714
problem.delcoefs, 715
problem.delCols, 222
problem.delConstraint, 483
problem.delCPCuts, 299
problem.delcpcuts, 716
problem.delCuts, 298
problem.delcuts, 717
problem.delGenCons, 188
problem.delgencons, 718
problem.delIndicators, 191
problem.delindicators, 719
problem.delObj, 258
problem.delobj, 721
problem.delPwlCons, 187
problem.delpwlcons, 720
problem.delQMatrix, 250
problem.delqmatrix, 722
problem.delRows, 220
problem.delSets, 226
problem.delSOS, 484
problem.deltolsets, 723
problem.delVariable, 485
problem.delvars, 724
problem.dumpControls, 189
problem.dumpcontrols, 725
problem.estimateRowDualRanges, 230
problem.estimaterowdualranges, 726
problem.evaluatecoef, 727

problem.evaluateformula, 728
problem.firstIIS, 208
problem.fixMIPEntities, 170
problem.fixmipentities, 729
problem.fixpenalties, 730
problem.FTran, 271
problem.ftran, 731
problem.getAttrib, 486
problem.getAttribInfo, 263
problem.getattribinfo, 732
problem.getBasis, 295
problem.getbasis, 733
problem.getbasisval, 734
problem.getCallbackDuals, 487
problem.getCallbackPresolveDuals, 334
problem.getCallbackPresolveRedCosts, 335
problem.getCallbackPresolveSlacks, 333
problem.getCallbackPresolveSolution, 332
problem.getCallbackRedCosts, 488
problem.getCallbackSlacks, 489
problem.getCallbackSolution, 490
problem.getccoef, 735
problem.getCoef, 285
problem.getcoef, 736
problem.getcoefformula, 737
problem.getcoefs, 738
problem.getcolinfo, 739
problem.getCols, 280
problem.getcols, 740
problem.getColType, 318
problem.getcoltype, 741
problem.getConstraint, 491
problem.getControl, 492
problem.getControlInfo, 262
problem.getcontrolinfo, 742
problem.getCPCutList, 301
problem.getcpcutlist, 743
problem.getCPCuts, 302
problem.getcpcuts, 744
problem.getCutList, 300
problem.getcutlist, 745
problem.getCutMap, 329
problem.getcutmap, 746
problem.getCutSlack, 328
problem.getcutslack, 747
problem.getdf, 749
problem.getDirs, 266
problem.getdirs, 748
problem.getDual, 750
problem.getDualRay, 293
problem.getdualray, 751
problem.getDuals, 493
problem.getGenCons, 186
problem.getgencons, 752
problem.getIISData, 215
problem.getiisdata, 754
problem.getIndex, 264
problem.getIndexFromName, 756
problem.getIndicators, 190
problem.getindicators, 757
problem.getInfeas, 198
problem.getinfeas, 758
problem.getLastBarSol, 206
problem.getlastbarsol, 759
problem.getlasterror, 760
problem.getLB, 278
problem.getlb, 761
problem.getLpSol, 331
problem.getlpsol, 762
problem.getlpsolval, 763
problem.getMessageStatus, 232
problem.getmessagestatus, 764
problem.getMIPEntities, 282
problem.getmipentities, 753
problem.getmipsol, 765
problem.getmipsolval, 766
problem.getMQObj, 286
problem.getmqobj, 767
problem.getNameList, 326
problem.getnamelist, 769
problem.getObj, 274
problem.getobj, 770
problem.getObjN, 275
problem.getobjn, 768
problem.getObjVal, 771
problem.getOutputEnabled, 503
problem.getPivotOrder, 268
problem.getpivotorder, 772
problem.getPivots, 309
problem.getpivots, 773
problem.getPresolveBasis, 317
problem.getpresolvebasis, 774
problem.getPresolveMap, 269
problem.getpresolvemap, 775
problem.getPresolveSol, 330
problem.getpresolvesol, 776
problem.getPrimalRay, 292
problem.getprimalray, 777
problem.getProbName, 261
problem.getProbStatus, 778
problem.getProbStatusString, 779
problem.getPwlCons, 184
problem.getpwlcons, 780
problem.getQObj, 265
problem.getqobj, 781
problem.getQRowCoeff, 319
problem.getqrowcoeff, 782
problem.getQRowQMatrix, 320
problem.getqrowqmatrix, 783
problem.getQRowQMatrixTriplets, 321
problem.getqrowqmatrixtriplets, 784
problem.getQRows, 323
problem.getqrows, 785
problem.getRCost, 786
problem.getRedCosts, 494
problem.getRHS, 276
problem.getrhs, 787
problem.getRHSRange, 277
problem.getrhsrange, 788
problem.getRowFlags, 283

