FICO

FICO® Xpress Optimization

Xpress Optimizer Python interface

User's manual

Last update 3 June 2017

Xpress Optimizer

Deliverable Version: A

Last Revised: 3 June 2017

Contents

1 Introduction **1**
- 1.1 Outline . . . 1
- 1.2 Installing the Python Xpress module . . . 1
  - 1.2.1 Installation from the Xpress Optimizer distribution . . . 1
  - 1.2.2 Installation from Conda . . . 2

2 Modeling an optimization problem **3**
- 2.1 Getting started . . . 3
- 2.2 Creating a problem . . . 3
- 2.3 Variables . . . 3
- 2.4 Constraints . . . 4
- 2.5 Objective function . . . 5
- 2.6 Special Ordered Sets (SOSs) . . . 6
- 2.7 Indicator constraints . . . 6
- 2.8 Modeling and solving nonlinear problems . . . 6
- 2.9 Solving a problem . . . 8
- 2.10 Querying a problem . . . 8
- 2.11 Reading and writing a problem . . . 10
- 2.12 Hints for building models efficiently . . . 10

3 Using Python numerical libraries **12**
- 3.1 Using `NumPy` in the Xpress Python interface . . . 12
- 3.2 Products of `NumPy` arrays . . . 13

4 Controls and Attributes **14**
- 4.1 Controls . . . 14
- 4.2 Examples . . . 15
- 4.3 Attributes . . . 15
- 4.4 Examples . . . 16
- 4.5 Accessing controls and attributes as object members . . . 16

5 Using Callbacks **19**
- 5.1 Introduction . . . 19

6 Examples of use **21**
- 6.1 Creating simple problems . . . 21
  - 6.1.1 Generating a small Linear Programming problem . . . 21
  - 6.1.2 A Mixed Integer Linear Programming problem . . . 22
- 6.2 Modeling examples . . . 23
  - 6.2.1 A simple model . . . 23
  - 6.2.2 Using IIS to investigate an infeasible problem . . . 23
  - 6.2.3 Modeling a problem using Python lists and vectors . . . 24
  - 6.2.4 A knapsack problem . . . 24
  - 6.2.5 A Min-cost-flow problem using NumPy . . . 25
  - 6.2.6 A nonlinear model . . . 26
  - 6.2.7 Finding the maximum-area *n*-gon . . . 26

6.2.8 Solving the *n*-queens problem . . . 27
6.2.9 Solving Sudoku problems . . . 27
6.3 Examples using NumPy . . . 28
6.3.1 Using NumPy multidimensional arrays to create variables . . . 28
6.3.2 Using the dot product to create arrays of expressions . . . 29
6.3.3 Using the Dot product to create constraints and quadratic functions . . . 29
6.3.4 Using NumPy to create quadratic optimization problems . . . 30
6.4 Advanced examples: callbacks and problem querying/modifying . . . 30
6.4.1 Visualize the branch-and-bound tree of a problem . . . 30
6.4.2 Query and modify a simple problem . . . 32
6.4.3 Change a problem after solution . . . 32
6.4.4 Combining modeling and API functions . . . 34
6.4.5 A simple Traveling Salesman Problem (TSP) solver . . . 34

7 Reference Manual **38**
7.1 Using this chapter . . . 38
Format of the reference . . . 39
7.2 Global methods of the Xpress module . . . 39
7.3 Methods of the problem class . . . 40
7.4 Methods for branching objects . . . 42
7.5 Methods for adding/removing callbacks of a problem object . . . 42
7.6 Methods to be used within a callback of a problem object . . . 42
xpress.free . . . 44
xpress.getbanner . . . 45
xpress.getcheckedmode . . . 46
xpress.getdaysleft . . . 47
xpress.getlasterror . . . 48
xpress.getlicerrmsg . . . 49
xpress.getversion . . . 50
xpress.init . . . 51
xpress.setcheckedmode . . . 52
xpress.setdefaults . . . 53
xpress.setdefaultcontrol . . . 54
xpress.Sum . . . 55
xpress.Dot . . . 56
xpress.Prod . . . 58
xpress.exp . . . 59
xpress.log . . . 60
xpress.log10 . . . 61
xpress.sin . . . 62
xpress.cos . . . 63
xpress.tan . . . 64
xpress.asin . . . 65
xpress.acos . . . 66
xpress.atan . . . 67
xpress.max . . . 68
xpress.min . . . 69
xpress.abs . . . 70
xpress.sign . . . 71
xpress.erf . . . 72
xpress.erfc . . . 73
xpress.sqrt . . . 74
xpress.user . . . 75
xpress.addcbmsghandler . . . 76
xpress.removecbmsghandler . . . 77

problem.addcbbariteration . . . 78
problem.addcbbarlog . . . 80
problem.addcbchgbranchobject . . . 81
problem.addcbcutlog . . . 82
problem.addcbdestroymt . . . 83
problem.addcbgapnotify . . . 84
problem.addcbgloballog . . . 86
problem.addcbinfnode . . . 87
problem.addcbintsol . . . 88
problem.addcblplog . . . 89
problem.addcbmessage . . . 90
problem.addcbmipthread . . . 91
problem.addcbnewnode . . . 92
problem.addcbnodecutoff . . . 93
problem.addcboptnode . . . 94
problem.addcbpreintsol . . . 95
problem.addcbprenode . . . 96
problem.addcbusersolnotify . . . 97
problem.addcoefs . . . 98
problem.addcols . . . 100
problem.addConstraint . . . 101
problem.addcuts . . . 102
problem.adddfs . . . 103
problem.addIndicator . . . 104
problem.addmipsol . . . 105
problem.addqmatrix . . . 106
problem.addrows . . . 107
problem.addSOS . . . 108
problem.addtolsets . . . 109
problem.addVariable . . . 110
problem.addvars . . . 111
problem.basisstability . . . 112
problem.btran . . . 113
problem.calcobjective . . . 114
problem.calcreducedcosts . . . 115
problem.calcslacks . . . 116
problem.calcsolinfo . . . 117
problem.cascade . . . 118
problem.cascadeorder . . . 119
problem.chgbounds . . . 120
problem.chgcoef . . . 121
problem.chgcoltype . . . 122
problem.chgcascadenlimit . . . 123
problem.chgccoef . . . 124
problem.chgdeltatype . . . 125
problem.chgdf . . . 126
problem.chgglblimit . . . 127
problem.chgmcoef . . . 128
problem.chgmqobj . . . 129
problem.chgnlcoef . . . 130
problem.chgobj . . . 131
problem.chgobjsense . . . 132
problem.chgqobj . . . 133
problem.chgqrowcoeff . . . 134
problem.chgrhs . . . 135

problem.chgrhsrange . . . 136
problem.chgrowstatus . . . 137
problem.chgrowtype . . . 138
problem.chgrowwt . . . 139
problem.chgtolset . . . 140
problem.chgvar . . . 142
problem.construct . . . 144
problem.copy . . . 145
problem.copycallbacks . . . 146
problem.copycontrols . . . 147
problem.delcoefs . . . 148
problem.delConstraint . . . 149
problem.delcpcuts . . . 150
problem.delcuts . . . 151
problem.delqmatrix . . . 152
problem.delSOS . . . 153
problem.deltolsets . . . 154
problem.delVariable . . . 155
problem.delvars . . . 156
problem.dumpcontrols . . . 157
problem.estimaterowdualranges . . . 158
problem.evaluatecoef . . . 159
problem.evaluateformula . . . 160
problem.filesol . . . 161
problem.fixglobals . . . 162
problem.fixpenalties . . . 163
problem.ftran . . . 164
problem.getAttrib . . . 165
problem.getattribinfo . . . 166
problem.getbasis . . . 167
problem.getccoef . . . 168
problem.getcoef . . . 169
problem.getcoefformula . . . 170
problem.getcoefs . . . 171
problem.getcolinfo . . . 172
problem.getcols . . . 173
problem.getcoltype . . . 174
problem.getConstraint . . . 175
problem.getControl . . . 176
problem.getcontrolinfo . . . 177
problem.getcpcutlist . . . 178
problem.getcpcuts . . . 179
problem.getcutlist . . . 180
problem.getcutmap . . . 181
problem.getcutslack . . . 182
problem.getdirs . . . 183
problem.getdf . . . 184
problem.getdtime . . . 185
problem.getDual . . . 186
problem.getdualray . . . 187
problem.getglobal . . . 188
problem.getiisdata . . . 189
problem.getIndex . . . 191
problem.getIndexFromName . . . 192
problem.getindicators . . . 193

problem.getinfeas . . . 194
problem.getlasterror . . . 195
problem.getlb . . . 196
problem.getlpsol . . . 197
problem.getmessagestatus . . . 198
problem.getmessagetype . . . 199
problem.getmipsol . . . 200
problem.getmqobj . . . 201
problem.getobj . . . 202
problem.getObjVal . . . 203
problem.getpivotorder . . . 204
problem.getpivots . . . 205
problem.getpresolvebasis . . . 206
problem.getpresolvemap . . . 207
problem.getpresolvesol . . . 208
problem.getprimalray . . . 209
problem.getProbStatus . . . 210
problem.getProbStatusString . . . 211
problem.getqobj . . . 212
problem.getqrowcoeff . . . 213
problem.getqrowqmatrix . . . 214
problem.getqrowqmatrixtriplets . . . 215
problem.getqrows . . . 216
problem.getRCost . . . 217
problem.getrhs . . . 218
problem.getrhsrange . . . 219
problem.getrowinfo . . . 220
problem.getrows . . . 221
problem.getrowstatus . . . 222
problem.getrowtype . . . 223
problem.getrowwt . . . 224
problem.getscaledinfeas . . . 225
problem.getSlack . . . 226
problem.getslpsol . . . 227
problem.getSolution . . . 228
problem.getSOS . . . 229
problem.gettolset . . . 230
problem.getub . . . 231
problem.getunbvec . . . 232
problem.getvar . . . 233
problem.getVariable . . . 235
problem.globalsol . . . 236
problem.hasdualray . . . 237
problem.hasprimalray . . . 238
problem.iisall . . . 239
problem.iisclear . . . 240
problem.iisfirst . . . 241
problem.iisisolations . . . 242
problem.iisnext . . . 243
problem.iisstatus . . . 244
problem.iiswrite . . . 245
problem.interrupt . . . 246
problem.loadbasis . . . 247
problem.loadbranchdirs . . . 248
problem.loadcoefs . . . 249

problem.loadcuts . . . 251
problem.loaddelayedrows . . . 252
problem.loaddfs . . . 253
problem.loaddirs . . . 254
problem.loadlpsol . . . 255
problem.loadmipsol . . . 256
problem.loadmodelcuts . . . 257
problem.loadpresolvebasis . . . 258
problem.loadpresolvedirs . . . 259
problem.loadproblem . . . 260
problem.loadsecurevecs . . . 262
problem.loadtolsets . . . 263
problem.loadvars . . . 264
problem.lpoptimize . . . 266
problem.mipoptimize . . . 267
problem.msaddcustompreset . . . 268
problem.msaddjob . . . 269
problem.msaddpreset . . . 270
problem.msclear . . . 271
problem.name . . . 272
problem.objsa . . . 273
problem.parsecformula . . . 274
problem.parseformula . . . 275
problem.postsolve . . . 276
problem.preparseformula . . . 277
problem.presolve . . . 278
problem.presolverow . . . 279
problem.printmemory . . . 280
problem.printevalinfo . . . 281
problem.printmsg . . . 282
problem.read . . . 283
problem.readbasis . . . 284
problem.readbinsol . . . 285
problem.readdirs . . . 286
problem.readslxsol . . . 287
problem.refinemipsol . . . 288
problem.reinitialize . . . 289
problem.removecbbariteration . . . 290
problem.removecbbarlog . . . 291
problem.removecbchgbranchobject . . . 292
problem.removecbcutlog . . . 293
problem.removecbdestroymt . . . 294
problem.removecbgapnotify . . . 295
problem.removecbgloballog . . . 296
problem.removecbinfnode . . . 297
problem.removecbintsol . . . 298
problem.removecblplog . . . 299
problem.removecbmessage . . . 300
problem.removecbmipthread . . . 301
problem.removecbnewnode . . . 302
problem.removecbnodecutoff . . . 303
problem.removecboptnode . . . 304
problem.removecbpreintsol . . . 305
problem.removecbprenode . . . 306
problem.removecbusersolnotify . . . 307

problem.repairinfeas . . . 308
problem.repairweightedinfeas . . . 310
problem.repairweightedinfeasbounds . . . 312
problem.reset . . . 314
problem.restore . . . 315
problem.rhssa . . . 316
problem.save . . . 317
problem.scale . . . 318
problem.scaling . . . 319
problem.setbranchbounds . . . 320
problem.setbranchcuts . . . 321
problem.setcbcascadeend . . . 322
problem.setcbcascadestart . . . 323
problem.setcbcascadevar . . . 324
problem.setcbcascadevarfail . . . 325
problem.setcbcoefevalerror . . . 326
problem.setcbconstruct . . . 327
problem.setcbdestroy . . . 329
problem.setcbdrcol . . . 330
problem.setcbformula . . . 331
problem.setcbiterend . . . 332
problem.setcbiterstart . . . 333
problem.setcbitervar . . . 334
problem.setcbmessage . . . 335
problem.setcbmsjobend . . . 336
problem.setcbmsjobstart . . . 337
problem.setcbmswinner . . . 338
problem.setcbslpend . . . 339
problem.setcbslpnode . . . 340
problem.setcbslpstart . . . 341
problem.setControl . . . 342
problem.setcurrentiv . . . 343
problem.setdefaultcontrol . . . 344
problem.setdefaults . . . 345
problem.setindicators . . . 346
problem.setlogfile . . . 347
problem.setmessagestatus . . . 348
problem.setObjective . . . 349
problem.setprobname . . . 350
problem.setuniqueprefix . . . 351
problem.solve . . . 352
problem.storebounds . . . 353
problem.storecuts . . . 354
problem.strongbranch . . . 355
problem.strongbranchcb . . . 356
problem.tokencount . . . 357
problem.tune . . . 358
problem.tunerreadmethod . . . 359
problem.tunerwritemethod . . . 360
problem.unconstruct . . . 361
problem.updatelinearization . . . 362
problem.validate . . . 363
problem.validatekkt . . . 364
problem.validaterow . . . 365
problem.validatevector . . . 366

problem.validformula . . . 367
problem.write . . . 368
problem.writebasis . . . 369
problem.writebinsol . . . 370
problem.writedirs . . . 371
problem.writeprtsol . . . 372
problem.writeslxsol . . . 373
problem.writesol . . . 374
branchobj.addbounds . . . 375
branchobj.addbranches . . . 376
branchobj.addcuts . . . 377
branchobj.addrows . . . 378
branchobj.getbounds . . . 379
branchobj.getbranches . . . 380
branchobj.getid . . . 381
branchobj.getlasterror . . . 382
branchobj.getrows . . . 383
branchobj.setpreferredbranch . . . 384
branchobj.setpriority . . . 385
branchobj.store . . . 386
branchobj.validate . . . 387

Appendix **388**

A Contacting FICO **388**
Product support . . . 388
Product education . . . 388
Product documentation . . . 388
Sales and maintenance . . . 389
Related services . . . 389
About FICO . . . 389

Index **390**

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 when creating optimization problems (Chapter 3);
- Setting and getting the value of controls, and getting the value of 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 7).

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 2.7, Python 3.4, and subsequent versions.

1.2 Installing the Python Xpress module

1.2.1 Installation from the Xpress Optimizer distribution

Using the Xpress Optimizer suite, the module is automatically installed by the installation script. If *xpress_dir* is the directory where the Xpress Optimizer suite was installed (for example `c:\xpressmp` for Windows users or `/opt/xpressmp` for *nix users), then the Python module is at *xpress_dir*`/lib` for *nix users and *xpress_dir*`\bin` for Windows users.

Documentation is available under *xpress_dir*`/docs/python`, both in PDF form (see *xpress_dir*`/docs/python/xpress_python_interface.pdf`) and as a set of HTML pages (see *xpress_dir*`/docs/python/dhtml/index.html`).

1.2.2 Installation from Conda

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

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

Conda packages for Python 2.7, 3.4, 3.5, and 3.6 are available, for Windows, Linux, and MacOS. The Conda package contains the Python interface module and its documentation, but not the Xpress Optimizer's libraries, its documentation, or a license. Please refer to the FICO Xpress Optimizer web pages for instructions on how to obtain a full installation of the Xpress optimization suite, which also contains an installation package that is compatible with Conda.

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 `myproblem` 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.

2.3 Variables

The Xpress type `var` allows for creating optimization variables. Note that 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. Below is the complete declaration with the list of all parameters (all of them are optional):

```
var (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 = xpress.var()`, its name, lower and upper bound can be accessed via `x.name`, `x.lb`, and `x.ub`. Note that, after a variable `x` has been added to one or more problems, a change in its feature will not be reflected in these problems, but only in the problems to which this variable is added subsequently.

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)
```

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.range; 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 constraint can be specified more naturally as a condition on an expression:

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

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

```
m.addConstraint (myconstr)
m.addConstraint (v1 + xp.tan (v2) <= 3)
m.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:

```
m.addConstraint (xp.Sum ([y[i]    for i in range (10)]) <= 1)
m.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.

2.5 Objective function

The objective function is any expression, so it can be constructed as for constraints. The method 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)
```

where objective is the expression defining the new objective and sense is either xpress.minimize or xpress.maximize. Examples follows (in the first, the objective function is to be minimized as per default, while the second example specifies the optimization sense as maximization).

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

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 vector to be nonzero. The Xpress Python interface allows for defining a SOS as follows:

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

One or more SOS can be added to a problem via the `addSOS` method:

```
set1 = xp.sos (x, [0.5 + i*0.1 for i in range(10)], type = 2)
m.addSOS (set1)
n = 10
w = [xp.var () for i in range (n)]
m.addSOS ([xpr.sos ([w[i],w[i+1]], [2,3], type = 2) for i in range (n-1)])
```

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 $x + y \geq 3$ should only be enforced if the binary variable u is equal to 1, then ($u = 1 \rightarrow x + y \geq 3$) 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):

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

2.8 Modeling and solving nonlinear problems

Version 8.3 of the Xpress Optimizer suite introduces nonlinear modeling in the Python interface. It allows for creating and solving nonlinear, possibly nonconvex problems with similar functions as for linear, quadratic, and conic problems and their mixed integer counterpart.

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

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

p.addVariable (x)

# 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)

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

x,y = xp.var (), xp.var ()

p = xp.problem ()

p.addVariables (x,y)

p.setObjective (user (mynorm, x, y))

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

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:

objtransrow : $-objtranscol + f(x) = 0$

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.

The reader can find more information on this in the Xpress Nonlinear reference manual.

2.9 Solving a problem

Simply call `solve()` 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. Note that in case of a nonconvex quadratic problem, the Xpress Nonlinear solver will be applied as the Xpress Optimizer solver cannot handle such problems.

```
m.solve ()
```

The status of a problem after solution can be inquired via the functions `getProbStatus()` and `getProbStatusString()` as follows:

```
import xpress as xp

m = xp.problem ()
m.read ("example3.lp")
m.solve ()

print ("problem status:            ", m.getProbStatus ())
print ("problem status, explained: ", m.getProbStatusString ())
```

The meaning of the returned value is explained in the Optimizer's reference manual. Note that the value and string returned by the above functions reflect the type of problem as input by the user, *not* the way the problem was last solved. If the user creates a MILP and then solves it as an LP with the flag `"l"`, then both `getProbStatus()` and `getProbStatusString()` yield the status of the MILP rather than the LP relaxation. At all effects, the call `p.getProbStatus()` returns `p.attributes.lpstatus` if `p` has continuous variables and `p.attributes.mipstatus` if `p` has integer variables.

2.10 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 `getSolution` and `getSlack` 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

v1 = xp.var ()
x  = [xp.var (lb = -1, ub = 1, vartype = xp.integer) for i in range(10)]

m  = xp.problem ()

m.addVariable (v1, x)

[...] # add constraints and objective
```

```
m.solve()

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]
```

Consider the last four commands. The first of them returns a 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 vector returned by `getSlack`) 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`. Finally, the fourth example shows that a range of indices can be specified in order to obtain a vector of values without specifying the corresponding variables. Recall that the column and row indices begin at 0.

The method `getSlack` works similarly, except constraints or integer indices must be passed. The following examples illustrate a few possible uses.

```
import xpress as xp

N = 10

x  = [xp.var (vartype = xp.binary) for i in range(N)]

m  = xp.problem ()

m.addVariable (x)

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

m.addConstraints (con1, con2)
m.setObjective (xp.Sum (x[i] for i in range (N))
m.solve ()

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

a = m.getSlack (con2)               # gets the slack of N-1 constraints con2
b = m.getSlack (0:2)                # gets the slack of con1 and con2[0]
```

In addition, for problems with only continuous variables, the two methods `getDual` and `getRCost` return the the vector (or a portion thereof) of dual variables and reduced costs, respectively. Their usage is similar to that for `getSolution` and `getSlack`.

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 `getSolution`, `getSlack`, 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)

x = [xp.var () for i in N]

p = xp.problem ()

p.addVariable (x)
m.addConstraints (xp.Sum (x[i] * i for i in N) <= n))
m.setObjective (xp.Sum (x[i] for i in N)
m.solve ()
```

```
for i in N:
    if m.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) vector:

```
sol = m.getSolution ()

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

2.11 Reading and writing a problem

After creating an empty problem, method, 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

m = xp.problem ()
m.read ("example2.lp")
m.solve ()

print (m.getSolution ())

m2 = xp.problem ()
v1 = xp.var ()
v2 = xp.var (vartype = xp.integer)

m2.addVariable (v1, v2)
m2.addConstraint (v1 + v2 <= 4)
m2.setObjective (v1**2 + v2)

m2.write ("twovarsproblem", "lp")
```

2.12 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 these methods, to ensure faster execution.

Let us consider an example:

```
import xpress as xp

N = 100000
S = range(N)

x = [xp.var()                    for i in S]
y = [xp.var(vartype = xp.binary) for i in S]

for i in S:
  m.addVariable (x[i])
```

```
  m.addVariable (y[i])

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

m.solve()
```

While the declaration of `x` and `y` is correct and efficient, the two subsequent loops are very inefficient: they imply `2N` calls to `addVariable` and `N` calls to `addConstraint`. Both methods add some overhead due to the conversion of Python object 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 first loop can be replaced by two calls to `addVariable`:

```
m.addVariable (x)
m.addVariable (y)
```

or, more compact and slightly more efficient:

```
m.addVariable (x,y)
```

The largest gain in performance, though, comes from replacing the second loop with a single call to `addConstraint`:

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

This line is equivalent to the second loop above, and it is much faster and more elegant.

When declaring `x` and `y` as NumPy vectors, an equally efficient and even more compact model can be written:

```
import xpress as xp
import numpy as np

N = 100000
S = range(N)

x = np.array ([xp.var ()                    for i in S])
y = np.array ([xp.var (vartype = xp.binary) for i in S])

m.addVariable (x,y)
m.addConstraint (x <= y)

m.solve()
```

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

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 works with NumPy versions 1.10 and above.

3.1 Using NumPy in 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)
x = np.array ([xp.var ()                    for i in S])
y = np.array ([xp.var (vartype = xp.binary) for i in S])
constr1 = x <= y
p = xp.problem ()
p.addVariable (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 $x_i \leq y_i$ for all i in the set S.

As happens for all NumPy operations, all operations are replicated 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 $x_i + y_i \leq 1$ 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 = np.array ([xp.var ()                    for i in range (200)]).reshape (4,5,10)
t = np.array ([xp.var (vartype = xp.binary) for i in range (200)]).reshape (4,5,10)
p.addVariable (z,t)
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))
```

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 $(i_1, i_2, \ldots, i_{k'})$ and $(j_1, j_2, \ldots, j_{k''})$, 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)
x     = np.array ([xp.var () for i in range(140)]).reshape (4,7,5)
y     = np.array ([xp.var () for i in range(80)]).reshape (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 defining the output 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 $(t - z)'Q(t - z) \leq 1$ 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.

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 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.

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.

4.1 Controls

Use the method `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 Xpress Optimizer's reference manual. The control parameters to be passed in `setControl` are lower-case strings:

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

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.

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

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

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 controls. 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 Xpress Optimizer's reference manual. As for controls, the attribute parameters to be passed in `getAttrib` are lower-case strings. For a problem `p` the call will be as follows:

```
nrows       = p.getAttrib ('nrow')
problemsize = p.getAttrib ('nrow', 'ncol')
```

4.4 Examples

```
import xpress as xp

p = xp.problem ()

p.read ("example.lp")

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

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

print (p.getAttrib (['nrow', 'ncol']))

# 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.heurstrategy)
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.

