FICO® Xpress Optimization

Last update 01 October 2022

41.01

FICO® Xpress Optimizer Python interface

User's manual

FICO®

Xpress Optimizer

Deliverable Version: A

Last Revised: 01 October 2022

Version 41.01 (FICO® Xpress 9.0)

Contents

CHAPTER 1

Introduction

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

1.1 Outline

The following chapters cover:

- Creating, handling, solving, and querying optimization problems (Chapter 2);
- Using Python numerical libraries such as NumPy to create optimization problems (Chapter 3);
- Setting and getting the value of parameters (controls and attributes) of a problem (Chapter 4);
- Using Python functions as callbacks for the Xpress Optimizer and the Xpress Nonlinear solver (Chapter 5);
- Several examples of usage of the Xpress Python interface (Chapter 6);
- A reference with all functions and parameters in the Python interface (Chapter 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 3.7 to 3.10.

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

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

1.2 Installing the Python Xpress module

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

The install comes with a copy of the *community* license, which allows for solving problems with up to 5000 between variables and constraints. If you already have an Xpress license, please make sure to set the XPAUTH_PATH environment variable to the full path to the license file, `xpauth.xpr`. See also Section 1.2.3 below.

The manual is located in the `xpress/doc` subdirectory of the Python installation folder, and its location can be identified by invoking the `xpress.manual()` function.

1.2.1 Installation from the Python Package Index (PyPI)

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

```
pip install xpress
```

Packages for Python 3.7 to 3.10 are available, for Windows, Linux, and MacOS. The package contains the Python interface module, its documentation in PDF format, the Xpress Optimizer's libraries, various examples of use, and a copy of the community license (see http://subscribe.fico.com/xpress-optimization-community-license). Online documentation can be viewed at the FICO Xpress Optimization Help page.

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

```
pip install xpress==8.11.2
```

1.2.2 Installation from Conda

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

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

The content of the Conda package is the same as that of the PyPI package. Similar to the PyPI package, Conda packages for Python 3.7 to 3.10 are available, for Windows, Linux, and MacOS. Similar to PyPI, the Conda installer fetches the latest version of the package but allows for installing earlier versions as in the following example (note that the Conda installer only uses a single "="):

```
conda install -c fico-xpress xpress=8.11.3
```

1.2.3 Troubleshooting the installation

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

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

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

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

- On Windows, the Xpress installer sets the `XPRESSDIR` environment variable to the installation directory, and the Xpress Python interface will look for a license file at `%XPRESSDIR%\bin\xpauth.xpr`.

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

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

CHAPTER 2

Modeling an optimization problem

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

2.1 Getting started

The Xpress Python module is imported as follows:

```
import xpress
```

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

```
import xpress as xp
```

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

```
from xpress import *
```

2.2 Creating a problem

Create an empty optimization problem `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. The synopsis of the `xpress.problem` method is as follows:

```
xpress.problem(*args, name='noname', sense=xpress.minimize)
```

The only two named arguments are `name` and `sense` and they denote the problem name and the optimization sense, respectively. The argument `args` is a list composed as follows:

- zero or more variables declared with `xpress.var` or `xpress.vars`;
- zero or more constraints created from functions of the variables;
- at most one function in the variables;
- at most one string.

The variables and constraints will be added to the problem as if they were with the `problem.addVariable` and `problem.addConstraint` functions, respectively, while the function is treated as the objective function and added to the problem as if with the `problem.setObjective` function. If the `sense` parameter is also added, this becomes the optimization sense. Because the arguments are scanned in the order they are received, the user ought to ensure that a constraint or the objective function are passed only after all of the variables containing them are passed.

Note that indicator constraints (see Section 2.8) cannot be added directly in the problem declaration but need to be added using `problem.addIndicator`.

The following is an example of the compact declaration: variables `x` and `y` are declared first, then the problem declaration is passed these variables and followed by two constraints and a function to be used as objective function. Note that because no optimization sense is given, minimization is assumed.

```
import xpress as xp
x = xp.var()
y = xp.var(lb=-1, ub=1)
prob = xp.problem(x, y, 2*x + y >= 1, x + 2*y >= 1, x + y, name='myproblem')
```

All operations for adding/deleting variables, constraint, SOS and others are allowed on problems declared this way; note that setting a new objective function with `problem.setObjective` resets the optimization sense, and sets it to `xpress.minimize` if none is given.

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 list 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)
```

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

```
v = xp.var(lb=-xp.infinity)
```

Variable names and Python objects

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

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

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

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

```
x = xp.var(name='var1')
y = xp.var(name='var2')
p = xp.problem(x, y, x + y >= 3, x + 2*y)
p.optimize()
print(p.getSolution([x, y]))
print(p.getSolution(['var1', 'var2']))
```

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

2.4 Constraints

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

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

The parameters are:

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

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

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

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

```
m.addConstraint (myconstr)
m.addConstraint(v1 + xp.tan(v2) <= 3)
m.addConstraint(x[i] + y[i] <= 2 for i in range(10))
myconstr = x1 + x2 * (x2 + 1) <= 4
m.addConstraint(myconstr)
```

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

```
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.

A set of variables or constraint can also be created using Python's fundamental data structure: lists and dictionaries, as well as NumPy's arrays. As described in Section 2.16 below, one can for example create a list of variables `x[i]`, all with upper bound 10, indexed from 0 to `k-1` as follows:

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

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

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

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

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

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

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

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

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

```
xpress.vars(*indices, name='x', lb=0, ub=xpress.infinity,
             threshold = -xpress.infinity, vartype=xpress.continuous)
```

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

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

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

```
y = xpress.vars(['a','b','c','d'], [100, 120, 150], vartype=xpress.integer)
```

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

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

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

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

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

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

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

is equivalent to the following:

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

2.5 Objective function

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

```
setObjective(objective, sense=xpress.minimize)
```

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

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

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

```
x = xp.vars(4,7,5)
```

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

```
x = xp.vars(4,7,5, name="")
```

2.6 Compact formulation

The interface allows for a more compact problem formulation where `xpress.problem` is passed all components of the problem: for instance, consider the code below:

```
import xpress as xp
```

```
x = xp.var(vartype=xp.integer, name='x1', lb=-10, ub=10)
y = xp.var(name='x2')
p = xp.problem(x, y, x**2 + 2*y, x + 3*y <= 4, name='myexample', sense=xp.maximize)
p.optimize()
```

The declaration of `p` is equivalent to the following:

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

2.7 Special Ordered Sets (SOSs)

A Special Order Set (SOS) is a modeling tool for constraining a small number of consecutive variables in a list to be nonzero. The 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 `problem.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([xp.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.

2.8 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)
```

2.9 Piecewise linear functions

Other types of constraints are available for modelling. *Piecewise linear* constraints allow to define a

variable as a piecewise linear function of another. The function does not have to be continuous, but please see the Optimizer's manual for information on how discontinuities are dealt with.

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

```
x = xp.var(lb=-xp.infinity)
y = xp.var()
z1 = xp.var(lb=-xp.infinity)
z2 = xp.var(lb=-xp.infinity)

p = xp.problem(x,y,z1,z2)

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

The above example creates variables `x`, `y`, `z1`, and `z2`, then constrains `z1` and `z2` to be (piecewise linear) functions of `x`, to be used with `y` in other constraints and in the objective function.

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

```
x = xp.var(lb=-xp.infinity)
y = xp.var()
p = xp.problem(x, y)

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

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

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

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

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

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

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

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

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

2.10 General constraints

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

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

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

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

```
x = [xp.var() for _ in range(7)]
y1 = xp.var()
y2 = xp.var()
y3 = xp.var()
type = [xpress.gencons_max, xpress.gencons_abs, xpress.gencons_and]
resultant = [y1, y2, y3]
colstart = [0, 2, 3]
col = [x[0], x[1], x[3], x[4], x[5], x[6]]
valstart = [0,1,1]
val = [46]
p = xp.problem(x, y1, y2, y3)
prob.addgencons(type, resultant, colstart, col, valstart, val);
prob.optimize()
```

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

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

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

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

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

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

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

2.11 Using `loadproblem` for efficiency

The high-level functions `problem.addConstraint` and `problem.addVariable` allow for efficient, concise, and understandable modeling of any optimization problem. An even faster way to create a problem is through the `problem.loadproblem` function, which uses a more direct interface to the Optimizer's libraries and is hence preferable with very large problems and when efficiency in model creation is necessary.

The functon `problem.loadproblem` can be used to create problems with linear and/or quadratic constraints, a linear and/or quadratic objective function, and with continuous and/or discrete variables. Its syntax with default parameter values allows for specifying only the components of interest. We refer the reader to its `entry` in Chapter 7, and present here a few examples of usages. More examples are shown in Chapter 6.

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

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

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

```
import xpress as xp
p = xp.problem()
p.loadproblem(probname='myexample',
              rowtype=['L', 'G'],    # constraint senses
              rhs=[4, 8],            # right-hand sides
              rng=None,              # no range rows
              objcoef=[0, 2],        # linear obj. coeff.
              start=[0, 2, 4],       # start pos. of all columns
              collen=None,           # unused
              rowind=[0, 1, 0, 1],   # row index in each column
              rowcoef=[1, 7, 3, 4],  # coefficients
              lb=[-10,0],            # variable lower bounds
              ub=[10,xp.infinity],   #          upper bounds
              objqcol1=[0],          # quadratic obj. terms, column 1
              objqcol2=[0],          #                       column 2
              objqcoef=[2],          #                       coeff
              coltype=['I'],         # variable types
              entind=[0],            # index of integer variable
              colnames=['x1', 'x2'])
```

Apart from the intuitive lists `qrtypes` (for constraint types: `'L'` for "lesser-than", `'G'` for "greater-than", `'E'` for "equal-to"), `rhs` (constraints' right-hand sides), `obj` (objective linear coefficients), `dlb` and `dub` (variables' lower and upper bounds), a few paramters deserve some attention. The three lists `mstart`, `mrwind`, `dmatval` describe the coefficient matrix: `mrwind` and `dmatval` contain, respectively, the *row* indices and the coefficients, while `mstart` is a list of $n + 1$ integers (where n is the number of variables,

i.e., the size of `obj`, `dlb`, and `dub`); `mstart[i]` indicates the position, within `mrwind` and `dmatval`, of the indices and coefficients of the `i`-th column. The last element `mstart[n+1]` indicates the number of nonzeros in the matrix.

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

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

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

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

2.12 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) 2*x1, 2*x2)

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

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

p = xp.problem()

p.addVariable(x,y)

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

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

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

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

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

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.

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

2.13 Solving a problem

Simply call `problem.optimize` to solve an optimization problem that was either built or read from a file. The type of solver is determined based on the type of problem: if at least one integer variable was declared, then the problem will be solved as a mixed integer (linear, quadratically constrained, or nonlinear) problem, while if all variables are continuous the problem is solved as a continuous optimization problem. If the problem is nonlinear in that it contains non-quadratic, non-conic nonlinear constraints, then the appropriate nonlinear solver of the Xpress Optimization suite will be called. 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.optimize ()
```

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

```
import xpress as xp

m = xp.problem()
m.read("example3.lp")
solvestatus, solstatus = m.optimize()

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

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

2.14 Querying a problem

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

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

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

Finally, an expression, which can be just a variable, will yield the value of the expression given the current solution.

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

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

```
import xpress as xp
import numpy as np

v1 = xp.var(name='Var1')
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.optimize()

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

Consider all lines after `m.optimize()`. The first of them returns a Python list of *ncol* floating point scalars, where *ncol* is the number of variables of the problem (*nrow* is the number of constraints, the size of the list returned by `problem.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`. The fourth example shows that a range of indices can be specified in order to obtain a list of values without specifying the corresponding variables. Recall that the column and row indices begin at 0. The fifth line shows that a variable can be passed by name, while the sixth line shows that passing a dictionary with variables, expression, indices, or variable names returns a dictionary with the same keys as the dictionary passed, but with its values set to the values of the corresponding variables/expressions.

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

The method `problem.getSlack` works with indices, constraint names, constraint objects, and lists thereof. 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.addConstraint(con1, con2)
m.setObjective(xp.Sum(x[i] for i in range(N))
m.optimize()

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 as a list of floats
b = m.getSlack(range(2))             # gets the slack of con1 and con2[0]
```

In addition, for problems with only continuous variables, the two methods `problem.getDual` and `problem.getRCost` return the list (or a portion thereof) of dual variables and reduced costs, respectively. Their usage is similar to that of `problem.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 `problem.getSolution`, `problem.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.addConstraint(xp.Sum(x[i] * i for i in N) <= n))
m.setObjective(xp.Sum(x[i] for i in N)
m.optimize()

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) list:

```
sol = m.getSolution()

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

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

2.15 Reading and writing a problem

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

which we refer.

```
import xpress as xp

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

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")
```

2.16 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.optimize()
```

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

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

m.optimize()
```

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

2.17 Exceptions

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

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

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

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

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 requires NumPy version 1.16 or greater, except on Python 3.10, where it requires NumPy version 1.21 or greater.

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

The NumPy arrays must have the attribute dtype equal to xpress.npvar (abbreviated to xp.npvar here) in order to use the matricial/vectorial form of the comparison (<=, =, >=), arithmetic (+, –, *, /, **), and logic (&, |) operators.

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

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

These can be written equivalently in the compact form as

```
import numpy as np
import xpress as xp
x = xp.vars(5, 4, name='v')
y = xp.vars(3, 3, 3, vartype=xp.binary)
z = xp.vars(1000, lb=-1, ub=1)
```

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

The main advantage of using `NumPy` operations is the ability to replicate them on each element of an array, taking into account all *broadcasting* features. For example, the following script "broadcasts" the right-hand side 1 to all elements of the array, thus creating the set of constraints $x_i + y_i \leq 1$ 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 = xp.vars(4, 5, 10)
t = xp.vars(4, 5, 10, vartype=xp.binary)
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))
```

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

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> sum(a)
array([5, 7, 9])
```

For the casual NumPy user, suffice it to say that the sum is done on the *first* dimension. Similarly, when creating a NumPy array of dimensions k of expressions, `xpress.Sum` returns a $(k - 1)$-array resulting from the sum across the first dimension.

It is important to note the following: NumPy does **not** use the `__iadd__` operator when computing these sums, but rather the `__add__` operator. For reasons discussed above and in the entry regarding the `xpress.Sum` operator, this can have a huge impact on performance. Consider the following example:

```
m,n = 1000,10
a = np.random.random((m,n))
x = xp.vars(m, n)
sum_0d = xp.Sum(xp.Sum(a*x))
sum_1d = xp.Sum(a*x)
```

The above example has a poor performance, and it is advised to avoid using `xpress.Sum` as such on a multi-dimensional array. If a scalar sum of all elements of the array is sought, such as `sum(sum(a))` for the numerical array above, we strongly advise to flatten the array first, and run instead `xpress.Sum(b.flatten())` if `b` is a multiarray of expressions. The multiarray has `dtype` equal to `xpress.npexpr` in order to be used for array operations. If only one pass is required, then it is better to explicitly create a vector whose elements are defined with a call to `xpress.Sum`:

```
prod = a*x
sum_0d = xp.Sum(prod.flatten())
sum_1d = np.array([xp.Sum(prod[i,j] for i in range(m)) for i in range(n)], dtype=xp.npexpr)
```

3.2 Products of NumPy arrays

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

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

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

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

```
coeff_pre = np.random.random((6,3,7))
x = xp.vars(4, 7, 5)
y = xp.vars(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, for example, by functions for adding constraints or variables, or functions for setting and modifying the objective function.

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

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

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

4.1 Controls

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

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

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

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

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

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

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

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

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

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

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

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

4.2 Examples

```
import xpress as xp

p = xp.problem()

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

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

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

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

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

p.setControl(myctrl)
```

4.3 Attributes

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

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

The first form is for obtaining the value of the attribute `attr`. The output will be the value of the attribute. The second and third forms are for retrieving `attr1` , `attr2` , ..., and `attrk`. Whether the attributes are

declared in a list or a tuple does not matter. The result will be a dictionary coupling each attribute with its value. The last form is to obtain all attributes; the result is a dictionary coupling all attributes with their respective value.

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

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

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

4.4 Examples

```
import xpress as xp

p = xp.problem()

p.read("example.lp")

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

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

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

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

attributes = p.getAttrib()
```

4.5 Accessing controls and attributes as object members

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

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

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

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

```
import xpress as xp
print(xp.controls.heuremphasis)
xp.controls.feastol = 1e-4 # Set new default to 1e-4
```

These "global" controls are maintained throughout while the Xpress module is loaded. Note that the

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

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

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

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

```
import xpress as xp

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

p1 = xp.problem("problem1")
p1.controls.miprelstop = 0.1 * xp.controls.miprelstop
p1.controls.feastol = 1e-5
p1.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.optimize()
p2.optimize()

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

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

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

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

When using the Python prompt in creating problems with the Xpress module, the name of controls and

attributes can be auto-completed by pressing TAB (note: this only works in Python 3.4 and subsequent versions). For instance,

```
>>> import xpress
>>> p = xp.problem()
>>> p.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, with linear or nonlinear constraints. Their main purpose is to provide the user with a point of entry into the branch-and-bound, which is the workhorse algorithm for MIPs.

Using callback functions is simple: the user first defines a function (say `myfunction`) that is to be run every time the branch-and-bound reaches a well-specified point; second, the user calls a function (such as `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();`
- 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.optimize()
```

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

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

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 list 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'], # rowtype
              [-2.4, -3, 4, 5],    # rhs
               None,                # rng
              [3,4,5],             # objcoef
              [0,2,4,8],           # start
```

```
                 None,               # collen
                [0,1,2,3,0,1,2,3],  # rowind
                [1,2,2,1,1,3,3,1],  # rowcoef
                [-1,-1,-1],         # lb
                [3,5,8],            # ub
                 colnames = ['X1','X2','X3'],                   # column names
                 rownames = ['row1','row2','row3','constr_04']) # row    names

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

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

```
x = xp.var()
p.addVariable(x)
p.setObjective(x**2 + 2*x + 444)
p.optimize()
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'], # rowtype
              [-2.4, -3, 4, 5],   # rhs
               None,               # rng
              [3,4,5],            # objcoef
              [0,2,4,8],          # start
               None,               # collen
              [0,1,2,3,0,1,2,3],  # rowind
              [1,1,1,1,1,1,1,1],  # rowcoef
              [-1,-1,-1],         # lb
              [3,5,8],            # ub
              [0,0,0,1,1,2],      # objqcol1
              [0,1,2,1,2,2],      # objqcol2
              [2,1,1,2,1,2],      # objqcoef
              [2,3],              # qrowind
              [2,3],              # nrowqcoefs
              [1,2,0,0,2],        # rowqcol1
              [1,2,0,2,2],        # rowqcol2
              [3,4,1,1,1],        # rowqcoef
              ['I','S'],          # coltype
              [0,1],              # entind
              [0,2],              # limit
              ['1','1'],          # settype
              [0,2,4],            # setstart
              [0,1,0,2],          # setind
              [1.1,1.2,1.3,1.4])  # refval

p.optimize()
```

6.2 Modeling examples

6.2.1 A simple model

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

solvestatus, solstatus = m.optimize()

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

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

6.2.2 Using IIS to investigate an infeasible problem

The problem modeled below is infeasible,

```
import xpress as xp

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.optimize()
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 list of N=5 variables and sets constraints and objective. We define all constraints and the objective function using a Python aggregate type.