problem.getrowinfo, 789
problem.getRows, 281
problem.getrows, 790
problem.getrowstatus, 791
problem.getRowType, 316
problem.getrowtype, 792
problem.getrowwt, 793
problem.getScale, 267
problem.getScaledInfeas, 199
problem.getscaledinfeas, 794
problem.getSlack, 795
problem.getSlacks, 495
problem.getslpsol, 796
problem.getSolution, 496
problem.getSOS, 497
problem.gettolset, 797
problem.getUB, 279
problem.getub, 798
problem.getUnbVec, 200
problem.getunbvec, 799
problem.getvar, 800
problem.getVariable, 498
problem.hasdualray, 801
problem.hasprimalray, 802
problem.IISAll, 212
problem.iisall, 803
problem.iisclear, 804
problem.iisfirst, 805
problem.IISIsolations, 214
problem.iisisolations, 806
problem.iisnext, 807
problem.IISStatus, 211
problem.iisstatus, 808
problem.iiswrite, 809
problem.interrupt, 260
problem.loadBasis, 224
problem.loadbasis, 810
problem.loadBranchDirs, 174
problem.loadbranchdirs, 811
problem.loadcoefs, 812
problem.loadCuts, 303
problem.loadcuts, 814
problem.loadDelayedRows, 172
problem.loaddelayedrows, 815
problem.loaddfs, 816
problem.loadDirs, 173
problem.loaddirs, 817
problem.loadLP, 164
problem.loadLPSol, 238
problem.loadlpsol, 818
problem.loadMIP, 176
problem.loadMipSol, 294
problem.loadmipsol, 819
problem.loadMIQCQP, 253
problem.loadMIQP, 168
problem.loadModelCuts, 171
problem.loadmodelcuts, 820
problem.loadPresolveBasis, 217
problem.loadpresolvebasis, 821
problem.loadPresolveDirs, 175
problem.loadpresolvedirs, 822
problem.loadproblem, 823
problem.loadQCQP, 251
problem.loadQP, 166
problem.loadSecureVecs, 218
problem.loadsecurevecs, 825
problem.loadtolsets, 826
problem.loadvars, 827
problem.lpOptimize, 192
problem.lpoptimize, 828
problem.mipOptimize, 193
problem.mipoptimize, 829
problem.msAddCustomPreset, 346
problem.msaddcustompreset, 830
problem.msAddJob, 344
problem.msaddjob, 831
problem.msAddPreset, 345
problem.msaddpreset, 832
problem.msClear, 353
problem.msclear, 833
problem.name, 834
problem.nextIIS, 209
problem.nlpAddFormulas, 356
problem.nlpCalcSlacks, 391
problem.nlpChgFormula, 358
problem.nlpchgformula, 835
problem.nlpChgFormulaStr, 357
problem.nlpchgformulastr, 836
problem.nlpCurrentIV, 350
problem.nlpDelFormulas, 362
problem.nlpEvaluateFormula, 354
problem.nlpGetFormula, 359
problem.nlpgetformula, 837
problem.nlpGetFormulaRows, 360
problem.nlpGetFormulaStr, 363
problem.nlpgetformulastr, 839
problem.nlpLoadFormulas, 361
problem.nlpOptimize, 349
problem.nlpoptimize, 840
problem.nlpPostsolveProb, 390
problem.nlpPrintEvalInfo, 347
problem.nlpSetInitVal, 364
problem.nlpsetinitval, 841
problem.nlpValidate, 348
problem.nlpValidateKKT, 352
problem.nlpValidateRow, 351
problem.nlpValidateVector, 355
problem.objSA, 246
problem.objsa, 842
problem.optimize, 194
problem.pivot, 237
problem.postSolve, 225
problem.postsolve, 843
problem.postSolveSol, 308
problem.presolve, 844
problem.presolveRow, 305
problem.presolverow, 845
problem.presolveSol, 307
problem.printevalinfo, 847
problem.printIIS, 210