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.read ("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.read ("example2.lp")

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

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

p1.solve ()
p2.solve ()
p2v.solve () # solve "example2.lp" with a less restrictive
             # feasibility tolerance
```

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.read ("example.lp")
print ("The problem has ",
    p.attributes.nrow, "rows and ",
    p.attributes.ncol, "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.read ("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
```

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.

5.1 Introduction

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, both 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 function 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 `addcbpreintsol`) with `myfunction` as its argument. Finally, the user runs the `solve` command 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()`); and
- 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.read ('myprob.lp') # reads in a problem, let's say a MIP

baz = foo ()

p.addcbpreintsol (preintsolcb, baz, 3)
p.solve ()
```

While the function argument is necessary for all `addcb*` 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 `addcb*` 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 `addcbpreintsol` 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 `addcb*`, a corresponding `removecb*` function is provided, for instance `removecbpreintsol`. 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 `addcb` function that match the function **and** the data.

The `None` keyword acts as a wildcard that matches any function/data object: if `removecb*` 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 `removecb*`.

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.

Most callback functions refer to a problem, therefore the `addcb*` method is called from a `problem` object. The only exception is the function `xpress.addcbmsghandler()`, which is called on the Xpress module itself and allows for providing a function that is called every time *any* output is produced within the Optimizer.

We refer to the Reference chapter of this manual for all information regarding callback functions and how to add/remove them from a problem.

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. The last line is to make the print statements, which are in Python 3 style here, work in Python 2.7 as well.

```
import xpress as xp
import numpy  as np
from __future__ import print_function
```

6.1 Creating simple problems

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.

6.1.1 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 vector with the `loadproblem` 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:	$3\,x_1 + 4\,x_2 + 5\,x_3$		
subject to:	$x_1 + x_3$	$\geq$	-2.4
	$2x_1 + 3x_3$	$\geq$	-3
	$2x_2 + 3x_3$	=	4
	$x_2 + x_3$	$\leq$	5
	$-1 \leq x_1 \leq 3$		
	$-1 \leq x_1 \leq 5$		
	$-1 \leq x_1 \leq 8$		

```
p = xp.problem ()

p.loadproblem ("",                # probname
               ['G','G','E', 'L'], # qrtypes
               [-2.4, -3, 4, 5],   # rhs
```

```
                None,               # range
                [3,4,5],            # obj
                [0,2,4,8],          # mstart
                None,               # mnel
                [0,1,2,3,0,1,2,3],  # mrwind
                [1,2,2,1,1,3,3,1],  # dmatval
                [-1,-1,-1],         # lb
                [3,5,8],            # ub
                colnames = ['X1','X2','X3'],                  # column names
                rownames = ['row1','row2','row3','constr_04']) # row    names

p.write ("loadlp", "lp")
p.solve ()
```

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 = xp.var()
p.addVariable (x)
p.setObjective (x**2 + 2*x + 444)
p.solve()
p.write ("updated", "lp")
```

6.1.2 A Mixed Integer Linear Programming problem

This example uses `loadproblem` 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.loadproblem ("",                # probname
               ['G','G','L', 'L'], # qrtypes
               [-2.4, -3, 4, 5],   # rhs
               None,               # range
               [3,4,5],            # obj
               [0,2,4,8],          # mstart
               None,               # mnel
               [0,1,2,3,0,1,2,3],  # mrwind
               [1,1,1,1,1,1,1,1],  # dmatval
               [-1,-1,-1],         # lb
               [3,5,8],            # ub
               [0,0,0,1,1,2],      # mqobj1
               [0,1,2,1,2,2],      # mqobj1
               [2,1,1,2,1,2],      # dqe
               [2,3],              # qcrows
               [2,3],              # qcnquads
               [1,2,0,0,2],        # qcmqcol1
               [1,2,0,2,2],        # qcmqcol2
               [3,4,1,1,1],        # qcdqval
               ['I','S'],          # qgtype
               [0,1],              # mgcols
               [0,2],              # dlim
               ['1','1'],          # qstype
               [0,2,4],            # msstart
               [0,1,0,2],          # mscols
               [1.1,1.2,1.3,1.4])  # dref

p.solve ()
```

6.2 Modeling examples

6.2.1 A simple model

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

```
N = 4
S = range (N)
v = [xp.var (name = "y{0}".format (i)) for i in S] # set name of a variable as

m = xp.problem ()

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

# Create a variable with name yi, where i is an index in S
v = [xp.var (name = "y{0}".format (i), lb = 0, ub = 2*N) for i in S]
```

The call below adds both v, a vector (list) of variables, and v1 and v2, two scalar variables.

```
m.addVariable (vb, v, v1, v2)

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.minimize)

m.solve ()

print ("status: ", m.getProbStatus ())
print ("string: ", m.getProbStatusString ())

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

6.2.2 Using IIS to investigate an infeasible problem

The problem modeled below is infeasible,

```
x0 = xp.var()
x1 = xp.var()
x2 = xp.var(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 = xp.problem ("ex-infeas")

minf.addVariable   (x0,x1,x2)
minf.addConstraint (c1,c2,c3,c4)

minf.solve()
minf.iisall()
print ("there are ", minf.attributes.numiis, " iis's")

miisrow = []
miiscol = []
constrainttype = []
colbndtype = []
duals = []
rdcs = []
isolationrows = []
isolationcols = []

# get data for the first IIS

minf.getiisdata (1, miisrow, miiscol, constrainttype, colbndtype,
                 duals, rdcs, isolationrows, isolationcols)

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

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

rowsizes  = []
colsizes  = []
suminfeas = []
numinfeas = []

print ("iisstatus:", minf.iisstatus (rowsizes, colsizes, suminfeas, numinfeas))
print ("vectors:", rowsizes, colsizes, suminfeas, numinfeas)
```

6.2.3 Modeling a problem using Python lists and vectors

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

```
N = 5
S = range (N)

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

m = xp.problem ("problem 1")

print ("variable:", v)

m.addVariable (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.solve ()

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

6.2.4 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.

```
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]

x = [xp.var (vartype = xp.binary) for i in S]
profit = xp.Sum (value[i] * x[i] for i in S)

p = xp.problem ("knapsack")
p.addVariable (x)
p.addConstraint (xp.Sum (weight[i] * x[i] for i in S) <= 130)
p.setObjective (profit, sense = xp.maximize)
p.solve ()
```

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

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

x = np.array ([xp.var (vartype = xp.binary) for i in S])
profit = xp.Dot (value, x)

p = xp.problem ("knapsack")
p.addVariable (x)
p.addConstraint (xp.Dot (weight, x) <= 130)
p.setObjective (profit, sense = xp.maximize)
p.solve ()
```

6.2.5 A Min-cost-flow problem using NumPy

This example solves a min-cost-flow problem using NumPy and the incidence matrix of the graph. It uses the `networkx` package, which might have to be installed using, for example, `pip`.

```
import networkx as netx

# 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

G = netx.DiGraph (E)

# Get NumPy representation
A = (netx.incidence_matrix (G, oriented = True).toarray())
```

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])

flow = np.array ([xp.var () for i in E]) # flow variables declared on arcs

p = xp.problem ('network flow')

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

p.solve ()
```

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

6.2.6 A nonlinear model

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

```
a,b = 1,100

x = xp.var (lb = -xp.infinity)
y = xp.var (lb = -xp.infinity)

p = xp.problem ()

p.addVariable (x,y)

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

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

p.solve ()

print ("status: ", p.getProbStatus ())
print ("string: ", p.getProbStatusString ())

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

6.2.7 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 $A_{(0,1)} + A_{(1,2)} + \ldots + A_{(N-1,0)}$, 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 $n - 1$.

```
rho   = [xp.var (lb = 1e-5,     ub = 1.0)     for i in Vertices]
theta = [xp.var (lb = -math.pi, ub = math.pi) for i in Vertices]

p = xp.problem ()

p.addVariable (rho, theta)

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.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.

6.2.8 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 dictionary 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)

x = {(i,j): xp.var (vartype = xp.binary, name='q{0}{1}'.format (i,j))
     for i in N for j in N}

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[i,j] for j in N for i in N if i+j == k) <= 1
              for k in range (1,2*n-2)]
diagonal2  = [xp.Sum (x[i,j] for j in N for i in N if i-j == k) <= 1
              for k in range (2-n,n-1)]

p = xp.problem()

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

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

p.solve ()
```

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 ('')
```

6.2.9 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)

x = {(i,j,k): xp.var (vartype = xp.binary, name='x{0}_{1}_{2}'.format(i,j,k))
     for i in N for j in N for k in N}

# 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 = xp.problem()

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

p.solve ()

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 ('')
```

6.3 Examples using NumPy

6.3.1 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 = np.array ([[[xp.var (vartype = xp.binary)
                 for i in S1]
                 for j in S2]
                 for k in S3])

m.addVariable (h)

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.solve ()
```

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

```
mstart1=[]
mclind1=[]
dqe1=[]
m.getqrowqmatrix (cons00, mstart1, mclind1, dqe1, 29, h[0][0][0], h[3][2][1])
print ("row 0:", mstart1, mclind1, dqe1)
```

6.3.2 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.

```
x = np.array ([xp.var() for i in range(5)])

p = xp.problem ()
p.addVariable (x)
p.addConstraint (xp.Sum (x) >= 2)

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

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)
```

6.3.3 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
x = np.array ([xp.var() for i in range(5)]) # vector of variables
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 = xp.problem ()

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

p.solve ()
```

6.3.4 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 $Qx \geq 0$, and nonnegativity on all variables. Report solution if available.

```
n = 10

Q  = np.arange (1, n**2 + 1).reshape (n, n)
x  = np.array ([xp.var () for i in range (n)])
x0 = np.random.random (n)

p = xp.problem ()

p.addVariable (x)

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.solve ('')

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

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

6.4 Advanced examples: callbacks and problem querying/modifying

6.4.1 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 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.

```
import networkx as netx
from matplotlib import pyplot as plt

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.

```
T = nx.Graph ()

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

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

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
```

6.4.2 Query and modify a simple problem

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

```
x = xp.var ()
y = xp.var ()

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

p = xp.problem ()

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

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

6.4.3 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.

```
p = xp.problem ()

N = 5
S = range (N)

x = [xp.var (vartype = xp.binary) for i in S]
```

```
p.addVariable (x)

# vectors can be used whole or addressed with their index

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], ['Y'], ['B'])
p.write ("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.

The same effect would be obtained with `p.addqmatrix (cc[0], [x[0],x[3],x[3]], [x[0],x[0],x[3]], [1,-1,1])`

As constraint vector `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:	$x_0 + 2\ x_1 + 3x_2$	$\geq$	4
	$4x_0 + 5x_1 + 6x_2 + 7\ x_3 + 8\ x_4 + y$	$\leq$	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 (qrtype = ['G', 'L'],
           rhs = [4, 4.4],
           mstart = [0, 3, 9],
           mclind = [x[0],x[1],x[2], x[0],x[1],x[2],x[3],x[4], 5],
           dmatval = [1,2,3,4,5,6,7,8,-3],
           names = ['newcon1', 'newcon2'])

p.solve ()
p.write ("amended", "lp")

slacks = []

p.calcslacks (solution = mysol, calculatedslacks = slacks)

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

The code below add 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], ['p1'], ['I'])
p.addcols ([4], [0,3], [c0,4,2], [-3, 2.4, 1.4], [0], [10], ['p2'], ['C'])
p.addcols ([4], [0,3], [c0,4,2], [-3,   2,   1], [0],  [1], ['p3'], ['S'])
p.addcols ([4], [0,3], [c0,4,2], [-3, 2.4,   4], [0],  [2], ['p4'], ['P'])
p.addcols ([4], [0,3], [c0,4,2], [-3,   2,   1], [0],  [2], ['p5'], ['R'])
```

```
p.solve ()

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],
           ['p1','p2','p3','p4','p5'],
           ['I','C','S','P','R'])
```

6.4.4 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.

```
p = xpress.problem ()

p.read ("example.lp")
p.solve ()
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.solve()
print ("New solution: ", p.getSolution ())
```

6.4.5 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.getlpsol (s, None, None, None)

    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 = [] # initialize s to an empty string to provide it as an output parameter

    prob.getlpsol (s, None, None, None)

    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)

x = {(i,j): xp.var (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 = xp.problem ()

p.addVariable (x)
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.maxtime = -2000 # 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.solve ()

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
```

CHAPTER 7

Reference Manual

7.1 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 Python 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.

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 Python lists, NumPy arrays, sequences, or generators when applicable. The output non-scalar parameters are stored as Python 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 (qrtype = type, rhs = rhs, mstart = mstart,
                   mclind = indices, dmatval = values)
```

Because the Python interface relies on the Xpress Optimizer low-level interface, it is advisable to complement the knowledge in this reference manual with that of the Xpress Optimizer reference manual.

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 an assignment Python command, 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 following as `"array"` 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`.

7.2 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` command. The invocation is therefore as in the example that follows:

```
import xpress as xp
print (xp.getlasterror ())
```

xpress.init	xpress.free	xpress.addcbmsghandler
xpress.getbanner	xpress.getcheckedmode	xpress.getdaysleft
xpress.getlasterror	xpress.getlicerrmsg	xpress.getversion
xpress.Sum	xpress.Dot	xpress.setcheckedmode
xpress.removecbmsghandler	xpress.setdefaultcontrol	xpress.setdefaults

7.3 Methods of the problem class

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
x = xp.var ()
p = xp.problem ()
p.setObjective (x + 3 * x**2 + 2)
```

addcols	addConstraint	addIndicator	addmipsol
addqmatrix	addrows	addSOS	addVariable
basisstability	btran	calcobjective	calcreducedcosts
calcslacks	calcsolinfo	chgbounds	chgcoef
chgcoltype	chgglblimit	chgmcoef	chgmqobj
chgobj	chgobjsense	chgqobj	chgqrowcoeff
chgrhs	chgrhsrange	chgrowtype	copy
copycontrols	delConstraint	delqmatrix	delSOS
delVariable	dumpcontrols	estimaterowdualranges	fixglobals
ftran			

getAttrib	getbasis	getcoef	getcols
getcoltype	getConstraint	getControl	getdirs
getDual	getdualray	getglobal	getiisdata
getIndex	getIndexFromName	getindicators	getinfeas
getlasterror	getlb	getlpsol	getmessagestatus
getmipsol	getmqobj	getobj	getObjVal
getpivotorder	getpivots	getpresolvebasis	getpresolvemap
getpresolvesol	getprimalray	getProbStatus	getProbStatusString
getqobj	getqrowcoeff	getqrowqmatrix	getqrowqmatrixtriplets
getqrows	getRCost	getrhs	getrhsrange
getrows	getrowtype	getscaledinfeas	getSlack
getSolution	getSOS	getub	getunbvec
getVariable	hasdualray	hasprimalray	

iisall	iisclear	iisfirst
iisisolations	iisnext	iisstatus
iiswrite	loadbasis	loadbranchdirs
loaddelayedrows	loaddirs	loadlpsol
loadmipsol	loadmodelcuts	loadpresolvebasis
loadpresolvedirs	loadproblem	loadsecurevecs
lpoptimize	mipoptimize	name
objsa	postsolve	presolverow
read	readbasis	readbinsol
readdirs	readslxsol	refinemipsol
repairinfeas	repairweightedinfeas	repairweightedinfeasbounds
restore	reset	rhssa
save	scale	setControl
setdefaults	setindicators	setlogfile
setmessagestatus	setObjective	setprobname
solve	strongbranch	write
writebasis	writebinsol	writedirs
writeprtsol	writeslxsol	writesol

The following table contains the problem functions to be called for nonlinear problems.

addcoefs	adddfs	addtolsets
addvars	cascade	cascadeorder
chgcascadenlimit	chgccoef	chgnlcoef
chgdeltatype	chgdf	chgrowstatus
chgrowwt	chgtolset	chgvar
construct	delcoefs	deltolsets
delvars	evaluatecoef	evaluateformula
filesol	fixpenalties	getccoef
getcoefformula	getcoefs	getcolinfo
getdf	getdtime	getmessagetype
getrowinfo	getrowstatus	getrowwt
getslpsol	gettolset	getvar
globalsol	loadcoefs	loaddfs
loadtolsets	loadvars	msaddcustompreset
msaddjob	msaddpreset	msclear
parsecformula	parseformula	preparseformula
presolve	printmemory	printevalinfo
printmsg	reinitialize	scaling
setcurrentiv	setuniqueprefix	tokencount
unconstruct	updatelinearization	validformula
validate	validatekkt	validaterow
validatevector		

7.4 Methods for branching objects

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	branchobj.addbranches	branchobj.addcuts
branchobj.addrows	branchobj.getbounds	branchobj.getbranches
branchobj.getid	branchobj.getlasterror	branchobj.getrows
branchobj.setpreferredbranch	branchobj.setpriority	branchobj.store
branchobj.validate		

7.5 Methods for adding/removing callbacks of a problem object

The following pages present methods that can be called from a problem **before** optimization has started, to add or remove callbacks. All these methods are part of the `problem` class and have to be instantiated from a `problem` object.

addcbbariteration	removecbbariteration
addcbbarlog	removecbbarlog
addcbchgbranchobject	removecbchgbranchobject
addcbcutlog	removecbcutlog
addcbdestroymt	removecbdestroymt
addcbgapnotify	removecbgapnotify
addcbgloballog	removecbgloballog
addcbinfnode	removecbinfnode
addcbintsol	removecbintsol
addcblplog	removecblplog
addcbmessage	removecbmessage
addcbmipthread	removecbmipthread
addcbnewnode	removecbnewnode
addcbnodecutoff	removecbnodecutoff
addcboptnode	removecboptnode
addcbpreintsol	removecbpreintsol
addcbprenode	removecbprenode
addcbusersolnotify	removecbusersolnotify

7.6 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.addcb*` methods. Calling these functions outside of a callback may result in an error and trigger termination of the optimization process. We provide two tables: one is for the Optimizer and another for the nonlinear solvers.

copycallbacks	delcpcuts	delcuts
getcpcutlist	getcpcuts	getcutlist
getcutmap	getcutslack	interrupt
loadcuts	setbranchbounds	setbranchcuts
storebounds	storecuts	strongbranchcb
addcuts		

setcbcascadeend	setcbcascadestart	setcbcascadevar
setcbcascadevarfail	setcbcoefevalerror	setcbconstruct
setcbdestroy	setcbdrcol	setcbformula
setcbiterend	setcbiterstart	setcbitervar
setcbmsjobend	setcbmsjobstart	setcbmswinner
setcbslpend	setcbslpnode	setcbslpstart

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.
Note that it is **unnecessary** to call this function upon exiting a block that uses the Xpress module, or when the optimizer is no longer used, as Python will release the Xpress environment when freeing the Xpress module. This function might be useful when a license is needed by another user or program, and one wishes to release the license.

Synopsis

```
xpress.free ()
```

Example

The following example shows how to call xpress.free and a possible use:

```
x = xp.var ()
y = xp.var ()
p = xp.problem () # This would imply a call to xp.init()
p.addVariable (x, y)
p.addConstraint (x+y <= 1)
p.setObjective (x+2*y, sense=xp.maximize)
p.solve ()
xp.free () # from this point on, the license
           # can be claimed by other users
```

Note that xpress.init is only useful when the user wants to claim a license that might be used by another program or user.

Further information

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 available, and referring to it may lead to errors. For instance, the following code results in an aborted run:

```
import xpress p = xpress.problem () xpress.free() xpress.init()
p.solve ()
```

Related topics

xpress.init

xpress.getbanner

Purpose

Returns the banner and copyright message.

Synopsis

```
i = xpress.getbanner ()
```

Example

```
print (xpress.getbanner ())
```

xpress.getcheckedmode

Purpose

Returns whether checking & validation of all Optimizer function calls is enabled for the current process. Checking & validation is enabled by default but can be disabled by `xpress.setcheckedmode`.

Synopsis

```
i = xpress.getcheckedmode ()
```

Related topics

`xpress.setcheckedmode`.

xpress.getdaysleft

Purpose

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 returns an error. The expiry information for evaluation licenses is also included in the Optimizer banner message.

xpress.getlasterror

Purpose

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

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 import of the Xpress Python module failed:

```
try:
   import xpress
except RuntimeError:
   print (xpress.getlicerrmsg ())
else:
   print ("all good")
```

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.

Synopsis

```
v = xpress.getversion ()
```

Example

```
print ("Using Xpress Optimizer version", xpress.getversion ())
```

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, as Python will claim a license automatically. This function might be useful when the user wants to reserve a license and prevent that it is claimed by user or program.

Synopsis

```
xpress.init ()
```

Example

The following example shows how to call `xpress.init` and why it could be useful:

```
xp.init () # reserves the license before creating variables
x = xp.var ()
y = xp.var ()
p = xp.problem () # This would imply a call to xp.init()
p.addVariable (x, y)
p.addConstraint (x+y <= 1)
p.setObjective (x+2*y, sense=xp.maximize)
p.solve ()
```

Note that the call to `xpress.init` is not necessary and should only be made when the user wants to claim a license that might be used by another program or user before the call to `xpress.problem`.

Related topics

`xpress.free`

xpress.setcheckedmode

Purpose

Disable/enable some of the checking & validation of function calls & 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 & 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 to disable function call checking & validation during application development.

Synopsis

```
xpress.setcheckedmode (checked_mode)
```

Argument

`checked_mode` 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.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 ('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.setdefaultcontrol`, `problem.setdefaults`, `problem.setdefaultcontrol`.

xpress.setdefaultcontrol

Purpose

Sets one of the module's controls to its default values. 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(xpress.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.Sum

Purpose

Alternative sum operator for an arbitrary number of objects created by a Python list, tuple, generator, NumPy array, dictionary, etc.

Synopsis

```
a = xpress.Sum (t1, t2, ...)
```

Example

The following are allowed uses of the `Sum` operator:

```
N = 20
S = range (S)
x = [xpress.var () for i in S]
y = [xpress.var (vartype = xpress.binary) for i in S]
p = xpress.problem ()
p.addVariable (x, y)
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])
                 <= log (10 + i) for i in S)
```

Further information

The `Sum` operator is functionally equivalent to Python's native `sum` operator. However, it is strongly advised to use the Xpress' `Sum` operator when constructing large expressions involving variables, as doing otherwise might slow down the execution significantly.

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 $T_1, T_2, \ldots, T_k$ of $d_1, d_2, \ldots, d_k$ dimensions is an object of $d_1 + d_2 + \ldots + d_k - 2(k-1)$ 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 (t1, t2, ..., out)
```

Argument

`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)

x  = np.array ([xp.var () for i in S])
x0 = np.random.random (N) # creates an N-vector of random numbers

p = xp.problem ()

# 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).reshape (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

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 $(n_1, n_2, \ldots, n_{d'})$ and $(m_1, m_2, \ldots, m_{d''})$, respectively, is a multi-dimensional array of dimension $d' + d'' - 2$, whose arity vector is $(n_1, n_2, \ldots, n_{d'-1}, m_1, m_2, \ldots, m_{d''-2}, m_{d''})$ and whose generic element is

$v_{i_1,i_2,\ldots,i_{d'-1},j_1,j_2,\ldots,j_{d''-2},j_{d''}} = \sum_{1 \le h \le n_{d'}} t'_{i_1,i_2,\ldots,i_{d'-1},h} \cdot t''_{j_1,j_2,\ldots,j_{d''-2},h,j_{d''}}$.

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 $\sum_{1 \le h \le n} v'_h \cdot v''_h$.

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

$C_{ij} = \sum_{1 \le h \le n} A_{ih} \cdot B_{hj}$.

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.

xpress.Prod

Purpose

Returns the product of a sequence of one or more expressions.

Synopsis

```
a = xpress.Prod (t1, t2, ...)
```

Example

The following are allowed uses of the `Prod` operator:

```
n=10
x = [xp.var() 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.exp

Purpose

Returns the exponential of a given expression.

Synopsis

```
a = xpress.exp (t)
```

Argument

`t` 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 (t)
```

Argument

`t` 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 (t1)
```

Argument

`t` Argument.

Related topics

`xpress.log`.

xpress.sin

Purpose

Returns the sine of a given expression.

Synopsis

```
a = xpress.sin (t)
```

Argument

`t` 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.cos

Purpose

Returns the cosine of a given expression.

Synopsis

```
a = xpress.cos (t)
```

Argument

`t` 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.tan

Purpose

Returns the tangent of a given expression.

Synopsis

```
a = xpress.tan (t)
```

Argument

`t` 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.asin

Purpose

Returns the arcsine of a given expression.

Synopsis

```
a = xpress.asin (t)
```

Argument

`t` 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.acos

Purpose

Returns the arccosine of a given expression.

Synopsis

```
a = xpress.acos (t)
```

Argument

`t` 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.atan

Purpose

Returns the arctangent of a given expression.

Synopsis

```
a = xpress.atan (t)
```

Argument

`t` 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.max

Purpose

Returns the maximum of one or more expressions.

Synopsis

```
a = xpress.max (t1, t2, ..., tn)
```

Argument

`t1, t2...` Arguments.