```
import xpress as xp

N = 5
S = range(N)

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.optimize()

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.

```
import xpress as xp

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

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.optimize()
```

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], dtype=xp.npvar)
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.optimize()
```

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.

```
import numpy as np
import xpress as xp

# digraph definition

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

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

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

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

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

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

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

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

flow = np.array([xp.var() for i in E], dtype=xp.npvar)  # 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.optimize()

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 $(a - x)^2 + b(y - x^2)^2$.

```
import xpress as xp

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

solvestatus, solstatus = p.optimize()

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

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

p = 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.optimize()
```

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

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

6.2.9 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.optimize()

print('Solution:')

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

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

6.3 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], dtype=xp.npvar)

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.optimize()
```

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

```
mstart1=[]
mclind1=[]
dqe1=[]
m.getqrowqmatrix(cons00, 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)], dtype=xp.npvar)

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.optimize()

A   = np.array(range(30)).reshape(6,5) # A is a 6x5 matrix
sol = np.array(p.getSolution()) # a vector of size 5
columns = A*sol         # not a matrix-vector product!
```

```
v = np.dot(A,sol)      # an array: matrix-vector product A*sol
w = xp.Dot(A,x)        # an array of expressions

print(v,w)
```

6.3.3 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)], dtype=xp.npvar)  # 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.optimize()
```

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)], dtype=xp.npvar)
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.optimize('')

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, and analysis

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

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

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

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

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

    card = 0

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

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

    card_subtree[node] = 1 + card
```

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

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

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

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

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

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

    if node in left.keys():

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

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

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

    if node in right.keys():

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

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

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

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

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

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

```
T = nx.Graph()

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

start_time = time.time()

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

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

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

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

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

6.4.2 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.

```
import xpress as xp

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

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

p.addVariable(x)

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

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

p.addConstraint(c0, cc)

p.setObjective(3 - x[0])

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

# add a variable with its coefficients

p.addcols([4], [0,3], [c0,4,2], [-3, 2.4, 1.4], [0], [2], ['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.

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

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

We then add the seventh and eighth constraints:

subject to:	$x_0 + 2\,x_1 + 3x_2$	$\geq$	4
	$4x_0 + 5x_1 + 6x_2 + 7\,x_3 + 8\,x_4 - 3\,y$	$\leq$	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.optimize()
p.write("amended", "lp")

slacks = []

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

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

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

```
p.addcols([4], [0,3], [c0,4,2], [-3,  -2,   1], [0],  [2], ['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.optimize()

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

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

```
p.addcols([4,4,4,4,4],
          [0,3,6,9,12,15],
          [c0,4,2,c0,4,2,c0,4,2,c0,4,2,c0,4,2],
          [3, -2, 1, -3, 2.4, 1.4, 3, 2, 1, -3, 2.4, 4, 3, 2, 1],
          [0,0,0,0,0],
          [2,10,1,2,2],
          ['p1','p2','p3','p4','p5'],
          ['I','C','S','P','R'])
```

6.4.4 Comparing the coefficients of two equally sized problems

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

```
import xpress as xp
import scipy.sparse

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

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

Next we obtain the matrix representation of the coefficient matrix for both problems. Let us suppose that, for memory reasons, we can only retrieve one million coefficients.

```
coef1, ind1, beg1 = [], [], []
coef2, ind2, beg2 = [], [], []

p1.getrows(beg1, ind1, coef1, 1000000, 0, p1.attributes.rows - 1)
p2.getrows(beg2, ind2, coef2, 1000000, 0, p2.attributes.rows - 1)
```

The function `problem.getrows` provides a richer output by filling up `ind1` and `ind2` with the Python objects (i.e. Xpress variables) corresponding to the variable indices rather than the numerical indices. We need to convert them to numerical indices using the `problem.getIndex` function.

```
ind1n = [p1.getIndex(v) for v in ind1]
ind2n = [p2.getIndex(v) for v in ind2]
```

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

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

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


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

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

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

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

6.4.5 Combining modeling and API functions

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

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

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

After solving the problem, we obtain its variables through `getVariable` and add a constraints so that their sum cannot be more than 1.1.

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

6.4.6 A simple Traveling Salesman Problem (TSP) solver

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

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

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

from matplotlib import pyplot as plt

import urllib.request as ul

filename = 'dj38.tsp'

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

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

G = nx.Graph()
```

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

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

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

```
            break

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

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

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

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

    s = []

    prob.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 as an empty list 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.optimize()

sol = p.getSolution()
```

```
# Read solution and store it in the graph

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

# Display best tour found

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

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

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

6.4.7 Solving a nonconvex MIQCQP

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

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

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

- $w_{ij} \geq l_j\, x_i + l_i\, x_j - l_j\, l_i$
- $w_{ij} \geq u_j\, x_i + u_i\, x_j - u_j\, u_i$
- $w_{ij} \leq l_j\, x_i + u_i\, x_j - l_j\, u_i$
- $w_{ij} \leq u_j\, x_i + l_i\, x_j - u_j\, l_i$

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

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

```
def create_prob(filename):

    [...]

    x = p.getVariable()

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

    [...]

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

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

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

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

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

    securecols += list(secureorig)

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

The creation of a single auxiliary variable is done in `addaux`, where its bounds are created and, depending on whether it is the product of two variables or the square of one, it receives a different treatment.

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

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

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

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

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

    return aux[i, j]
```

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

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

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

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

```
                J = p.getIndex(ind[j])

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

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

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

                if row < 0:
                    p.chgobj([y], [coe])
                else:
                    rcols.append(y)
                    rrows.append(row)
                    rcoef.append(coe)

        if row >= 0:

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

        else:

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

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

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

```
def solveprob(p, aux):

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

    p.mipoptimize()
```

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

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

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

    x = prob.getVariable()  # presolved variables

    rowmap = []
    colmap = []

    prob.getpresolvemap(rowmap, colmap)

    invcolmap = [-1 for _ in lb]

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

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

    sol = np.array(sol)

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

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

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

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

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

```
    if i == j:

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

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

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

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

            assert(brvarind >= 0)

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

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

            b.addbranches(2)

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

            # New variable bounds are not enough, add new McCormick
```

```
            # inequalities for y = x**2: suppose x0,y0 are the current
            # solution values for x,y, yp = x0**2 and xu,yu = xu**2 are their
            # upper bound, and similar for lower bound. Then these two
            # rows must be added, one for each branch:
            #
            # y - yp <= (yl-yp)/(xl-x0) * (x - x0)  <===>
            # (yl-yp)/(xl-x0) * x - y >= (yl-yp)/(xl-x0) * x0 - yp
            #
            # y - yp <= (yu-yp)/(xu-x0) * (x - x0)  <===>
            # (yu-yp)/(xu-x0) * x - y >= (yu-yp)/(xu-x0) * x0 - yp
            #
            # Obviously do this only for finite bounds

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

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

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

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

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

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

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

            return b
```

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

```
        else:

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

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

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

            alpha = 0.2

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

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

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

            b.addbranches(2)

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

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

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

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

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

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

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

    global Aux_i, Aux_j, Aux_ind

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

    sol = []

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

    sol = np.array(sol)

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

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

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

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

```
def cbfindsol(prob, aux):
```

```
    sol = []

    try:
        prob.getlpsol(x=sol)
    except:
        return 0

    xnew = sol[:]

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

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

    prob.addmipsol(xnew)

    return 0
```

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

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

    cuts = []

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

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

        if i == j:

            # Separate quadratic term

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

                xk = sol[i]

                ox = xk
                oy = ox ** 2

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

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

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

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

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

        # Done creating cuts. Add them to the problem

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

        return 0
```

Another useful component of any nonconvex solver is a procedure to tighten the variable bounds based on information that is known on other variables. For example, if new bounds are inferred on w_{ij}, possible tighter lower or upper bounds can be deduced on x_i and/or x_j.

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

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

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

    feastol = prob.controls.feastol

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

        auxind = prob.getIndex(a)

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

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

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

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

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

            if uba < -feastol:
```

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

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

            else:

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

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

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

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

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

                if lbj >= 0 and lbi >= 0:

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

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

        # Done creating cuts. Add them to the problem

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

            mcolsp, dvalp = [], []
            drhsp, status = prob.presolverow(s, I, C, r, prob.attributes.cols,
                                             mcolsp, dvalp)
            if status >= 0:

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

                    assert(dvalp[0] > 0)

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

        return 0
```

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

6.5 Translated Mosel examples

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

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

CHAPTER 7

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)
```

As of version 8.12, the names in the C API have undergone a change in order to have more expressive names in the C API. The Python API was updated accordingly. The old names still work but are now deprecated. This reference documentation and all error messages refer to the new names.

In the Python version, the `prob` pointer is not provided as obviously `addrows` is a method of the `problem` class. The C variables `n` and `nnz`, which are assigned to arguments `newrow` and `newnz`, respectively, of the call to `XPRSaddrows`, are not necessary in the Python call as the length of `rhs`, `mstart`, etc. is inferred from the passed lists. If the lengths of all lists passed as arguments are not consistent with one another, an error will be returned.

Because lists (or tuples, generators, iterators, sequences) can be used as parameters of all functions in this manual, their size does not need to be passed explicitly as it is detected from the parameter itself. The interface will check the consistency and the content if the vector is referred to the variables or constraints, and will return an error in case of a mismatch.

When passing (lists, arrays, dictionaries of) variables, constraints, or SOSs, there are three ways of referring to these entities: by numerical index, by Python object, and by name. For instance, consider the `problem.getSolution` method, which admits both an empty argument and one or a list of variables. If we define a variable with a name as follows

```
x = xpress.var(name='myvar')
p = xpress.problem()
p.addVariable (x)
```

then we can refer to its index (which will be 0 here as it is the first variable added to the problem), by its object name, i.e., `x`, and by its given name `"myvar"`, in the three following (equivalent) ways:

```
print('x is ', p.getSolution(x))
print('x is ', p.getSolution(0))
print('x is ', p.getSolution('myvar'))
```

Another difference between the Python methods and their C API counterpart is that some *output* arguments are no longer passed (by reference) as arguments to the Python functions but rather are (part of) the value returned by the function. Where multiple scalar output parameters are returned by the C API function, some Python functions return a *tuple* containing all such output values.

The non-scalar parameters can instead be specified as lists, NumPy arrays, sequences, or generators when applicable. The output non-scalar parameters are stored as lists.

Optional parameters can be specified as `None` or skipped, provided the subsequent arguments are explicitly declared with their parameter name as Python allows:

```
p.addrows(rowtype=type, rhs=rhs, start=mstart,
          colind=indices, rowcoef=values)
```

Because the Python interface relies on the Xpress Optimizer C Application Program Interface, it is advisable to complement the knowledge in this reference manual with that of the Xpress Optimizer reference manual.

Format of the reference

The descriptions in the following pages report, for each function:

- Name;
- A short description of its purpose;
- Its synopsis, i.e., how it must be called. If it returns a value, then it will be presented as a Python assignment statement, otherwise it will be just shown as a call without a returned value; also, if it is a module function rather than a problem-specific function, it will be prefixed by `xpress`;
- A description of its arguments and whether each argument is optional;
- Error values;
- Associated controls;
- A sample usage of the function;
- Further useful information about the function;
- Related functions, parameters.

Note that all arguments defined in the remainder of this chapter as "array" or "vector" can be many other Python non-scalar objects: lists, generators, and NumPy arrays are admissible as parameters, except when specified (e.g. `getControl`). However, for simplicity we refer to non-scalar arguments as array.

Finally, some attributes and controls are referred to as uppercase words for clarity. For example, `ROWS` indicates the attribute "rows" of a problem, hence it is equivalent to `problem.attributes.rows`.

7.2 Classes of the Xpress module

Below is a list of classes used in all operations of the `xpress` module. While for a few of these classes an explicit constructor exists (for instance, `xpress.problem` and `xpress.sos`), objects of other classes, like `xpress.linterm` and `xpress.expression`, cannot be created with a constructor methods but are created using algebraic operators applied to constants, variables, and other expressions.

xpress.attr	xpress.branchobj	xpress.ctrl
xpress.constraint	xpress.expression	xpress.linterm
xpress.nonlin	xpress.problem	xpress.poolcut
xpress.quadterm	xpress.sos	xpress.var
xpress.voidstar	xpress.xprsobject	

7.3 Global methods of the Xpress module

Below is a list of functions that are invoked from the Xpress module, i.e., they are not methods of the `problem` or the `branchobj` class and can be invoked after the `import` statement. The invocation is therefore as in the example that follows:

```
import xpress as xp
print(xp.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.And	xpress.Or	xpress.pwl
xpress.setdefaultcontrol	xpress.setdefaults	xpress.featurequery
xpress.removecbmsghandler	xpress.setarchconsistency	xpress.manual
xpress.examples	xpress.getcomputeallowed	xpress.setcomputeallowed

7.4 Methods of the class `problem`

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	addgencons	addIndicator
addmipsol	addObjective	addpwlcons	addqmatrix
addrows	addsetnames	addSOS	addVariable
basisstability	btran	calcobjective	calcreducedcosts
calcslacks	calcsolinfo	chgbounds	chgcoef
chgcoltype	chgglblimit	chgmcoef	chgmqobj
chgobj	chgobjsense	chgqobj	chgqrowcoeff
chgrhs	chgrhsrange	chgrowtype	copy
copycontrols	crossoverlpsol	delConstraint	delgencons
delpwlcons	delqmatrix	delSOS	delVariable
dumpcontrols	estimaterowdualranges	fixmipentities	ftran

getAttrib	getbasis	getbasisval	getcoef
getcols	getcoltype	getConstraint	getControl
getdirs	getDual	getdualray	getgencons
getmipentities	getiisdata	getIndex	getIndexFromName
getindicators	getinfeas	getlastbarsol	getlasterror
getlb	getlpsol	getlpsolval	getmessagestatus
getmipsol	getmipsolval	getmqobj	getnamelist
getobj	getObjVal	getOutputEnabled	getpivotorder
getpivots	getpresolvebasis	getpresolvemap	getpresolvesol
getprimalray	getProbStatus	getProbStatusString	getpwlcons
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	loadpresolvedir
loadproblem	loadsecurevecs	lpoptimize	mipoptimize
name	objsa	optimize	postsolve
presolverow	read	readbasis	readbinsol
readdirs	readslxsol	refinemipsol	repairinfeas
repairweightedinfeas	repairweightedinfeasbounds	restore	reset
rhssa	save	scale	setControl
setdefaults	setindicators	setlogfile	setmessagestatu
setObjective	setOutputEnabled	setprobname	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
fixpenalties	getccoef	getcoefformula
getcoefs	getcolinfo	getdf
getrowinfo	getrowstatus	getrowwt
getslpsol	gettolset	getvar
loadcoefs	loaddfs	
loadtolsets	loadvars	msaddcustompreset
msaddjob	msaddpreset	msclear
presolve	printmemory	printevalinfo
reinitialize	scaling	setcurrentiv
unconstruct	updatelinearization	validate
validatekkt	validaterow	validatevector

7.5 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		

7.6 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
addcbchecktime	removecbchecktime
addcbcutlog	removecbcutlog
addcbdestroymt	removecbdestroymt
addcbgapnotify	removecbgapnotify
addcbmiplog	removecbmiplog
addcbinfnode	removecbinfnode
addcbintsol	removecbintsol
addcblplog	removecblplog
addcbmessage	removecbmessage
addcbmipthread	removecbmipthread
addcbnewnode	removecbnewnode
addcbnodecutoff	removecbnodecutoff
addcbnodelpsolved	removecbnodelpsolved
addcboptnode	removecboptnode
addcbpreintsol	removecbpreintsol
addcbprenode	removecbprenode
addcbusersolnotify	removecbusersolnotify

7.7 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	setcbintsol
setcbiterend	setcbiterstart	setcbitervar
setcbmessage	setcbmsjobend	setcbmsjobstart
setcbmswinner	setcboptnode	setcbprenode
setcbpreupdatelinearization	setcbslpend	setcbslpnode
setcbslpstart		

7.8 Xpress base classes

xpress.attr

Purpose

Internal object class used for the attributes of an `xpress.problem`. The user can read attributes from a problem, but cannot create objects of this class. Also, an attribute of a problem may be read, but it cannot be set.