problem.printmemory, 846
problem.read, 848
problem.readBasis, 196
problem.readbasis, 849
problem.readBinSol, 197
problem.readbinsol, 850
problem.readDirs, 180
problem.readdirs, 851
problem.readProb, 162
problem.readSlxSol, 195
problem.readslxsol, 852
problem.reinitialize, 853
problem.removeAfterObjectiveCallback, 439
problem.removeBarIterationCallback, 421
problem.removeBarlogCallback, 400
problem.removeBeforeObjectiveCallback, 437
problem.removecbafterobjective, 876
problem.removecbbariteration, 854
problem.removecbbarlog, 855
problem.removecbbeforeobjective, 875
problem.removecbchecktime, 856
problem.removecbchgbranchobject, 857
problem.removecbcutlog, 858
problem.removecbcutround, 859
problem.removecbdestroymt, 860
problem.removecbgapnotify, 861
problem.removecbinfnode, 863
problem.removecbintsol, 864
problem.removecblplog, 865
problem.removecbmessage, 866
problem.removecbmiplog, 862
problem.removecbmipthread, 867
problem.removecbnewnode, 868
problem.removecbnodecutoff, 869
problem.removecbnodelpsolved, 870
problem.removecboptnode, 871
problem.removecbpreintsol, 872
problem.removecbprenode, 873
problem.removecbpresolve, 877
problem.removecbusersolnotify, 874
problem.removeChangeBranchObjectCallback, 427
problem.removeCheckTimeCallback, 441
problem.removeComputeRestartCallback, 435
problem.removeCutlogCallback, 398
problem.removeCutRoundCallback, 443
problem.removeGapNotifyCallback, 429
problem.removeInfnodeCallback, 406
problem.removeIntsolCallback, 410
problem.removeLplogCallback, 394
problem.removeMessageCallback, 415
problem.removeMiplogCallback, 396
problem.removeMipThreadCallback, 417
problem.removeMipThreadDestroyCallback, 419
problem.removeMsJobEndCallback, 467
problem.removeMsJobStartCallback, 465
problem.removeMsWinnerCallback, 469
problem.removeNewnodeCallback, 425
problem.removeNlpCoefEvalErrorCallback, 471
problem.removeNodecutoffCallback, 408
problem.removeNodeLPSolvedCallback, 433
problem.removeOptnodeCallback, 402
problem.removePreIntsolCallback, 413
problem.removePrenodeCallback, 404
problem.removePresolveCallback, 423
problem.removeSlpCascadeEndCallback, 445
problem.removeSlpCascadeStartCallback, 447
problem.removeSlpCascadeVarCallback, 449
problem.removeSlpCascadeVarFailCallback, 451
problem.removeSlpConstructCallback, 453
problem.removeSlpDrColCallback, 463
problem.removeSlpIntSolCallback, 455
problem.removeSlpIterEndCallback, 457
problem.removeSlpIterStartCallback, 459
problem.removeSlpIterVarCallback, 461
problem.removeSlpPreUpdateLinearizationCallback, 473
problem.removeUserSolNotifyCallback, 431
problem.repairInfeas, 243
problem.repairinfeas, 878
problem.repairWeightedInfeas, 239
problem.repairweightedinfeas, 880
problem.repairWeightedInfeasBounds, 241
problem.repairweightedinfeasbounds, 882
problem.reset, 499
problem.restore, 236
problem.rhsSA, 248
problem.rhssa, 884
problem.save, 500
problem.saveAs, 235
problem.scale, 179
problem.scaling, 885
problem.setcbcascadeend, 669
problem.setcbcascadestart, 670
problem.setcbcascadevar, 671
problem.setcbcascadevarfail, 672
problem.setcbcoefevalerror, 673
problem.setcbconstruct, 674
problem.setcbdestroy, 675
problem.setcbdrcol, 676
problem.setcbintsol, 677
problem.setcbiterend, 678
problem.setcbiterstart, 679
problem.setcbitervar, 680
problem.setcbmessage, 681
problem.setcbmsjobend, 682
problem.setcbmsjobstart, 683
problem.setcbmswinner, 684
problem.setcboptnode, 685
problem.setcbprenode, 686
problem.setcbpreupdatelinearization, 687
problem.setcbslpend, 688
problem.setcbslpnode, 689