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 (t1, t2, ..., tn)
```

Argument

`t1, t2...` Arguments.

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.abs

Purpose

Returns the absolute value of a given expression

Synopsis

```
a = xpress.abs (t)
```

Argument

`t` Argument of the abs() function.

Further information

Using Python's `math` library operator `math.abs` is only advisable when the argument is not an expression that depends on variables. Python's native `abs` operator is equivalent to `xpress.abs` for arguments that are functions of variables.

xpress.sign

Purpose

Returns the sign of an expression: 1 if positive, -1 if negative, 0 if zero.

Synopsis

```
a = xpress.sign (t)
```

Argument

`t` Argument of the sign function.

xpress.erf

Purpose

Returns the error function with an expression as its argument.

Synopsis

```
a = xpress.erf (t)
```

Argument

`t` 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 (t)
```

Argument

`t` 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.sqrt

Purpose

Returns the square root of an expression.

Synopsis

```
a = xpress.sqrt (t)
```

Argument

`t` 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.user

Purpose

Creates an expression that is computed by means of a user-specified function.

Synopsis

```
def f (a1, a2, ..., an):  [...]  a = xpress.user (f, t1, t2, ..., tn)
```

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.

Example

The following code shows how to define user functions:

```
import math
def mynorm (v):
  return math.sqrt (sum (v[i] for i in range (len (v))))

def weighted_sum (t1, t2, t3 = 0):
  return 2*t1 + 3*t2 + 4*t3

x = [xp.var() for i in range (20)]

f1 = xp.user (mynorm, x)
f2 = xp.user (weighted_sum, x[0], x[1], x[2])

# doesn't use optional arg
f3 = xp.user (weighted_sum, x[0], x[1])
```

Further information

User functions **must** return a Float, as the behaviour is otherwise undefined.

xpress.addcbmsghandler

Purpose

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 (f_msghandler, object, priority)
ret = f_msghandler (vObject, vUserContext, vSystemThreadId, sMsg, iMsgType,
       iMsgNumber)
```

Arguments

`f_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.

`object` 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.

Further information

To send all messages to a log file the built in message handler `logfilehandler` can be used. This can be done with:

```
 xpress.addcbmsghandler (logfilehandler,
'log.txt', 0)
```

Related topics

`xpress.removecbmsghandler`.

xpress.removecbmsghandler

Purpose

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 (f_msghandler, object)
```

Arguments

`f_msghandler` The callback function to remove. If `None` then all message callback functions added with the given user-defined object value will be removed.

`object` 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 pointer `f_msghandler` will be removed.

Related topics

`xpress.addcbmsghandler`.

problem.addcbbariteration

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 addcbbariteration.

Synopsis

```
problem.addcbbariteration (f_bariteration, object, priority)
barrier_action = f_bariteration (my_prob, my_object)
```

Arguments

`f_bariteration` 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 `object` 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;

`object` 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:
xprob.addcbbariteration (BarrierIterCallback, myobj, 0)
```

Further information

1. Only the following functions are expected to be called from the callback: `problem.getlpsol` and the attribute/control value retrieving and setting routines.
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.getlpsol`.

Related topics

`problem.removecbbariteration`.

problem.addcbbarlog

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 addcbbarlog.

Synopsis

```
problem.addcbbarlog (f_barlog, object, priority)
ret = f_barlog (my_prob, my_object)
```

Arguments

`f_barlog`	The callback function itself. This takes two arguments, `my_prob` and `my_object`, and has an integer return value. If the value returned by `f_barlog` 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, `f_barlog`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `addcbbarlog`.
`object`	A user-defined object to be passed to the callback function, `f_barlog`.
`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.removecbbarlog`, `problem.addcbgloballog`, `problem.addcblplog`, `problem.addcbmessage`.

problem.addcbchgbranchobject

Purpose

Declares a callback function that will be called every time the Optimizer has selected a global 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.addcbchgbranchobject.

Synopsis

```
problem.addcbchgbranchobject (f_chgbranchobject, object, priority)
newobject = f_chgbranchobject (my_prob, my_object, obranch)
```

Arguments

f_chgbranchobject — 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, f_chgbranchobject.

my_object — The user defined object passed as object 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.

object — A user-defined object to be passed to the callback function, f_chgbranchobject.

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 global 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.removecbchgbranchobject.

problem.addcbcutlog

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.addcbcutlog`.

Synopsis

```
problem.addcbcutlog (f_cutlog, object, priority)
ret = f_cutlog (my_prob, my_object)
```

Arguments

`f_cutlog`	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, `f_cutlog`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `addcbcutlog`.
`object`	A user-defined object to be passed to the callback function, `f_cutlog`.
`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 `f_cutlog` should return a non-zero value to stop cutting on the current node.

Related topics

`problem.removecbcutlog`.

problem.addcbdestroymt

Purpose

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 (f_destroymt, object, priority)
f_destroymt (my_prob, my_object)
```

Arguments

`f_destroymt`	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 `object` when setting up the callback with `addcbdestroymt`.
`object`	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.removecbdestroymt`, `problem.addcbmipthread`.

problem.addcbgapnotify

Purpose

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 (f_gapnotify, object, priority)
(RelGapNotify, AbsGapNotify, AbsGapNotifyObj, AbsGapNotifyBound) = f_gapnotify
       (my_prob, my_object)
```

Arguments

`f_gapnotify` The callback function.

`object` A user-defined object that wil be passed into the callback `f_gpanotify`.

`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 `object` 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.

`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.

`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.

`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.

`object` A user-defined object to be passed to the callback function, `f_gapnotify`.

`priority` 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.

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
    relgap = abs ((obj - bound) / obj)

    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.removecbgapnotify.

problem.addcbgloballog

Purpose

Declares a global log callback function, called each time the global log is printed. This callback function will be called in addition to any callbacks already added by addcbgloballog.

Synopsis

```
problem.addcbgloballog (f_globallog, object, priority)
ret = f_globallog (my_prob, my_object)
```

Arguments

`f_globallog`	The callback function which takes two arguments, `my_prob` and `my_object`, and has an integer return value. This function is called whenever the global log is printed as determined by the `MIPLOG` control.
`my_prob`	The problem passed to the callback function, `f_globallog`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `addcbgloballog`.
`object`	A user-defined object to be passed to the callback function, `f_globallog`.
`priority`	An integer that determines the order in which multiple global 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 global search the node number and its depth:

```
prob.controls.miplog = 3
prob.addcbgloballog (globalLog, None, 0)
prob.mipoptimize ('')
```

The callback function may resemble:

```
def globalLog (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 global search will be interrupted.

Related topics

problem.removecbgloballog, problem.addcbbarlog, problem.addcblplog, problem.addcbmessage.

problem.addcbinfnode

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. This callback function will be called in addition to any callbacks already added by addcbinfnode.

Synopsis

```
problem.addcbinfnode (f_infnode, object, priority)
f_infnode (my_prob, my_object)
```

Arguments

`f_infnode`	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, `f_infnode`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `addcbinfnode`.
`object`	A user-defined object to be passed to the callback function, `f_infnode`.
`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 global 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.removecbinfnode, problem.addcboptnode, problem.addcbintsol, problem.addcbnodecutoff.

problem.addcbintsol

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 addcbintsol.

Synopsis

```
problem.addcbintsol (f_intsol, object, priority)
f_intsol (my_prob, my_object)
```

Arguments

`f_intsol`	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, `f_intsol`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `addcbintsol`.
`object`	A user-defined object to be passed to the callback function, `f_intsol`.
`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 global search:

```
prob.addcbintsol (printsol, None, 0)
prob.mipoptimize ("")
```

The callback function might resemble:

```
def printsol (my_prob, object):

    cols = my_prob.attributes.originalcols
    objval = my_prob.attributes.lpobjval

    x = []
    my_prob.getlpsol (x, None, None, None)

    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 either `problem.getlpsol` or `problem.getpresolvesol`. `problem.getmipsol` always returns the last integer solution found and, if called from the `intsol` callback, it will not necessarily return the solution that caused the invocation of the callback (for example, it is possible that when solving with multiple MP threads, another thread finds a new integer solution before the user calls `problem.getmipsol`).
3. This callback is called after a new integer solution was found by the Optimizer. Use a callback set by `problem.addcbpreintsol` 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.removecbintsol`, `problem.addcbpreintsol`.

problem.addcblplog

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 addcblplog.

Synopsis

```
problem.addcblplog (f_lplog, object, priority)
ret = f_lplog (my_prob, my_object)
```

Arguments

f_lplog	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, f_lplog.
my_object	The user-defined object passed as object when setting up the callback with addcblplog.
object	A user-defined object to be passed to the callback function, f_lplog.
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.removecblplog, problem.addcbbarlog, problem.addcbgloballog, problem.addcbmessage.

problem.addcbmessage

Purpose

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 (f_message, object, priority)
f_message (my_prob, my_object, msg, msgtype)
```

Arguments

`f_message`	The callback function which takes five arguments, `my_prob`, `my_object`, `msg`, `len` 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 `object` when setting up the callback with `addcbmessage`.
`msg`	A null terminated character array (string) containing the message, which may simply be a new line.
`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.
`object`	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. 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. 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.removecbmessage`, `problem.addcbbarlog`, `problem.addcbgloballog`, `problem.addcblplog`, `problem.setlogfile`.

problem.addcbmipthread

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 addcbmipthread.

Synopsis

```
problem.addcbmipthread (f_mipthread, object, priority)
f_mipthread (my_prob, my_object, thread_prob)
```

Arguments

`f_mipthread`	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
`object`	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.removecbmipthread`, `problem.addcbdestroymt`.

problem.addcbnewnode

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 addcbnewnode.

Synopsis

```
problem.addcbnewnode (f_newnode, object, priority)
f_newnode (my_prob, my_object, parentnode, newnode, branch)
```

Arguments

`f_newnode`	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, `f_newnode`.
`my_object`	The user-defined object passed as `object` 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 global entity this will be either `0` or `1`.
`object`	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 global entity, `branch` will be either zero or one, depending on whether the new node corresponds to branching the global entity up or down.
2. When branching on a `branchobject`, `branch` refers to the given branch index of the object.

Related topics

`problem.removecbnewnode`.

problem.addcbnodecutoff

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 addcbnodecutoff.

Synopsis

```
problem.addcbnodecutoff (f_nodecutoff, object, priority)
f_nodecutoff (my_prob, my_object, node)
```

Arguments

`f_nodecutoff` 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, `f_nodecutoff`.

`my_object` The user-defined object passed as `object` when setting up the callback with `addcbnodecutoff`.

`node` The number of the node that is cut off.

`object` A user-defined object to be passed to the callback function, `f_nodecutoff`.

`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 global 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.removecbnodecutoff`, `problem.addcboptnode`, `problem.addcbinfnode`, `problem.addcbintsol`.

problem.addcboptnode

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 addcboptnode.

Synopsis

```
problem.addcboptnode (f_optnode, object, priority)
feas = f_optnode (my_prob, my_object)
```

Arguments

`f_optnode`	The callback function which takes three arguments, `my_prob`, `my_object` and `feas`, and has no return value.
`my_prob`	The problem passed to the callback function, `f_optnode`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `addcboptnode`.
`feas`	The feasibility status. If set to a nonzero value by the user, the current node will be declared infeasible.
`object`	A user-defined object to be passed to the callback function, `f_optnode`.
`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, feas):

    node = prob.attributes.currentnode
    print ("NodeOptimal: node number", node)
    objval = prob.attributes.lpobjval
    print ("Objective function value =", objval)
```

Related topics

problem.removecboptnode, problem.addcbinfnode, problem.addcbintsol, problem.addcbnodecutoff, CALLBACKCOUNT_OPTNODE.

problem.addcbpreintsol

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 function will be called in addition to any integer solution callbacks already added by addcbpreintsol.

Synopsis

```
problem.addcbpreintsol (f_preintsol, object, priority)
(ifreject, newcutoff) = f_preintsol (my_prob, my_object, isheuristic, cutoff)
```

Arguments

`f_preintsol`	The callback function which takes five arguments, `my_prob`, `my_object`, `isheuristic`, `ifreject` 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.
`my_prob`	The problem passed to the callback function, `f_preintsol`.
`my_object`	The user-defined object passed as object when setting up the callback with `addcbpreintsol`.
`isheuristic`	Set to 1 if the solution was found using a heuristic. Otherwise, it will be the global feasible solution to the current node of the global search.
`ifreject`	Set this to 1 if the solution should be rejected.
`cutoff`	The current `cutoff` value.
`newcutoff`	The new `cutoff` value, to be used by the Optimizer if the solution is accepted. The returned `newcutoff` value will not be updated if the solution is rejected.
`object`	A user-defined object to be passed to the callback function, `f_preintsol`.
`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.getlpsol` or `problem.getpresolvesol`. `problem.getmipsol` will not return the newly found solution because it has not been saved at this point.

Related topics

`problem.removecbpreintsol`, `problem.addcbintsol`.

problem.addcbprenode

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 addcbprenode.

Synopsis

```
problem.addcbprenode (f_prenode, object, priority)
nodinfeas = f_prenode (my_prob, my_object)
```

Arguments

`f_prenode`	The callback function, which takes three arguments, `my_prob`, `my_object` and `nodinfeas`, and has no return value. This function is called before a node is reoptimized and the node may be made infeasible by setting `*nodinfeas` to `1`.
`my_prob`	The problem passed to the callback function, `f_prenode`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `addcbprenode`.
`nodinfeas`	The feasibility status. If set to a nonzero value by the user, the current node will be declared infeasible by the Optimizer.
`object`	A user-defined object to be passed to the callback function, `f_prenode`.
`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.removecbprenode`, `problem.addcbinfnode`, `problem.addcbintsol`, `problem.addcbnodecutoff`, `problem.addcboptnode`.

problem.addcbusersolnotify

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 addcbusersolnotify.

Synopsis

```
problem.addcbusersolnotify (f_usersolnotify, object, priority)
f_usersolnotify (my_prob, my_object, solname, status)
```

Arguments

f_usersolnotify — 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, f_usersolnotify.

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 globals.
- 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 globals 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.

object — A user-defined object to be passed to the callback function, f_usersolnotify.

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.removecbusersolnotify, problem.addmipsol.

problem.addcoefs

Purpose

Add non-linear coefficients to the SLP problem

Synopsis

```
problem.addcoefs (rowindex, colindex, factor, fstart, parsed, type, value)
```

Arguments

`rowindex`	Array holding the indices of rows for the coefficient.
`colindex`	Array holding the indices of columns for the coefficient.
`factor`	Array holding factor by which formula is scaled. If `None`, a value of 1.0 will be used.
`FormulaStart`	Integer array of length `nSLPCoef+1` holding the start position in the arrays `Type` and `Value` of the formula for the coefficients. `FormulaStart` should have an extra entry containing the next position after the end of the last formula.
`parsed`	Integer 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 = []

n = 0
ncoef = 0

formulastart[ncoef], ncoef = n, ncoef + 1
Type[n], Value[n], n = xslp_op_col, 3, n+1
Type[n],           n = xslp_op_eof,    n+1


formulastart[ncoef], ncoef = n, ncoef + 1

Type[n], Value[n], n = xslp_op_col, 2,             n+1
Type[n], Value[n], n = xslp_op_col, 2,             n+1
Type[n], Value[n], n = xslp_op_op,  xslp_MULTIPLY, n+1
Type[n],           n = xslp_op_eof,                n+1

formulastart[ncoef], ncoef = n, ncoef + 1

Type[n], Value[n], n = xslp_op_col, 2, n+1
Type[n],           n = xslp_op_eof,    n+1

formulastart [ncoef] = n

p.addcoefs (rowindex, colindex,  None, formulastart, 1, 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`=1), 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 `FormulaStart`. For possible token types and values see the chapter on "Formula Parsing".

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.chgnlcoef`, `problem.chgccoef`, `problem.delcoefs`, `problem.getcoefformula`, `problem.getccoef`, `problem.loadcoefs`

problem.addcols

Purpose

Add columns to the problem after passing it to the Optimizer using the input routines.

Synopsis

```
problem.addcols (objx, mstart, mrwind, dmatval, bdl, bdu, names, types)
```

Arguments

`objx`	Array containing the objective function coefficients of the new columns.
`mstart`	Array containing the offsets in the `mrwind` and `dmatval` arrays of the start of the elements for each column.
`mrwind`	Array containing the row indices for the elements in each column.
`dmatval`	Array containing the element values.
`bdl`	Array containing the lower bounds on the added columns.
`bdu`	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.

Example

In this example, we consider the two problems:

(a)	maximize:	2x + y			(b)	maximize:	2x + y + 3z		
	subject to:	x + 4y	$\leq$	24		subject to:	x + 4y + 2z	$\leq$	24
		y	$\leq$	5			y + z	$\leq$	5
		3x + y	$\leq$	20			3x + y	$\leq$	20
		x + y	$\leq$	9			x + y + 3z	$\leq$	9
							z	$\leq$	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], mstart = [0,3], mrwind = [0, 1, 3],
           matval = [2,1,3], bdl = [-xpress.infinity], bdu = [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.addConstraint

Purpose

Adds one or more constraints to the problem.

Synopsis

```
problem.addConstraint (c1, c2, ...)
```

Argument

`c1,c2...` Constraints or list/tuples/array of constraints created with the `xpress.constraint()` call.

Example

```
N = 20
x = [xpress.var () for i in range (N)]
c = [x[i] <= x[i+1] for i in range (N-1)]
c2 = x[0] >= x[19]
p = xpress.problem ()
p.addVariable (x)
p.addConstraint (x[2] == x[4])
p.addConstraint (c, c2)
```

Further information

All arguments can be single constraints or lists, tuples, or NumPy arrays of constraints created as `xpress.constraint` objects. Arguments do not need to be declared prior to the call.

problem.addcuts

Purpose

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 (mtype, rtype, rhs, mstart, mcols, matval)
```

Arguments

`mtype`	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`.
`rtype`	Character array of length `ncuts` containing the row types: `L` indicates a $\leq$ row; `G` indicates $\geq$ row; `E` indicates an = row.
`rhs`	Array of length `ncuts` containing the right hand side elements for the cuts.
`mstart`	Array containing offset into the `mcols` and `dmatval` arrays indicating the start of each cut. This array is of length `ncuts+1` with the last element, `mstart[ncuts]`, being where cut `ncuts+1` would start.
`cols`	Array of length `mstart[ncuts]` containing the column indices in the cuts.
`matval`	Array of length `mstart[ncuts]` containing the matrix values for the cuts.

Further information

1. The columns and elements of the cuts must be stored contiguously in the `mcols` and `dmatval` arrays passed to `addcuts`. The starting point of each cut must be stored in the `mstart` array. To determine the length of the final cut, the `mstart` array must be of length `ncuts+1` with the last element of this array containing the position in `mcols` and `dmatval` where the cut `ncuts+1` would start. `mstart[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.

Related topics

`problem.addrows`, `problem.delcpcuts`, `problem.delcuts`, `problem.getcpcutlist`, `problem.getcutlist`, `problem.loadcuts`, `problem.storecuts`, Section "Working with the cut manager" of the Xpress Optimizer reference manual.

problem.adddfs

Purpose

Add a set of distribution factors

Synopsis

```
problem.adddfs (colindex, rowindex, value)
```

Arguments

`colindex`	Array of indices of columns whose distribution factor is to be changed.
`rowindex`	Array of indices of the rows 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.addIndicator

Purpose

Adds one or more indicator constraints to the problem.

Synopsis

```
problem.addIndicator (c1, c2, ...)
```

Argument

`c1,c2...` Tuples containing an indicator constraints, or list/tuples/array of tuples containing a binary condition and a constraint.

Example

```
x = xpress.var (vartype = xpress.binary)
y = xpress.var (lb = 10, ub = 20)
z = xpress.var ()
ind1 = (x==1, y+z <= 40)
p = xpress.problem ()
p.addVariable (x,y,z)
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.addmipsol

Purpose

Adds a new feasible, infeasible or partial MIP solution for the problem to the Optimizer.

Synopsis

```
problem.addmipsol (mipsolval, mipsolcol, solname)
```

Arguments

`mipsolval`	Array containing solution values.
`mipsolcol`	Optional integer array containing the column indices for the solution values provided in `mipsolval`. It is optional when the length of mipsolval is equal to `COLS`, in which case it is assumed that `mipsolval` provides a complete solution vector.
`solname`	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, global 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.addcbusersolnotify` callback function can be used to discover the outcome of a loaded solution. The optional name provided as `solname` will be returned in the callback function.
5. If one or more solutions are loaded during the `problem.addcboptnode` 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.addcbusersolnotify`, `problem.addcboptnode`.

problem.addqmatrix

Purpose

Adds a new quadratic matrix into a row defined by triplets.

Synopsis

```
problem.addqmatrix (irow, mqc1, mqc2, dqe)
```

Arguments

`irow`	Index of the row where the quadratic matrix is to be added.
`mqc1`	First index in the triplets.
`mqc2`	Second index in the triplets.
`dqe`	Coefficients in the triplets.

Further information

1. The triplets should define the whole quadratic expression. This means that to add $x^2 + 4xy$ the `dqe` arrays shall contain the coefficients 1 and 4.
2. The matrix defined by `mqc1`, `mqc2` and `dqe` should be positive semi-definite for $\leq$ and negative semi-definite for $\geq$ rows.
3. The row must not be an equality or a ranged row.

Related topics

`problem.loadproblem`, `problem.getqrowcoeff`, `problem.chgqrowcoeff`, `problem.getqrowqmatrix`, `problem.getqrowqmatrixtriplets`, `problem.getqrows`, `problem.chgqobj`, `problem.chgmqobj`, `problem.getqobj`.

problem.addrows

Purpose

Adds rows and their coefficient to the problem.

Synopsis

```
problem.addrows (qrtype, rhs, mstart, mclind, dmatval, range = None, names = None)
```

Arguments

`qrtype`	Character array containing the row types: `L` indicates a $\leq$ row; `G` indicates $\geq$ row; `E` indicates an = row. `R` indicates a range constraint; `N` indicates a nonbinding constraint.
`rhs`	Array containing the right hand side elements.
`mstart`	Array containing the offsets in the `mclind` and `dmatval` arrays of the start of the elements for each row.
`mclind`	Array containing the (contiguous) column indices for the elements in each row.
`dmatval`	Array containing the (contiguous) element values.
`range`	(optional) Array containing the row range elements. Optional. 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.

Example

Suppose the current problem is:

maximize:	2x + y + 3z		
subject to:	x + 4y + 2z	$\leq$	24
	y + z	$\leq$	5
	3x + y	$\leq$	20
	x + y + 3z	$\leq$	9

Then the following adds the row $8x + 9y + 10z \leq 25$ to the problem and names it `NewRow`:

```
p = xpress.problem ()
p.addrows (['L'], [25], None, [0,3], [0,1,2],
           dmatval = [8, 9, 10], 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.addSOS

Purpose

Adds one or more Special Ordered Set (SOS) to the problem.

Synopsis

```
problem.addSOS (s1, s2, ...)
```

Argument

`s1,s2...` Special Ordered Sets defined prior to the call or (see example below) defined directly in the call.

Example

```
N = 20
x = [xpress.var () for i in range (N)]
p = xpress.problem ()
p.addVariable (x)
s = xpress.sos ([x], [i+2 for i in range (N)])
p.addSOS (s)
p.addSOS ([x[0], x[2]], [4,6])
```

Further information

All arguments can be single SOSs or lists, tuples, or NumPy arrays of SOSs created as `xpress.sos` objects. As for constraints, a SOS does not need to be declared prior to being added as an argument.

problem.addtolsets

Purpose

Add sets of standard tolerance values to an SLP problem

Synopsis

```
problem.addtolsets (tol)
```

Argument

`slptol` Array of 9*h* 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 zero 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.addVariable

Purpose

Adds one or more variables to the problem.

Synopsis

```
problem.addVariable (v1, v2, ...)
```

Argument

`v1,v2...` Variables or list/tuples/array of variables created with the `xpress.var()` call.

Example

```
x = xpress.var (vartype = xpress.binary)
Y = [xpress.var () for i in range (20)]
p = xpress.problem ()
p.addVariable (x, Y)
```

Further information

All arguments can be single variables or lists, tuples, or NumPy arrays of variables created as `xpress.var` objects.

problem.addvars

Purpose

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 variable as follows:
`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   = [0,2]
initvalue = [0,1.42]

p.addvars (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 `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

Returns various measures for the stability of the current basis, including the basis condition number.

Synopsis

```
x = problem.basisstability (type, norm, ifscaled)
```

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.
`ifscaled`	If the stability values are to be calculated in the scaled or the unscaled matrix.	

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.btran

Purpose

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 `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.calcobjective

Purpose

Returns the objective value of a given solution.

Synopsis

```
objval = problem.calcobjective (solution)
```

Argument

`solution` Array of length `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.calcreducedcosts

Purpose

Returns the reduced cost values for a given (row) dual solution.