Example

The following example creates a problem and then prints one of its attributes:

```
import xpress as xp

x = [xp.var() for _ in range(10)]

p = xp.problem(x)

print(p.attributes.cols, "variables")  # will print "10 variables"
```

Related topics

`problem.getAttrib`.

xpress.branchobj

Purpose

Class for branching objects. These objects are created by the user within a callback when directing a branch-and-bound solve toward different branching decisions.

Synopsis

```
b = xpress.branchobj(prob, branches=None, isoriginal=True)
```

Arguments

`prob` Problem object.

`branches` List or tuple of branching decisions. If it is a tuple, its members are constraints of distinct branches; if it is a list, its members must be either tuples of branching constraints, each tuple for a single branch.

`isoriginal`

`False`	Column indices should refer to the current (presolved) node problem;
`True`	Column indices should refer to the original matrix.

xpress.constraint

Purpose

Class for linear, quadratic, and nonlinear constraints.

Synopsis

```
c = xpress.constraint (constraint=None, body=None, lb=-xpress.infinity,
      ub=xpress.infinity, sense=None, rhs=None, name='')
```

Arguments

`constraint` The constraint, written as a ==, <=, or >= condition between two expressions. Variables can appear on either or both sides of the sign. Example: x1 + 2 * x2 <= 4

`body` An expression indicating the function to be constrained between `lb` and `ub` or by `rhs` with an assigned `sense`. It should not be used when `constraint` is defined. Example: 3 * x1 + x2

`lb` Lower bound on `body`.

`ub` Upper bound on `body`.

`sense` Sign of the constraint: one of `xpress.leq`, `xpress.eq`, `xpress.geq`, or `xpress.rng`.

`rhs` Right-hand side of the constraint if `sense` is defined. It may not be specified if `lb` or `ub` are.

`name` Name of the constraint (string)..

Example

Constraint declared without the explicit constructor:

```
myconstr = x1 + x2 * (x2 + 1) <= 4
```

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

```
m.addConstraint(myconstr)
m.addConstraint(v1 + v2 <= 3)
m.addConstraint(x[i] + y[i] <= 2 for i in range (10))
```

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

```
m.addConstraint(xp.Sum ([y[i]    for i in range (10)]) <= 1)
m.addConstraint(xp.Sum ([x[i]**2 for i in range (9)])  <= x[9])
```

Further information

1. Parameters lb, ub, and rhs must be constant.
2. A constraint can be specified more naturally as a condition on a linear or quadratic expression:
3. 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.

Related topics

`problem.addConstraint`.

xpress.ctrl

Purpose

Internal object class used for the controls of an `xpress.problem`. The user can read and write controls for a problem, but cannot create objects of this class.

Example

The following example creates a problem and then reads and sets a few of its controls:

```
import xpress as xp

x = [xp.var() for _ in range(10)]

p = xp.problem(x, xp.Sum(x) >= 1)

print('miprelstop is currently", p.controls.miprelstop)

p.controls.miprelstop = 1e-7
p.controls.xslp_solver = 0

# An equivalent way to do the two lines above
p.setControl({'miprelstop': 1e-7, 'xslp_solver': 0})
```

Related topics

`problem.setControl`, `problem.getControl`.

xpress.expression

Purpose

Class for linear and quadratic expressions. These can be used and combined to create constraints and objective function of an optimization problem. The user cannot explicitly create an object of this class, but applying sum, multiplication, and squaring of variables and constants gives rise to an object of this type. It can also be used for *type hinting*.

Example

An expression can be created as follows:

```
import xpress as xp

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

e = x**2 + 2*y - 5
```

xpress.linterm

Purpose

Internal class for a first-degree monomial, i.e., the product of a constant by a variable. It can be used and combined to create constraints and objective function of an optimization problem. The user cannot explicitly create an object of this class.

Example

Example declaration:

```
import xpress as xp

x = xp.var()

l = 2*x  # l is of type xpress.linterm
```

xpress.nonlin

Purpose

Internal class for objects representing functions which are neither quadratic nor linear nor constant. It can be used and combined to create constraints and objective function of an optimization problem. The user cannot explicitly create an object of this class.

Example

The following creates a nonlinear expression and sets it as the objective function of a problem:

```
import xpress as xp

x = xp.var()

obj = x**4 + xp.exp(x)

p = xp.problem(x, obj)
```

xpress.poolcut

Purpose

Class for poolcut objects. These are used by callback functions when creating cuts within a Branch-and-bound.

Synopsis

```
c = xpress.poolcut()
```

Further information

These objects are created by the Optimizer within callbacks and can be used by Python callback functions to store and pass pool cuts.

xpress.problem

Purpose

Class for all optimization problems solved by the Xpress Optimizer.

Synopsis

```
p = xpress.problem(*elements=None, name='', sense=xpress.minimize)
```

Arguments

`elements` Variables, constraints, SOSs, or objective function of the problem. These can be specified as single objects or lists and arrays thereof. They can be listed in the same order as would be added to the problem through `problem.addVariable`, `problem.addConstraint`, `problem.addSOS`, `problem.setObjective`, i.e. by making sure that the variables appearing in a constraint or objective function appear beforehand in the list.

`name` Name of the problem, displayed on solve log or saved in the `.lp` or `.mps` file when saved with `problem.write`.

`sense` Optimization sense. Can be `xpress.minimize` (default) or `xpress.maximize`.

Example 1

An object of class `xpress.problem` can be created from scratch or read from a file. It contains a set of variables and constraints, and may have an objective function. An empty optimization problem is created as follows:

```
myproblem = xp.problem()
```

A name can be assigned to a problem upon creation:

```
myproblem = xp.problem(name='My first problem')
```

The problem has no variables or constraint at this point.

Example 2

Simply call optimize() to solve an optimization problem that was either built or read from a file. The type of solver is determined based on the type of problem: if at least one integer variable was declared, then the problem will be solved as a mixed integer (linear or quadratically constrained) problem, while if all variables are continuous the problem is solved as a linear or quadratic optimization problem.

```
m.optimize()
```

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

```
import xpress as xp

m = xp.problem()
m.read("example3.lp")
solvestatus, solstatus = m.optimize()

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

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

Example 3

It is useful, after solving a problem, to obtain the value of an optimal solution. After solving a continuous or mixed integer problem, the two methods `problem.getSolution` and `problem.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.optimize()

print(m.getSolution ())             # prints a list with an optimal solution
print("v1 is", m.getSolution(v1))   # only prints the value of v1
a = m.getSolution(x)                # gets the values of all variables in the v
b = m.getSolution(0:4)              # gets the value of v1 and x[0], x[1], x[2]
```

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

xpress.quadterm

Purpose

Internal class for objects representing monomials of degree two. It can be used and combined to create constraints and objective function of an optimization problem. The user cannot explicitly create an object of this class.

Example

Example declaration:

```
import xpress as xp

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

q1 = 2*x*y  # bilinear term
q2 = 3*x**2  # quadratic term
```

xpress.sos

Purpose

Python object class for Special Order Sets (SOS). An SOS is a modeling tool for constraining a small number of consecutive variables in a vector to be nonzero.

Synopsis

```
s = xpress.sos(indices, weights, type=1, name='')
```

Arguments

`indices` List of variables composing the SOS.

`weights` List of weights (one per variable). These define the order for SOS2 constraints and may be used in branching for both types.

`type` Type of SOS. Can be 1 (default) or 2.

`name` Name of the SOS.

Example

The following are example declarations of SOS:

```
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, "mysos")
```

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 ([xp.sos([w[i],w[i+1]], [2,3]) for i in range(n-1)])
```

Related topics

`problem.addSOS`.

xpress.var

Purpose

Class for optimization variables.

Synopsis

```
x = xpress.var(name='', lb=0, ub=xpress.infinity,
      threshold=-xpress.infinity, vartype=xpress.continuous)
```

Arguments

`name` a Python UTF-8 string containing the name of the variable (its ASCII version will be saved if written onto a file); a default name is assigned if the user does not specify it.

`lb` Lower bound (0 by default).

`ub` Upper bound (+infinity by default).

`threshold` the threshold for semi-continuous, semi-integer, and partially integer variables; it must be between its lower and its upper bound; it has no default, so if a variable is defined as `xpress.partiallyinteger` the threshold must be specified.

`vartype`

`xpress.continuous`	for continuous variables;
`xpress.binary`	for binary variables;
`xpress.integer`	for integer variables;
`xpress.semicontinuous`	for semi-continuous variables;
`xpress.semiinteger`	for semi-integer variables;
`xpress.partiallyinteger`	for partially integer variables.

Example

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)
```

Further information

Variables are not tied to a problem but may exist globally in a Python program. In order for them to be included into a problem, they have to be explicitly added to that problem using `problem.addVariable`.

Related topics

`problem.addVariable`, `xpress.vars`.

xpress.voidstar

Purpose

Internal class for unspecified objects in the Xpress Optimizer Library. This is an internal class and the user cannot create an object of this class.

xpress.xprsobject

Purpose

Internal class for Xpress objects used within an optimization problem solved by the Xpress Optimizer. A user cannot declare an object of this class.

7.9 Xpress object functions

object.extractLinear

Purpose

Returns the variables and coefficients of the linear part of any expression.

Synopsis

```
vars, coef = a.extractLinear()
```

Arguments

`a`	An expression or variable.
`vars`	A list containing the variable objects composing the linear expression in `a`.
`coef`	A list containing the corresponding coefficients in the linear expression.

Example

The following code snippets show what is the expected result of applying `extractLinear`:

```
import xpress as xp

x = xp.var()
y = xp.var(name='myvar')

a = x + 2*y
b = 3*x
c = y**2 + x**2 - 6*x
d = x**5 - 7*x  # nonlinear expression

print (a.extractLinear())  # will print "([C1, myvar], [1, 2])"
assert (a.extractLinear() == ([x, y], [1, 2]))

print (b.extractLinear())  # will print "([C1], [3])"
print (c.extractLinear())  # will print "([C1], [-6])"
print (d.extractLinear())  # will print "([C1], [-7])"
```

Further information

1. Note that this operator returns variable objects, not indices, in the `vars` portion of the output tuple. To obtain indices, use the `problem.getIndex` function. Printing these lists will show the name of the associated variables, as determined by the user when creating the variable with the `name` argument or, if `name` was not provided, it will show the name as determined by the Optimizer's library (default variable names are "C"+index). See also the Modelling chapter.

2. This operator is most useful only for linear expressions with more than one element. For nonlinear expressions, the function attempts to extract as much linear information it can, but will not be able to infer linearity apart from the most obvious cases. For example, for the expression `x**4 + xp.log(xp.exp(y))`, which contains the linear term `y`, the function will return `([],[])`.

object.extractQuadratic

Purpose

Returns the variables and coefficients of the quadratic part of any expression.

Synopsis

```
vars1, vars2, coef = a.extractQuadratic()
```

Arguments

`a` An expression or variable.

`vars1` A list containing the first variables of each bilinear term composing the quadratic expression in `a`.

`vars2` A list containing the second variables of each bilinear term of the quadratic expression in `a`.

`coef` A list containing the corresponding coefficients in the quadratic expression.

Example

The following code snippets show what is the expected result of applying `extractQuadratic`:

```
import xpress as xp

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

a = x + 2*y + x*y + 8 * x**2
b = 3*x**2 + z + 4
c = y**2 + x**2 - 6*x*y
d = x**5 - 7*x*y - 4*x*y*z  # nonlinear expression
e = x*y + y*x  # note: same bilinear term added twice. This is compressed to 2

print (a.extractQuadratic())  # will print "([C1, C1], [C2,C1], [1,8])"
assert (a.extractQuadratic() == ([x,x], [y,x], [1,8]))

print (b.extractQuadratic())  # will print "([C1], [C1], [3])"
print (c.extractQuadratic())  # will print "([C2, C1], [C2, C1], [1, 1])"
print (d.extractQuadratic())  # will print "([C1], [C2], [-7])"
print (e.extractQuadratic())  # will print "([C1], [C2], [2])"
```

Further information

1. Similar to `object.extractLinear`, this operator returns variable objects, not indices, in the `vars` portion of the output tuple. To obtain indices, use the `problem.getIndex` function. Printing these lists will show the name of the associated variables, as determined by the user when creating the variable with the `name` argument or, if `name` is not provided, it will show the name as determined by the Optimizer's library (default variable names are "C"+index). See also the Modelling chapter.

2. This operator is most useful only for quadratic expressions with more than one element. For nonlinear, non-quadratic expressions, the function attempts to extract as much quadratic information it can, but will not be able to detect quadratic/bilinear expressions apart from the most obvious cases. For example, for the expression `x**4 + xp.sqrt(y**4)`, which contains the quadratic term `y**2`, the function will return `([],[])`.

7.10 Xpress operators

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

Python's native `abs` operator is equivalent to `xpress.abs` for arguments that are functions of variables.

xpress.acos

Purpose

Returns the arccosine of a given expression.

Synopsis

```
a = xpress.acos(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.And

Purpose

Returns a logical AND of two or more binary variables or expressions.

Synopsis

`xpress.And(variables)`

Argument

`variables` A list/array of binary variables or binary expressions

Example

The following example shows how to use `and` to model various logical constraints:

```
N = 10

x = xp.vars(N, vartype=xp.binary)  # Creates N binary variables

c = [1, 4, 7, 3, 5, 7, 8, 4, 4, 9]

p = xp.problem(x)  # Creates a problem with x, y

# Sets a linear objective
p.setObjective (xp.Sum(c[i] * x[i] for i in range(N)))

# Linear constraint
p.addConstraint (xp.Sum(x) <= 6)

# Constrains the first x variable to be the conjunction of all other x's
p.addConstraint (x[0] == xp.And(x[1:]))

# Forces the logical AND between some logical expressions to
# be zero, i.e., at least one of them must be zero

p.addConstraint (xp.And([x[1] | x[4], x[2] | x[1], x[3] | x[6]]) == 0)
```

Further information

1. For `AND` functions, all variables and expressions must be binary; an error will be generated otherwise.
2. A function call `xpress.And(x1,x2,...,xk)` is equivalent to `x1 and (x2 and (x3 and ... xk))...)`.
3. Note that since `x1`, `x2`, ..., `xk`, are binary variables, `xpress.And(x1,x2,...,xk)` is equivalent to `xpress.min(x1,x2,...,xk)`.

Related topics

`problem.addgencons`, `problem.delgencons`, `problem.getgencons`, `xpress.Or`.

xpress.asin

Purpose

Returns the arcsine of a given expression.

Synopsis

```
a = xpress.asin(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.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.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.Dot

Purpose

Alternative dot-product operator for an arbitrary number of NumPy single- or multi-dimensional arrays. Following the convention for dot-product, the result of `Dot` for a list of k objects $T_1, T_2, ..., T_k$ of $d_1, d_2, ..., d_k$ dimensions is an object of $d_1 + d_2 + ... + d_k - 2(k - 1)$ dimensions. For each i-th factor in [1,2,...,$k - 1$], the arity of the last dimension of T_i must match the arity of the penultimate dimension of T_{i+1} (or its arity if T_{i+1} is single-dimensional, i.e., a vector).

Synopsis

```
a = xpress.Dot(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], dtype=xp.npvar)
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))
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 $(n_1, n_2, ..., n_{d'})$ and $(m_1, m_2, ..., m_{d''})$, respectively, is a multi-dimensional array of dimension $d' + d'' - 2$, whose arity vector is $(n_1, n_2, ..., n_{d'-1}, m_1, m_2, ..., m_{d''-2}, m_{d''})$ and whose generic element is

$v_{i_1,i_2,...,i_{d'-1}j_1j_2,...j_{d''-2}j_{d''}} = \sum_{1 \le h \le n_{d'}} t'_{i_1,i_2,...,i_{d'-1},h} \cdot t''_{j_1j_2,...j_{d''-2},hj_{d''}}$.

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.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.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.

Further information

Using Python's `math` library operator `math.log10` is only advisable when the argument is not an expression that depends on variables.

Related topics

`xpress.log`.

xpress.max

Purpose

Returns the maximum of one or more expressions.

Synopsis

```
a = xpress.max(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.Or

Purpose

Returns a logical OR of two or more binary variables or expressions.

Synopsis

`xpress.Or(variables)`

Argument

`variables` A list/array of binary variables or binary expressions

Example

The following example shows how to use `or` to model various logical constraints:

```
N = 10

x = xp.vars(N, vartype=xp.binary)  # Creates N binary variables

c = [1, 4, 7, 3, 5, 7, 8, 4, 4, 9]

p = xp.problem(x)  # Creates a problem with x, y

# Sets a linear objective
p.setObjective (xp.Sum(c[i] * x[i] for i in range(N)))

# Linear constraint
p.addConstraint (xp.Sum(x) <= 6)

# Constrains the first x variable to be the conjunction of all other x's
p.addConstraint (x[0] == xp.Or(x[1:]))

# Forces the logical OR between some logical expressions to
# be one, i.e., at least one of them must be one

p.addConstraint (xp.Or([x[1] & x[4], x[2] & x[1], x[3] & x[6]]) == 1)
```

Further information

1. For `OR` functions, all variables and expressions must be binary; an error will be generated otherwise.
2. A function call `xpress.Or(x1,x2,...,xk)` is equivalent to `x1 or (x2 or (x3 or ... xk))...)`.
3. Note that since `x1`, `x2`, ..., `xk`, are binary variables, `xpress.Or(x1,x2,...,xk)` is equivalent to `xpress.max(x1,x2,...,xk)`.