problem.setcbslpstart, 690
problem.setControl, 501
problem.setcurrentiv, 886
problem.setDefaultControl, 160
problem.setdefaultcontrol, 887
problem.setDefaults, 161
problem.setdefaults, 888
problem.setIndicators, 182
problem.setindicators, 889
problem.setLogFile, 159
problem.setlogfile, 890
problem.setMessageStatus, 231
problem.setmessagestatus, 891
problem.setObjective, 502
problem.setOutputEnabled, 504
problem.setProbName, 158
problem.setprobname, 892
problem.slpAddCoefs, 371
problem.slpCascadeOrder, 377
problem.slpCascadeSol, 376
problem.slpChgCascadeNLimit, 381
problem.slpChgCoef, 373
problem.slpchgcoef, 700
problem.slpChgCoefStr, 372
problem.slpchgcoefstr, 691
problem.slpChgDeltaType, 380
problem.slpChgRowStatus, 378
problem.slpChgRowWt, 379
problem.slpConstruct, 382
problem.slpDelCoefs, 368
problem.slpEvaluateCoef, 385
problem.slpFixPenalties, 389
problem.slpGetCoefFormula, 365
problem.slpgetcoefformula, 838
problem.slpGetCoefs, 366
problem.slpGetCoefStr, 369
problem.slpgetcoefstr, 893
problem.slpGetColInfo, 374
problem.slpGetRowInfo, 375
problem.slpGetRowStatus, 383
problem.slpGetRowWT, 384
problem.slpLoadCoefs, 367
problem.slpReInitialize, 386
problem.slpSetDetRow, 370
problem.slpSetInitStepBounds, 392
problem.slpUnConstruct, 387
problem.slpUpdateLinearization, 388
problem.sparseBTran, 272
problem.sparseFTran, 273
problem.storeCuts, 304
problem.storecuts, 894
problem.strongBranch, 228
problem.strongbranch, 895
problem.strongBranchCB, 229
problem.strongbranchcb, 896
problem.triggerRestart, 325
problem.tune, 202
problem.tuneProbSetFile, 203
problem.tuneprobsetfile, 897
problem.tunerReadMethod, 205
problem.tunerreadmethod, 898
problem.tunerWriteMethod, 204
problem.tunerwritemethod, 899
problem.unconstruct, 900
problem.updatelinearization, 901
problem.validate, 902
problem.validatekkt, 903
problem.validaterow, 904
problem.validatevector, 905
problem.write, 906
problem.writeBasis, 287
problem.writebasis, 907
problem.writeBinSol, 289
problem.writebinsol, 908
problem.writeDirs, 181
problem.writedirs, 909
problem.writeIIS, 213
problem.writeProb, 310
problem.writePrtSol, 290
problem.writeprtsol, 910
problem.writeSlxSol, 291
problem.writeslxsol, 911
problem.writeSol, 288
problem.writesol, 912

V

var.getRedCost, 507
var.getSolution, 506

X

xpress.abs, 117
xpress.acos, 118
xpress.addcbmsghandler, 924
xpress.AllowCompute, 583
xpress.AlwaysNeverAutomatic, 537
xpress.And, 119
xpress.array, 79
xpress.asin, 120
xpress.atan, 121
xpress.attr, 96
xpress.BacktrackAlg, 539
xpress.BARHGOps, 591
xpress.BarHGPrecision, 594
xpress.BarOrder, 534
xpress.BarPresolveOps, 562
xpress.BasisStatus, 592
xpress.BranchChoice, 540
xpress.branchobj, 97
xpress.CholeskyAlgorithm, 541
xpress.Clamping, 580
xpress.ComputeLog, 584
xpress.constants, 596
xpress.constraint, 98
xpress.cos, 122
xpress.CrossoverDynamicReduction, 542
xpress.ctrl, 100
xpress.CutSelect, 558
xpress.CutStrategy, 555
xpress.DefaultAlg, 535
xpress.Dot, 123