Synopsis

```
problem.calcreducedcosts (duals, solution, calculateddjs)
```

Arguments

`duals`	Array of length `ROWS` that holds the dual solution to calculate the reduced costs for.
`solution`	Optional array of length `COLS` that holds the primal solution. This is necessary for quadratic problems.
`calculateddjs`	Array of length `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.calcslacks

Purpose

Calculates the row slack values for a given solution.

Synopsis

```
problem.calcslacks (solution, calculatedslacks)
```

Arguments

`solution` Array of length `COLS` that holds the solution to calculate the slacks for.

`calculatedslacks` Array of length `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.calcsolinfo

Purpose

Returns the required property of a solution, like maximum infeasibility of a given primal and dual solution.

Synopsis

```
val = problem.calcsolinfo (solution, dual, property)
```

Arguments

`solution`	Array of length `COLS` that holds the solution.	
`dual`	Array of length `ROWS` that holds the dual solution.	
`property`	`xpress.solinfo_absprimalinfeas`	absolute primal infeasibility.
	`xpress.solinfo_relprimalinfeas`	relative primal infeasibility.
	`xpress.solinfo_absdualinfeas`	absolute dual infeasibility.
	`xpress.solinfo_reldualinfeas`	relative dual 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.cascade

Purpose

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 (colnum, None, None, None, None,
  None, None, value, None, None, None,
  None)

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.cascadeorder`.

Related topics

`problem.cascadeorder`

problem.cascadeorder

Purpose

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.cascade` to calculate consistent values for the dependent variables.

```
int ColNum;
for colnum in range (1, nCol):
  p.chgvar (colnum, None, None, None, None,
            None, None, [DefaultValue [ColNum]], None, None, None,
            None)
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.cascade`

problem.chgbounds

Purpose

Changes the bounds on columns in the problem.

Synopsis

```
problem.chgbounds (mindex, qbtype, bnd)
```

Arguments

`mindex`	Array containing the indices of the columns on which the bounds will change.
`qbtype`	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.
`bnd`	Array giving the new bound values.

Example

The following changes column `0` of the current problem to have an upper bound of `0.5`:

```
p.chgbounds ([0,1,2],['L','U','B'],[1,2,3])
```

Further information

1. A column index may appear twice in the `mindex` 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 double constant `xpress.infinity` can be used to represent plus and minus infinity in the bound (`bnd`) 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.chgcoef

Purpose

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 (irow, icol, dval)
```

Arguments

`irow`	Row index for the coefficient.
`icol`	Column index for the coefficient.
`dval`	New value for the coefficient. If `dval` 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 = xpress.var ()
c = x + x**2 <= 3
p.addVariable (x)
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.chgcoltype

Purpose

Changes the type of a column in the problem.

Synopsis

```
problem.chgcoltype (mindex, qctype)
```

Arguments

`mindex` Array containing the indices of the columns.

`qctype` 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 bound will be set to `1.0`.

Example

The following changes the type of variable `x` from binary to integer:

```
p = xpress.problem ()
x = xpress.var (vartype = xp.binary)
p.addVariable (x)
p.chgcoltype ([x],['I'])
```

Further information

1. The column types can only be changed before the global 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.

Related topics

`problem.addcols`, `problem.chgrowtype`, `problem.getcoltype`.

problem.chgcascadenlimit

Purpose

Set a variable specific cascade iteration limit

Synopsis

```
problem.chgcascadenlimit (icol, cascadenlimit)
```

Arguments

`icol` The index of the column corresponding to the SLP variable for which the cascading limit is to be imposed.

`cascadenlimit` 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.cascadeorder`

problem.chgccoef

Purpose

Add or change a single matrix coefficient using a character string for the formula

Synopsis

```
problem.chgccoef (rowindex, colindex, factor, formula)
```

Arguments

`rowindex`	The index of the matrix row for the coefficient.
`colindex`	The index of the matrix column for the coefficient.
`factor`	Constant multiplier for the formula. If `factor` is `None`, a value of 1.0 will be used.
`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 `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.addcoefs`, `problem.delcoef`, `problem.chgnlcoef`, `problem.getcoefformula`, `problem.loadcoefs`

problem.chgdeltatype

Purpose

Changes the type of the delta assigned to a nonlinear variable

Synopsis

```
problem.chgdeltatype (vars, deltatypes, values)
```

Arguments

`Vars`	Indices of the variables to change the deltas for.
`DeltaTypes`	Type if the delta variable: `0` Differentiable variable, default. `1` Variable defined over the grid size given in `values`. `2` Variable where a minimum perturbation size given in `values` may be required before a significant change in the problem is achieved. `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.

Further information

Changing the delta type of a variables makes the variable nonlinear.

Related topics

problem.chgdf

Purpose

Set or change a distribution factor

Synopsis

```
problem.chgdf (colindex, rowindex, value)
```

Arguments

`colindex`	The index of the column whose distribution factor is to be set or changed.
`rowindex`	The index of the row where the distribution applies.
`value`	Address of a double precision variable holding 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

Changes semi-continuous or semi-integer lower bounds, or upper limits on partial integers.

Synopsis

```
problem.chgglblimit (mindex, dlimit)
```

Arguments

`mindex`	Array containing the indices of the semi-continuous, semi-integer or partial integer columns that should have their limits changed.
`dlimit`	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.getglobal`.

problem.chgmcoef

Purpose

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 (mrow, mcol, dval)
```

Arguments

`mrow`	Array containing the row indices of the coefficients to be changed.
`mcol`	Array containing the column indices of the coefficients to be changed.
`dval`	Array containing the new coefficient values. If an element of `dval` 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.chgmqobj

Purpose

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 (mqcol1, mqcol2, dval)
```

Arguments

`mqcol1`	Array containing the column index of the first variable in each quadratic term.
`mqcol2`	Array containing the column index of the second variable in each quadratic term.
`dval`	New values for the coefficients. If an entry in `dval` 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: $[4x_1^2 + 6x_1x_2$

```
p.chgmqobj ([x1,x1], [x1,x2], [4,3])
```

Further information

1. The columns in the arrays `mqcol1` and `mqcol2` must already exist in the matrix. If the columns do not exist, they must be added.
2. `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.chgnlcoef

Purpose

Add or change a single matrix coefficient using a parsed or unparsed formula

Synopsis

```
problem.chgnlcoef (rowindex, colindex, factor, parsed, type, value)
```

Arguments

`rowindex`	The index of the matrix row for the coefficient.
`colindex`	The index of the matrix column for the coefficient.
`factor`	Address of a double precision variable holding the constant multiplier for the formula. If `Factor` is `None`, a value of 1.0 will be used.
`parsed`	Integer 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.xslp_op_ifun,  xp.xslp_op_var, xp.xslp_op_rb, xp.xslp_op_eof]
value = [xp.xslp_ifun_sin, 1,              0,             0]

Factor = 2.5
p.chgnlcoef (1, 3, Factor, 0, type, value)
```

`problem.getindex` is used to retrieve the index for the internal function `sin`. The "nocase" version matches the function name regardless of the (upper or lower) case of the name. Tokens of type `xpress.xslp_op_var` always count from 1, so `Col1` is 1. The formula is written in unparsed form (`parsed` = 0) 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 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.addcoefs`, `problem.chgccoef`, `problem.delcoefs`, `problem.getcoefformula`, `problem.loadcoefs`

problem.chgobj

Purpose

Change the objective function coefficients.

Synopsis

```
problem.chgobj (mindex, obj)
```

Arguments

`mindex`	Arraycontaining the indices of the columns 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.
`obj`	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.chgobjsense

Purpose

Changes the problem's objective function sense to minimize or maximize.

Synopsis

```
problem.chgobjsense (sense)
```

Argument

`objsense`	`xpress.minimize` or `xpress.maximize` to change into a minimization or maximization problem, respectively.

Example

Changing three coefficients of the objective function with `chgobj`:

```
p.chgobjsense (xpress.maximize) # optimize in this general direction
```

Related topics

`problem.lpoptimize`, `problem.mipoptimize`.

problem.chgqobj

Purpose

Change a single quadratic coefficient in the objective function corresponding to the variable pair `(icol,jcol)` of the Hessian matrix.

Synopsis

```
problem.chgqobj (icol, jcol, dval)
```

Arguments

`icol`	Column index for the first variable in the quadratic term.
`jcol`	Column index for the second variable in the quadratic term.
`dval`	New value for the coefficient in the quadratic Hessian matrix. If an entry in `dval` is `0`, the corresponding entry will be deleted.

Example

The following code adds the terms $[6x_1^2 + 3x_1x_2 + 3x_2x_1] / 2$ to the objective function:

```
p.chgqobj (x1, x1, 6)
p.chgqobj (x1, x2, 3)
```

Further information

1. The columns `icol` and `jcol` must already exist in the matrix..
2. If `icol` is not equal to `jcol`, then both the matrix elements `(icol, jcol)` and `(jcol, icol)` are changed to leave the Hessian symmetric.

Related topics

`problem.chgcoef`, `problem.chgmcoef`, `problem.chgmqobj`, `problem.chgobj`, `problem.getqobj`.

problem.chgqrowcoeff

Purpose

Changes a single quadratic coefficient in a row.

Synopsis

```
problem.chgqrowcoeff (irow, icol, jcol, dval)
```

Arguments

`irow`	Index of the row where the quadratic matrix is to be changed.
`icol`	First index of the coefficient to be changed.
`jcol`	Second index of the coefficient to be changed.
`dval`	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

Changes right–hand side values of the problem.

Synopsis

```
problem.chgrhs (mindex, rhs)
```

Arguments

`mindex`	Array containing the indices of the rows 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.chgrhsrange

Purpose

Change the range for one or more rows of the problem.

Synopsis

```
problem.chgrhsrange (mindex, rng)
```

Arguments

`mindex`	Array containing the indices of the rows on which the range elements will change.
`rng`	Array containing the range values.

Example

Here, the constraint `cons1` x + y $\leq$ 10 is changed to 8 $\leq$ x + y $\leq$ 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
$r \geq 0$	= *b*, $\leq b$	$b - r \leq \sum a_j x_j \leq b$
$r \geq 0$	$\geq b$	$b \leq \sum a_j x_j \leq b + r$
$r < 0$	= *b*, $\leq b$	$b \leq \sum a_j x_j \leq b - r$
$r < 0$	$\geq b$	$b + r \leq \sum a_j x_j \leq b$

Related topics

`problem.chgcoef`, `problem.chgmcoef`, `problem.chgrhs`, `problem.getrhsrange`.

problem.chgrowstatus

Purpose

Change the status setting of a constraint

Synopsis

```
problem.chgrowstatus (rowindex, status)
```

Arguments

`rowindex` 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.getrow`) 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.getrowstatus`

problem.chgrowtype

Purpose

Changes the type of a row in the problem.

Synopsis

```
problem.chgrowtype (mindex, qrtype)
```

Arguments

`mindex`	Array containing the indices of the rows.
`qrtype`	Character array giving the new row types: `L` indicates a $\leq$ row; `E` indicates an = row; `G` indicates a $\geq$ 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.chgrowwt

Purpose

Set or change the initial penalty error weight for a row

Synopsis

```
problem.chgrowwt (rowindex, value)
```

Arguments

`RowIndex`	The index of the row whose weight is to be set or changed.
`Value`	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.rowinfo`.

Related topics

`problem.getrowwt`, `problem.rowinfo`

problem.chgtolset

Purpose

Add or change a set of convergence tolerances used for SLP variables

Synopsis

```
problem.chgtolset (ntol, status, tols)
```

Arguments

`ntol`	Tolerance set for which values are to be changed. A zero value for `nSLPTol` will create a new set.
`status`	Address of an integer holding a bitmap describing which tolerances are active in this set. See below for the settings.
`tols`	Array of 9 double precision 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

Define a column as an SLP variable or change the characteristics and values of an existing SLP variable

Synopsis

```
problem.chgvar (colindex, detrow, initstepbound, stepbound, penalty, damp, initvalue,
       value, tolset, history, converged, vartype)
```

Arguments

`colindex` The index of the matrix column.

`detrow` Address of an integer holding the index of the determining row. Use -1 if there is no determining row. May be `None` if not required.

`initstepbound` Address of a double precision variable holding the initial step bound size. May be `None` if not required.

`stepbound` Address of a double precision variable holding the current step bound size. Use zero to disable the step bounds. May be `None` if not required.

`penalty` Address of a double precision variable holding the weighting of the penalty cost for exceeding the step bounds. May be `None` if not required.

`damp` Address of a double precision variable holding the damping factor for the variable. May be `None` if not required.

`initvalue` Address of a double precision variable holding the initial value for the variable. May be `None` if not required.

`value` Address of a double precision variable holding the current value for the variable. May be `None` if not required.

`tolset` Address of an integer holding 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` Address of an integer holding the history value for this variable. May be `None` if not required.

`converged` Address of an integer holding the convergence status for this variable. May be `None` if not required.

`vartype` Address of an integer holding 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 (25, None, None, None, None,
          None, 1.42, None, 2,
          None, None, 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.construct`. 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

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.writeprob ("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.copy

Purpose

Obtains a copy of a problem.

Synopsis

```
p = problem.copy ()
```

Example

```
 p = xpress.problem () x = [xpress.var () for
_ in range (10)] p.addVariable (x) p.addConstraint (xpress.Sum (x) <=
10) p2 = p.copy () # add a constraint that won't be in p
p2.addConstraint (xpress.Sum (x) >= 6) # x[0] is deleted from p2
p2.delVariable (x[0])
```

Further information

The objects of the copied problem (variables, constraints, SOSs) are the same as the source problem, i.e., the one of which a copy was created. Therefore, any object that existed in the source problem can be addressed and used in the copy problem.

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.copycontrols

Purpose

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.copycallbacks`.

problem.delcoefs

Purpose

Delete coefficients from the current problem

Synopsis

```
problem.delcoefs (rowindex, colindex)
```

Arguments

`rowindex`	Row indices of the SLP coefficients to delete.
`colindex`	Column indices of the SLP coefficients to delete.

Related topics

`problem.addcoefs`, `problem.chgnlcoef`, `problem.chgccoef`, `problem.getcoefformula`, `problem.getccoef`, `problem.loadcoefs`

problem.delConstraint

Purpose

Delete one or more constraints from the problem.

Synopsis

```
problem.delConstraint (c1, c2, ...)
```

Example

```
N = 20
x = [xpress.var () for i in range (N)]
p = xpress.problem ()
p.addVariable (x)
p.addConstraint (x[i] >= x[i+1] for i in range (N-1))
p.delConstraint (2) # deletes x[2] >= x[3]
```

Further information

1. All arguments can be single constraints or lists, tuples, or NumPy arrays of variables. They can also be constraint indices (from 0 to ROWS-1). The index of variables, constraints, and SOSs can be obtained with problem.getIndex.
2. Indicator constraints are indexed as constraints, hence they can also be deleted with this function.

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 `delcpcuts`.

Synopsis

```
problem.delcpcuts (itype, interp, cutind)
```

Arguments

`itype` User defined cut type to match against.

`interp` Way in which the cut `itype` 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 `itype`;
- `3` treat cut types as bit maps - delete if all bits match those set in `itype`.

`cutind` Array containing the cuts which are to be deleted.

Related topics

`problem.addcuts`, `problem.delcuts`, `problem.loadcuts`, Section "Working with the cut manager" of the Xpress 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.

Synopsis

```
problem.delcuts (ibasis, itype, interp, delta, cutind)
```

Arguments

ibasis	Ensures the basis will be valid if set to 1. If set to 0, cuts with non-basic slacks may be deleted.
itype	User defined type of the cut to be deleted.
interp	Way in which the cut itype 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 itype; 3 treat cut types as bit maps - delete if all bits match those set in itype.
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 ibasis 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 ibasis 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.

Related topics

problem.addcuts, problem.delcpcuts, problem.getcutlist, problem.loadcuts, Section "Working with the cut manager" of the Xpress Optimizer reference manual.

problem.delqmatrix

Purpose

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.delSOS

Purpose

Delete one or more SOSs from the problem.

Synopsis

```
problem.delSOS (s1, s2, ...)
```

Example

```
N = 20
x = [xpress.var () for i in range (N)]
p = xpress.problem ()
p.addVariable (x)
s = xpress.sos (x, i+1 for i in range (N))
p.addSOS (s)
p.delSOS (s)
```

Further information

All arguments can be single SOSs or lists, tuples, or NumPy arrays of SOSs. They can also be constraint indices (from 0 to `ROWS`-1). The index of variables, constraints, and SOSs can be obtained with `problem.getIndex`.

problem.deltolsets

Purpose

Delete tolerance sets from the current problem

Synopsis

```
problem.deltolsets (index)
```

Argument

`tolsetindex` Indices of tolerance sets to delete.

Related topics

`problem.addtolsets`, `problem.chgtolset`, `problem.gettolset`, `problem.loadtolsets`

problem.delVariable

Purpose

Delete one or more variables from the problem.

Synopsis

```
problem.delVariable (x1, x2, ...)
```

Example

```
N = 20

x = [xpress.var () for i in range (N)]
p = xpress.problem ()
p.addVariable (x)
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

All arguments can be single variables or lists, tuples, or NumPy arrays of variables. They can also be variable indices (from 0 to `COLS`-1). The index of variables, constraints, and SOSs can be obtained with `problem.getIndex`.

problem.delvars

Purpose

Convert SLP variables to normal columns. Variables must not appear in SLP structures

Synopsis

```
problem.delvars (index)
```

Argument

`colindex` 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

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.estimaterowdualranges

Purpose

Performs a dual side range sensitivity analysis, i.e. calculates estimates for the possible ranges for dual values.

Synopsis

```
problem.estimaterowdualranges (rowIndices, iterationLimit, minDualActivity,
       maxDualActivity)
```

Arguments

`rowIndices` Row indices to analyze.

`iterationLimit` Effort limit expressed as simplex iterations per row.

`minDualActivity` Estimated lower bounds on the possible dual ranges.

`maxDualActivity` 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.evaluatecoef

Purpose

Evaluate a coefficient using the current values of the variables

Synopsis

```
value = problem.evaluatecoef (rowindex, colindex)
```

Arguments

`rowindex`	Integer index of the row.
`colindex`	Integer index of the column.
`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 (5, None, None, None, None,
          None, None, 1.42, None, None, None,
          None)
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.evaluateformula` `problem.getvar`

problem.evaluateformula

Purpose

Evaluate a formula using the current values of the variables

Synopsis

```
result = problem.evaluateformula (parsed, type, value)
```

Arguments

`parsed`	integer indicating whether the formula of the item is in internal unparsed format (`Parsed`=0) or parsed (reverse Polish) format (`Parsed`=1).
`type`	Integer array of token types for the formula.
`value`	Double 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.xslp_op_var, xp.xslp_op_var, xp.xslp_op_op,       xp.xslp_op_eof]
value = [3,              6,              xp.xslp_ifun_divide, 0]

value = p.evaluateformula (1, type, value)
```

Further information

The formula in `Type` and `Value` 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.evaluatecoef`

problem.filesol

Purpose

Prints the last SLP iteration's solution to file

Synopsis

```
problem.filesol (filename)
```

Argument

`filename` Name of the file to write the solution into

Further information

For SLP variables, the initial values are also printed.

Related topics

`problem.writeprob`

problem.fixglobals

Purpose

Fixes all the global entities to the values of the last found MIP solution. This is useful for finding the reduced costs for the continuous variables after the global variables have been fixed to their optimal values.

Synopsis

```
problem.fixglobals (ifround)
```

Argument

`ifround`	If all global entities should be rounded to the nearest discrete value in the solution before being fixed.

Example

This example performs a global search on problem `myprob` and then uses `fixglobals` before solving the remaining linear problem:

```
p.read ("myprob", "")
p.mipoptimize ()
p.fixglobals (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 global 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 `fixglobals`.

Related topics

`problem.mipoptimize`.

problem.fixpenalties

Purpose

Fixe 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 `getlpsolution` instead.

problem.ftran

Purpose

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 `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.getAttrib

Purpose

Retrieves one or more attributes of a problem.

Synopsis

```
a = problem.getAttrib (attr1, attr2, ...)
```

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

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.

problem.getattribinfo

Purpose

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.getbasis

Purpose

Returns the current basis into the user's data arrays.

Synopsis

```
problem.getbasis (rstatus, cstatus)
```

Arguments

`rstatus` Array of length `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.

`cstatus` Array of length `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.getccoef

Purpose

Retrieve a single matrix coefficient as a formula in a character string

Synopsis

```
(factor, formula) = problem.getccoef (rowindex, colindex, flen)
```

Arguments

`RowIndex`	Integer holding the row index for the coefficient.
`ColIndex`	Integer holding the column index for the coefficient.
`Factor`	Address of a double precision variable to receive the value of the constant factor multiplying the formula in the coefficient.
`Formula`	Character buffer in which the formula will be placed in the same format as used for input from a file. The formula will be null terminated.
`fLen`	Maximum length of returned formula.

Return value

`0`	Normal return.
`1`	Formula is too long for the buffer and has been truncated.
`other`	Error.

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 `flen - 1`, the formula is truncated to the last token that will fit, and the (partial) formula is terminated with a null character.

Related topics

`problem.chgccoef`, `problem.chgnlcoef`, `problem.getcoefformula`

problem.getcoef

Purpose

Returns a single coefficient in the constraint matrix.

Synopsis

```
coef = problem.getcoef (irow, icol)
```

Arguments

`irow`	Row of the constraint matrix.
`icol`	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.getcoefformula

Purpose

Retrieve a single matrix coefficient as a formula split into tokens

Synopsis

```
(factor, tokencount, type, value) = problem.getcoefformula (rowindex, colindex,
       parsed, bufsize)
```

Arguments

`rowindex`	The row index for the coefficient.
`colindex`	The column index for the coefficient.
`factor`	The value of the constant factor multiplying the formula in the coefficient.
`parsed`	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).
`bufsize`	Maximum number of tokens to return, i.e. length of the Type and Value arrays.
`tokencount`	Number of tokens returned in Type and Value.
`type`	Array to hold 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.chgccoef`, `problem.chgnlcoef`, `problem.getccoef`

problem.getcoefs

Purpose

Retrieve the list of positions of the nonlinear coefficients in the problem

Synopsis

```
problem.getcoefs (rowindex, colindex)
```

Arguments

`rowindices`	Row positions of the coefficients. May be `None` if not required.
`colindices`	Column positions of the coefficients. May be `None` if not required.

Related topics

`problem.getccoef`, `problem.getcoefformula`

problem.getcolinfo

Purpose

Get current column information.

Synopsis

```
problem.getcolinfo (infotype, colindex);
```

Arguments

`InfoType`	Type of information (see below)
`ColIndex`	Index of the column whose information is to be handled
`Info`	Address of information to be set or 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 column information handling:

`xpress.colinfo_value`	Get the current value of the column
`xpress.colinfo_rdj`	Get the current reduced cost of the column
`xpress.colinfo_deltaindex`	Get the delta variable index associated to the column
`xpress.colinfo_delta`	Get the delta value (change since previous value) of the column
`xpress.colinfo_deltadj`	Get the delta variables reduced cost
`xpress.colinfo_updaterow`	Get the index of the update (or step bound) row associated to the column
`xpress.colinfo_sb`	Get the step bound on the variable
`xpress.colinfo_sbdual`	Get the dual multiplier of the step bound row for the variable

problem.getcols

Purpose

Returns the nonzeros in the constraint matrix for the columns in a given range.