Related topics

`problem.addgencons`, `problem.delgencons`, `problem.getgencons`, `xpress.And`.

xpress.pwl

Purpose

Returns a piecewise linear function over a variable.

Synopsis

`xpress.pwl(dict)`

Argument

`dict`	Python dictionary containing, as keys, two-elements tuples, and, as values, linear expressions in a variable. If the piecewise linear function has only constant values (i.e. it is a piecewise constant function), the input variable can be specified with the key-value pair `None: x`.

Example

The following example shows various usages of `xpress.pwl` to model nonlinear functions as piecewise-linear functions :

```
x = xp.var()  # Nonnegative variable
y = xp.var(lb=-xp.infinity)  # dependent variable, unrestricted
t = xp.var()
w = xp.var()

p = xp.problem(x, y)

# Sets a piecewise linear objective: a ramp function
p.setObjective (xp.pwl({(-xp.infinity, -1):  -2,
                        (-1, 1):            2*x,
                        (1, xp.infinity)]:    2}))

p.addConstraint (t == xp.pwl({(1,2): 4*x, (2,4): 2, (4,5): -1}))

# Piecewise CONSTANT function: add a key-value pair None: x to specify
# input variable
p.addConstraint (t == xp.pwl({(1,2): 4, (2,4): 2, (4,5): -1, None: x}))

p.addConstraint (xp.pwl({(-1,0): x, (0,1): 2*x, (1,10): 2}) <=
                 xp.pwl({(0,10): 2*z, (10,20): z+2, (20,xp.infinity): 4}))
```

Further information

1. A piecewise linear function must use only one variable in all of the dictionary's values;
2. All values in the dictionary must be either constants or linear functions;
3. The intervals, specified as two-element tuples in the dictionary's keys, must be pairwise disjoint, i.e., they must not overlap.
4. Discontinuities in the function are allowed, i.e., one can declare a function as follows: `xp.pwl({(1, 2): 2*x + 4, (2,3): x - 1})`, which is obviously discontinuous at 2. The value of the function if the optimal solution has `x=2` will be then either 8 or 1.

Related topics

`problem.addpwlcons`, `problem.delpwlcons`, `problem.getpwlcons`.

xpress.Prod

Purpose

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

Synopsis

```
a = xpress.Prod(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.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.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.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.Sum

Purpose

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

Synopsis

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

Example

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

```
import math
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])
                <= math.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.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.user

Purpose

Creates an expression that is computed by means of a user-specified function. The user function can be defined to either provide or not provide the value of all derivatives w.r.t. the variables.

Synopsis

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

Arguments

`f`	User function; must be a Python function with as many (possibly optional) arguments as specified in the declaration.	
`t1,...,tn`	Arguments of the user function.	
`derivatives`	`"never"`	`f` does not return derivatives;
	`"always"`	`f` always returns derivatives;
	`"ondemand"`	`f` returns derivatives when they are requested by the solver (see notes below).

Example

The following code shows how to define user functions and use them in an optimization problem:

```
import math

def mynorm(*v):
  return math.sqrt(sum(e**2 for e in v))

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

def ondemand_derivatives(t1, t2, *deltas):
  val = 2*t1 + 4*t1*t2
  if not deltas:
    # No derivatives needed
    return val
  else:
    # Calculate whichever derivatives are needed
    d1, d2 = deltas
    return (val,
      2 + 4*t2 if d1 != 0 else None,
      4*t1 if d2 != 0 else None)

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

f1 = xp.user(mynorm, *x)
f2 = xp.user(weighted_sum, x[4], x[5], x[6], derivatives="always")
f3 = xp.user(ondemand_derivatives, x[0], x[1], derivatives="ondemand")

p = xp.problem()

p.addVariable(x)
p.addConstraint(f3 >= 4)
p.addConstraint(f2 == 1)
```

```
p.setObjective(f1)
p.optimize()
print('solution:', p.getSolution(x))
```

Further information

1. User functions **must** produce a Float, as the behaviour is otherwise undefined. If the `derivatives` parameter is set to `"never"` (the default), then the function should simply return the function value. If `derivatives="always"`, the function must return a tuple consisting of the function value and the derivatives of the function w.r.t. all variables in the list of arguments. If `derivatives="ondemand"`, the function will either be called with `numArgs` arguments, or with `2 * numArgs` arguments, depending on whether derivatives are required by the solver. When derivatives are not required, only the input values will be passed to the function, and the function can simply return the function value. When derivatives are required, the function will be passed the input value arguments followed by a delta argument for every input argument, and the function must return a tuple, as when `derivatives="always"`. Note that for this reason, the delta arguments must be declared as optional, either by providing default values, or by using variable-length arguments syntax (`*deltas`). The function only needs to populate the tuple with a derivative where the corresponding delta argument is nonzero. (Where the delta argument is zero, the function should provide some placeholder value such as `None` or zero.) The delta can be used as a suggested perturbation for numerical differentiation (a negative sign indicates that if a one-sided derivative is calculated, then a backward one is preferred).

2. The variables on which the function is defined cannot be passed as lists or numpy arrays. Lists of variables can be passed by unpacking the list: `xpress.user(lambda *y: sum(y), *list_of_vars)` Note that in this case the wrapped function must be variadic.

Related topics

`problem.setcbpreupdatelinearization`.

7.11 Xpress base functions

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

Arguments

`msghandler` The callback function which takes six arguments, `vObject`, `vUserContext`, `vSystemThreadId`, `sMsg`, `iMsgType` and `iMsgNumber`. Use `None` to cancel a callback function.

`vObject` The object sending the message.

`vUserContext` The user-defined object passed to the callback function.

`vSystemThreadId` The system id of the thread sending the message cast to a `void *`.

`sMsg` A string containing the message, which may simply be a new line. When the callback is called for the first time `sMsg` will be empty.

`iMsgType` Indicates the type of output message:

- `1` information messages;
- `2` (not used);
- `3` warning messages;
- `4` error messages.

When the callback is called for the first time `iMsgType` will be a negative value.

`iMsgNumber` The number associated with the message. If the message is an error or a warning then you can look up the number in the section Optimizer Error and Warning Messages for advice on what it means and how to resolve the associated issue.

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

`priority` An integer that determines the order in which multiple message handler callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

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.evaluate

Purpose

Returns the evaluation of one or more expressions for a given assignment of values to optimization variables.

Synopsis

```
v = xpress.evaluate(*args, problem=None, solution=None)
```

Arguments

`args` One or more objects to be evaluated. These can be variables, linear or nonlinear expressions; they can also be tuples, lists, dictionaries, or NumPy arrays of variables and expressions.

`problem` The `xpress.problem` object this function is referring to. If `problem` is not `None`, then `solution` is either `None` (in which case the current solution is used) or it is to be intended referred to the indices of variables in `problem`. If `problem` is `None`, `solution` must provide this information directly, i.e. it must be a dictionary mapping variable objects to their value

`solution` Either a list or NumPy array of values (in which case `problem` must not be `None`) or a dictionary mapping variable objects to their value. As mentioned above, if it is `None` then `problem` must be passed and the assignment for the function is assumed to be the solution as retrieved via `problem.getSolution`

Further information

1. Variable assignments do not have to correspond to a feasible solution.
2. At least one of the arguments `problem` and `solution` must be specified, because the objects in `*args` contain variables, and all variables could be used in zero or more problems. Only the following cases are allowed:
   - `problem=None` and `solution` is a dictionary mapping variables to values; the dictionary must have a key for each variable appearing in `*args`;
   - `problem` is not `None` but `solution=None`; then `solution` is taken as the result of `problem.getSolution`; this call is equivalent to `p.getSolution(*args)`;
   - `problem` is not `None` and `solution` is either a list or a NumPy array; then the size of `solution` must match the number of variables of `problem` and the order of values in the list/array is the same order in which the variables were added to `problem`.
3. Variables assignment do not have to correspond to a feasible solution.
4. When using `evaluate` with piecewise linear functions that have a step discontinuity, for example with the constraint `y == xp.pwl({(0,3): x, (3,5): 10*x})`, if at an optimal solution `x=3`the Optimizer library will compute a value for `y` that is anywhere between 3 and 30, because of numerical issues associated with discontinuities.

   In such cases, `evaluate` is unaware of the link between the function and `y` and, by convention, will return a value of the function that correspond to the second interval, i.e., the function will be evaluated at 30. In order to obtain the value of the piecewise linear function, evaluate should be run on `y` instead.

Example

The following examples are valid uses of `xpress.evaluate`:

```
import xpress as xp

x = xp.var()
y = xp.var(vartype=xp.binary)

# Uses evaluate without a problem but by assigning the variables
# explicitly. Note that the dictionary is necessary as no problem is
```

```
# defined. The result should be [5.4, 124.71633781453677].

v1 = xp.evaluate([x + y, x**3 - xp.cos(x)], solution={x:5, y:0.4})

p = xp.problem(x, y)  # Create a problem and add variables x and y

# Similar to the computation of v1 but with a vector of numbers; the
# order in which the variables were added to p means that this x=2,
# y=3 here. The result should be {'exp1':11, 'exp2':6, 'exp3':9}.

v2 = xp.evaluate({'exp1':x + 3*y, 'exp2':x*y, 'exp3':y**2},
                 problem=p, solution=[2,3])

p.addConstraint(x + y >= 3)
p.setObjective(x + 2*y)

p.optimize()

l = np.array([x**2 * y, x * y**2, x**3], dtype=xp.npexpr)

# No solution is passed, so the solution of p as computed with optimize()
# above is used. It is easy to show that the solution is x=3, y=0, so
# the result is np.array([0, 0, 27]).

v3 = xp.evaluate(l, problem=p)
```

Related topics

problem.getSolution.

xpress.examples

Purpose

Returns the full path to the directory of examples of the Xpress Python interface module.

Synopsis

```
xpress.examples()
```

Further information

The `modeling_examples/` subdirectory contains some of the Mosel examples translated into their Python counterpart.

xpress.featurequery

Purpose

Returns `True` if the provided feature is available in the current license used by the optimizer, `False` otherwise.

Synopsis

```
xpress.featurequery(feature)
```

Argument

`feature` The feature string to be checked in the license.

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`.
It is not recommended to call this function directly: it is better to call `xpress.init` in a `with` statement, which implicitly calls `xpress.free` at the end of the statement.

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.optimize()
xp.free() # from this point on, the license
          # can be claimed by other users
```

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 valid, and attempting to use it will raise an `xpress.ModelError`.

Related topics

`xpress.init`

xpress.getbanner

Purpose

Returns the banner and copyright message.

Synopsis

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

Example

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

xpress.getcomputeallowed

Purpose

Queries whether the current application is allowed to use the Insight Compute interface.

Synopsis

```
isComputeAllowed = xpress.getcomputeallowed()
```

Return value

Whether to allow use of Insight Compute, will be one of the following values:

1 Always allow solves to be sent to Compute.

0 Never allow solves to be sent to Compute.

-1 Allow solves to be sent to Compute only from non-OEM applications.

Related topics

`xpress.setcomputeallowed`.

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, it 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 as a string of the form 15.10.03, where 15 is the major release, 10 is the minor release, and 03 is the build number.

Synopsis

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

Example

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

xpress.init

Purpose

Initializes the Xpress environment prior to creating or reading a problem.
Note that it is **not** necessary to call this function after importing the Xpress module and before creating or solving a problem, since the environment will be automatically initialized when it is needed for the first time. However, you may want to call `xpress.init` in a `with` statement, which allows you to:

- detect initialization errors (which will be raised as an `xpress.ModelError`);
- specify the path to your license file;
- explicitly acquire the Xpress license;
- automatically release the Xpress license at the end of the `with` statement.

Synopsis

```
xpress.init(lic_path=None)
```

Argument

`lic_path` (optional) Path to the Xpress license file.

Example

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

```
try:
  with xp.init():     # Acquire the Xpress license
    x = xp.var()
    y = xp.var()
    p = xp.problem()  # This would imply a call to xp.init() if it had not alr
    p.addVariable(x, y)
    p.addConstraint(x+y <= 1)
    p.setObjective(x+2*y, sense=xp.maximize)
    p.optimize()
  # Xpress license is implicitly released here
catch xp.ModelError:
  print('Failed to initialize Xpress')
```

Related topics

`xpress.free`

xpress.manual

Purpose

Returns the full path to the PDF reference manual of the Python interface.

Synopsis

```
xpress.manual()
```

Further information

Note that only the manual of the Python interface (in PDF format) is included in the PyPI and conda package downloaded from these repositories; the PDF version of all other Xpress-related documentation is contained in the Xpress distribution, and the on-line, HTML format documentation is available on the FICO web pages.

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(msghandler, data)
```

Arguments

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

`data` The object value that the callback was added with. If `None`, then the object value will not be checked and all message callbacks with the function `msghandler` will be removed.

Related topics

`xpress.addcbmsghandler`.

xpress.setarchconsistency

Purpose

Sets whether to force the same execution path on various CPU architecture extensions, in particular (pre-)AVX and AVX2.

Synopsis

```
xpress.setarchconsistency(ifArchConsistent=False)
```

Argument

`ifArchConsistent` Whether to force the same execution path:
`False` Do not force the same execution path (default behavior);
`True` Force the same execution path.

Further information

Note that using this general environment API function is different from setting the `xpress.controls.cpuplatform` control. Setting this control selects a vectorization instruction set for the barrier method.

Related topics

`xpress.getcomputeallowed`.

xpress.setcomputeallowed

Purpose

Set whether the current application is allowed to use the Insight Compute interface.

Synopsis

```
xpress.setcomputeallowed(isComputeAllowed)
```

Argument

`isComputeAllowed` Whether to allow use of Insight Compute, must be one of the following values:

- `1` Always allow solves to be sent to Compute.
- `0` Never allow solves to be sent to Compute.
- `-1` Allow solves to be sent to Compute only from non-OEM applications.

Further information

1. This function controls whether this process would be allowed to use the Insight Compute Interface if the user tries to enable it.
2. If the user enables the Insight Compute Interface but the value specified through this function does not allow the Insight Compute Interface to be used, any solves will terminate with an immediate error. This function can be used to prevent solves from being sent to Insight Compute but cannot be used to force solves to be performed locally. The purpose of this function is to allow an application to prevent the optimization model being sent to the Insight Compute Interface.

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 checking and validation for function calls, invalid usage of Xpress Optimizer functions may not be detected and may cause the Xpress Optimizer process to behave unexpectedly or crash. It is not recommended to disable function call checking & validation during application development.

Synopsis

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

Argument

`checked_mode` Pass as False or 0 to disable much of the validation for all Xpress function calls from the current process. Pass True or 1 to re-enable validation. By default, validation is enabled.

Related topics

`xpress.getcheckedmode`.

xpress.setdefaults

Purpose

Sets the module's controls to their default values. This affects all problems created after calling `setdefaults`, not before.

Synopsis

```
xpress.setdefaults()
```

Example

The following creates two problems, one before and one after calling `setdefaults()`:

```
xpress.controls.presolve = 0
p1 = xpress.problem()
xpress.setdefaults()
p2 = xpress.problem()
print('Check p1.controls.presolve is 0:          ', p1.controls.presolve)
print('Check p2.controls.presolve is its default:', p2.controls.presolve)
```

Related topics

`xpress.setdefaultcontrol`, `problem.setdefaults`, `problem.setdefaultcontrol`.

xpress.setdefaultcontrol

Purpose

Sets one of the module's controls to its default values. This affects all problems created after calling `setdefaults`, not before.

Synopsis

```
xpress.setdefaultcontrol(ipar)
```

Argument

`ipar` Name of the control to be set to default.

Example

The following creates two problems, one before and one after calling `setdefaultcontrol('presolve')`:

```
xpress.controls.presolve = 0
p1 = xpress.problem()
xpress.setdefaultcontrol('presolve')
p2 = xpress.problem()
print('I bet p1.controls.presolve is 0:          ', p1.controls.presolve)
print('I bet p2.controls.presolve is its default:', p2.controls.presolve)
```

Related topics

`xpress.setdefaults`, `problem.setdefaults`, `problem.setdefaultcontrol`.

xpress.vars

Purpose

Creates a dictionary or NumPy array of variables. Similar to the creation of a single variable with `xpress.var`, `vars` allows for using one or more index sets, specified as sets, lists, `range` objects, or any iterable object. Specifying a number `k` as an argument is equivalent to `range(k)` but can be used to create NumPy multiarrays of variables, and allows for more efficient creation. The result is otherwise a Python dictionary of variables, whose keys are tuple of indices. A collection of variables `x` that is created with `vars` can be indexed, for instance, as `x[i,j]` where `i` and `j` are indices in the lists provided.

Synopsis

```
x = xpress.vars(*indices, name="x", lb=0, ub=xpress.infinity, threshold=0,
      vartype=xpress.continuous)
```

Arguments

`indices`	One or more lists, sets, ranges, or iterable objects to be combined; in alternative, one can specify one or more numbers `k` to signify the range `0..k-1`. Using only numbers as argument will yield a NumPy multiarray with the dimensions as specified by the arguments themselves.
`name`	Prefix to be added to the name of each variable; see notes for more information.
`lb`	Lower bound for all variables.
`ub`	Upper bound for all variables.
`threshold`	Threshold for all variables; only used if the variables are partially integer.
`vartype`	Type of all variables, similar to the definition of single variables.