xpress.DualGradient, 543
xpress.DualizeOps, 560
xpress.DualStrategy, 544
xpress.eq, 92
xpress.erf, 125
xpress.erfc, 126
xpress.evaluate, 80
xpress.examples, 82
xpress.exp, 127
xpress.expression, 101
xpress.FeasibilityPump, 545
xpress.featureQuery, 75
xpress.featurequery, 925
xpress.free, 83
xpress.GenConsType, 579
xpress.geq, 94
xpress.getBanner, 71
xpress.getbanner, 926
xpress.getCheckedMode, 70
xpress.getcheckedmode, 928
xpress.getComputeAllowed, 78
xpress.getcomputeallowed, 927
xpress.getConstraintOperatorsEnabled, 89
xpress.getDaysLeft, 74
xpress.getdaysleft, 929
xpress.getlasterror, 930
xpress.getlicerrmsg, 931
xpress.getOutputEnabled, 87
xpress.getVersion, 72
xpress.getversion, 932
xpress.getVersionNumbers, 73
xpress.getversionnumbers, 933
xpress.Globalboundingbox, 586
xpress.GlobalLSHEURStrategy, 590
xpress.HeuristicSearchSelect, 546
xpress.IISOps, 588
xpress.IISSolStatus, 531
xpress.init, 84
xpress.is_equal, 934
xpress.isEqual, 91
xpress.leq, 93
xpress.linterm, 102
xpress.log, 128
xpress.log10, 129
xpress.LPFlags, 578
xpress.LPStatus, 529
xpress.max, 130
xpress.min, 131
xpress.MipPresolve, 552
xpress.MipRestart, 563
xpress.MIPStatus, 530
xpress.MIPStopStage, 595
xpress.MultiObjOps, 587
xpress.Namespaces, 585
xpress.ndarray, 110
xpress.NodeSelectionCriteria, 547
xpress.nonlin, 103
xpress.ObjControl, 582
xpress.ObjSense, 569
xpress.OnOff, 538
xpress.OptimizeType, 533
xpress.Or, 132
xpress.OutputDetail, 548
xpress.ParameterType, 570
xpress.poolcut, 104
xpress.PreProbing, 549
xpress.Presolve, 553
xpress.PresolveCoefElim, 564
xpress.PresolveDomColumn, 566
xpress.PresolveDomRow, 565
xpress.PresolveOperations, 550
xpress.PresolveState, 551
xpress.Pricing, 554
xpress.PrimalUnshift, 567
xpress.problem, 105
xpress.Prod, 134
xpress.pwl, 133
xpress.QConvexity, 571
xpress.quadterm, 107
xpress.RefineOps, 559
xpress.removecbmsghandler, 935
xpress.RepairIndefiniteQuadratic, 568
xpress.RowFlag, 581
xpress.Scaling, 557
xpress.SDPCutStrategy, 593
xpress.setArchConsistency, 76
xpress.setarchconsistency, 936
xpress.setCheckedMode, 69
xpress.setcheckedmode, 938
xpress.setComputeAllowed, 77
xpress.setcomputeallowed, 937
xpress.setConstraintOperatorsEnabled, 90
xpress.setDefaultControl, 86
xpress.setdefaultcontrol, 939
xpress.setDefaults, 85
xpress.setdefaults, 940
xpress.setOutputEnabled, 88
xpress.sign, 135
xpress.sin, 136
xpress.SolAvailable, 532
xpress.SolInfo, 572
xpress.SolStatus, 527
xpress.SolveStatus, 528
xpress.sos, 108
xpress.sqrt, 137
xpress.StopType, 536
xpress.Sum, 138
xpress.tan, 139
xpress.TreeDiagnostics, 561
xpress.TunerHistory, 576
xpress.TunerMethod, 574
xpress.TunerMode, 573
xpress.TunerRootAlg, 577
xpress.TunerTarget, 575
xpress.user, 140
xpress.UserSolStatus, 589
xpress.var, 109
xpress.VariableSelection, 556
xpress.vars, 941
xpress.voidstar, 111

xpress.xprsobject, 112