Synopsis

```
problem.getcols (mstart, mrwind, dmatval, size, first, last)
```

Arguments

`mstart`	Array which will be filled with the indices indicating the starting offsets in the `mrwind` and `dmatval` arrays for each requested column. It must be of length at least `last-first+2`. Column `i` starts at position `mstart[i]` in the `mrwind` and `dmatval` arrays, and has `mstart[i+1]-mstart[i]` elements in it. May be `None` if not required.
`mrwind`	Array of length `size` which will be filled with the row indices of the nonzero coefficents for each column. May be `None` if not required.
`dmatval`	Array of length `size` which will be filled with the nonzero coefficient values. May be `None` if not required.
`size`	The size of the `mrwind` and `dmatval` 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 `mstart` vector of the problem:

```
p = xpress.problem ()
p.read ("example", "l")
mstart = []
p.getcols (mstart, 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 `mstart`, `mrwind` and `dmatval` by `None`, as in the example. In this case, `size` 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.getcoltype

Purpose

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.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.

Synopsis

```
c = problem.getControl (ctrl1, ctrl2, ...)
```

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.

Related topics

problem.setControl.

problem.getcontrolinfo

Purpose

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.getcpcutlist

Purpose

Returns a list of cut indices from the cut pool.

Synopsis

```
ncuts = problem.getcpcutlist (itype, interp, delta, size, cutind, viol)
```

Arguments

`itype`	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 `itype`; `3` treat cut types as bit maps - get cut if all bits match those set in `itype`.
`delta`	Only those cuts with a signed violation greater than delta will be returned.
`size`	Maximum number of cuts to be returned.
`mcutind`	Array of length `size` where the cuts will be returned.
`dviol`	Array of length `size` 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 size of the (signed) violation required. If unviolated cuts are required as well, `delta` may be set to `_MINUSINFINITY` which is defined in the library header file.
2. If the number of active cuts is greater than `size`, only `size` cuts will be returned. Otherwise only the existing cuts will be used to fill in the positions of `mcutind`.
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`, Section "Working with the cut manager" of the Xpress Optimizer reference manual.

problem.getcpcuts

Purpose

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 `mcols` and `dmatval` 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 `mstart` parameter.

Synopsis

```
problem.getcpcuts (mindex, size, type, rtype, mstart, mcols, matval, drhs)
```

Arguments

`mindex`	Array of length `ncuts` containing the cuts.
`size`	Maximum number of column indices of the cuts to be returned.
`type`	Array of length at least `ncuts` where the cut types will be returned.
`rtype`	Character array of length at least `ncuts` where the sense of the cuts (`L`, `G`, or `E`) will be returned.
`mstart`	Array of length at least `ncuts+1` containing the offsets into the `mcols` and `dmatval` arrays. The last element indicates where cut `ncuts+1` would start.
`cols`	Array of length `size` where the column indices of the cuts will be returned.
`matval`	Array of length `size` where the matrix values will be returned.
`rhs`	Array of length at least `ncuts` where the right hand side elements for the cuts will be returned.

Related topics

`problem.getcpcutlist`, `problem.getcutlist`, Section "Working with the cut manager" of the Xpress Optimizer reference manual.

problem.getcutlist

Purpose

Retrieves a list of cuts for the cuts active at the current node.

Synopsis

```
problem.getcutlist (itype, interp, size, cutind)
```

Arguments

`itype` 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 `itype`;
- `3` treat cut types as bit maps - get cut if all bits match those set in `itype`.

`size` Maximum number of cuts to be retrieved.

`cutind` Array of length `size` where the pointers to the cuts will be returned.

Further information

If the number of active cuts is greater than `size`, then `size` 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`, Section "Working with the cut manager" of the Xpress Optimizer reference manual.

problem.getcutmap

Purpose

Returns 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.

Synopsis

```
problem.getcutmap (cuts, cutmap)
```

Arguments

`cuts`	Array with the cuts for which the row index is requested.
`cutmap`	Array of length `ncuts`, where the row indices are returned.

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`, Section "Working with the cut manager" of the Xpress Optimizer reference manual.

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).

Synopsis

```
slack = problem.getcutslack (cut)
```

Arguments

`cuts`	Pointer of the cut for which the slack is to be calculated.
`slack`	Double pointer where the value of the slack is returned.

Related topics

`problem.getcpcutlist`, `problem.delcpcuts`, `problem.getcutlist`, `problem.loadcuts`, `problem.getcutmap`, `problem.getcpcuts`, Section "Working with the cut manager" of the Xpress Optimizer reference manual.

problem.getdirs

Purpose

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 (mcols, mpri, qbr, dupc, ddpc)
```

Arguments

`mcols` 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.

`mpri` Array containing the priorities for the columns and sets. May be `None` if not required.

`qbr` 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.

`dupc` Array containing the up pseudo costs for the columns and sets. May be `None` if not required.

`ddpc` 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 `xprobem.attributes.mipents`.

Related topics

`problem.loaddirs`, `problem.loadpresolvedirs`.

problem.getdf

Purpose

Get a distribution factor

Synopsis

```
value = problem.getdf (colindex, rowindex)
```

Arguments

`colindex`	The index of the column whose distribution factor is to be retrieved.
`rowindex`	The index of the row 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.getdtime

Purpose

Retrieve a double precision time stamp in seconds

Synopsis

```
seconds = problem.getdtime ()
```

Argument

`seconds` Address of double precision variable of the time in seconds.

Example

The following example measures the elapsed time to read a problem:

```
start = p.getdtime ()
p.read ("NewMat","")
finish = p.getdtime ();
print ("Elapsed time to read = {0} secs".format (finish - start))
```

Further information

The timing information returned is provided by the operating system and is typically accurate to no more than 1 millisecond.

The clock is not initialized when Xpress Nonlinear starts, so it is necessary to save an initial time and then measure all times by difference.

Related topics

`problem.gettime`

problem.getDual

Purpose

Return the dual for all constraints of the problem w.r.t. the solution found by `solve()`; this only works on continuous optimization problems.

Synopsis

```
d = problem.getDual ()
```

Example

```
p.solve ()
print ("duals:", p.getDual ())
```

Related topics

`problem.getlpsol`, `problem.getSlack`, `problem.getRCost`, `problem.getProbStatus`, `problem.getProbStatusString`.

problem.getdualray

Purpose

Retrieves a dual ray (dual unbounded direction) for the current problem, if the problem is found to be infeasible.

Synopsis

```
problem.getdualray (dray)
```

Argument

`dray` Array of length `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.lp_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.getglobal

Purpose

Retrieves global information about a problem. It must be called before `problem.mipoptimize` if the presolve option is used.

Synopsis

```
problem.getglobal (qgtype, mgcols, dlim, qstype, msstart, mscols, dref)
```

Arguments

`qgtype` 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.

`mgcols` Array where the column indices of the global entities will be returned.

`dlim` 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).

`qstype` Character array where the set types will be returned. The set types will be one of:
- `1` SOS1 type sets;
- `2` SOS2 type sets.

`msstart` Array where the offsets into the `mscols` and `dref` 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 `mscols` and `dref` arrays.

`mscols` Array of length `SETMEMBERS` where the columns in each set will be returned.

`dref` Array of length `SETMEMBERS` where the reference row entries for each member of the sets will be returned.

Example

The following obtains the SOS information:

```
qstype = []
mstart = []
mscols = []
dref   = []
p.getglobal (None, None, None, qstype, mstart, mscols, dref)
```

Further information

All arguments may be `None` if not required.

Related topics

`problem.loadproblem`.

problem.getiisdata

Purpose

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 (num, miisrow, miiscol, constrainttype, colbndtype, duals, rdcs,
       isolationrows, isolationcols)
```

Arguments

`num` The ordinal number of the IIS to get data for.

`miisrow` Indices of rows in the IIS. Can be `None` if not required.

`miiscol` Indices of bounds (columns) in the IIS. Can be `None` if not required.

`constrainttype` 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;
- `I` for an indicator row.

Can be `None` if not required.

`colbndtype` 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.

`rdcs` 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.

```
miisrow = []
miiscol = []
constrainttype = []
colbndtype = []
duals = []
rdcs = []
```

```
isolationrows = []
isolationcols = []
p.getiisdata (1, miisrow, miiscol, constrainttype, colbndtype,
              duals, rdcs, isolationrows, isolationcols)
```

Further information

1. IISs are numbered from 1 to `NUMIIS`. Index number 0 refers to the IIS approximation.
2. If `miisrow` and `miiscol` both are `None`, only the `rownumber` and `colnumber` are returned.
3. The arrays may be `None` if not required. However, arrays `constrainttype`, `duals` and `isolationrows` are only returned if `miisrow` is not `None`. Similarly, arrays `colbndtype`, `rdcs` and `isolationcols` are only returned if `miiscol` is not `None`.
4. For the initial IIS approximation (`num` = 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 `miisrow` array.
6. For non-LP IISs, some column indices may appear more than once in the `miiscol` 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.iisclear`, `problem.iisfirst`, `problem.iisisolations`, `problem.iisnext`, `problem.iisstatus`, `problem.iiswrite`.

problem.getIndex

Purpose

Returns the numerical index for a specified row, column, or set of the optimizer.

Synopsis

```
ind = problem.getIndex (obj)
```

Argument

`obj` Python object with the column, row, or SOS

Example

The following example adds a constraint to a problem and then retrieves its index:

```
x = xpress.var ()
c = x**2 + 2*x >= 5
p.addVariable (x)
p.addConstraint (c)
print ("c has index", p.getIndex (c))
```

Related topics

`problem.getIndexFromName`, `problem.getVariable`, `problem.getConstraint`.

problem.getIndexFromName

Purpose

Returns the index for a specified row or column name.

Synopsis

```
ind = problem.getIndexFromName (type, name)
```

Arguments

`type`	1	if a row index is required;
	2	if a column index is required.
`name`	String containing name of the item sought.	

Example

The following example retrieves the index of column "`xnew`":

```
x = xpress.var (name = 'xnew')
[...]
print ("variable's index: ", p.getIndexFromName ('xnew'))
```

Related topics

`problem.getIndexFromName`, `problem.getVariable`, `problem.getConstraint`.

problem.getindicators

Purpose

Returns the indicator constraint condition (indicator variable and complement flag) associated to the rows in a given range.

Synopsis

```
problem.getindicators (inds, comps, first, last)
```

Arguments

`inds` Array of length `last-first+1` where the column indices of the indicator variables are to be placed.

`comps` 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 `inds` 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.

```
inds = []
comps = []
p.getindicators (inds, comps, 2, 4)
print ("indices:",          inds)
print ("complement flags:", comps)
```

Related topics

`problem.setindicators`.

problem.getinfeas

Purpose

Returns a list of infeasible primal and dual variables.

Synopsis

```
problem.getinfeas (mx, mslack, mdual, mdj)
```

Arguments

`mx`	Array to store the primal infeasible variables. May be `None` if not required.
`mslack`	Array to store the primal infeasible rows. May be `None` if not required.
`mdual`	Array to store the dual infeasible rows. May be `None` if not required.
`mdj`	Array to store the dual infeasible variables. May be `None` if not required.

Example

```
mx = []
mslack = []
p.getinfeas (mx, mslack, None, None)
print ("getinfeas --> mx and mslack:", mx, mslack)
```

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.iisclear`, `problem.iisfirst`, `problem.iisisolations`, `problem.iisnext`, `problem.iisstatus`, `problem.iiswrite`.

problem.getlasterror

Purpose

Returns the error message corresponding to the last error triggered by a library function.

Synopsis

```
s = problem.getlasterror ()
```

Example

The following shows how this function might be used in error-checking:

```
p.solve ()
print ("Current error status:", p.getlasterror ())
```

Related topics

ERRORCODE, problem.addcbmessage, problem.setlogfile.

problem.getlb

Purpose

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.getlpsol

Purpose

Used to obtain the LP solution values following optimization.

Synopsis

```
problem.getlpsol (x, slack, dual, dj)
```

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.
`dual`	Array to store the values of the dual variables ($c_B^T B^{-1}$). May be `None` if not required.
`dj`	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 commands will get the LP solution (`x`) at the top node of a MIP and the optimal MIP solution (`y`):

```
p.mipoptimize ("l") # only solve the LP relaxation
x = []
p.getlpsol (x)
print ("root LP solution:", x)
p.mipoptimize () # solve the MIP problem
p.getmipsol (x)
print ("final MIP solution", x)
```

Further information

1. If called during a global callback the solution of the current node will be returned.
2. When an integer solution is found during a global 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`.
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.getpresolvesol`, `problem.getmipsol`, `problem.writeprtsol`, `problem.writesol`.

problem.getmessagestatus

Purpose

Returns the current suppression status of a message: non-zero if the message is not suppressed; `0` otherwise.

Synopsis

```
status = problem.getmessagestatus (errcode)
```

Argument

`errcode` 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.getmessagetype

Purpose

Retrieve the message type corresponding to a message number

Synopsis

```
type = problem.getmessagetype (code)
```

Arguments

`code`	The message number.
`type`	The message type.

Example

The following example retrieves the last error message and finds its type.

```
(code, a) = p.getlasterror ()
type = p.getmessagetype (code)
print ("Error of type ", type)
```

Further information

The possible values returned in `type` are:

0	no such message number
1	information
3	warning
4	error

Related topics

`problem.getlasterror`

problem.getmipsol

Purpose

Used to obtain 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 commands will get the solution (`x`) of the last MIP solution for a problem:

```
x = []
p.mipoptimize ()
p.getmipsol (x)
print ("solution:", x)
```

Related topics

`problem.getpresolvesol`, `problem.writeprtsol`, `problem.writesol`.

problem.getmqobj

Purpose

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 (mstart, mclind, dobjval, size, first, last)
```

Arguments

`mstart`	Array which will be filled with indices indicating the starting offsets in the `mclind` and `dobjval` arrays for each requested column. It must be length of at least `last-first+2`. Column `i` starts at position `mstart[i]` in the `mrwind` and `dmatval` arrays, and has `mstart[i+1]-mstart[i]` elements in it. May be `None` if `size` is `0`.
`mclind`	Array which will be filled with at most `size` column indices of the nonzero elements in the lower triangular part of `Q`. May be `None` if `size` is `0`.
`dobjval`	Array which will be filled with at most `size` nonzero element values. May be `None` if `size` is `0`.
`size`	The maximum number of elements to be returned (size of the arrays).
`first`	First column in the range.
`last`	Last column in the range.

Further information

The objective function is of the form $c^T x+0.5x^T Qx$ 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.getobj

Purpose

Returns the objective function coefficients for the columns in a given range.

Synopsis

```
problem.getobj (obj, first, last)
```

Arguments

`obj`	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:

```
obj = []
p.getobj (obj, 0, 4)
```

Related topics

`problem.chgobj`.

problem.getObjVal

Purpose

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 `solve()` command is run:

```
p.solve ()
print ("optimal solution:", p.getObjVal ())
```

Related topics

problem.solve.

problem.getpivotorder

Purpose

Returns the pivot order of the basic variables.

Synopsis

```
problem.getpivotorder (mpiv)
```

Argument

`mpiv` Array where the pivot order will be returned.

Example

The following returns the pivot order of the variables into an array `pPivot` :

```
mpiv = []
p.getpivotorder (mpiv)
```

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.getpivots

Purpose

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 `in` entered the basis; and an integer where the actual number of potential leaving variables will be returned.

Synopsis

```
dobj, npiv = problem.getpivots (in, outlist, x, maxpiv)
```

Arguments

`in`	Index of the specified row or column to enter basis.
`outlist`	Array of length at least `maxpiv` to hold list of potential leaving variables. May be `None` if not required.
`x`	Array of length `ROWS+SPAREROWS+COLS` to hold the values of all the variables that would result if `in` entered the basis. May be `None` if not required.
`maxpiv`	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 `in` 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 in `npiv`. A list of at most `maxpiv` of these candidates is returned in `outlist` which must be at least `maxpiv` long. If variable `in` 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 `dobj` and the values of the variables into `x`.

2. 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.getpivotorder`.

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.

Synopsis

```
problem.getpresolvebasis (rstatus, cstatus)
```

Arguments

`rstatus` Array of length `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 `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 global search:

```
cs = []
p = xpress.problem ()
p.read ("global1", "")
p.mipoptimize ()
p.getpresolvebasis (cstatus = cs)
```

Related topics

`problem.getbasis`, `problem.loadbasis`, `problem.loadpresolvebasis`.

problem.getpresolvemap

Purpose

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.mipoptimize ("l")
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

Returns the solution for the presolved problem from memory.

Synopsis

```
problem.getpresolvesol (x, slack, dual, dj)
```

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.
`dual`	Array to store the values of the dual variables. May be `None` if not required.
`dj`	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.mipoptimize ("l")
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 global callback the solution of the current node will be returned.
4. When an integer solution is found during a global 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

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 `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.lp_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.getProbStatus

Purpose

Returns the problem status before or after a `solve ()` command. The returned number corresponds to the problem status described in the Xpress Optimizer reference manual. If the problem is an LP, the returned value is equal to `p.attributes.lpstatus` if the problem is an LP, and to `p.attrobutes.mipstatus` if the problem is a MIP.

Synopsis

```
s = problem.getProbStatus ()
```

Example

```
p = xpress.problem ()
p.read ("example2", "")
p.solve ()
print ("solution status code: ", p.getProbStatus (), " -->",
       p.getProbStatusString ())
```

Related topics

`problem.solve`, `problem.getSolution`, `problem.getDual`, `problem.getSlack`, `problem.getRCost`, `problem.getProbStatusString`.

problem.getProbStatusString

Purpose

Returns the string corresponding to the the problem status before or after a `solve ()` command.

Synopsis

```
s = problem.getProbStatusString ()
```

Example

```
p = xpress.problem ()
p.read ("example2", "")
p.solve ()
print ("solution status code: ", p.getProbStatus (), " -->",
       p.getProbStatusString ())
```

Related topics

problem.solve, problem.getSolution, problem.getDual, problem.getSlack, problem.getRCost, problem.getProbStatus.

problem.getqobj

Purpose

Returns a single quadratic objective function coefficient corresponding to the variable pair `(icol, jcol)` of the Hessian matrix.

Synopsis

```
objqcoef = problem.getqobj (icol, jcol)
```

Arguments

`icol`	Column index for the first variable in the quadratic term.
`jcol`	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.getqrowcoeff

Purpose

Returns a single quadratic constraint coefficient corresponding to the variable pair (`icol`, `jcol`) of the Hessian of a given constraint.

Synopsis

```
coeff = problem.getqrowcoeff (row, icol, jcol)
```

Arguments

`row`	The quadratic row where the coefficient is to be looked up.
`icol`	Column index for the first variable in the quadratic term.
`jcol`	Column index for the second variable in the quadratic term.

Example

The following returns the coefficient of the $\texttt{dist}^2$ 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 $\texttt{x}^2$ 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.getqrowqmatrix

Purpose

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 (irow, mstart, mclind, dqe, size, first, last)
```

Arguments

`irow`	Index of the row for which the quadratic coefficients are to be returned.
`mstart`	Array which will be filled with indices indicating the starting offsets in the `mclind` and `dobjval` arrays for each requested column. It must be length of at least `last-first+2`. Column `i` starts at position `mstart[i]` in the `mrwind` and `dmatval` arrays, and has `mstart[i+1]-mstart[i]` elements in it. May be `None` if size is 0.
`mclind`	Array of length size which will be filled with the column indices of the nonzero elements in the lower triangular part of Q. May be `None` if size is 0.
`dqe`	Array of length size which will be filled with the nonzero element values. May be `None` if size is 0.
`size`	Maximum number of elements to be returned in `mclind` and `dqe`.
`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.getqrowqmatrixtriplets

Purpose

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 (irow, mqcol1, mqcol2, dqe)
```

Arguments

`irow`	Index of the row for which the quadratic coefficients are to be returned.
`nqelem`	Argument used to return the number of quadratic coefficients in the row.
`mqcol1`	First index in the triplets. May be `None` if not required.
`mqcol2`	Second index in the triplets. May be `None` if not required.
`dqe`	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.getqrows

Purpose

Returns the list indices of the rows that have quadratic coefficients.

Synopsis

```
problem.getqrows (qcrows)
```

Argument

`qcrows`	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.getRCost

Purpose

Return the reduced cost of all variables of the problem w.r.t. the solution found by `solve()`. This function only works on continuous optimization problems.

Synopsis

```
r = problem.getRCost ()
```

Example

```
p.solve ()
print ("reduced costs:", p.getRCost ())
```

Related topics

`problem.solve`, `problem.getlpsol`, `problem.getSolution`, `problem.getDual`, `problem.getSlack`, `problem.getProbStatus`, `problem.getProbStatusString`.

problem.getrhs

Purpose

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.getrhsrange

Purpose

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.getrowinfo

Purpose

Get current row information.

Synopsis

```
info = problem.getrowinfo (infotype, rowindex);
```

Arguments

`infotype`	Type of information (see below)
`rowindex`	Index of the row 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 `xpress.undefined`.

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

Returns the nonzeros in the constraint matrix for the rows in a given range.

Synopsis

```
problem.getrows (mstart, mclind, dmatval, size, first, last)
```

Arguments

`mstart`	Array which will be filled with the indices indicating the starting offsets in the `mclind` and `dmatval` arrays for each requested row. It must be of length at least `last-first+2`. Column `i` starts at position `mstart[i]` in the `mrwind` and `dmatval` arrays, and has `mstart[i+1]-mstart[i]` elements in it. May be `None` if not required.
`mclind`	Arrays which will be filled with at most `size` column indices of the nonzero elements for each row. May be `None` if not required.
`dmatval`	Array which will be filled with at most `size` nonzero element values. May be `None` if not required.
`size`	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.getrowstatus

Purpose

Retrieve the status setting of a constraint

Synopsis

```
status = problem.getrowstatus (rowIndex)
```

Arguments

`rowindex`	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.chgrowstatus`

problem.getrowtype

Purpose

Returns the row types for the rows in a given range.

Synopsis

```
problem.getrowtype (qrtype, first, last)
```

Arguments

`qrtype`	Character array of length `last-first+1` characters where the row types will be returned: `N` indicates a free constraint; `L` indicates a $\leq$ constraint; `E` indicates an = constraint; `G` indicates a $\geq$ 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 three rows of the problem into an array `qrt` :

```
qrt = []
p.getrowtype (qrt, 0, 3)
```

Related topics

`problem.chgrowtype`, `problem.getrows`.

problem.getrowwt

Purpose

Get the initial penalty error weight for a row

Synopsis

```
value = problem.getrowwt (rowindex)
```

Arguments

`rowindex`	The index of the row 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.getrowinfo`.

Related topics

`problem.chgrowwt`, `problem.getrowinfo`

problem.getscaledinfeas

Purpose

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 (mx, mslack, mdual, mdj)
```

Arguments

`mx`	Array to store the primal infeasible variables. May be `None` if not required.
`mslack`	Array to store the primal infeasible rows. May be `None` if not required.
`mdual`	Array to store the dual infeasible rows. May be `None` if not required.
`mdj`	Array to store the dual infeasible variables. May be `None` if not required.

Example

```
mx     = []
mslack = []
mdual  = []
mdj    = []
p.getscaledinfeas (mx, mslack, mdual, mdj)
```

Related topics

problem.getinfeas, problem.getiisdata, problem.iisall, problem.iisclear, problem.iisfirst, problem.iisisolations, problem.iisnext, problem.iisstatus, problem.iiswrite.

problem.getSlack

Purpose

Return the slack for all constraints of the problem w.r.t. the solution found by `solve()`. This function works both with continuous and mixed-integer optimization problems.

Synopsis

```
s = problem.getSlack ()
```

Example

```
p.solve ()
print ("slack:", p.getSlack ())
```

Related topics

`problem.solve`, `problem.getlpsol`, `problem.getmipsol`, `problem.getSolution`, `problem.getDual`, `problem.getRCost`, `problem.getProbStatus`, `problem.getProbStatusString`.

problem.getslpsol

Purpose

Obtain the solution values for the most recent SLP iteration

Synopsis

```
problem.getslpsol (x, slack, dual, dj)
```

Arguments

`x`	Array of length `xslp_originalcols` to hold the values of the primal variables. May be `None` if not required.
`slack`	Array of length `xslp_originalrows` to hold the values of the slack variables. May be `None` if not required.
`dual`	Array of length `xslp_originalrows` to hold the values of the dual variables. May be `None` if not required.
`dj`	Array of length `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  = []
dj   = []
p.getslpsol (val,None,None,dj)
```

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.

problem.getSolution

Purpose

Returns the solution to an optimization problem if called after the `solve()` function has terminated. This function works with both continuous and mixed-integer optimization problems.

Synopsis

```
x = problem.getSolution ()
```

Example

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

Related topics

problem.getlpsol, problem.getmipsol, problem.getDual, problem.getSlack, problem.getRCost, problem.getProbStatus, problem.getProbStatusString.

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.gettolset

Purpose

Retrieve the values of a set of convergence tolerances for an SLP problem

Synopsis

```
status = problem.gettolset (nslptol, tols)
```

Arguments

`nslptol`	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

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.getunbvec

Purpose

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.getvar

Purpose

Retrieve information about an SLP variable

Synopsis

```
(detrow, initstepbound, stepbound, penalty, damp, initvalue, value, tolset, history,
       converged, vartype, delta, penaltydelta, updaterow, oldvalue) = problem.getvar
       (colindex)
```

Arguments

`colindex` The index of the column.

`detrow` Address of an integer to receive the index of the determining row. May be `None` if not required.

`initstepbound` Address of a double precision variable to receive the value of the initial step bound of the variable. May be `None` if not required.

`stepbound` Address of a double precision variable to receive the value of the current step bound of the variable. May be `None` if not required.

`penalty` Address of a double precision variable to receive the value of the penalty delta weighting of the variable. May be `None` if not required.

`damp` Address of a double precision variable to receive the value of the current damping factor of the variable. May be `None` if not required.

`initvalue` Address of a double precision variable to receive the value of the initial value of the variable. May be `None` if not required.

`value` Address of a double precision variable to receive the current activity of the variable. May be `None` if not required.

`tolset` Address of an integer to receive the index of the tolerance set of the variable. May be `None` if not required.

`history` Address of an integer to receive the SLP history of the variable. May be `None` if not required.

`converged` Address of 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` Address of 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` Address of an integer to receive the index of the delta vector for the variable. May be `None` if not required.