Example

The following creates a dictionary containing 6 variables whose indices vary in the set `{(0,'a'),(0,'b'),(0,'c'),(1,'a'),(1,'b'),(1,'c')}`:

```
x = xpress.vars([0,1],['a','b','c'])
```

The following creates a dictionary containing 6 variables whose indices vary in the set `{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)}`:

```
x = xpress.vars(2,3)
```

The code below creates a dictionary containing 5 integer variables with names `'y(a)', 'y(b)', 'y(c)', 'y(d)', 'y(e)'` and creates a constraint to bound their sum:

```
x = xpress.vars(['a','b','c','d','e'],
name='y', vartype=xpress.integer)
con1 = xpress.Sum(x) <= 4
```

The code below creates a dictionary whose keys range from 0 to 4:

```
x = xpress.vars(range(5),
name='y', vartype=xpress.integer)
con1 = xpress.Sum(x) <= 4
```

The following example creates a Numpy multiarray of dimensions `3,7,4` without assigning names to the variables:

```
x = xpress.vars(3,7,4, name="", lb=-1, ub=1)
```

Note that specifying anything other than a number yields a dictionary rather than a Numpy multiarray. Finally, the following creates a variable indexed by the set defined right before:

```
S = set()
S.add('john')
S.add('cleese')
x = xpress.vars(S, name='y', vartype=xpress.integer)
```

Further information

1. The name of each variable is created by concatenating its indices together. If the `name` argument is given as a non-empty string, this will be prepended to the name of each variable. If the `name` argument is given as an empty string, no names will be assigned to the variables. This option can be used to create large arrays of variables more quickly, since it will not be necessary to calculate a name for each variable.
2. All lists must contain non-repeated elements to avoid having variables with equal names. If a list in the argument is, for instance, `['a','b','a']`, an error is returned.

Related topics

`xpress.var`

xpress.getOutputEnabled

Purpose

Returns True if Optimizer messages will be written to the Python output stream, False otherwise.

Synopsis

```
enabled = xpress.getOutputEnabled()
```

Related topics

`xpress.setOutputEnabled`, `problem.getOutputEnabled`, `problem.setOutputEnabled`.

xpress.setOutputEnabled

Purpose

Enables or disables writing Optimizer messages to the Python output stream.

Synopsis

```
xpress.setOutputEnabled(enabled)
```

Argument

`enabled` True if Optimizer messages should be written to the Python output stream, False otherwise.

Related topics

`xpress.getOutputEnabled`, `problem.getOutputEnabled`, `problem.setOutputEnabled`.

7.12 Xpress problem methods

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(callback, data, priority)
barrier_action = callback(my_prob, my_object)
```

Arguments

`callback` The callback function itself. This takes two arguments, `my_prob` and `my_object`, and returns an integer return value. This function is called at every barrier iteration.

`my_prob` The problem passed to the callback function, `fubi`.

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

`barrier_action` Defines a return value controlling barrier:

- `<0` continue with the next iteration;
- `=0` let barrier decide (use default stopping criteria)
- `1` barrier stops with status not defined;
- `2` barrier stops with optimal status;
- `3` barrier stops with dual infeasible status;
- `4` barrier stops wih primal infeasible status;

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

`priority` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

This simple example demonstrates how the solution might be retrieved for each barrier iteration.

```
# Barrier iteration callback
def BarrierIterCallback(my_prob, my_object):

    current_iteration = my_prob.attributes.bariter

    PrimalObj = my_prob.attributes.barprimalobj
    DualObj   = my_prob.attributes.bardualobj

    Gap = DualObj - PrimalObj

    PrimalInf        = my_prob.attributes.barprimalinf
    DualInf          = my_prob.attributes.bardualinf
    ComplementaryGap = my_prob.attributes.barcgap

    # decide if stop or continue
    barrier_action = 0
    if(current_iteration >= 50 or
        Gap <= 0.1 * max(abs(PrimalObj), abs(DualObj))):
        barrier_action = 2

    return barrier_action
```

```
# To set callback:
prob.addcbbariteration(BarrierIterCallback, myobj, 0)
```

Further information

1. Only the 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(callback, data, priority)
ret = callback(my_prob, my_object)
```

Arguments

`callback`	The callback function itself. This takes two arguments, `my_prob` and `my_object`, and has an integer return value. If the value returned by `callback` is nonzero, the solution process will be interrupted. This function is called at every barrier iteration.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbbarlog`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple barrier log callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

This simple example prints a line to the screen for each iteration of the algorithm.

```
prob.addcbbarlog(barLog, None, 0)
prob.lpoptimize('b')
```

The callback function might resemble:

```
def barLog(prob, object):
    print('Next barrier iteration')
```

Further information

If the callback function returns a nonzero value, the Optimizer run will be interrupted.

Related topics

`problem.removecbbarlog`, `problem.addcbmiplog`, `problem.addcblplog`, `problem.addcbmessage`.

problem.addcbchecktime

Purpose

Declares a callback function which is called every time the Optimizer checks if the time limit has been reached. This callback function will be called in addition to any callbacks already added by `addcbchecktime`.

Synopsis

```
problem.addcbchecktime(callback, data, priority)
ret = callback(my_prob, my_object)
```

Arguments

`callback`	The callback function which takes two arguments, `my_prob` and `my_data`, and has an integer return value. If the value returned by `callback` is nonzero, the solution process will be interrupted. This function is called every time the Optimizer checks against the time limit.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbchecktime`.
`data`	The user-defined object passed as `data` when setting up the callback with `addcbchecktime`.
`priority`	An integer that determines the order in which multiple checktime callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Further information

If the callback function returns a nonzero value the solution process will be interrupted.

Related topics

`problem.removecbchecktime`.

problem.addcbchgbranchobject

Purpose

Declares a callback function that will be called every time the Optimizer has selected a MIP entity for branching. Allows the user to inspect and override the Optimizer's branching choice. This callback function will be called in addition to any callbacks already added by `problem.addcbchgbranchobject`.

Synopsis

```
problem.addcbchgbranchobject(callback, data, priority)
newobject = callback(my_prob, my_object, obranch)
```

Arguments

`callback`	The callback function, which takes three arguments: `my_prob`, `my_object`, and `obranch`. This function is called every time the Optimizer has selected a candidate entity for branching.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user defined object passed as `data` when setting up the callback with `addcbchgbranchobject`.
`obranch`	The candidate branching object selected by the Optimizer.
`newobject`	New branching object to replace the Optimizer's selection. Can be `None`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Further information

1. The branching object given by the Optimizer provides a linear description of how the Optimizer intends to branch on the selected candidate. This will often be one of standard MIP entities of the current problem, but can also be e.g. a split disjunction or a structural branch, if those features are turned on.
2. The functions `branchobj.getbranches`, `branchobj.getbounds` and `branchobj.getrows` can be used to inspect the given branching object.
3. Refer to the `branchobj` class to learn how to create a new branching object to replace the Optimizer's selection. Note that the new branching object should be created with a priority value no higher than the current object to guarantee it will be used for branching.

Related topics

`problem.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(callback, data, priority)
ret = callback(my_prob, my_object)
```

Arguments

`callback`	The callback function which takes two arguments, `my_prob` and `my_object`, and has an integer return value.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbcutlog`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple cut log callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Further information

The callback `callback` should return a nonzero value to stop cutting on the current node.

Related topics

`problem.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(callback, data, priority)
callback(my_prob, my_object)
```

Arguments

`callback`	The callback function which takes two arguments, `my_prob` and `my_object`, and has no return value.
`my_prob`	The thread problem passed to the callback function.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbdestroymt`.
`data`	A user-defined object to be passed to the callback function.
`priority`	An integer that determines the order in which multiple callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Further information

This callback is useful for freeing up any user data created in the MIP thread callback.

Related topics

`problem.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(callback, data, priority)
(RelGapNotify, AbsGapNotify, AbsGapNotifyObj, AbsGapNotifyBound) =
      callback(my_prob, my_object)
```

Arguments

`callback` The callback function.

`data` A user-defined object that will be passed into the callback `callback`.

`priority` An integer that determines the order in which multiple gap notification callbacks will be invoked. The callback added with the higher priority will be called before a callback with a lower priority. Set to `0` if not required.

`my_prob` The current problem.

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

`RelGapNotify` The value the `miprelgapnotify` control will be set to after this callback. May be modified within the callback in order to set a new notification target. Can be `None`.

`AbsGapNotify` The value the `mipabsgapnotify` control will be set to after this callback. May be modified within the callback in order to set a new notification target. Can be `None`.

`AbsGapNotifyObj` The value the `mipabsgapnotifyobj` control will be set to after this callback. May be modified within the callback in order to set a new notification target. Can be `None`.

`AbsGapNotifyBound` The value the `mipabsgapnotifybound` control will be set to after this callback. May be modified within the callback in order to set a new notification target. Can be `None`.

Example

The following example prints a message when the gap reaches 10% and 1%

```
def gapnotify(prob, object):

    obj = prob.attributes.mipobjval
    bound = prob.attributes.bestbound

    # If no solutions were found, just return a tuple of None's
    if prob.attributes.mipsols == 0:
        return None, None, None, None

    if obj != 0 and bound != 0:
        relgap = abs((obj - bound) / max(abs(obj), abs(bound)))
    else:
        relgap = 0

    newRelGapNotifyTarget = -1

    if relgap <= 0.1:
        print('Gap reached 10%')
        newRelGapNotifyTarget = 0.1
```

```
            if relgap <= 0.01:
                print('Gap reached 1%')
                newRelGapNotifyTarget = -1 # Don't call gapnotify again

            # return a quadruple with new values, or
            # None for those that should not be set
            return (newRelGapNotifyTarget, None, None, None)

        prob.controls.miprelgapnotify = 0.1
        prob.addcbgapnotify(gapnotify, None, 0)
        prob.mipoptimize('')
```

Further information

The target values that caused the callback to be triggered will automatically be reset to prevent the same callback from being fired again.

Related topics

MIPRELGAPNOTIFY, MIPABSGAPNOTIFY, MIPABSGAPNOTIFYOBJ, MIPABSGAPNOTIFYBOUND, problem.removecbgapnotify.

problem.addcbmiplog

Purpose

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

Synopsis

```
problem.addcbmiplog(callback, data, priority)
ret = callback(my_prob, my_object)
```

Arguments

`callback`	The callback function which takes two arguments, `my_prob` and `my_object`, and has an integer return value. If the value returned by `callback` is nonzero, the solution process will be interrupted. This function is called whenever the MIP log is printed as determined by the `MIPLOG` control.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbmiplog`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple MIP log callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following example prints at each node of the tree search the node number and its depth:

```
prob.controls.miplog = 3
prob.addcbmiplog(mipLog, None, 0)
prob.mipoptimize('')
```

The callback function may resemble:

```
def mipLog(prob, object):

    nodedepth = prob.attributes.nodedepth
    node      = prob.attributes.currentnode

    print('Node {0} with depth {1} has been processed'.format
          (node, nodedepth))

    return 0
```

Further information

If the callback function returns a nonzero value, the tree search will be interrupted.

Related topics

`problem.removecbmiplog`, `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(callback, data, priority)
callback(my_prob, my_object)
```

Arguments

`callback`	The callback function which takes two arguments, `my_prob` and `my_object`, and has no return value. This function is called after the current node has been found to be infeasible.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbinfnode`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple user infeasible node callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following notifies the user whenever an infeasible node is found during the tree search:

```
prob.addcbinfnode(nodeInfeasible, None, 0)
prob.mipoptimize("")
```

The callback function may resemble:

```
def nodeInfeasible(prob, object):
    node = prob.attributes.currentnode
    print("Node {0} infeasible".format(node))
```

Related topics

`problem.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(callback, data, priority)
callback(my_prob, my_object)
```

Arguments

`callback`	The callback function which takes two arguments, `my_prob` and `my_object`, and has no return value. This function is called if the current node is found to have an integer feasible solution, i.e. every time an integer feasible solution is found.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbintsol`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple integer solution callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following example prints integer solutions as they are discovered in the tree search:

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

The callback function might resemble:

```
def printsol(my_prob, object):

    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(callback, data, priority)
ret = callback(my_prob, my_object)
```

Arguments

`callback`	The callback function which takes two arguments, `my_prob` and `my_object`, and has an integer return value. This function is called every `LPLOG` simplex iterations including iteration `0` and the final iteration.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcblplog`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple lplog callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following code sets a callback function, `lpLog`, to be called every `10` iterations of the optimization:

```
prob.controls.lplog = 10
prob.addcblplog(lpLog, None, 0)
prob.read("problem", "")
prob.mipoptimize("")
```

The callback function may resemble:

```
def lpLog(my_prob, object):

    iter = my_prob.attributes.simplexiter
    obj  = my_prob.attributes.lpobjval

    print("At iteration {0} objval is {1}".format(iter, obj))
    return 0
```

Further information

If the callback function returns a nonzero value, the solution process will be interrupted.

Related topics

`problem.removecblplog`, `problem.addcbbarlog`, `problem.addcbmiplog`, `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(callback, data, priority)
callback(my_prob, my_object, msg, msgtype)
```

Arguments

`callback`	The callback function which takes four arguments: `my_prob`, `my_object`, `msg`, and `msgtype`, and has no return value. Use a `None` value to cancel a callback function.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbmessage`.
`msg`	A string containing the message.
`msgtype`	Indicates the type of output message: `1` information messages; `2` (not used) `3` warning messages; `4` error messages. A negative value indicates that the Optimizer is about to finish and the buffers should be flushed at this time if the output is being redirected to a file.
`data`	A user-defined object to be passed to the callback function.
`priority`	An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following example simply sends all output to the screen (`stdout`):

```
prob.addcbmessage(Message, None, 0)
```

The callback function might resemble:

```
def Message(my_prob, object, msg, msgtype):

    print('{0}: {1}'.format(msgtype, msg))
```

Further information

1. The Xpress Python API registers a message callback that prints messages to `stdout`. This callback cannot be removed explicity but can be disabled using `xpress.setOutputEnabled`.
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.addcbmiplog`, `problem.addcblplog`, `problem.setlogfile`, `xpress.setOutputEnabled`.

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(callback, data, priority)
callback(my_prob, my_object, thread_prob)
```

Arguments

`callback`	The callback function which takes three arguments, `my_prob`, `my_object` and `thread_prob`, and has no return value.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed to the callback function.
`thread_prob`	The problem for the MIP thread
`data`	A user-defined object to be passed to the callback function.
`priority`	An integer that determines the order in which multiple callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following example clears the message callback for each of the MIP threads:

```
prob.addcbmipthread(mipthread, None, 0)

def mipthread(my_prob, my_object, mipthread):
    my_prob.removecbmessage(mipthread, None)
```

Further information

This function will be called when a new MIP worker problem is created. Each worker problem receives a unique identifier that can be obtained through the `MIPTHREADID` attribute. Worker problems can be matched with different system threads at different points of a solve, so the system thread that is responsible for executing the callback is not necessarily the same thread used for all subsequent callbacks for the same worker problem. On the other hand, worker problems are always assigned to a single thread at a time and the same nodes are always solved on the same worker problem in repeated runs of a deterministic MIP solve. A worker problem therefore acts as a virtual thread through the node solves.

Related topics

`problem.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(callback, data, priority)
callback(my_prob, my_object, parentnode, newnode, branch)
```

Arguments

`callback`	The callback function, which takes five arguments: `myprob`, `my_object`, `parentnode`, `newnode` and `branch`. This function is called every time a new node is created through branching.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbnewnode`.
`parentnode`	Unique identifier for the parent of the new node.
`newnode`	Unique identifier assigned to the new node.
`branch`	The sequence number of the new node amongst the child nodes of `parentnode`. For regular branches on a MIP entity this will be either `0` or `1`.
`data`	A user-defined object to be passed to the callback function.
`priority`	An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Further information

1. For regular branches on a MIP entity, `branch` will be either zero or one, depending on whether the new node corresponds to branching the MIP entity up or down.
2. When branching on a `branchobject`, `branch` refers to the given branch index of the object.

Related topics

`problem.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(callback, data, priority)
callback(my_prob, my_object, node)
```

Arguments

`callback`	The callback function, which takes three arguments, `my_prob`, `my_object` and `node`, and has no return value. This function is called every time a node is cut off as the result of an improved integer solution being found.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbnodecutoff`.
`node`	The number of the node that is cut off.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple node-optimal callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following notifies the user whenever a node is cutoff during the tree search:

```
prob.addcbnodecutoff(Cutoff, None, 0)
mipoptimize(prob, "")
```

The callback function might resemble:

```
def Cutoff(prob, object, node):

    print("Node {0} cutoff".format(node))
```

Further information

This function allows the user to keep track of the eligible nodes. Note that the LP solution will not be available from this callback.

Related topics

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

problem.addcbnodelpsolved

Purpose

Declares a node LP solved callback function, called during the branch and bound search, after the LP relaxation has been solved for the current node, but before any internal cuts and heuristics have been applied. This callback function will be called in addition to any callbacks already added by XPRSaddcbnodelpsolved.

Synopsis

```
problem.addcbnodelpsolved(callback, data, priority)
callback(my_prob, my_object)
```

Arguments

`callback`	The callback function, which takes two arguments, `my_prob` and `my_object`, and has no return value.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbnodecutoff`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple node-optimal callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Related topics

`problem.removecbnodelpsolved`, `problem.addcboptnode`

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(callback, data, priority)
infeas = callback(my_prob, my_object)
```

Arguments

`callback`	The callback function which takes two arguments, `my_prob` and `my_object`, and returns an integer. If the value returned by `callback` is nonzero, the solution process will be interrupted.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcboptnode`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple node-optimal callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following prints the optimal objective value of the node LP relaxations:

```
prob.addcboptnode(nodeOptimal, None, 0)
prob.mipoptimize("")
```

The callback function might resemble:

```
def nodeOptimal(prob, object):

    node = prob.attributes.currentnode
    print("NodeOptimal: node number", node)
    objval = prob.attributes.lpobjval
    print("Objective function value =", objval)
    return 0
```

Related topics

problem.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(callback, data, priority)
(ifreject, newcutoff) = callback(my_prob, my_object, soltype, cutoff)
```

Arguments

`callback` The callback function which takes four arguments, `my_prob`, `my_object`, `soltype` and `cutoff`, returns a tuple of two elements. This function is called when an integer solution is found, but before the solution is accepted by the Optimizer, allowing the user to reject the solution.

`my_prob` The problem passed to the callback function, `callback`.

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

`soltype` The type of MIP solution that has been found: Set to 1 if the solution was found using a heuristic. Otherwise, it will be the integer feasible solution to the current node of the tree search.

- `0` The continuous relaxation solution to the current node of the tree search, which has been found to be integer feasible.
- `1` A MIP solution found by a heuristic.
- `2` A MIP solution provided by the user.
- `3` A solution resulting from refinement of primal or dual violations of a previous MIP solution.

`cutoff` The current `cutoff` value.

`ifreject` If a nonzero value is returned in the first position of the tuple, the solution will be rejected.

`newcutoff` A new `cutoff` value can be returned in the second position of the tuple, to be used by the Optimizer if the solution is accepted. The returned `newcutoff` value will not be updated if the solution is rejected.

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

`priority` An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Further information

1. If a solution is rejected, the Optimizer will drop the found solution without updating any attributes, including the cutoff value. To change the cutoff value when rejecting a solution, the control `MIPABSCUTOFF` should be set instead.
2. When a node solution is rejected (`isheuristic = 0`), the node itself will be dropped without further branching.
3. To retrieve the integer solution, use either `problem.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(callback, data, priority)
nodinfeas = callback(my_prob, my_object)
```

Arguments

`callback`	The callback function, which takes two arguments, `my_prob`, `my_object` and and returns an integer. This function is called before a node is reoptimized and the node may be made infeasible by returning `1` from the callback.
`my_prob`	The problem passed to the callback function, `callback`.
`my_object`	The user-defined object passed as `data` when setting up the callback with `addcbprenode`.
`data`	A user-defined object to be passed to the callback function, `callback`.
`priority`	An integer that determines the order in which multiple preprocess node callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following example notifies the user before each node is processed:

```
prob.addcbprenode(preNode, None, 0)
prob.mipoptimize("")
```

The callback function might resemble:

```
def preNode(prob, object):

    return 0 # set to 1 if node is infeasible
```

Related topics

`problem.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(callback, data, priority)
callback(my_prob, my_object, solname, status)
```

Arguments

callback The callback function which takes four arguments, my_prob, my_object, id and status and has no return value.

my_prob The problem passed to the callback function, callback.

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

solname The string name assigned to the solution when it was loaded into the Optimizer using problem.addmipsol.

status One of the following status values:

- 0 An error occured while processing the solution.
- 1 Solution is feasible.
- 2 Solution is feasible after reoptimizing with fixed MIP entities.
- 3 A local search heuristic was applied and a feasible solution discovered.
- 4 A local search heuristic was applied but a feasible solution was not found.
- 5 Solution is infeasible and a local search could not be applied.
- 6 Solution is partial and a local search could not be applied.
- 7 Failed to reoptimize the problem with MIP entities fixed to the provided solution. Likely because a time or iteration limit was reached.
- 8 Solution is dropped. This can happen if the MIP problem is changed or solved to completion before the solution could be processed.

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

priority An integer that determines the order in which multiple callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Further information

If presolve is turned on, any solution added with problem.addmipsol will first be presolved before it can be checked. The value returned in status refers to the presolved solution, which might have had values adjusted due to bound changes, fixing of variables, etc.

Related topics

problem.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 rows (or their indices or names) for the coefficient.
`colindex`	Array holding the columns (or their indices or names) for the coefficient.
`factor`	Array holding factor by which formula is scaled. If `None`, a value of 1.0 will be used.
`fstart`	Integer array holding the start position in the arrays `Type` and `Value` of the formula for the coefficients. `fstart` should have an extra entry containing the next position after the end of the last formula.
`parsed`	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 `fstart`. For possible token types and values see the chapter on Formula Parsing in the SLP reference manual.

The `addcoef` function loads additional items into the SLP problem. The corresponding `loadcoefs` function deletes any existing items first.

The behaviour for existing coefficients is additive: the formula defined in the parameters are added to any existing formula coefficients. However, due to performance considerations, such duplications should be avoided when possible.

Related topics

`problem.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(objcoef, start, rowind, rowcoef, lb, ub, names, types)
```

Arguments

`objcoef` Array containing the objective function coefficients of the new columns.

`start` Array containing the offsets in the `rowind` and `rowcoef` arrays of the start of the elements for each column.

`rowind` Array containing the rows (i.e. `xpress.constraint` objects, indices, or names) for the elements in each column.

`rowcoef` Array containing the element values.

`lb` Array containing the lower bounds on the added columns.

`ub` Array containing the upper bounds on the added columns.

`names` (optional) Array containing the names of the columns added.

`types` (optional) Array of characters containing the types of the newly added columns:

- `C` indicates a continuous variable (default);
- `I` indicates an integer variable;
- `B` indicates a binary variable;
- `S` indicates a semi-continuous variable;
- `R` indicates a semi-continuous integer variable;
- `P` indicates a partial integer variable.

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], start=[0,3], rowind=[0, 1, 3],
          matval=[2,1,3], lb=[-xpress.infinity], ub=[12],
          names=['john_cleese'], types=['C'])
```

Further information

1. The constant `xpress.infinity` can be used to represent infinite bounds.
2. If the columns are added to a MIP problem, then they will be continuous variables unless `types` is specified. Use `problem.chgcoltype` to impose integrality conditions on such new columns.

Related topics

`problem.addrows`, `problem.chgcoltype`.

problem.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(cuttype, rowtype, rhs, start, colind, cutcoef)
```

Arguments

`cuttype` Array containing the user assigned cut types. The cut types can be any integer chosen by the user, and are used to identify the cuts in other cut manager routines using user supplied parameters. The cut type can be interpreted as an integer or a bitmap - see `problem.delcuts`.

`rowtype` Character array containing the row types:
- `L` indicates a $\leq$ row;
- `G` indicates $\geq$ row;
- `E` indicates an = row.

`rhs` Array containing the right hand side elements for the cuts.

`start` Array containing offset into the `colind` and `cutcoef` arrays indicating the start of each cut. This array is of length `ncuts+1` with the last element, `start[ncuts]`, being where cut `ncuts+1` would start.

`colind` Array containing the columns (i.e. `xpress.var` objects, indices, or names) in the cuts.

`cutcoef` Array containing the matrix values for the cuts.

Further information

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

2. The cuts added to the matrix are always added at the end of the matrix and the number of rows is always set to the original number of cuts added. If `ncuts` have been added, then the rows `0`,...,`ROWS-ncuts-1` are the original rows, whilst the rows `ROWS-ncuts`,...,`ROWS-1` are the added cuts. The number of cuts can be found by consulting the `CUTS` problem attribute.

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 columns (i.e. `xpress.var` objects, indices, or names) whose distribution factor is to be changed.
`rowindex`	Array of rows (i.e. `xpress.constraint` objects, indices, or names) where each distribution factor applies.
`value`	Array holding the new values of the distribution factors.

Example

The following example adds distribution factors as follows:
column 282 in row 134 = 0.1
column 282 in row 136 = 0.15
column 285 in row 133 = 1.0.

```
colindex = [282, 282, 285]
rowindex = [134, 136, 133]
value    = [0.1, 0.15,  1]
p.adddfs(colindex,rowindex,value)
```

Further information

The *distribution factor* of a column in a row is the matrix coefficient of the corresponding delta vector in the row. Distribution factors are used in conventional recursion models, and are essentially normalized first-order derivatives. Xpress SLP can accept distribution factors instead of initial values, provided that the values of the variables involved can all be calculated after optimization using determining rows, or by a callback.

The `problem.adddfs` functions load additional items into the SLP problem. The corresponding `problem.loaddfs` functions delete any existing items first.

Related topics

`problem.chgdf`, `problem.getdf`, `problem.loaddfs`

problem.addgencons

Purpose

Adds one or more general constraints to the problem. Each general constraint `y = f(x1, ..., xn, c1, ..., cn)` consists of one or more (input) columns xi, zero or more constant values ci and a resultant (output column) y. General constraints can be defined using operators such as `maximum` and `minimum` (at least one input column of any contype and arbitrary number of input values), `and` and `or` (at least one binary input column, no constant values, binary resultant) and `absolute value` (exactly one input column of arbitrary contype, no constant values).

Synopsis

```
problem.addgencons (contype, resultant, colstart, colind, valstart, val)
```

Arguments

`contype`	list or array containing the types of the general constraints: `xpress.gencons_max (0)` indicates a `maximum` constraint; `xpress.gencons_min (1)` indicates a `minimum` constraint; `xpress.gencons_and (2)` indicates an `and` constraint; `xpress.gencons_or (3)` indicates an `or` constraint; `xpress.gencons_abs (4)` indicates an `absolute value` constraint.
`resultant`	Array/list containing the output variables (or indices thereof) of the general constraints.
`colstart`	Array/list containing the start index of each general constraint in the `colind` array.
`colind`	Array/list containing the input variables in all general constraints.
`valstart`	Array/list containing the start index of each general constraint in the `val` array (may be `None`).
`val`	Array/list containing the constant values in all general constraints (may be `None`).

Example

This adds two new general constraints `x2 = max(x0, x1, 5)` and `x3 = |x1|`:

```
contype = [xpress.gencons_max, xpress.gencons_abs]
resultant = [2, 3]
colstart = [0, 2]
colind = [0, 1, 1]
valstart = [0, 1]
val = [5.0]

prob.addgencons(contype, resultant, colstart, colind, valstart, val);
prob.optimize()
```

Further information

General constraints must be set up before solving the problem. They are converted to additional binary variables, indicator and linear constraints with the exact formulation and number of added entities depending on the performed presolving.

Note that using non-binary variables in `and`/`or` constraints or adding constant values to them or `absolute value` constraints will give an error at solve time.

Related topics

`problem.getgencons`, `problem.delgencons`, `xpress.And`, `xpress.Or`, `xpress.max`, `xpress.min`, `xpress.abs`.

problem.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(solval, colind, name)
```

Arguments

`solval`	Array containing solution values.
`colind`	Optional integer array containing the columns (i.e. `xpress.var` objects, indices, or names) for the solution values provided in `solval`. It is optional when the length of solval is equal to `COLS`, in which case it is assumed that `solval` provides a complete solution vector.
`name`	An optional name to associate with the solution.

Further information

1. The function returns immediately after passing the solution to the Optimizer. The solution is placed in a pool until the Optimizer is able to analyze the solution during a MIP solve.
2. If the provided solution is found to be infeasible, a limited local search heuristic will be run in an attempt to find a close feasible integer solution.
3. If a partial solution is provided, discrete columns will be fixed to any provided values and a limited local search will be run in an attempt to find integer feasible values for the remaining unspecified columns. Values provided for continuous column in partial solutions are currently ignored.
4. The `problem.addcbusersolnotify` callback function can be used to discover the outcome of a loaded solution. The optional name provided as `name` will be returned in the callback function.
5. If one or more solutions are loaded during the `problem.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.addobj

Purpose

Appends an objective function with the given coefficients to a multi-objective problem. The weight and priority of the objective are set to the given values.

Synopsis

```
problem.addobj(colind, objcoef, priority=0, weight=1)
```

Arguments

`colind`	Integer array of length `ncols` containing the indices of the columns whose objective coefficients will change. An index of `-1` indicates that the fixed part of the objective function on the right hand side should change.
`objcoef`	Double array of length `ncols` giving the new objective function coefficients.
`priority`	The priority for the objective function. During optimization, objectives with the same priority are combined together in a weighted sum.
`weight`	The weight for the objective function. If the weight is negative, the sense of this objective is reversed.

Example

Adding a second objective function to a problem:

```
colind = [0, 2, 5]
objcoef = [25.0, 5.3, 0.0]
p.addobj(colind, objcoef, 1, 1)
```

Related topics

`problem.addObjective`, `problem.setObjective`, `problem.chgobjn`, `problem.getobjn`, `problem.delobj`, `problem.chgobj`.

problem.addObjective

Purpose

Adds one or more objective functions to the problem.

Synopsis

```
problem.addObjective(obj1, obj2, ..., priority=None, weight=None,
       abstol=None, reltol=None)
```

Arguments

`obj1,obj2,...` Objectives to add to the problem. An error will be returned if any variable in any objective was not already added to the problem via `addVariable`.

`priority` (optional) Priority for the new objectives (only relevant for multi-objective problems).

`weight` (optional) Weight for the new objectives (only relevant for multi-objective problems).

`abstol` (optional) Absolute tolerance for the new objectives (only relevant for multi-objective problems).

`reltol` (optional) Relative tolerance for the new objectives (only relevant for multi-objective problems).

Example

The following example adds two objective functions to the problem:

```
x1 = xpress.var()
x2 = xpress.var()
p = xpress.problem()
p.addVariable(x1, x2)
p.addObjective(2*x1**2 + 3*x1*x2 + 5*x2**2 + 4*x1 + 4)
p.addObjective(x1**2)
```

Related topics

`problem.setObjective`, `problem.addobj`, `problem.chgobjn`, `problem.delobj`, `problem.chgobj`.

problem.addpwlcons

Purpose

Adds one or more piecewise linear constraints to the problem. Each piecewise linear constraint `y = f(x)` consists of an (input) column x, a resultant (output column) y and a piecewise linear function f. The piecewise linear function f is described by a number of breakpoints, which are given as combinations of x- and y-values. Discontinuous piecewise linear functions are supported, in this case both the left and right limit at a given point need to be entered as breakpoints. To differentiate between left and right limit, the breakpoints need to be given as a list with non-decreasing x-values.

Synopsis

```
problem.addpwlcons(colind, resultant, start, xval, yval)
```

Arguments

`colind`	Integer array (or list) containing the input variables x of the piecewise linear functions.
`resultant`	Integer array containing the output variables y of the piecewise linear functions.
`start`	Integer array containing the start index of each piecewise linear constraint in the `xval` and `yval` arrays.
`xval`	Array containing the x-values of the breakpoints.
`yval`	Array containing the y-values of the breakpoints.

Example

This adds a new piecewise linear constraint `y = f(x)`, where

```
f(x) = -x           if x < 0
f(x) = 1      if 0 <= x <= 2
f(x) = 2x-3         if x > 2
```

```
colind = [x]
resultant = [y]
start = [0]
xval = [-1, 0, 0, 2, 3]
yval = [1, 0.5, 1, 1, 3]

prob.addpwlcons(colind, resultant, start, xval, yval)
prob.setObjective(y)   # the piecewise linear function is to be minimized
prob.mipoptimize()
```

Further information

Piecewise linear constraints must be set up before solving the problem. They are converted to additional linear constraints, continuous variables and SOS2 constraints, with the exact formulation and number of added entities depending on the convexity of the piecewise linear function and some presolving steps that are applied.

Related topics

`problem.getpwlcons`, `problem.delpwlcons`, `xpress.pwl`.

problem.addqmatrix

Purpose

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

Synopsis

```
problem.addqmatrix(row, rowqcol1, rowqcol2, rowqcoef)
```

Arguments

`row`	Row (i.e. `xpress.constraint` object, index, or name) where the quadratic matrix is to be added.
`rowqcol1`	Array with first variables (i.e. `xpress.var`objects, indices, or names) in the triplets.
`rowqcol2`	Array with second variables (i.e. `xpress.var`objects, indices, or names) index in the triplets.
`rowqcoef`	Array of coefficients in the triplets.

Further information

1. The triplets should define the upper triangular part of the quadratic expression. This means that to add $x^2 + 4xy$ the `rowqcoef` array shall contain the coefficients 1 and 2.
2. The matrix defined by `rowqcol1`, `rowqcol2` and `rowqcoef` 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(rowtype, rhs, start, colind, rowcoef, range=None,
       names=None)
```

Arguments

`rowtype`	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.
`start`	Array containing the offsets in the `colind` and `rowcoef` arrays of the start of the elements for each row.
`colind`	Array containing the (contiguous) columns (i.e. `xpress.var`objects, indices, or names) for the elements in each row.
`rowcoef`	Array containing the (contiguous) element coefficients.
`range`	(optional) Array containing the row range elements. The values in the `range` array will only be read for `'R'` type rows. The entries for other type rows will be ignored.
`names`	(optional) Array of names to be assigned to each new row.