`penaltydelta` Address of 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` Address of an integer to receive the index of the update row for the variable. May be `None` if not required.

`oldvalue` Address of 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 `colindex` 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.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

`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`.

`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`

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.globalsol

Purpose

Initiate the Xpress Nonlinear mixed integer SLP (MISLP) algorithm

Synopsis

```
problem.globalsol ()
```

Example

The following example solves the continuous relaxation of the problem and then finds the integer solution.

```
p.nlpoptimize ()
p.globalsol ()
```

Further information

The current Xpress Nonlinear mixed integer problem will be maximized or minimized using the algorithm defined by the control variable `xslp_mipalgorithm`.

It is recommended that `problem.nlpoptimize` be used first to obtain a converged solution to the relaxed problem. If this is not done, ensure that `xslp_ojsense` is set appropriately.

See the chapter on Mixed Integer Non-Linear Programming in the SLP Reference Manual for more information about the Xpress Nonlinear MISLP algorithms.

Related topics

`xslp_MIPALGORITHM`, `xslp_OBJSENSE`

problem.hasdualray

Purpose

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

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

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` command 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.iisfirst`, `problem.iisnext`, or `problem.iisall` functions.

Related topics

`problem.getiisdata`, `problem.iisclear`, `problem.iisfirst`, `problem.iisisolations`, `problem.iisnext`, `problem.iisstatus`, `problem.iiswrite`.

problem.iisclear

Purpose

Resets the search for Irreducible Infeasible Sets (IIS).

Synopsis

```
problem.iisclear ()
```

Further information

1. The information stored internally about the IISs identified by problem.iisfirst, problem.iisnext or problem.iisall are cleared. Functions problem.getiisdata, problem.iisstatus, problem.iiswrite and problem.iisisolations cannot be called until the IIS identification procedure is started again.

2. This function is automatically called by problem.iisfirst and problem.iisall.

Related topics

problem.getiisdata, problem.iisall, problem.iisfirst, problem.iisisolations, problem.iisnext, problem.iisstatus, problem.iiswrite.

problem.iisfirst

Purpose

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 (iismode)
```

Argument

`iismode`	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.iisnext` 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.iisfirst` 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.iisclear`, `problem.iisisolations`, `problem.iisnext`, `problem.iisstatus`, `problem.iiswrite`.

problem.iisisolations

Purpose

Performs the isolation identification procedure for an Irreducible Infeasible Set (IIS).

Synopsis

```
problem.iisisolations (num)
```

Argument

`num`	The number of the IIS identified by either `problem.iisfirst`, `problem.iisnext`, 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 `num` parameter cannot be zero: the concept of isolations is meaningless for the initial infeasible subproblem.

Related topics

`problem.getiisdata`, `problem.iisall`, `problem.iisclear`, `problem.iisfirst`, `problem.iisnext`, `problem.iisstatus`, `problem.iiswrite`.

problem.iisnext

Purpose

Continues the search for further Irreducible Infeasible Sets (IIS), or calls problem.iisfirst 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.iisfirst 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.iisnext 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.iisfirst or problem.iisnext, the generation process has to be started from scratch.

Related topics

problem.getiisdata, problem.iisall, problem.iisclear, problem.iisfirst, problem.iisisolations, problem.iisstatus, problem.iiswrite.

problem.iisstatus

Purpose

Returns statistics on the Irreducible Infeasible Sets (IIS) found so far by problem.iisfirst, problem.iisnext, or problem.iisall. The returned value is the number of IISs found so far.

Synopsis

```
iiscount = problem.iisstatus (rowsizes, colsizes, suminfeas, numinfeas)
```

Arguments

rowsizes	Number of rows in the IISs.
colsizes	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.iisclear, problem.iisfirst, problem.iisisolations, problem.iisnext, problem.iiswrite.

problem.iiswrite

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.

Synopsis

```
problem.iiswrite (num, fn, type, typeflags)
```

Arguments

`num`	The ordinal number of the IIS to be written.
`fn`	The name of the file to be created.
`type`	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.
`typeflags`	Flags passed to the `problem.write` 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:
   (a) save the IIS using MPS hexadecimal format to eliminate rounding errors associated with conversion between internal and decimal representation.
   (b) turn presolve off since the nature of an IIS makes it necessary that during their identification the presolve is turned off.
   (c) 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.iisfirst`, `problem.iisnext`, 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.iiswrite` corresponding to an IIS contain no objective function, since infeasibility is independent from the objective.

Related topics

`problem.getiisdata`, `problem.iisall`, `problem.iisclear`, `problem.iisfirst`, `problem.iisisolations`, `problem.iisnext`, `problem.iisstatus`.

problem.interrupt

Purpose

Interrupts the Optimizer algorithms.

Synopsis

```
problem.interrupt (reason)
```

Argument

`reason`	The reason for stopping. Possible reasons are:	
	`xpress.stop_timelimit`	time limit hit;
	`xpress.stop_ctrlc`	control C hit;
	`xpress.stop_nodelimit`	node limit hit;
	`xpress.stop_iterlimit`	iteration limit hit;
	`xpress.stop_mipgap`	MIP gap is sufficiently small;
	`xpress.stop_sollimit`	solution limit hit;
	`xpress.stop_user`	user interrupt.

Further information

The `interrupt` command can be called from any callback.

problem.loadbasis

Purpose

Loads a basis as specified by the user.

Synopsis

```
problem.loadbasis (rstatus, cstatus)
```

Arguments

`rstatus` Array of length `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.

`cstatus` Array of length `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.loadbranchdirs

Purpose

Loads directives into the current problem to specify which global entities the Optimizer should continue to branch on when a node solution is global feasible.

Synopsis

```
problem.loadbranchdirs (mcols, mbranch)
```

Arguments

`mcols` Array containing the column numbers. A negative value indicates a set number (the first set being -1, the second -2, and so on).

`mbranch` Array containing either 0 or 1 for the entities given in `mcols`. Entities for which mbranch is set to 1 will be branched on until fixed before a global feasible solution is returned. If `mbranch` is `None`, the branching directive will be set for all entities in `mcols`.

Related topics

`problem.loaddirs`, `problem.readdirs`.

problem.loadcoefs

Purpose

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 of length `nSLPCoef+1` holding the start position in the arrays `Type` and `Value` of the formula for the coefficients. `FormulaStart[nSLPCoef]` should be set to the next position after the end of the last formula.
`parsed`	Integer indicating whether the token arrays are formatted as internal unparsed (`Parsed`=0) or internal parsed reverse Polish (`Parsed`=1).
`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 = []

n = 0
ncoef = 0

formulastart[ncoef], ncoef = n, ncoef + 1
Type[n], Value[n], n = xslp_op_col, 3, n+1
Type[n],           n = xslp_op_eof,    n+1


formulastart[ncoef], ncoef = n, ncoef + 1

Type[n], Value[n], n = xslp_op_col, 2,             n+1
Type[n], Value[n], n = xslp_op_col, 2,             n+1
Type[n], Value[n], n = xslp_op_op,  xslp_MULTIPLY, n+1
Type[n],           n = xslp_op_eof,                n+1

formulastart[ncoef], ncoef = n, ncoef + 1

Type[n], Value[n], n = xslp_op_col, 2, n+1
Type[n],           n = xslp_op_eof,    n+1

formulastart [ncoef] = n
```

```
p.loadcoefs (rowindex, colindex, None, formulastart, 1, 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`=1), 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 `FormulaStart`. For possible token types and values see the chapter on "Formula Parsing".

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.addcoefs`, `problem.chgnlcoef`, `problem.chgccoef`, `problem.getcoefformula`, `problem.getccoef`

problem.loadcuts

Purpose

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 (itype, interp, cutind)
```

Arguments

`itype` 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 `itype`;
- `3` treat cut types as bit maps - `0` load cut if all bits match those set in `itype`.

`mcutind` Array containing the cuts to be loaded into the matrix.

Related topics

`problem.addcuts`, `problem.delcpcuts`, `problem.delcuts`, `problem.getcpcutlist`, Section "Working with the cut manager" of the Xpress Optimizer reference manual.

problem.loaddelayedrows

Purpose

Specifies that a set of rows in the problem will be treated as delayed rows during a global 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 (mrows)
```

Argument

`mrows` An array of row indices to treat as delayed rows.

Example

This sets the first six matrix rows as delayed rows in the global 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.loaddfs

Purpose

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

Loads directives into the problem.

Synopsis

```
problem.loaddirs (mcols, mpri, qbr, dupc, ddpc)
```

Arguments

`mcols` Array containing the column numbers. A negative value indicates a set number (the first set being `-1`, the second `-2`, and so on).

`mpri` Array containing the priorities for the columns or sets. Priorities must be between 0 and 1000. May be `None` if not required.

`qbr` 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.

`dupc` Array containing the up pseudo costs for the columns or sets. May be `None` if not required.

`ddpc` 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.loadlpsol

Purpose

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, dual, dj)
```

Arguments

`x`	Optional: Array of length `COLS` (for the original problem and not the presolve problem) containing the values of the variables.
`slack`	Optional: double array of length `ROWS` containing the values of slack variables.
`dual`	Optional: double array of length `ROWS` containing the values of dual variables.
`dj`	Optional: double array of length `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, dual, None)
```

Further information

1. At least one of variables `x` and dual variables `dual` 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 dual variables `dual` is `None`, both dual variables and reduced costs will be set to zero.
5. When reduced costs `dj` is `None`, it will be computed from dual variables `dual`. If reduced costs are provided, dual variables cannot be omitted.

Related topics

`problem.getlpsol`.

problem.loadmipsol

Purpose

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 (dsol)
```

Argument

`dsol`	Array of length `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 (dsol)
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.getmipsol`, `problem.addcboptnode`.

problem.loadmodelcuts

Purpose

Specifies that a set of rows in the problem will be treated as model cuts.

Synopsis

```
problem.loadmodelcuts (mrows)
```

Argument

`mrows` An array of row indices to be treated as cuts.

Example

This sets the first six matrix rows as model cuts in the global 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 global 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

Loads a presolved basis from the user's areas.

Synopsis

```
problem.loadpresolvebasis (rstatus, cstatus)
```

Arguments

`rstatus` 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.

`cstatus` 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.mipoptimize ("l")
rs = []
cs = []
p1.getpresolvebasis (rs, cs)

p2.read ("myprob2", "")
p2.mipoptimize ("l")
p2.loadpresolvebasis (rs, cs)
```

Related topics

`problem.getbasis`, `problem.getpresolvebasis`, `problem.loadbasis`.

problem.loadpresolvedirs

Purpose

Loads directives into the presolved matrix.

Synopsis

```
problem.loadpresolvedirs (mcols, mpri, qbr, dupc, ddpc)
```

Arguments

`mcols` Array containing the column numbers. A negative value indicates a set number (`-1` being the first set, `-2` the second, and so on).

`mpri` Array containing the priorities for the columns or sets. May be `None` if not required.

`qbr` 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.

`dupc` Array containing the up pseudo costs for the columns or sets. May be `None` if not required.

`ddpc` 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.mipoptimize ("l")
p.loadpresolvedirs ([0], [1], None, None, None)
p.mipoptimize ("")
```

Related topics

`problem.getdirs`, `problem.loaddirs`.

problem.loadproblem

Purpose

Load an optimization problem, possibly with quadratic objective and/or constraints, and integer variables.

Synopsis

```
problem.loadproblem (probname, qrtypes, rhs, range, obj, mstart, mnel, mrwind,
       dmatval, dlb, dub, mqcol1, mqcol2, dqe, qcrows, qcnquads, qcmqcol1, qcmqcol2,
       qcdqval, qgtype, mgcols, dlim, qstype, msstart, mscols, dref, colnames,
       rownames)
```

Arguments

`probname` A string of up to 200 characters containing the problem name.

`qrtype` 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.

`range` 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.

`obj` Array containing the objective function coefficients.

`mstart` Array containing the offsets in the `mrwind` and `dmatval` 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 `mnel` is `None`, `ncol+1`. If `mnel` is `None` the extra entry of `mstart`, `mstart[ncol]`, contains the position in the `mrwind` and `dmatval` arrays at which an extra column would start, if it were present.

`mnel` Array containing the number of nonzero elements in each column. May be `None` if all elements are contiguous and `mstart[ncol]` contains the offset where the elements for column `ncol+1` would start. This array is not required if the non-zero coefficients in the `mrwind` and `dmatval` arrays are continuous, and the `mstart` array has `ncol+1` entries as described above. It may be `None` if not required.

`mrwind` 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 `mrwind` is `mstart[ncol-1]+mnel[ncol-1]` or, if `mnel` is `None`, `mstart[ncol]`.

`dmatval` Array containing the nonzero element values; length as for `mrwind`.

`dlb` Array containing the lower bounds on the columns. Use `-xpress.infinity` to represent a lower bound of minus infinity.

`dub` Array containing the upper bounds on the columns. Use `xpress.infinity` to represent an upper bound of plus infinity.

`mqc1` (optional) Array with the first variable in each quadratic term.

`mqc2` (optional) Array with the second variable in each quadratic term.

`dqe` (optional) Array with the quadratic coefficients.

`qcrows` (optional) Integer containing the indices of rows with quadratic matrices in them. Note that the rows are expected to be defined in `qrtype` as type `L`.

`qcnquads` (optional) Array containing the number of nonzeros in each quadratic constraint matrix.

`qcmqcol1` (optional) Array with a number of elements equal to the sum of the elements in `qcnquads` (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 `qcnquads[0]-1`, for the second matrix from `qcnquads[0]` to `qcnquads[0]+ qcnquads[1]-1`, etc.

`qcmqcol2` (optional) Array containing the second index for the quadratic constraint matrices.

`qcdqval` (optional) Array containing the coefficients for the quadratic constraint matrices.

`qgtype` 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.

`mgcols` (optional) Array containing the variables of the global entities.

`dlim` (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.

`qstype` (optional) Character array of length equal to the number of sets specified, `nsets`, and specifies the set types:
`1` SOS1 type sets;
`2` SOS2 type sets.
May be `None` if not required.

`msstart` (optional) Array containing the offsets in the `mscols` and `dref` 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.

`mscols` (optional) Array of length `msstart[nsets]-1` containing the columns in each set. May be `None` if not required.

`dref` (optional) Array of length `msstart[nsets]-1` containing the reference row entries for each member of the sets. May be `None` if not required.

`colname` (optional) Array of containing the column names for all variables added.

`rowname` (optional) Array of containing the row names for all constraints added.

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. The row and column indices are from `0` to `nrow-1` and `0` to `ncol-1` respectively.

4. The row and column indices are from `0` to `nrow-1` and `0` to `ncol-1` respectively.

5. 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.loadproblem`, `problem.read`.

problem.loadsecurevecs

Purpose

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 (mrow, mcol)
```

Arguments

`mrow`	Array containing the rows to be marked. May be `None` if not required.
`mcol`	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 (mrow = [0,1,2,3,4,5], mcol = [0,1,2,3])
p.mipoptimize ("")
```

problem.loadtolsets

Purpose

Load sets of standard tolerance values into an SLP problem

Synopsis

```
problem.loadtolsets (slptol)
```

Argument

`slptol` Array of 9*h* 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 zero 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

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:
`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   = [0,2]
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 function 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 global 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, Chapter 4 of the Xpress Optimizer reference manual.

problem.mipoptimize

Purpose

This function begins a global 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 global 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 — stop after having solved the initial continous relaxation.

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 global 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 global search is given by DEFAULTALG. The default is to use the dual simplex algorithm.

Related topics

problem.mipoptimize.

problem.msaddcustompreset

Purpose

A combined version of msaddjob and 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.
`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.msaddjob

Purpose

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 pointer 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.msaddpreset

Purpose

Loads a preset of jobs into the multistart job pool.

Synopsis

```
problem.msaddpreset (description, preset, count, job_object)
```

Arguments

`description`	Text description of the preset. 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.
`job_object`	Job-specific user context pointer 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.msclear

Purpose

Removes all scheduled jobs from the multistart job pool

Synopsis

```
problem.msclear ()
```

Related topics

problem.msaddjob, problem.msaddpreset, problem.msaddcustompreset

problem.name

Purpose

Returns the name of the problem as a Python string.

Synopsis

```
brian = problem.name ()
```

Related topics

problem.setprobname.

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.

Synopsis

```
problem.objsa (mindex, lower, upper)
```

Arguments

`mindex`	Array containing the indices of the columns 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 $5.0 \leq C_2 \leq 7.0$, $3.8 \leq C_8 \leq 5.2$ and $5.7 \leq C_6$, 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.rhssa`.

problem.parsecformula

Purpose

Parse a formula written as a character string into internal parsed (reverse Polish) format

Synopsis

```
ntoken = problem.parsecformula (formula, type, value)
```

Arguments

`ntoken`	Number of tokens in the parsed formula (not counting the terminating `xslp_op_eof` token).
`formula`	Character string containing the formula, written in the same free-format style as used in formulae in Extended MPS format, with spaces separating tokens.
`Type`	Array of token types providing the parsed formula.
`Value`	Array of values corresponding to the types in `Type`.

Example

Assuming that `x` and `y` are already defined as columns, the following example converts the formula "sin(x+y)" into internal parsed format, and then writes it out as a sequence of tokens.

```
type  = []
value = []
ntoken = p.parsecformula ("sin ( x + y )", Type, Value)
i = 0
while type[i] != xslp_op_eof:
  str = p.itemname (type[i], value[i])
  printf (str)
  i += 1
```

Further information

Tokens are identified by name, so any columns or user functions which appear in the formula must already have been defined. Unidentified tokens will appear as type `xslp_UNKNOWN`.

Related topics

`problem.parseformula`, `problem.preparseformula`

problem.parseformula

Purpose

Parse a formula written as an unparsed array of tokens into internal parsed (reverse Polish) format

Synopsis

```
ntoken = problem.parseformula (intype, invalue, type, value)
```

Arguments

`intype`	Array of token types providing the unparsed formula.
`invalue`	Array of values corresponding to the types in `inType`.
`ntoken`	The number of tokens in the parsed formula (not counting the terminating `xslp_op_eof` token).
`type`	Array of token types providing the parsed formula.
`value`	Array of values corresponding to the types in `type`.

Example

Assuming that `x` and `y` are already defined as columns with index `iX` and `iY` respectively, the following example converts the formula "sin(x+y)" into internal parsed format, and then writes it out as a sequence of tokens.

```
intype  = []
invalue = []

intype  = [xslp_op_ifun, xslp_op_lb, xslp_op_col, xslp_op_op,
           xslp_op_col, xslp_op_rb, xslp_op_eof]
invalue = [xslp_op_sin,  0,          iX         , xslp_op_plus,
           iY,          0,          0]

type  = []
value = []

ntoken = p.parseformula (intype, invalue, type, value);

i=0
while type[n] != xslp_op_eof:
  str = p.itemname (type[i], value[i])
  print (str);
  i += 1
```

Further information

For possible token types and values, see the chapter on "Formula Parsing" in the SLP reference manual.

Related topics

`problem.parsecformula`, `problem.preparseformula`

problem.postsolve

Purpose

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.preparseformula

Purpose

Perform an initial scan of a formula written as a character string, identifying the operators but not attempting to identify the types of the individual tokens

Synopsis

```
(type, value, stringtable) = problem.preparseformula (formula)
```

Arguments

`formula`	Character string containing the formula, written in the same free-format style as formulae in Extended MPS format, with spaces separating tokens.
`type`	Array of token types providing the parsed formula.
`value`	Array of values corresponding to the types in `Type`.
`stringtable`	Character buffer to receive the names of the unidentified tokens.

Example

The following example converts the formula "sin(x+y)" into internal parsed format without trying to identify the tokens apart from operands and numbers, and then writes it out as a sequence of tokens.

```
(type, value, stab) = p.preparseformula ("sin ( x + y )")

i=0
while type[i] != xslp_op_eof:
  if type[n] == xslp_UNKNOWN:
    print ("? ", value[i])
  else:
    str = p.itemname (type[i], value[i])
    printf (str)
  i += 1
```

Further information

Only operands and numbers are identified by `preparseformula`. All other operands, including names of variables, functions are left as strings of type `xslp_op_unknown`. The `Value` of such a type is the index in `stringtable` of the start of the token name.

The parsed formula can be converted into a calculable formula by replacing the `xslp_op_unknown` tokens by the correct types and values.

Related topics

`problem.parsecformula`, `problem.parseformula`

problem.presolve

Purpose

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.solve ("")
```

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.construct`, 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.construct` or any of the SLP optimization procedures..

Related topics

`xslp_presolve`

problem.presolverow

Purpose

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 (qrtype, mcolso, dvalo, drhso, maxcoeffs, mcolsp,
       dvalp)
```

Arguments

`qrtype`	The type of the row: `L` indicates a $\leq$ row; `G` indicates a $\geq$ row.
`mcolso`	Array containing the column indices of the row to presolve.
`dvalo`	Array containing the non-zero coefficients of the row to presolve.
`drhso`	The right-hand side constant of the row to presolve.
`maxcoeffs`	Maximum number of elements to return in the `mcolsp` and `dvalp` arrays.
`mcolsp`	Array which will be filled with the column indices of the presolved row.
`dvalp`	Array which will be filled with the coefficients of the presolved row.

Example

Adding the row $2x_1 + x_2 \leq 1$ 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 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.

Related topics

`problem.addcuts`, `problem.loadsecurevecs`, `problem.setbranchcuts`, `problem.storecuts`.

problem.printmemory

Purpose

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

Print a summary of any evaluation errors that may have occurred during solving a problem

Synopsis

```
problem.printevalinfo ()
```

Related topics

problem.setcbcoefevalerror

problem.printmsg

Purpose

Print a message string according to the current settings for Xpress Nonlinear output

Synopsis

```
problem.printmsg (msgtype, msg)
```

Arguments

`MsgType` Integer containing the message type. The following types are system-defined:

- `1` Information message
- `3` Warning message
- `4` Error message

Other message types can be used and passed to a user-supplied message handler.

`Msg` Character string containing the message.

Example

The following example checks the SLP optimization status and prints an informative message for some of the possible values.

```
status = p.attributes.xslp_status
if status == 0:
  p.printmsg (1, "Fully converged solution")
if status & xp.xslp_maxtime:
  p.printmsg (3, "Max time exceeded")
if status & xp.xslp_convergedobjucc:
  p.printmsg (1, "Solution with unimportant unconverged values")
```

Further information

If `msgtype` is outside the range 1 to 4, any message handler written to handle the standard message types may not print the message correctly. One of the uses of the fucntion is to provide a unified means of logging from the callbacks.

problem.read

Purpose

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)
```

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.

Example

Read problem `problem1.lp` and output an optimal solution:

```
p.read ("problem1", "l")
p.solve ("", "")
print ("solution of problem1.lp:", p.getSolution ())
```

Related topics

`problem.write`.

problem.readbasis

Purpose

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.readbinsol

Purpose

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 commands:

```
p.read ("myprob", "")
p.readbinsol ("", "")
p.writeprtsol ("", "")
```

Related topics

`problem.getlpsol`, `problem.getmipsol`, `problem.writebinsol`, `problem.writesol`, `problem.writeprtsol`.

problem.readdirs

Purpose

Reads a directives file to help direct the global 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 command must be issued 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 global entity. The *lower* the number, the *more* likely the entity is to be selected for branching; the *higher*, the *less* likely. By default, all global 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 global 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 global 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 global entities whose names start with `x1` a priority of `2`.

Related topics

`problem.loaddirs`.

problem.readslxsol

Purpose

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 global 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.readbinsol` `problem.addmipsol`, `problem.addcbusersolnotify`.

problem.refinemipsol

Purpose

Runs the MIP solution refiner.

Synopsis

```
problem.refinemipsol (options, flags, solution, refined_solution)
```

Arguments

`options` Refinement options:
- `0` Reducing MIP fractionality is priority.
- `1` Reducing LP infeasiblity is priority

`flags` Flags passed to any optimization calls during refinement.

`solution` The MIP solution to refine. Must be a valid MIP solution.

`refined_solution` The refined MIP solution in case of success

`refinestatus` Refinement results:
- `0` An error has occurred
- `1` The solution has been refined
- `2` Current solution meets target criteria
- `3` Solution cannot be refined

Further information

The function provides a mechanism to refine the MIP solution by attempting to round any fractional global entity and by attempting to reduce LP infeasiblity.

Related topics

`REFINEOPS`.

problem.reinitialize

Purpose

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.createprob, problem.destroyprob, problem.unconstruct, problem.setcurrentiv,

problem.removecbbariteration

Purpose

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 (f_bariteration, object)
```

Arguments

`f_bariteration` The callback function to remove. If `None` then all bariteration callback functions added with the given user-defined object value will be removed.

`object` 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 pointer `f_bariteration` will be removed.