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], [0,3], [0,1,2],
          rowcoef=[8, 9, 10], range=None, names=['NewRow'])
```

Further information

Range rows are automatically converted to type `L`, with an upper bound in the slack. This must be taken into consideration, when retrieving row type, right–hand side values or range information for rows.

Related topics

`problem.addcols`, `problem.addcuts`.

problem.addsetnames

Purpose

When a model with MIP entities is loaded, any special ordered sets may not have names associated with them. If you wish names to appear in the ASCII solutions files, the names for a range of sets can be added with this function.

Synopsis

```
problem.addsetnames(names, first=0, last=problem.attributes.sets - 1);
```

Arguments

`names`	A list of strings contatining all names to be assigned.
`first`	(Optional) first of the set range.
`last`	(Optional) last of the set range.

Example

Add set names (`set1` and `set2`) to a problem:

```
snames = ["set1", "set2"]
...
p.addsetnames(snames, 0, 1);
```

Further information

If `first` is not provided, it is considered equal to 0; if `last` is omitted, a value of `problem.attributes.sets - 1` is used.

Related topics

`problem.loadproblem`,

problem.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 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` constructor or the `xpress.vars` function.

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` Array holding the initial value for each SLP variable (use the `VarType` bit map to indicate if a value is being provided)
May be `None` if not required.

`stepbound` Array holding the initial step bound size for each SLP variable (a zero value means that no initial step bound size has been specified). If a value of `xpress.infinity` is used for a value in `stepbound`, the delta will never have step bounds applied, and will almost always be regarded as converged.
May be `None` if not required.

Example

The following example loads two SLP variables into the problem. They correspond to columns 23 and 25 of the underlying LP problem. Column 25 has an initial value of 1.42; column 23 has no specific initial value

```
colindex  = [23,25]
vartype   = [0,2]
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, scaled)
```

Arguments

`type`	`0`	Condition number of the basis.
	`1`	Stability measure for the solution relative to the current basis.
	`2`	Stability measure for the duals relative to the current basis.
	`3`	Stability measure for the right hand side relative to the current basis.
	`4`	Stability measure for the basic part of the objective relative to the current basis.
`norm`	`0`	Use the infinity norm.
	`1`	Use the 1 norm.
	`2`	Use the Euclidian norm for vectors and the Frobenius norm for matrices.
`scaled`	If the stability values are to be calculated in the scaled or the unscaled matrix.	

Further information

1. The condition number (`type` = `0`) of an invertible matrix is the norm of the matrix multiplied with the norm of its inverse. This number is an indication of how accurate the solution can be calculated and how sensitive it is to small changes in the data. The larger the condition number is, the less accurate the solution is likely to become.
2. The stability measures (`type` = `1...4`) are using the original matrix and the basis to recalculate the various vectors related to the solution and the duals. The returned stability measure is the norm of the difference of the recalculated vector to the original one.

problem.bndsa

Purpose

Returns upper and lower sensitivity ranges for specified variables' lower and upper bounds. If the bounds are varied within these ranges the current basis remains optimal and feasible.

Synopsis

```
problem.bndsa(colind, lblower=None, lbupper=None, ublower=None,
      ubupper=None)
```

Arguments

`colind`	A list or Numpy array of the variables (or their indices or names), for which the sensitivity range is requested.
`lblower`	Array (to be passed as a list, possibly empty) that will contain the variable lower bound lower ranges.
`lbupper`	Array for the variable lower bound upper ranges.
`ublower`	Array for the variable upper bound lower ranges.
`ubupper`	Array for the variable upper bound upper ranges.

Example

`problem.bndsa` can only be called when an optimal solution to the current LP has been found. It cannot be used when the problem is MIP presolved.

```
x = xp.vars(10)
[...]
ll, lu, ul, uu = [], [], [], []
p.bndsa(x, ll, lu, ul, uu)
print("ranges:", ll, lu, ul, uu)
```

Further information

If the problem is in a presolved state, `btran` works with the basis for the presolved problem.

Related topics

`problem.rhssa`, `problem.objsa`.

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 `problem.attributes.rows` containing the values by which the basis inverse is to be multiplied. The transformed values will also be returned in this array.

Example

Get the (unscaled) tableau row `z` of constraint number `irow`, assuming that all arrays have been dimensioned.

```
y = [0,1,0,0]
p.btran(y)
print("btran result:", y)
```

Further information

If the problem is in a presolved state, `btran` works with the basis for the presolved problem.

Related topics

`problem.ftran`.

problem.calcobjn

Purpose

Returns the value of a given objective. A solution can optionally be provided, otherwise the current solution will be used.

Synopsis

```
objval = problem.calcobjn(objidx, solution)
```

Arguments

`objidx`	Index of the objective to calculate.
`solution`	Array of length `problem.attributes.cols` that holds the solution.

Further information

The calculations are always carried out in the original problem, even if the problem is currently presolved.

Related topics

`problem.setObjective`

problem.calcobjective

Purpose

Returns the objective value of a given solution.

Synopsis

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

Argument

`solution` Array of length `problem.attributes.cols` that holds the solution.

Further information

The calculations are always carried out in the original problem, even if the problem is currently presolved.

Related topics

`problem.calcslacks`, `problem.calcreducedcosts`.

problem.calcreducedcosts

Purpose

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

Synopsis

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

Arguments

`duals`	Array of length `problem.attributes.rows` that holds the dual solution to calculate the reduced costs for.
`solution`	Optional array of length `problem.attributes.cols` that holds the primal solution. This is necessary for quadratic problems.
`djs`	Array of length `problem.attributes.cols` in which the calculated reduced costs are returned.

Example

```
p = xpress.problem()
p.read("silly_walks.lp") # assume problem has 4 constraints
dj = []
p.calcreducedcosts([0,1,1,1], None, dj)
print("red. cost:", dj)
```

Further information

1. The calculations are always carried out in the original problem, even if the problem is currently presolved.
2. If using the function during a solve (e.g. from a callback), use ORIGINALCOLS and ORIGINALROWS to retrieve the non-presolved dimensions of the problem.

Related topics

`problem.calcslacks`, `problem.calcobjective`.

problem.calcslacks

Purpose

Calculates the row slack values for a given solution.

Synopsis

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

Arguments

`solution`	Array of length `problem.attributes.cols` that holds the solution to calculate the slacks for.
`slacks`	Array of length `problem.attributes.rows` in which the calculated row slacks are returned.

Further information

1. The calculations are always carried out in the original problem, even if the problem is currently presolved.
2. If using the function during a solve (e.g. from a callback), use ORIGINALCOLS and ORIGINALROWS to retrieve the non-presolved dimensions of the problem.

Related topics

`problem.calcreducedcosts`, `problem.calcobjective`.

problem.calcsolinfo

Purpose

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

Synopsis

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

Arguments

`solution`	Array of length `problem.attributes.cols` that holds the solution.	
`duals`	Array of length `problem.attributes.rows` that holds the duals solution.	
`property`	`xpress.solinfo_absprimalinfeas`	absolute primal infeasibility.
	`xpress.solinfo_relprimalinfeas`	relative primal infeasibility.
	`xpress.solinfo_absdualinfeas`	absolute duals infeasibility.
	`xpress.solinfo_reldualinfeas`	relative duals infeasibility.
	`xpress.solinfo_maxmipfractional`	absolute MIP infeasibility (fractionality).

Further information

The calculations are always carried out in the original problem, even if the problem is currently presolved.

Related topics

`problem.calcslacks`, `problem.calcobjective`, `problem.calcreducedcosts`.

problem.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(col=colnum)

p.cascade()
```

`problem.getvar` and `problem.chgvar` are being used to get and change the current value of a single variable. Provided no other values have been changed since the last execution of `cascade`, values will be changed only for variables which depend on column 91.

Further information

See the section on cascading for an extended discussion of the types of cascading which can be performed.

`cascade` is called automatically during the SLP iteration process and so it is not normally necessary to perform an explicit cascade calculation.

The variables are re-calculated in accordance with the order generated by `problem.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.

```
for colnum in range(1, nCol):
  p.chgvar(col=colnum, value=DefaultValue[ColNum])
p.cascadeorder()
p.cascade()
```

Further information

`cascadeorder` is called automatically at the start of the SLP iteration process and so it is not normally necessary to perform an explicit cascade ordering.

Related topics

`problem.cascade`

problem.chgbounds

Purpose

Changes the bounds on columns in the problem.

Synopsis

```
problem.chgbounds(colind, bndtype, bndval)
```

Arguments

`colind` Array containing the columns (i.e. `xpress.var` objects, indices, or names) on which the bounds will change.

`bndtype` Character array indicating the type of bound to change:
- `U` indicates a change in the upper bound;
- `L` indicates a change in the lower bound;
- `B` indicates a change in both bounds, i.e. the column is fixed.

`bndval` Array giving the new bound values.

Example

The following changes the lower bound of variable `v1` to 2, upper bound of variable `v2` to 5, and fixes variable `v3` to 3:

```
p.chgbounds([v1,v2,v3],['L','U','B'],[2,5,3])
```

Further information

1. A column may appear twice in the `colind` array so it is possible to change both the upper and lower bounds on a variable in one go.
2. `chgbounds` may be applied to the problem in a presolved state, in which case it expects references to the presolved problem.
3. The constant `xpress.infinity` can be used to represent plus and minus infinity in the bound (`bndval`) array.
4. If the upper bound on a binary variable is changed to be greater than 1 or the lower bound is changed to be less than 0 then the variable will become an integer variable.

Related topics

`problem.getlb`, `problem.getub`.

problem.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(row, col, coef)
```

Arguments

`row`	Row (i.e. `xpress.constraint` object, index, or name) for the coefficient.
`col`	Column (i.e. `xpress.var` object, index, or name) for the coefficient.
`coef`	New value for the coefficient. If `coef` is zero, any existing coefficient will be deleted.

Example

In the following, the constraint is introduced in the problem and then its linear coefficient for `x` is changed to 3:

```
p = xpress.problem()
x = 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(colind, coltype)
```

Arguments

`colind` Array containing the columns (i.e. `xpress.var` objects, indices, or names) whose type is to be changed.

`coltype` Character array giving the new column types:

- `C` indicates a continuous column;
- `B` indicates a binary column;
- `I` indicates an integer column.
- `S` indicates a semi–continuous column. The semi–continuous lower bound will be set to `1.0`.
- `R` indicates a semi–integer column. The semi–integer lower bound will be set to `1.0`.
- `P` indicates a partial integer column. The partial integer 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 tree search is started.
2. Calling `chgcoltype` to change any variable into a binary variable causes the bounds previously defined for the variable to be deleted and replaced by bounds of 0 and 1.
3. Calling `chgcoltype` to change a continuous variable into an integer variable cause its lower bound to be rounded up to the nearest integer value and its upper bound to be rounded down to the nearest integer value.

Related topics

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

problem.chgcascadenlimit

Purpose

Set a variable specific cascade iteration limit

Synopsis

```
problem.chgcascadenlimit(col, limit)
```

Arguments

`col`	The column corresponding to the SLP variable for which the cascading limit is to be imposed.
`limit`	The new cascading iteration limit.

Further information

A value set by this function will overwrite the value of the control `xslp_cascadenlimit` for this variable. To remove any previous value set by this function, use an iteration limit of 0.

Related topics

`problem.cascadeorder`

problem.chgccoef

Purpose

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

Synopsis

```
problem.chgccoef(row, col, factor, formula)
```

Arguments

`row`	The row (i.e. `xpress.constraint` object, index, or name) for the coefficient.
`col`	The column (i.e. `xpress.var` object, index, or name) for the coefficient.
`factor`	Constant multiplier for the formula. If `factor` is `None`, a value of 1.0 will be used.
`formula`	String holding the formula, with the tokens separated by spaces.

Example

Assuming that the columns of the matrix are named `Col1`, `Col2`, etc, the following example puts the formula `2.5*sin(Col1)` into the coefficient in row 1, column 3.

```
Formula = "sin ( Col1 )"
Factor = 2.5
p.chgccoef(1, 3, Factor, Formula)
```

Note that all the tokens in the formula (including mathematical operators and separators) are separated by one or more spaces.

Further information

If the coefficient already exists as a constant or formula, it will be changed into the new coefficient. If it does not exist, it will be added to the problem.

A coefficient is made up of two parts: `Factor` and `Formula`. `Factor` is a constant multiplier which can be provided in the `Factor` variable. If Xpress Nonlinear can identify a constant factor in the Formula, then it will use that as well, to minimize the size of the formula which has to be calculated.

This function can only be used if all the operands in the formula can be correctly identified as constants, existing columns, character variables or functions. Therefore, if a formula refers to a new column, that new item must be added to the Xpress Nonlinear problem first.

Related topics

`problem.addcoefs`, `problem.delcoefs`, `problem.chgnlcoef`, `problem.getcoefformula`, `problem.loadcoefs`

problem.chgdeltatype

Purpose

Changes the type of the delta assigned to a nonlinear variable

Synopsis

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

Arguments

`varind` 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(col, row, value)
```

Arguments

`col`	The column (i.e. `xpress.var` object, index, or name) whose distribution factor is to be set or changed.
`row`	The row (i.e. `xpress.constraint` object, index, or name) where the distribution applies.
`value`	The new value of the distribution factor. May be `None` if not required.

Example

The following example retrieves the value of the distribution factor for column 282 in row 134 and changes it to be twice as large.

```
value = p.getdf(282,134)
value *= 2
p.chgdf(282,134,value)
```

Further information

The *distribution factor* of a column in a row is the matrix coefficient of the corresponding delta vector in the row. Distribution factors are used in conventional recursion models, and are essentially normalized first-order derivatives. Xpress Nonlinear can accept distribution factors instead of initial values, provided that the values of the variables involved can all be calculated after optimization using determining rows, or by a callback.

Related topics

`problem.adddfs`, `problem.getdf`, `problem.loaddfs`

problem.chgglblimit

Purpose

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

Synopsis

```
problem.chgglblimit(colind, limit)
```

Arguments

`colind`	Array containing the indices of the semi-continuous, semi-integer or partial integer columns that should have their limits changed.
`limit`	Array giving the new limit values.

Further information

1. The new limits are not allowed to be negative.
2. Partial integer limits can be at most 2^{28}.

Related topics

`problem.chgcoltype`, `problem.getmipentities`.

problem.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(rowind, colind, rowcoef)
```

Arguments

`rowind` Array containing the rows (i.e. `xpress.constraint` objects, indices, or names) of the coefficients to be changed.

`colind` Array containing the columns (i.e. `xpress.var` objects, indices, or names) of the coefficients to be changed.

`rowcoef` Array containing the new coefficient values. If an element of `rowcoef` is zero, the coefficient will be deleted.

Example

```
con1 = x + y + z <= 2
con2 = x + y >= 1
con3 = x + 3*y == 1
p.addVariable(x,y,z)
p.addConstraint(con1, con2, con3)
p.chgmcoef([con1,con1,con1,con2,con3], [x,y,z,x,x], [-2, -3, -3.2, 1, 3])
```

This changes five coefficients, three of which in the first constraint and one in each of the second and third constraints.

Further information

`chgmcoef` is more efficient than repeated calls to `problem.chgcoef` and should be used in its place if many coefficients are to be changed.

Related topics

`problem.chgcoef`, `problem.chgmqobj`, `problem.chgobj`, `problem.chgqobj`, `problem.chgrhs`, `problem.getcols`, `problem.getrhs`.

problem.chgobjn

Purpose

Modifies one or more coefficients of an objective function in a multi-objective problem. If the objective already exists, any coefficients not present in the `colind` and `objcoef` arrays will unchanged. If the objective does not exist, it will be added to the problem.

Synopsis

```
problem.chgobjn(objidx, colind, objcoef)
```

Arguments

`objidx`	Index of the objective function to add or modify.
`colind`	Integer array of length `ncols` containing the indices of the columns whose objective coefficients will change. An index of `-1` indicates that the fixed part of the objective function on the right hand side should change.
`objcoef`	Double array of length `ncols` giving the new objective function coefficients.

Example

Changing three coefficients of the first objective function:

```
colind = [0, 2, 5]
objcoef = [25.0, 5.3, 0.0]
p.chgobjn(0, colind, objcoef)
```

Further information

1. When `objidx=0`, this function is equivalent to `problem.chgobj`.
2. Any objectives with `idx < objidx` that do not already exist will be added to the problem with all zero coefficients.

Related topics

`problem.addObjective`, `problem.setObjective`, `problem.addobj`, `problem.getobjn`, `problem.delobj`, `problem.chgobj`.

problem.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(objqcol1, objqcol2, objqcoef)
```

Arguments

`objqcol1`	Array containing the column index of the first variable in each quadratic term.
`objqcol2`	Array containing the column index of the second variable in each quadratic term.
`objqcoef`	New values for the coefficients. If an entry in `objqcoef` is `0`, the corresponding entry will be deleted. These are the coefficients of the lower triangular part of the Hessian of the objective function.

Example

The following code results in an objective function with terms: $[4x_1^2 + 6x_1x_2$

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

Further information

1. The columns in the arrays `objqcol1` and `objqcol2` 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(row, col, factor, parsed, type, value)
```

Arguments

`row`	The index of the matrix row for the coefficient.
`col`	The index of the matrix column for the coefficient.
`factor`	The constant multiplier for the formula. If `factor` is `None`, a value of 1.0 will be used.
`parsed`	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 string form.

Further information

If the coefficient already exists as a constant or formula, it will be changed into the new coefficient. If it does not exist, it will be added to the problem.

A coefficient is made up of two parts: `Factor` and `Formula`. `Factor` is a constant multiplier which can be provided in the `factor` variable. If Xpress Nonlinear can identify a constant factor in the Formula, then it will use that as well, to minimize the size of the formula which has to be calculated.

Related topics

`problem.addcoefs`, `problem.chgccoef`, `problem.delcoefs`, `problem.getcoefformula`, `problem.loadcoefs`

problem.chgobj

Purpose

Change the objective function coefficients.

Synopsis

```
problem.chgobj(colind, objcoef)
```

Arguments

`colind`	Array containing the columns (i.e. `xpress.var` objects, indices, or names) on which the range elements will change. An index of `-1` indicates that the fixed part of the objective function on the right hand side should change.
`objcoef`	Array giving the new objective function coefficient.

Example

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

```
p.chgobj([x1,x2,x3,-1], [3.5, -2, 0, 224])
```

Further information

The value of the fixed part of the objective function can be obtained using the `OBJRHS` problem attribute.

Related topics

`problem.chgcoef`, `problem.chgmcoef`, `problem.chgmqobj`, `problem.chgqobj`, `problem.getobj`.

problem.chgobjsense

Purpose

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

Synopsis

```
problem.chgobjsense(objsense)
```

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 `(objqcol1,objqcol2)` of the Hessian matrix.

Synopsis

```
problem.chgqobj(objqcol1, objqcol2, objqcoef)
```

Arguments

`objqcol1`	Column index for the first variable in the quadratic term.
`objqcol2`	Column index for the second variable in the quadratic term.
`objqcoef`	New value for the coefficient in the quadratic Hessian matrix. If an entry in `objqcoef` is `0`, the corresponding entry will be deleted.

Example

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

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

Further information

1. The columns `objqcol1` and `objqcol2` must already exist in the matrix..
2. If `objqcol1` is not equal to `objqcol2`, then both the matrix elements `(objqcol1, objqcol2)` and `(objqcol2, objqcol1)` 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(row, rowqcol1, rowqcol2, rowqcoef)
```

Arguments

`row`	Row (i.e. `xpress.constraint` object, index, or name) where the quadratic matrix is to be changed.
`rowqcol1`	First index of the coefficient to be changed.
`rowqcol2`	Second index of the coefficient to be changed.
`rowqcoef`	The new coefficient.