Related topics

`problem.addcbbariteration`.

problem.removecbbarlog

Purpose

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 (f_barlog, object)
```

Arguments

`f_barlog`	The callback function to remove. If `None` then all barrier log callback functions added with the given user-defined object value will be removed.
`object`	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 pointer `f_barlog` will be removed.

Related topics

`problem.addcbbarlog`.

problem.removecbchgbranchobject

Purpose

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 (f_chgbranchobject, object)
```

Arguments

`f_chgbranchobject` The callback function to remove. If `None` then all branch object callback functions added with the given user-defined object value will be removed.

`object` 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 pointer `f_chgbranchobject` will be removed.

Related topics

`problem.addcbchgbranchobject`.

problem.removecbcutlog

Purpose

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 (f_cutlog, object)
```

Arguments

`f_cutlog`	The callback function to remove. If `None` then all cut log callback functions added with the given user-defined object value will be removed.
`object`	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 pointer `f_cutlog` will be removed.

Related topics

`problem.addcbcutlog`.

problem.removecbdestroymt

Purpose

Removes a slave 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 (f_destroymt, object)
```

Arguments

`f_destroymt`	The callback function to remove. If `None` then all thread destruction callback functions added with the given user-defined object value will be removed.
`object`	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 pointer `f_destroymt` will be removed.

Related topics

`problem.addcbdestroymt`.

problem.removecbgapnotify

Purpose

Removes a callback function previously added by problem.addcbgapnotify. The specified callback function will no longer be removed after it has been returned.

Synopsis

```
problem.removecbgapnotify (f_gapnotify, object)
```

Arguments

f_gapnotify	The callback function to remove. If None then all gapnotify callback functions added with the given user-defined pointer value will be removed.
object	The user-defined pointer value that the callback was added with. If None then the pointer value will not be checked and all the gapnotify callbacks with the function pointer f_gapnotify will be removed.

Related topics

problem.addcbgapnotify.

problem.removecbgloballog

Purpose

Removes a global log callback function previously added by `addcbgloballog`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbgloballog (f_globallog, object)
```

Arguments

`f_globallog`	The callback function to remove. If `None` then all global log callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If `None`, then the object value will not be checked and all global log callbacks with the function pointer `f_globallog` will be removed.

Example

The following code sets and removes a callback function:

```
prob.controls.miplog = 3
prob.addcbgloballog (globalLog, None, 0)
prob.mipoptimize ("")
prob.removecbgloballog (globalLog, None)
```

Related topics

`problem.addcbgloballog`.

problem.removecbinfnode

Purpose

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 (f_infnode, object)
```

Arguments

`f_infnode` 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.

`object` 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 pointer `f_infnode` will be removed.

Related topics

`problem.addcbinfnode`.

problem.removecbintsol

Purpose

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 (f_intsol, object)
```

Arguments

`f_intsol` The callback function to remove. If `None` then all integer solution callback functions added with the given user-defined object value will be removed.

`object` 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 pointer `f_intsol` will be removed.

Related topics

`problem.addcbintsol`.

problem.removecblplog

Purpose

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 (f_lplog, object)
```

Arguments

`f_lplog` The callback function to remove. If `None` then all lplog callback functions added with the given user-defined object value will be removed.

`object` 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 pointer `f_lplog` 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.addcblplog`.

problem.removecbmessage

Purpose

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 (f_message, object)
```

Arguments

`f_message`	The callback function to remove. If `None` then all message callback functions added with the given user-defined object value will be removed.
`object`	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 pointer `f_message` will be removed.

Related topics

`problem.addcbmessage`.

problem.removecbmipthread

Purpose

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 (f_mipthread, object)
```

Arguments

`f_mipthread`	The callback function to remove. If `None` then all variable branching callback functions added with the given user-defined object value will be removed.
`object`	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 pointer `f_mipthread` will be removed.

Related topics

`problem.addcbmipthread`.

problem.removecbnewnode

Purpose

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 (f_newnode, object)
```

Arguments

`f_newnode`	The callback function to remove. If `None` then all separation callback functions added with the given user-defined object value will be removed.
`object`	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 pointer `f_newnode` will be removed.

Related topics

`problem.addcbnewnode`.

problem.removecbnodecutoff

Purpose

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 (f_nodecutoff, object)
```

Arguments

`f_nodecutoff` The callback function to remove. If `None` then all node-cutoff callback functions added with the given user-defined object value will be removed.

`object` 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 pointer `f_nodecutoff` will be removed.

Related topics

`problem.addcbnodecutoff`.

problem.removecboptnode

Purpose

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 (f_optnode, object)
```

Arguments

`f_optnode`	The callback function to remove. If `None` then all node-optimal callback functions added with the given user-defined object value will be removed.
`object`	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 pointer `f_optnode` will be removed.

Related topics

`problem.addcboptnode`.

problem.removecbpreintsol

Purpose

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 (f_preintsol, object)
```

Arguments

`f_preintsol`	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.
`object`	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 pointer `f_preintsol` will be removed.

Related topics

`problem.addcbpreintsol`.

problem.removecbprenode

Purpose

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 (f_prenode, object)
```

Arguments

`f_prenode`	The callback function to remove. If `None` then all preprocess node callback functions added with the given user-defined object value will be removed.
`object`	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 pointer `f_prenode` will be removed.

Related topics

`problem.addcbprenode`.

problem.removecbusersolnotify

Purpose

Removes a user solution notification callback previously added by problem.addcbusersolnotify. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbusersolnotify (f_usersolnotify, object)
```

Arguments

`f_usersolnotify` 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.

`object` 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 pointer `f_usersolnotify` will be removed.

Related topics

problem.addcbusersolnotify.

problem.repairinfeas

Purpose

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 (pflags, oflags, gflags, lrp, grp, lbp, ubp, delta)
```

Arguments

`pflags` 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.

`oflags` 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.

`gflags` Specifies if the global search should be done:
- `g` do the global search (default);
- `l` solve as a linear model ignoring the discreteness of variables.

`lrp` Preference for relaxing the less or equal side of row.

`grp` Preference for relaxing the greater or equal side of a row.

`lbp` Preferences for relaxing lower bounds.

`ubp` 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 pflags 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
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. If an irreducible infeasible set (IIS) has been identified, the generated IIS(s) are accesible through the IIS retrieval fucntions, see NUMIIS and problem.getiisdata.

Related topics

problem.repairweightedinfeas.

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 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 (lrp_array, grp_array, lbp_array,
       ubp_array, phase2, delta, optflags)
```

Arguments

`lrp_array` Array of size `ROWS` containing the preferences for relaxing the less or equal side of row.

`grp_array` Array of size `ROWS` containing the preferences for relaxing the greater or equal side of a row.

`lbp_array` Array of size `COLS` containing the preferences for relaxing lower bounds.

`ubp_array` 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.

`optflags` 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
`lrp_array`	`= rows`	$a^T x$ `- aux_var = b`	`1/lrp*aux_var`	`1/lrp*aux_var`^{2}
`lrp_array`	`<= rows`	$a^T x$ `- aux_var <= b`	`1/lrp*aux_var`	`1/lrp*aux_var`^{2}
`grp_array`	`= rows`	$a^T x$ `+ aux_var = b`	`1/grp*aux_var`	`1/grp*aux_var`^{2}
`grp_array`	`>= rows`	$a^T x$ `+ aux_var >= b`	`1/grp*aux_var`	`1/grp*aux_var`^{2}
`ubp_array`	`upper bounds`	x_i `- aux_var <= u`	`1/ubp*aux_var`	`1/ubp*aux_var`^{2}
`lbp_array`	`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 fucntions, see `NUMIIS` and `problem.getiisdata`.

Related topics

`problem.repairinfeas`, `problem.repairweightedinfeasbounds`.

problem.repairweightedinfeasbounds

Purpose

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 (lrp_array, grp_array, lbp_array,
       ubp_array, lrb_array, grb_array, lbb_array, ubb_array, phase2, delta, optflags)
```

Arguments

lrp_array	Array of size ROWS containing the preferences for relaxing the less or equal side of row.
grp_array	Array of size ROWS containing the preferences for relaxing the greater or equal side of a row.
lbp_array	Array of size COLS containing the preferences for relaxing lower bounds.
ubp_array	Array of size COLS containing preferences for relaxing upper bounds.
lrb_array	Array of size ROWS containing the upper bounds on the amount the less or equal side of a row can be relaxed.
grb_array	Array of size ROWS containing the upper bounds on the amount the greater or equal side of a row can be relaxed.
lbb_array	Array of size COLS containing the upper bounds on the amount the lower bounds can be relaxed.
ubb_array	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.
optflags	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=lrp_array[j]` and `grp=grp_array[j]`, or variable `i` with bound preferences `ubp=ubp_array[i]` and `lbp=lbp_array[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 fucntions, see `NUMIIS` and `problem.getiisdata`.

Related topics

`problem.repairinfeas`.

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.solve ()
x0 = p.getSolution ()
p.reset ()
p.read ("problem1", "")
p.solve ()
x1 = p.getSolution ()
```

Related topics

`problem.read`.

problem.restore

Purpose

Restores the Optimizer's data structures from a file created by `problem.save`. Optimization may then recommence from the point at which the file was created.

Synopsis

```
problem.restore (probname, flags)
```

Arguments

`probname`	A string of up to 200 characters containing the problem name.	
`flags`	`f`	Force the restoring of a save file even if its from a different version.

Example

```
p.restore ("", "")
```

Further information

1. This routine restores the data structures from the file *probname*`.svf` that was created by a previous execution of `save`. The file *probname*`.sol` is also required and, if recommencing optimization in a global search, the files *problem_name*`.glb` and *problem_name*`.ctp` are required too. Note that `.svf` files 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 use of the 'f' flag is not recommended and can cause unexpected results.

Related topics

`problem.save`.

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.

Synopsis

```
problem.rhssa (mindex, lower, upper)
```

Arguments

`mindex`	Array containing the indices of the rows 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 $5.0 \leq \text{rhs}_2$, $3.8 \leq \text{rhs}_8 \leq 5.2$ and $5.7 \leq \text{rhs}_6$, 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.objsa`.

problem.save

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.

Synopsis

```
problem.save ()
```

Example

```
p.save ()
```

Further information

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`. Under such circumstances, the file *problem_name*`.sol` and, if a branch and bound search is in progress, the global files *problem_name*`.glb` and *problem_name*`.ctp` are also required. These files will be present after execution of `save`, but will be modified by subsequent optimization, so no optimization calls may be made after the call to `save`. 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.

Related topics

`problem.restore`.

problem.scale

Purpose

Re-scales the current problem.

Synopsis

```
problem.scale (mrscal, mcscal)
```

Arguments

`mrscal`	Array of size `ROWS` containing the exponents of the powers of `2` with which to scale the rows, or `None` if not required.
`mcscal`	Array of size `COLS` containing the exponents of the powers of `2` with which to scale the columns, or `None` if not required.

Example

```
p.read ("prob1", "")
p.scale ([1] * p.attributes.rows, [3] * p.attributes.cols)
p.lpoptimize ("")
```

This reads the MPS file `prob1.mat`, rescales the problem and seeks the minimum objective value.

Further information

1. If `mrscal` and `mcscal` are both non-`None` then they will be used to scale the problem. Otherwise the problem will be scaled according to the control `SCALING`. This routine may be useful when the current problem has been modified by calls to routines such as `problem.chgmcoef` and `problem.addrows`.

2. `scale` cannot be called if the current problem is presolved.

Related topics

`problem.read`.

problem.scaling

Purpose

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.setbranchbounds

Purpose

Specifies the bounds previously stored using `problem.storebounds` that are to be applied in order to branch on a user global entity.

Synopsis

```
problem.setbranchbounds (mindex)
```

Argument

`mindex` Object previously defined in a call to `problem.storebounds` that references the stored bounds to be used to separate the node.

Related topics

`problem.loadcuts`, `problem.storebounds`, Section "Working with the cut manager" of the Xpress Optimizer reference manual.

problem.setbranchcuts

Purpose

Specifies the cuts in the cut pool that are to be applied in order to branch on a user global entity..

Synopsis

```
problem.setbranchcuts (mindex)
```

Argument

`mindex` Array containing the pointers to the cuts in the cut pool that are to be applied. Typically obtained from `problem.storecuts`.

Related topics

`problem.getcpcutlist`, `problem.storecuts`, Section "Working with the cut manager" of the Xpress Optimizer reference manual.

problem.setcbcascadeend

Purpose

Set a user callback to be called at the end of the cascading process, after the last variable has been cascaded

Synopsis

```
problem.setcbcascadeend (userfunc, object)
value = userfunc (prob, myobject)
```

Arguments

`userfunc`	The function to be called at the end of the cascading process. `userfunc` returns an integer value. The return value is noted by Xpress SLP but it has no effect on the optimization.
`prob`	The problem passed to the callback function.
`myobject`	The user-defined object passed as `object` to `setcbcascadeend`.
`object`	User-defined object, which can be used for any purpose by the function. `object` is passed to `userfunc` 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.cascade`, `problem.setcbcascadestart`, `problem.setcbcascadevar`, `problem.setcbcascadevarfail`

problem.setcbcascadestart

Purpose

Set a user callback to be called at the start of the cascading process, before any variables have been cascaded

Synopsis

```
problem.setcbcascadestart (userfunc, object)
retval = userfunc (my_prob, my_object)
```

Arguments

`userfunc`	The function to be called at the start of the cascading process. `userfunc` 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 `object` to `setcbcascadestart`.
`object`	Address of a user-defined object, which can be used for any purpose by the function. `object` is passed to `userfunc` 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.cascade`, `problem.setcbcascadeend`, `problem.setcbcascadevar`, `problem.setcbcascadevarfail`

problem.setcbcascadevar

Purpose

Set a user callback to be called after each column has been cascaded

Synopsis

```
problem.setcbcascadevar (userfunc, object)
retval = userfunc (my_prob, my_object, colindex)
```

Arguments

`userfunc`	The function to be called after each column has been cascaded. `userfunc` 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 `object` to `problem.setcbcascadevar`.
`ColIndex`	The number of the column which has been cascaded.
`Object`	User-defined object, which can be used for any purpose by the function. `object` is passed to `userfunc` 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 (myprob, ColNum, None, None, None, None,
              None, None, obj[iCol], None, None, None,
              None)
  return 0
```

The `object` argument is used here to hold the address of 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.cascade`, `problem.setcbcascadeend`, `problem.setcbcascadestart`, `problem.setcbcascadevarfail`

problem.setcbcascadevarfail

Purpose

Set a user callback to be called after cascading a column was not successful

Synopsis

```
problem.setcbcascadevarfail (userfunc, object)
retval = userfunc (my_prob, my_object, colindex)
```

Arguments

`userfunc`	The function to be called after cascading a column was not successful. `userfunc` 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 `object` to `setcbcascadevarfail`.
`ColIndex`	The number of the column which has been cascaded.
`Object`	Address of a user-defined object, which can be used for any purpose by the function. `object` is passed to `userfunc` 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.cascade`, `problem.setcbcascadeend`, `problem.setcbcascadestart`, `problem.setcbcascadevar`

problem.setcbcoefevalerror

Purpose

Set a user callback to be called when an evaluation of a coefficient fails during the solve

Synopsis

```
problem.setcbcoefevalerror (userfunc, object)
retval = userfunc (my_prob, my_object, rowindex, colIndex)
```

Arguments

`userfunc`	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 `object` to `setcbcoefevalerror`.
`RowIndex`	The row position of the coefficient.
`ColIndex`	The column position of the coefficient.
`Object`	Address of a user-defined object, which can be used for any purpose by the function. `object` is passed to `userfunc` 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.printevalinfo`

problem.setcbconstruct

Purpose

Set a user callback to be called during the Xpress SLP augmentation process

Synopsis

```
problem.setcbconstruct (userfunc, object)
retval = userfunc (my_prob, my_object)
```

Arguments

`userfunc`	The function to be called during problem augmentation. `userfunc` 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 `object` to `setcbconstruct`.
`object`	Address of a user-defined object, which can be used for any purpose by the function. `object` is passed to `userfunc` 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.construct` to recalculate the initial matrix. The second time it is called it frees the allocated memory and returns to `problem.construct` to proceed with the rest of the augmentation.

```
def CBConstruct (myprob, obj) {
  if (obj == 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 (i, None, None, None, None,
                   None, None, cValue[i], None, None, None,
                   None)
    # set Object non-null 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.construct` 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.construct`

problem.setcbdestroy

Purpose

Set a user callback to be called when an SLP problem is about to be destroyed

Synopsis

```
problem.setcbdestroy (userfunc, object) int (XPRS_CC *userfunc) (XSLPprob my_prob,
       void *my_object), void *Object)
```

Arguments

`userfunc`	The function to be called when the SLP problem is about to be destroyed. `userfunc` 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 `object` to `setcbdestroy`.
`object`	Address of a user-defined object, which can be used for any purpose by the function. `object` is passed to `userfunc` 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 != None):
    Obj.inuse = 0
  return 0
```

The `object` argument is used here to hold the address of 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.

Related topics

`problem.destroyprob`

problem.setcbdrcol

Purpose

Set a user callback used to override the update of variables with small determining column

Synopsis

```
problem.setcbdrcol (userfunc, object)
newvalue = userfunc (my_prob, my_object, colindex, drcolindex, drcolvalue, vlb, vub)
```

Arguments

`userfunc`	The function to be called after each column has been cascaded. `userfunc` 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 `object` to `setcbcascadevar`.
`ColIndex`	The index of the column 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.
`object`	Address of a user-defined object, which can be used for any purpose. by the function. `object` is passed to `userfunc` 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.cascade`, `problem.setcbcascadeend`, `problem.setcbcascadestart`

problem.setcbformula

Purpose

Set a callback to be used in formula evaluation when an unknown token is found

Synopsis

```
problem.setcbformula (userfunc, object)
(retval = result) = userfunc (my_prob, my_object, value)
```

Arguments

`userfunc`	The function to be called during formula evaluation. `userfunc` returns an integer value. At present the value is ignored.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `object` to `setcbformula`.
`value`	The `Value` of the unknown token.
`result`	Address of a double precision value to hold the result of the calculation.
`object`	Address of a user-defined object, which can be used for any purpose by the function. `object` is passed to `userfunc` as `my_object`.

Example

The following example sets a callback to process unknown tokens in formulae. It then creates a formula with an unknown token, and evaluates it.

```
def MyCB (MyProb, MyObject, MyValue):
  if (MyValue == None):
    Result = 1
  else:
    Result = 0
  return (0, Result)

p.setcbformula (MyCB,None);

nToken = 0;
type  = [xslp_op_con, xslp_op_unknown, xslp_op_op,   xslp_op_eof]
value = [10,          z,               xslp_op_plus, 0]

answer = p.evaluateformula (1, type, value)

printf ("Answer:", answer)
```

This demonstrates how the `value` of an unknown token can be set in any way, as long as the routine that sets the token up and the callback agree on how it is to be interpreted. In this case, the value actually contains the address of a character string, which is converted by the callback into a real number.

Related topics

problem.evaluateformula

problem.setcbiterend

Purpose

Set a user callback to be called at the end of each SLP iteration

Synopsis

```
problem.setcbiterend (userfunc, object)
retval = userfunc (my_prob, my_object)
```

Arguments

`userfunc`	The function to be called at the end of each SLP iteration. `userfunc` 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 `object` to `setcbiterend`.
`object`	Address of a user-defined object, which can be used for any purpose by the function. `object` is passed to `userfunc` 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 `object` 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.setcbitervarF`

problem.setcbiterstart

Purpose

Set a user callback to be called at the start of each SLP iteration

Synopsis

```
problem.setcbiterstart (userfunc, object)
retval = userfunc (my_prob, my_object)
```

Arguments

`userfunc`	The function to be called at the start of each SLP iteration. `userfunc` 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 `object` to `setcbiterstart`.
`object`	Address of a user-defined object, which can be used for any purpose by the function. `object` is passed to `userfunc` 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.getsol(xprob, Obj, None, None, None)
  return 0
```

The `object` argument is used here to hold the address of 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.setcbitervarF`

problem.setcbitervar

Purpose

Set a user callback to be called after each column has been tested for convergence

Synopsis

```
problem.setcbitervar (userfunc, object)
retval = userfunc (my_prob, my_object, colindex)
```

Arguments

`userfunc` The function to be called after each column has been tested for convergence. `userfunc` 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 `object` to `setcbitervar`.

`ColIndex` The number of the column which has been tested for convergence.

`object` A user-defined object, which can be used for any purpose by the function. `object` is passed to `userfunc` 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 address of 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.setcbitervarF`

problem.setcbmessage

Purpose

Set a user callback to be called whenever Xpress Nonlinear outputs a line of text

Synopsis

```
problem.setcbmessage (userfunc, object)
userfunc (my_prob, my_object, msg, msgtype)
```

Arguments

`userfunc` The function to be called whenever Xpress Nonlinear outputs a line of text. `userfunc` does not return a value.

`my_prob` The problem passed to the callback function.

`my_object` The user-defined object passed as `object` to `setcbmessage`.

`msg` Character buffer holding the 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` which can be passed using `problem.printmsg`

`object` Address of a user-defined object, which can be used for any purpose by the function. `object` is passed to `userfunc` 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 then screen output is automatically disabled.

Output can be directed into a log file by using `problem.setlogfile`.

Related topics

`problem.setcbmessageF`, `problem.setlogfile`,

problem.setcbmsjobend

Purpose

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 (userfunc, object)
status = userfunc (my_prob, my_object, job_object, description)
```

Arguments

`userfunc`	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 `object` 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.setcbmsjobstart`, `problem.setcbmswinner`

problem.setcbmsjobstart

Purpose

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 (userfunc, object)
status = userfunc (my_prob, my_object, job_object, description)
```

Arguments

`userfunc`	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 `object` 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.setcbmsjobend`, `problem.setcbmswinner`

problem.setcbmswinner

Purpose

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 (userfunc, object)
userfunc (my_prob, my_object, job_object, description)
```

Arguments

`userfunc`	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 `object` 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.setcbmsjobstart`, `problem.setcbmsjobend`

problem.setcbslpend

Purpose

Set a user callback to be called at the end of the SLP optimization

Synopsis

```
problem.setcbslpend (userfunc, object)
userfunc (my_prob, my_object)
```

Arguments

`userfunc`	The function to be called at the end of the SLP optimization. `userfunc` 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 `object` to `setcbslpend`.
`object`	Address of a user-defined object, which can be used for any purpose by the function. `object` is passed to `userfunc` 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 != None)
    Obj = []
  else Obj = None
  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

Set a user callback to be called during MISLP after the SLP optimization at each node.

Synopsis

```
problem.setcbslpnode (userfunc, object)
(retval, feas) = userfunc (my_prob, my_object)
```

Arguments

`userfunc`	The function to be called after the set-up of the SLP problem to be solved at a node. `userfunc` 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 `object` to `setcbslpnode`.
`feas`	Address of an integer containing the feasibility flag. If `userfunc` sets the flag nonzero, the node is declared infeasible.
`object`	Address of a user-defined object, which can be used for any purpose by the function. `object` is passed to `userfunc` 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.

Related topics

`problem.setcboptnode`, `problem.setcbprenode`

problem.setcbslpstart

Purpose

Set a user callback to be called at the start of the SLP optimization

Synopsis

```
problem.setcbslpstart (userfunc, object)
retval = userfunc (my_prob, my_object)
```

Arguments

`userfunc`	The function to be called at the start of the SLP optimization. `userfunc` 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 `object` to `setcbslpstart`.
`object`	User-defined object, which can be used for any purpose by the function. `object` is passed to `userfunc` 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.setControl

Purpose

Sets one or more controls of a problem.

Synopsis

```
problem.setControl (string, value)
```

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.

Related topics

`problem.getControl`.

problem.setcurrentiv

Purpose

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.reinitialize, problem.unconstruct

problem.setdefaultcontrol

Purpose

Sets one control to its default values. Must be called before the problem is read or loaded by `problem.read` 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

Sets all controls to their default values. Must be called before the problem is read or loaded by problem.read and 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

Specifies that a set of rows in the problem will be treated as indicator constraints during a global 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 global 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 (mrows, inds, comps)
```

Arguments

`mrows` Array containing the indices of the rows that define the linear inequality part for the indicator constraints.

`inds` Array containing the column indices of the indicator variables.

`comps` Array with the complement flags:

- `0` not an indicator constraint (in this case the corresponding entry in the `inds` 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 global 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.setlogfile

Purpose

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.addcbmessage`, 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.addcbmessage`.

problem.setmessagestatus

Purpose

Manages suppression of messages.

Synopsis

```
problem.setmessagestatus (errcode, status)
```

Arguments

`errcode`	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`	Non-zero 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.setObjective

Purpose

Sets the objective function of the problem.

Synopsis

```
problem.setObjective (expr)
```

Argument

`expr`	A linear or quadratic function of the variables that were added to the problem prior to this call. An error will be returned if any variable in the linear or quadratic part of the objective was not added to the problem via `addVariable`.

Example

The following example sets the objective function of the problem to $[2x_1^2 + 3x_1x_2 + 5x_2^2 + 4x_1 + 4]$:

```
x1 = xpress.var ()
x2 = xpress.var ()
p = xpress.problem ()
p.addVariables (x1, x2)
p.setObjective (2*x1**2 + 3*x1*x2 + 5*x2**2 + 4*x1 + 4)
```

Further information

Multiple calls to `setObjective` are allowed, and each replaces the old objective function with a new one.

Related topics

`problem.addVariable`.

problem.setprobname

Purpose

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.read, problem.name, MAXPROBNAMELENGTH.

problem.setuniqueprefix

Purpose

Find a prefix character string which is different from all the names currently in use within the SLP problem

Synopsis

```
problem.setuniqueprefix ()
```

Example

The following example reads a problem from file and then finds a unique prefix so that new names can be added without fear of duplications:

```
p.read ("Matrix", "")
p.setuniqueprefix ()
s = p.attributes.xslp_uniqueprefix
print ("No names start with ", s)
```

Further information

The unique prefix may be more than one character in length, and may change if new names are added to the problem. The value of the unique prefix can be obtained from the string attribute `xslp_uniqueprefix`.

problem.solve

Purpose

Solves the current problem.

Synopsis

```
problem.solve (flags)
```

Argument

`flags`	(optional) a string with flags expressed as characters.

problem.storebounds

Purpose

Stores bounds for node separation using user separate callback function.

Synopsis

```
problem.storebounds (mcols, type, bds, mindex)
```

Arguments

`nbnds`	Number of bounds to store.
`mcols`	Array containing the column indices.
`qbtype`	Array containing the bounds types: `U` indicates an upper bound; `L` indicates a lower bound.
`dbds`	Array containing the bound values.
`mindex`	Object that the user will use to reference the stored bounds for the Optimizer in `problem.setbranchbounds`.

Related topics

`problem.setbranchbounds`.

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 or problem.setbranchcuts before they become active.

Synopsis

```
problem.storecuts (nodupl, type, rtype, rhs, mstart, mindex, mcols, matval)
```

Arguments

nodupl	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.
mtype	Array of length ncuts containing the cut types. The cut types can be any integer and are used to identify the cuts.
qrtype	Character array of length ncuts containing the row types: L indicates a $\leq$ row; E indicates an = row; G indicates a $\geq$ row.
drhs	Array containing the right hand side elements for the cuts.
mstart	Array containing offsets into the mcols and dmtval arrays indicating the start of each cut. This array is of length ncuts+1 where ncuts is the length of drhs, with the last element mstart[ncuts] being where cut ncuts+1 would start.
mindex	Array of length ncuts where the pointers to the cuts will be returned.
mcols	Array of length mstart[ncuts] containing the column indices in the cuts.
dmatval	Array of length mstart[ncuts] containing the matrix values for the cuts.

Further information

1. storecuts can be used to eliminate duplicate cuts. If the nodupl 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 nodupl 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 mindex array. If the cut is not added to the cut pool because a stronger cut exits a NULL will be returned. The mindex array can be passed directly to problem.loadcuts or problem.setbranchcuts to load the most recently stored cuts into the matrix.

3. The columns and elements of the cuts must be stored contiguously in the mcols and dmtval arrays passed to storecuts. The starting point of each cut must be stored in the mstart array. To determine the length of the final cut the mstart 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.setbranchcuts, Section "Working with the cut manager" of the Xpress Optimizer reference manual.

problem.strongbranch

Purpose

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 (mbndind, cbndtype, dbndval, itrlimit, dsbobjval, msbstatus)
```

Arguments

`mbndind`	Array containing the indices of the columns on which the bounds will change.
`cbndtype`	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.
`dbndval`	Array giving the new bound values.
`itrlimit`	Maximum number of LP iterations to perform for each bound change.
`dsobjval`	Objective value of each LP after performing the strong branching iterations.
`msbstatus`	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:

```
dsbobjval = []
msbstatus = []
p.strongbranch ([0,0,1,0], ['','','',''], [1,0,2,1],
                1000, dsbobjval, msbstatus)
```

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

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 (bndind, bndtype, bndval, itrlimit, objval, status, sbsolvecb,
       vContext)
ret = sbsolvecb (prob, vContext, ibnd)
```

Arguments

`bndind`	Array of size `nbnds` containing the indices of the columns on which the bounds will change.
`bndtype`	Character array of length `nbnds` 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 of length `nbnds` giving the new bound values.
`itrlimit`	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.
`sbsolvecb`	Function to be called after each strong branch has been reoptimized.
`vContext`	User context to be provided for `sbsolvecb`.
`ibnd`	The index of bound for which `sbsolvecb` 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.getlpsol`.

problem.tokencount

Purpose

Count the number of tokens in a free-format character string

Synopsis

```
token = problem.tokencount (record)
```

Argument

`record` The character string to be processed. This must be terminated with a null character.

Return value

The number of tokens (strings separated by one or more spaces) in `record`.

Example

The following example counts the number of tokens in the string "sin (x + y)":

```
int nToken;
nToken = p.tokencount ("sin ( x + y )")
```

Further information

`Record` should follow the conventions for Extended MPS Format, with each token being separated by one or more spaces from the previous token.

Related topics

`problem.qparse`

problem.tune

Purpose

Begin a tuner sesssion 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 `xpress.attributes.objsense`.

Synopsis

```
problem.tune (flags)
```

Argument

`flags` Flags to 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`);
- `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.

Example

```
p.tune ('dp')
```

This tunes the current problem. The problem type is automatically determined. If it is an LP problem, it will be solved with a concurrent run of the dual and primal simplex method. If it is a MIP problem, the initial LP relaxation of the MIP will be solved with a concurrent run of primal and dual simplex.

Further information

Please refer to the Xpress Optimizer reference manual for a detailed guide of how to use the tuner.

problem.tunerreadmethod

Purpose

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.

problem.tunerwritemethod

Purpose

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.

problem.unconstruct

Purpose

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.createprob, problem.destroyprob, problem.reinitialize, problem.setcurrentiv,

problem.updatelinearization

Purpose

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.construct`.

Related topics

`problem.construct`

problem.validate

Purpose

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

`XSLPvalidate` 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 `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

`xslp_validationindex_A`, `xslp_validationindex_R`, `xslp_validationtol_A`, `xslp_validationtol_R`

problem.validatekkt

Purpose

Validates the first order optimality conditions also known as the Karush-Kuhn-Tucker (KKT) conditions versus the currect solution

Synopsis

```
problem.validatekkt (calculationmode, respectbasisstatus, updatemultipliers,
       kktviolationtarget)
```

Arguments

`calculationmode` 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.

`respectbasisstatus` 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.

`updatemultipliers` 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.

`kktviolationtarget` 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 `kktviolationtarget` are automatically relaxed if the desired accuracy cannot be achieved.

problem.validaterow

Purpose

Prints an extensive analysis on a given constraint of the SLP problem

Synopsis

```
problem.validate (row)
```

Argument

`row` The index of the row 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

Validate the feasibility of constraints for a given solution

Synopsis

```
(suminf, sumscaledinf, obj) = problem.validate (vector)
```

Arguments

`vector`	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.validate`, and will update `xslp_validationindex_a` and `xslp_validationindex_r`.

Related topics

`Xslp_Validationindex_a`, `xslp_validationindex_r`, `xslp_validationtol_a`, `xslp_validationtol_r`

problem.validformula

Purpose

Check a formula in internal (parsed or unparsed) format for unknown tokens

Synopsis

```
(token, name, stringtable) = problem.validformula (type, value)
```

Arguments

`type`	Array of token types providing the formula.
`value`	Array of values corresponding to the types in `inType`
`ntoken`	Number of the first invalid token in the formula. A value of zero means that the formula is valid. May be `None` if not required.
`name`	Character buffer to hold the name of the first invalid token. May be `None` if not required.
`stringtable`	Character buffer holding the names of the unidentified tokens (this can be created by `problem.preparseformula`).

Related topics

`problem.preparseformula`

problem.write

Purpose

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.read` 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.read`.

problem.writebasis

Purpose

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. The Optimizer input commands might then be:

```
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.writebinsol

Purpose

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.getlpsol`, `problem.getmipsol`, `problem.readbinsol`, `problem.writesol`, `problem.writeprtsol`.

problem.writedirs

Purpose

Writes the global 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.writeprtsol

Purpose

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 command 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.getlpsol`, `problem.getmipsol`, `problem.readbinsol`, `problem.writebinsol`, `problem.writesol`.

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.

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 global 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.writesol

Purpose

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 or goal programming 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 command 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.getlpsol`, `problem.getmipsol`, `problem.writerange`, `problem.writeprtsol`.

branchobj.addbounds

Purpose

Adds new bounds to a branch of a user branching object.

Synopsis

```
branchobj.addbounds (ibranch, bndtype, bndcol, bndval)
```

Arguments

`ibranch` 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 of length `nbounds` indicating the type of bounds to add:

- `L` Lower bound.
- `U` Upper bound.

`bndcol` Array of length `nbounds` containing the columns for the new bounds.

`bndval` Array of length `nbounds` giving the bound values.

branchobj.addbranches

Purpose

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

Adds stored user cuts as new constraints to a branch of a user branching object.

Synopsis

```
branchobj.addcuts (ibranch, cutind)
```

Arguments

`ibranch`	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.addrows

Purpose

Adds new constraints to a branch of a user branching object.

Synopsis

```
branchobj.addrows (ibranch, rtype, rhs, beg, mcol, val)
```

Arguments

`ibranch`	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.
`rtype`	Character array of length `nrows` indicating the type of constraints to add: `L` Less than type. `G` Greater than type. `E` Equality type.
`rhs`	Array of length `nrows` containing the right hand side values.
`beg`	Array of length `nrows` containing the offsets of the `mcol` and `dval` arrays of the start of the non zero coefficients in the new constraints.
`mcol`	Array of length `nelems` containing the column indices for the non zero coefficients.
`dval`	Array of length `nelems` containing the non zero coefficient values.

Example

The following function will create a branching object that branches on constraints $x_1 + x_2 \geq 1$ or $x_1 + x_2 \leq 0$:

```
def CreateConstraintBranch (mip, icol):

    # Create the new object with two empty branches.
    bo = xpress.branchobj (mip, isoriginal = True)
    bo.addbranches (2)

    # Add the constraint of the branching object:
    # x1 + x2 >= 1
    # x1 + x2 <= 0
    bo.addrows (0, 1, 2, ['G'], [1.0], [0], [0,1], [1.0,1.0])
    bo.addrows (1, 1, 2, ['L'], [0.0], [0], [0,1], [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

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 (ibranch, nbounds_size, bndtype, bndcol, bndval)
```

Arguments

`ibranch` The number of the branch to get the bounds for.

`nbounds_size` Maximum number of bounds to return.

`bndtype` Character array of length `nbounds_size` where the types of bounds twill be returned:

- `L` Lower bound.
- `U` Upper bound.

`bndcol` Array of length `nbounds_size` where the column indices will be returned.

`bndval` Array of length `nbounds_size` where the bound values will be returned.

Related topics

`branchobj.addbounds`.

branchobj.getbranches

Purpose

Returns the number of branches of a branching object.

Synopsis

```
branchobj.getbranches ()
```

Related topics

branchobj.addbranches.

branchobj.getid

Purpose

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

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

Returns the constraints for a branch of a user branching object.

Synopsis

```
branchobj.getrows (ibranch, nrows_size, nelems_size, rtype, rrhs, rbeg, mcol, dval)
```

Arguments

`ibranch`	The number of the branch to get the constraints from.
`nrows_size`	Maximum number of rows to return.
`nelems_size`	Maximum number of non zero coefficients to return.
`rtype`	Character array of length `nrows_size` where the types of the rows will be returned: `L` Less than type. `G` Greater than type. `E` Equality type.
`rhs`	Array of length `nrows_size` where the right hand side values will be returned.
`mbeg`	Array of length `nrows_size` which will be filled with the offsets of the `mcol` and `dval` arrays of the start of the non zero coefficients in the returned constraints.
`mcol`	Array of length `nelems_size` which will be filled with the column indices for the non zero coefficients.
`dval`	Array of length `nelems_size` which will be filled with the non zero coefficient values.

Related topics

`branchobj.addrows`.

branchobj.setpreferredbranch

Purpose

Specifies which of the child nodes corresponding to the branches of the object should be explored first.

Synopsis

```
branchobj.setpreferredbranch (ibranch)
```

Argument

`ibranch`	The number of the branch to mark as preferred.

branchobj.setpriority

Purpose

Sets the priority value of a user branching object.

Synopsis

```
branchobj.setpriority (ipriority)
```

Argument

`ipriority`	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 global 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.

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 function set by `problem.addcboptnode`.

Synopsis

```
status = branchobj.store ()
```

Argument

`status` 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.

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`.
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`.
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.

Related topics

`branchobj.validate`.

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.

Synopsis

```
status = branchobj.validate ()
```

Argument

`status` 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.

APPENDIX A

Contacting FICO

FICO provides clients with support and services for all our products. Refer to the following sections for more information.

Product support

FICO 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 purchased a FICO product and have an active support or maintenance contract. You can find support contact information on the Product Support home page (www.fico.com/support).

On the Product Support home page, you can also register for credentials to log on to FICO Online Support, our web-based support tool to access Product Support 24x7 from anywhere in the world. Using FICO Online Support, you can enter cases online, track them through resolution, find articles in the FICO Knowledge Base, and query known issues.

Please include *'Xpress'* in the subject line of your support queries.

Product education

FICO Product Education is the principal provider of product training for our clients and partners. Product Education 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 Product Education homepage at www.fico.com/en/product-training or email producteducation@fico.com.

Product 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.

Sales and maintenance

USA, CANADA AND ALL AMERICAS

Email: XpressSalesUS@fico.com

WORLDWIDE

Email: XpressSalesUK@fico.com

Tel: +44 207 940 8718
Fax: +44 870 420 3601

Xpress Optimization, FICO
FICO House
International Square
Starley Way
Birmingham B37 7GN
UK

Related services

Strategy Consulting: Included in your contract with FICO may be a specified amount of consulting time to assist you in using FICO Optimization Modeler to meet your business needs. Additional consulting time can be arranged by contract.

Conferences and Seminars: FICO offers conferences and seminars on our products and services. For announcements concerning these events, go to www.fico.com or contact your FICO account representative.

About FICO

FICO (NYSE:FICO) delivers superior predictive analytics solutions that drive smarter decisions. The company's groundbreaking use of mathematics to predict consumer behavior has transformed entire industries and revolutionized the way risk is managed and products are marketed. FICO's innovative solutions include the FICO® Score—the standard measure of consumer credit risk in the United States—along with industry-leading solutions for managing credit accounts, identifying and minimizing the impact of fraud, and customizing consumer offers with pinpoint accuracy. Most of the world's top banks, as well as leading insurers, retailers, pharmaceutical companies, and government agencies, rely on FICO solutions to accelerate growth, control risk, boost profits, and meet regulatory and competitive demands. FICO also helps millions of individuals manage their personal credit health through www.myfico.com. Learn more at www.fico.com. FICO: Make every decision count™.

Index

B

branchobj.addbounds, 375
branchobj.addbranches, 376
branchobj.addcuts, 377
branchobj.addrows, 378
branchobj.getbounds, 379
branchobj.getbranches, 380
branchobj.getid, 381
branchobj.getlasterror, 382
branchobj.getrows, 383
branchobj.setpreferredbranch, 384
branchobj.setpriority, 385
branchobj.store, 386
branchobj.validate, 387

P

problem.addcbbariteration, 78
problem.addcbbarlog, 80
problem.addcbchgbranchobject, 81
problem.addcbcutlog, 82
problem.addcbdestroymt, 83
problem.addcbgapnotify, 84
problem.addcbgloballog, 86
problem.addcbinfnode, 87
problem.addcbintsol, 88
problem.addcblplog, 89
problem.addcbmessage, 90
problem.addcbmipthread, 91
problem.addcbnewnode, 92
problem.addcbnodecutoff, 93
problem.addcboptnode, 94
problem.addcbpreintsol, 95
problem.addcbprenode, 96
problem.addcbusersolnotify, 97
problem.addcoefs, 98
problem.addcols, 100
problem.addConstraint, 101
problem.addcuts, 102
problem.adddfs, 103
problem.addIndicator, 104
problem.addmipsol, 105
problem.addqmatrix, 106
problem.addrows, 107
problem.addSOS, 108
problem.addtolsets, 109
problem.addVariable, 110
problem.addvars, 111
problem.basisstability, 112
problem.btran, 113
problem.calcobjective, 114
problem.calcreducedcosts, 115
problem.calcslacks, 116
problem.calcsolinfo, 117
problem.cascade, 118
problem.cascadeorder, 119
problem.chgbounds, 120
problem.chgcascadenlimit, 123
problem.chgccoef, 124
problem.chgcoef, 121
problem.chgcoltype, 122
problem.chgdeltatype, 125
problem.chgdf, 126
problem.chgglblimit, 127
problem.chgmcoef, 128
problem.chgmqobj, 129
problem.chgnlcoef, 130
problem.chgobj, 131
problem.chgobjsense, 132
problem.chgqobj, 133
problem.chgqrowcoeff, 134
problem.chgrhs, 135
problem.chgrhsrange, 136
problem.chgrowstatus, 137
problem.chgrowtype, 138
problem.chgrowwt, 139
problem.chgtolset, 140
problem.chgvar, 142
problem.construct, 144
problem.copy, 145
problem.copycallbacks, 146
problem.copycontrols, 147
problem.delcoefs, 148
problem.delConstraint, 149
problem.delcpcuts, 150
problem.delcuts, 151
problem.delqmatrix, 152
problem.delSOS, 153
problem.deltolsets, 154
problem.delVariable, 155
problem.delvars, 156
problem.dumpcontrols, 157
problem.estimaterowdualranges, 158
problem.evaluatecoef, 159
problem.evaluateformula, 160
problem.filesol, 161
problem.fixglobals, 162
problem.fixpenalties, 163
problem.ftran, 164
problem.getAttrib, 165
problem.getattribinfo, 166
problem.getbasis, 167
problem.getccoef, 168
problem.getcoef, 169
problem.getcoefformula, 170

problem.getcoefs, 171
problem.getcolinfo, 172
problem.getcols, 173
problem.getcoltype, 174
problem.getConstraint, 175
problem.getControl, 176
problem.getcontrolinfo, 177
problem.getcpcutlist, 178
problem.getcpcuts, 179
problem.getcutlist, 180
problem.getcutmap, 181
problem.getcutslack, 182
problem.getdf, 184
problem.getdirs, 183
problem.getdtime, 185
problem.getDual, 186
problem.getdualray, 187
problem.getglobal, 188
problem.getiisdata, 189
problem.getIndex, 191
problem.getIndexFromName, 192
problem.getindicators, 193
problem.getinfeas, 194
problem.getlasterror, 195
problem.getlb, 196
problem.getlpsol, 197
problem.getmessagestatus, 198
problem.getmessagetype, 199
problem.getmipsol, 200
problem.getmqobj, 201
problem.getobj, 202
problem.getObjVal, 203
problem.getpivotorder, 204
problem.getpivots, 205
problem.getpresolvebasis, 206
problem.getpresolvemap, 207
problem.getpresolvesol, 208
problem.getprimalray, 209
problem.getProbStatus, 210
problem.getProbStatusString, 211
problem.getqobj, 212
problem.getqrowcoeff, 213
problem.getqrowqmatrix, 214
problem.getqrowqmatrixtriplets, 215
problem.getqrows, 216
problem.getRCost, 217
problem.getrhs, 218
problem.getrhsrange, 219
problem.getrowinfo, 220
problem.getrows, 221
problem.getrowstatus, 222
problem.getrowtype, 223
problem.getrowwt, 224
problem.getscaledinfeas, 225
problem.getSlack, 226
problem.getslpsol, 227
problem.getSolution, 228
problem.getSOS, 229
problem.gettolset, 230
problem.getub, 231
problem.getunbvec, 232
problem.getvar, 233
problem.getVariable, 235
problem.globalsol, 236
problem.hasdualray, 237
problem.hasprimalray, 238
problem.iisall, 239
problem.iisclear, 240
problem.iisfirst, 241
problem.iisisolations, 242
problem.iisnext, 243
problem.iisstatus, 244
problem.iiswrite, 245
problem.interrupt, 246
problem.loadbasis, 247
problem.loadbranchdirs, 248
problem.loadcoefs, 249
problem.loadcuts, 251
problem.loaddelayedrows, 252
problem.loaddfs, 253
problem.loaddirs, 254
problem.loadlpsol, 255
problem.loadmipsol, 256
problem.loadmodelcuts, 257
problem.loadpresolvebasis, 258
problem.loadpresolvedirs, 259
problem.loadproblem, 260
problem.loadsecurevecs, 262
problem.loadtolsets, 263
problem.loadvars, 264
problem.lpoptimize, 266
problem.mipoptimize, 267
problem.msaddcustompreset, 268
problem.msaddjob, 269
problem.msaddpreset, 270
problem.msclear, 271
problem.name, 272
problem.objsa, 273
problem.parsecformula, 274
problem.parseformula, 275
problem.postsolve, 276
problem.preparseformula, 277
problem.presolve, 278
problem.presolverow, 279
problem.printevalinfo, 281
problem.printmemory, 280
problem.printmsg, 282
problem.read, 283
problem.readbasis, 284
problem.readbinsol, 285
problem.readdirs, 286
problem.readslxsol, 287
problem.refinemipsol, 288
problem.reinitialize, 289
problem.removecbbariteration, 290
problem.removecbbarlog, 291
problem.removecbchgbranchobject, 292
problem.removecbcutlog, 293
problem.removecbdestroymt, 294
problem.removecbgapnotify, 295

problem.removecbgloballog, 296
problem.removecbinfnode, 297
problem.removecbintsol, 298
problem.removecblplog, 299
problem.removecbmessage, 300
problem.removecbmipthread, 301
problem.removecbnewnode, 302
problem.removecbnodecutoff, 303
problem.removecboptnode, 304
problem.removecbpreintsol, 305
problem.removecbprenode, 306
problem.removecbusersolnotify, 307
problem.repairinfeas, 308
problem.repairweightedinfeas, 310
problem.repairweightedinfeasbounds, 312
problem.reset, 314
problem.restore, 315
problem.rhssa, 316
problem.save, 317
problem.scale, 318
problem.scaling, 319
problem.setbranchbounds, 320
problem.setbranchcuts, 321
problem.setcbcascadeend, 322
problem.setcbcascadestart, 323
problem.setcbcascadevar, 324
problem.setcbcascadevarfail, 325
problem.setcbcoefevalerror, 326
problem.setcbconstruct, 327
problem.setcbdestroy, 329
problem.setcbdrcol, 330
problem.setcbformula, 331
problem.setcbiterend, 332
problem.setcbiterstart, 333
problem.setcbitervar, 334
problem.setcbmessage, 335
problem.setcbmsjobend, 336
problem.setcbmsjobstart, 337
problem.setcbmswinner, 338
problem.setcbslpend, 339
problem.setcbslpnode, 340
problem.setcbslpstart, 341
problem.setControl, 342
problem.setcurrentiv, 343
problem.setdefaultcontrol, 344
problem.setdefaults, 345
problem.setindicators, 346
problem.setlogfile, 347
problem.setmessagestatus, 348
problem.setObjective, 349
problem.setprobname, 350
problem.setuniqueprefix, 351
problem.solve, 352
problem.storebounds, 353
problem.storecuts, 354
problem.strongbranch, 355
problem.strongbranchcb, 356
problem.tokencount, 357
problem.tune, 358
problem.tunerreadmethod, 359
problem.tunerwritemethod, 360
problem.unconstruct, 361
problem.updatelinearization, 362
problem.validate, 363
problem.validatekkt, 364
problem.validaterow, 365
problem.validatevector, 366
problem.validformula, 367
problem.write, 368
problem.writebasis, 369
problem.writebinsol, 370
problem.writedirs, 371
problem.writeprtsol, 372
problem.writeslxsol, 373
problem.writesol, 374

X

xpress.abs, 70
xpress.acos, 66
xpress.addcbmsghandler, 76
xpress.asin, 65
xpress.atan, 67
xpress.cos, 63
xpress.Dot, 56
xpress.erf, 72
xpress.erfc, 73
xpress.exp, 59
xpress.free, 44
xpress.getbanner, 45
xpress.getcheckedmode, 46
xpress.getdaysleft, 47
xpress.getlasterror, 48
xpress.getlicerrmsg, 49
xpress.getversion, 50
xpress.init, 51
xpress.log, 60
xpress.log10, 61
xpress.max, 68
xpress.min, 69
xpress.Prod, 58
xpress.removecbmsghandler, 77
xpress.setcheckedmode, 52
xpress.setdefaultcontrol, 54
xpress.setdefaults, 53
xpress.sign, 71
xpress.sin, 62
xpress.sqrt, 74
xpress.Sum, 55
xpress.tan, 64
xpress.user, 75