Further information

1. This function may be used to add new nonzero coefficients, or even to define the whole quadratic expression with it. Doing that, however, is significantly less efficient than adding the whole expression with `problem.addqmatrix`.
2. The row must not be an equality or a ranged row.

Related topics

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

problem.chgrhs

Purpose

Changes right–hand side values of the problem.

Synopsis

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

Arguments

`rowind`	Array containing the rows (i.e. `xpress.constraint` objects, indices, or names) whose right hand side will change.
`rhs`	Array containing the right hand side values.

Example

Here we change the three right hand sides in rows `2`, `6`, and `8` to new values:

```
p.chgrhs([2,8,6], [5, 3.8, 5.7])
```

Related topics

`problem.chgcoef`, `problem.chgmcoef`, `problem.chgrhsrange`, `problem.getrhs`, `problem.getrhsrange`.

problem.chgrhsrange

Purpose

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

Synopsis

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

Arguments

`rowind`	Array containing the rows (i.e. `xpress.constraint` objects, indices, or names) on which the range elements will change.
`rng`	Array containing the range values.

Example

Here, the constraint `cons1` $x + y \leq 10$ 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(row, status)
```

Arguments

`row` The index of the matrix row to be changed.

`status` The bitmap with the new status settings. If the status is to be changed, always get the current status first (use `problem.getrowstatus`) 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(rowind, rowtype)
```

Arguments

`rowind` Array containing the rows (i.e. `xpress.constraint` objects, indices, or names).

`rowtype` Character array giving the new row types:

- `L` indicates a $\leq$ 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(row, weight)
```

Arguments

`row`	The row (i.e. `xpress.constraint` object, index, or name) whose weight is to be set or changed.
`weight`	The new value of the weight. May be `None` if not required.

Example

The following example sets the initial weight of row number 2 to a fixed value of 3.6 and the initial weight of row 4 to a value twice the calculated default value.

```
p.chgrowwt(2, -3.6)
p.chgrowwt(4,2)
```

Further information

A positive value is interpreted as a multiplier of the default row weight calculated by Xpress SLP.

A negative value is interpreted as a fixed value: the absolute value is used directly as the row weight.

The initial row weight is used only when the augmented structure is created. After that, the current weighting can be accessed and changed using `problem.getrowinfo`.

Related topics

`problem.getrowwt`, `problem.getrowinfo`

problem.chgtolset

Purpose

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

Synopsis

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

Arguments

`tolset`	Tolerance set for which values are to be changed. A zero value for `tolset` will create a new set.
`status`	A bitmap describing which tolerances are active in this set. See below for the settings.
`tols`	Array of 9 values holding the values for the corresponding tolerances.

Example

The following example creates a new tolerance set with the default values for all tolerances except the relative delta tolerance, which is set to 0.005. It then changes the value of the absolute delta and absolute impact tolerances in tolerance set 6 to 0.015

```
Tols = 9*[0]
Tols[2] = 0.005
Status = 1<<2

p.chgtolset(0, 1<<2, Tols)
Tols[1] = 0.015
Tols[5] = 0.015
Status = 1<<1 | 1<<5
p.chgtolset(6, Status, Tols)
```

Further information

The bits in `status` are set to indicate that the corresponding tolerance is to be changed in the tolerance set. The meaning of the bits is as follows:

Entry / Bit	Tolerance	XSLP constant	XSLP bit constant
0	Closure tolerance (TC)	`xslp_TOLSET_TC`	`xslp_TOLSETBIT_TC`
1	Absolute delta tolerance (TA)	`xslp_TOLSET_TA`	`xslp_TOLSETBIT_TA`
2	Relative delta tolerance (RA)	`xslp_TOLSET_RA`	`xslp_TOLSETBIT_RA`
3	Absolute coefficient tolerance (TM)	`xslp_TOLSET_TM`	`xslp_TOLSETBIT_TM`
4	Relative coefficient tolerance (RM)	`xslp_TOLSET_RM`	`xslp_TOLSETBIT_RM`
5	Absolute impact tolerance (TI)	`xslp_TOLSET_TI`	`xslp_TOLSETBIT_TI`
6	Relative impact tolerance (RI)	`xslp_TOLSET_RI`	`xslp_TOLSETBIT_RI`
7	Absolute slack tolerance (TS)	`xslp_TOLSET_TS`	`xslp_TOLSETBIT_TS`
8	Relative slack tolerance (RS)	`xslp_TOLSET_RS`	`xslp_TOLSETBIT_RS`

The xslp_TOLSET constants can be used to access the corresponding entry in the value arrays, while the xslp_TOLSETBIT constants are used to set or retrieve which tolerance values are used for a given SLP variable. The members of the `Tols` array corresponding to nonzero bit settings in `Status` will be used to change the tolerance set. So, for example, if bit 3 is set in `Status`, then `Tols[3]` will replace the current value of the absolute coefficient tolerance. If a bit is not set in `Status`, the value of the corresponding element of `Tols` is unimportant.

Related topics

`problem.addtolsets`, `problem.deltolsets`, `problem.gettolset`, `problem.loadtolsets`

problem.chgvar

Purpose

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

Synopsis

```
problem.chgvar(col=None, detrow=None, initstepbound=None, stepbound=None,
       penalty=None, damp=None, initial=None, value=None, tolset=None,
       history=None, converged=None, vartype=None)
```

Arguments

`col` The index of the matrix column.

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

`initstepbound` The initial step bound size. May be `None` if not required.

`stepbound` The current step bound size. Use zero to disable the step bounds. May be `None` if not required.

`penalty` The weighting of the penalty cost for exceeding the step bounds. May be `None` if not required.

`damp` The damping factor for the variable. May be `None` if not required.

`initial` The initial value for the variable. May be `None` if not required.

`value` The current value for the variable. May be `None` if not required.

`tolset` The index of the tolerance set for this variable. Use zero if there is no specific tolerance set. May be `None` if not required.

`history` The history value for this variable. May be `None` if not required.

`converged` The convergence status for this variable. May be `None` if not required.

`vartype` A bitmap defining the existence of certain properties for this variable:
`Bit 1:` Variable has a delta vector
`Bit 2:` Variable has an initial value
`Bit 14:` Variable is the reserved "=" column
May be `None` if not required.

Example

The following example sets an initial value of 1.42 and tolerance set 2 for column 25 in the matrix.

```
p.chgvar(col=25, initial=1.42, tolset=2,
           vartype=1<<1 | 1<<2)
```

Note that bits 1 and 2 of `vartype` are set, indicating that the variable has a delta vector and an initial value. For columns already defined as SLP variables, use `problem.getvar` to obtain the current value of `vartype` because other bits may already have been set by the system.

Further information

If any of the arguments is `None` then the corresponding information for the variable will be left unaltered. If the information is new (i.e. the column was not previously defined as an SLP variable) then the default values will be used.

Changing `Value`, `History` or `Converged` is only effective during SLP iterations.

Changing `initvalue` and `initstepbound` is only effective before `problem.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.write("augment","l")
```

The "l" flag causes output of the current linear problem (which is now the augmented structure and the current linearization) rather than the original nonlinear problem.

Further information

`construct` adds new rows and columns to the SLP matrix and calculates initial values for the non-linear coefficients. Which rows and columns are added will depend on the setting of `xslp_augmentation`. Names for the new rows and columns are generated automatically, based on the existing names and the string control variables `xslp_xxxformat`.

Once `construct` has been called, no new rows, columns or non-linear coefficients can be added to the problem. Any rows or columns which will be required must be added first. Non-linear coefficients must not be changed; constant matrix elements can generally be changed after `construct`, but not after `problem.presolve` if used.

`construct` is called automatically by the SLP optimization procedure, and so only needs to be called explicitly if changes need to be made between the augmentation and the optimization.

Related topics

`problem.presolve`

problem.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.crossoverlpsol

Purpose

Provides a basic optimal solution for a given solution of an LP problem. This function behaves like the crossover after the barrier algorithm.

Synopsis

```
status = problem.crossoverlpsol()
```

Argument

`status` One of:

- `0` The crossover was successful.
- `1` The crossover was not performed because the problem has no solution.

Example

This example loads a problem, loads a solution for the problem and then uses `crossoverlpsol` to find a basic optimal solution.

```
p = xp.problem()
p.read('problem.mps')
status = p.loadlpsol(x, None, dual, None)
status = p.crossoverlpsol()
```

A solution can also be loaded from an ASCII solution file using `problem.readslxsol`.

Further information

1. The crossover performs two phases: a crossover phase for finding a basic solution and a clean-up phase for finding a basic optimal solution. Setting `algaftercrossover` to `0` will allow the crossover to skip the clean-up phase.
2. The given solution is expected to be feasible or nearly feasible, otherwise the crossover may take a long time to find a basic feasible solution. More importantly, the given solution is expected to have a small duality gap. A small duality gap indicates that the given solution is close to the optimal solution. If the given solution is far away from the optimal solution, the clean-up phase may need many simplex iterations to move to a basic optimal solution.

Related topics

`problem.loadlpsol`, `problem.readslxsol`

problem.delcoefs

Purpose

Delete coefficients from the current problem

Synopsis

```
problem.delcoefs(rowind, colind)
```

Arguments

`rowind`	rows (i.e. `xpress.constraint` objects, indices, or names) of the SLP coefficients to delete.
`colind`	columns (i.e. `xpress.var` objects, indices, or names) of the SLP coefficients to delete.

Related topics

`problem.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 constraints. 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(cuttype, interp, cutind)
```

Arguments

`cuttype` User defined cut type to match against.

`interp` Way in which the cut `cuttype` is interpreted:

- `-1` match all cut types;
- `1` treat cut types as numbers;
- `2` treat cut types as bit maps - delete if any bit matches any bit set in `cuttype`;
- `3` treat cut types as bit maps - delete if all bits match those set in `cuttype`.

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

Related topics

`problem.addcuts`, `problem.delcuts`, `problem.loadcuts`, 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(basis, cuttype, interp, delta, cutind)
```

Arguments

`basis` Ensures the basis will be valid if set to `1`. If set to `0`, cuts with non-basic slacks may be deleted.

`cuttype` User defined type of the cut to be deleted.

`interp` Way in which the cut `cuttype` is interpreted:

- `-1` match all cut types;
- `1` treat cut types as numbers;
- `2` treat cut types as bit maps - delete if any bit matches any bit set in `cuttype`;
- `3` treat cut types as bit maps - delete if all bits match those set in `cuttype`.

`delta` Only delete cuts with an absolute slack value greater than `delta`. To delete all the cuts, this argument should be set to `-xpress.infinity`.

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

Further information

1. It is usually best to drop only those cuts with basic slacks, otherwise the basis will no longer be valid and it may take many iterations to recover an optimal basis. If the `basis` parameter is set to `1`, this will ensure that cuts with non-basic slacks will not be deleted even if the other parameters specify that these cuts should be deleted. It is highly recommended that the `basis` parameter is always set to `1`.
2. The cuts to be deleted can also be specified by the size of the slack variable for the cut. Only those cuts with a slack value greater than the `delta` parameter will be deleted.
3. A list of indices of the cuts to be deleted can also be provided. The list of active cuts at a node can be obtained with the `problem.getcutlist` function.

Related topics

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

problem.delgencons

Purpose

Delete general constraints from a problem.

Synopsis

```
problem.delgencons(conind)
```

Argument

`conind` An integer array containing the general constraints to delete.

Example

In this example, general constraints `0` and `2` are deleted from the problem:

```
conind = [0, 2]
prob.delgencons(conind)
```

Further information

After general constraints have been deleted from a problem, the indices of the remaining constraints are reduced down so that the general constraints are always numbered from `0` to `prob.attributes.gencons - 1` where `prob.attributes.gencons` contains the number of non-deleted general constraints in the problem.

Related topics

`problem.addgencons`, `problem.getgencons`, `xpress.And`, `xpress.Or`, `xpress.max`, `xpress.min`, `xpress.abs`.

problem.delindicators

Purpose

Delete indicator constraints. This turns the specified rows into normal rows (not controlled by indicator variables).

Synopsis

```
problem.delindicators(first=None, last=None);
```

Arguments

`first`	First row in the range.
`last`	Last row in the range (inclusive).

Example

In this example, if any of the first two rows of the matrix is an indicator constraint, they are turned into normal rows:

```
prob.delindicators(0,1)
```

Further information

This function has no effect on rows that are not indicator constraints.

Related topics

`problem.getindicators`, `problem.setindicators`.

problem.delpwlcons

Purpose

Delete piecewise linear constraints from a problem.

Synopsis

```
problem.delpwlcons(pwlind)
```

Argument

`pwlind` An integer array containing the piecewise linear constraints to delete.

Example

In this example, piecewise linear constraints `0` and `2` are deleted from the problem:

```
pwlind = [0,2]
prob.delpwlcons(pwlind)
```

Further information

After piecewise linear constraints have been deleted from a problem, the indices of the remaining constraints are reduced so that the piecewise linear constraints are always numbered from `0` to `problem.attributes.pwlcons - 1` where `problem.attributes.pwlcons` is the problem attribute containing the number of non-deleted piecewise linear constraints in the problem.

Related topics

`problem.addpwlcons`, `problem.getpwlcons`, `xpress.pwl`.

problem.delobj

Purpose

Removes an objective function from a multi-objective problem. Any objectives with `index > objidx` will be shifted down. Deleting the last objective function in the problem causes all the objective coefficients to be zeroed, but `OBJECTIVES` remains set to `1`.

Synopsis

```
problem.delobj(objidx)
```

Argument

`objidx` Index of the objective to remove.

Example

Removing the second objective function from a problem:

```
p.delobj(1)
```

Related topics

`problem.addObjective`, `problem.setObjective`, `problem.chgobjn`, `problem.addobj`, `problem.getobjn`.

problem.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(tolind)
```

Argument

`tolind` 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(colind)
```

Argument

`colind` Columns to be converted to linear ones.

Further information

The SLP variables to be converted to linear, non SLP columns must not be in use by any other SLP structure (coefficients, initial value formulae, delayed columns). Use the appropriate deletion or change functions to remove them first.

Related topics

`problem.addvars`, `problem.chgvar`, `problem.getvar`, `problem.loadvars`

problem.dumpcontrols

Purpose

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(rowind, iterlim, mindual, maxdual)
```

Arguments

`rowind`	rows (i.e. `xpress.constraint` objects, indices, or names) to analyze.
`iterlim`	Effort limit expressed as simplex iterations per row.
`mindual`	Estimated lower bounds on the possible dual ranges.
`maxdual`	Estimated upper bounds on the possible dual ranges.

Further information

This function may provide better results for individual row dual ranges when called for a larger number of rows.

Related topics

`problem.lpoptimize`, `problem.strongbranch`

problem.evaluatecoef

Purpose

Evaluate a coefficient using the current values of the variables

Synopsis

```
value = problem.evaluatecoef(row, col)
```

Arguments

`row`	Row (i.e. `xpress.constraint` object, index, or name).
`col`	Column (i.e. `xpress.var` object, index, or name).
`value`	The result of the calculation.

Example

The following example sets the value of column 5 to 1.42 and then calculates the coefficient in row 2, column 3. If the coefficient depends on column 5, then a value of 1.42 will be used in the calculation.

```
p.chgvar(col=5, value=1.42)
value = p.evaluatecoef(2, 3)
```

Further information

The values of the variables are obtained from the solution, or from the `Value` setting of an SLP variable (see `problem.chgvar` and `problem.getvar`).

Related topics

`problem.chgvar`, `problem.evaluateformula`, `problem.getvar`

problem.evaluateformula

Purpose

Evaluate a formula using the current values of the variables

Synopsis

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

Arguments

`parsed`	integer indicating whether the formula of the item is in internal unparsed format (`Parsed`=0) or parsed (reverse Polish) format (`Parsed`=1).
`type`	Integer array of token types for the formula.
`values`	Array of values corresponding to `Type`.
`result`	The result of the calculation.

Example

The following example calculates the value of column 3 divided by column 6.

```
type   = [xp.xslp_op_var, xp.xslp_op_var, xp.xslp_op_op,       xp.xslp_op_eof]
values = [3,              6,              xp.xslp_ifun_divide, 0]

result  = p.evaluateformula(1, type, values)
```

Further information

The formula in `type` and `values` must be terminated by an `xslp_op_eof` token.

The formula cannot include "complicated" functions, such as user functions which return more than one value.

Related topics

`problem.evaluatecoef`

problem.fixmipentities

Purpose

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

Synopsis

```
problem.fixmipentities(options)
```

Argument

`options` Options for fixing the MIP entities, evaluated as a bit string whose bits have the following meaning:

Bit	Meaning
`0`	If all MIP entities should be rounded to the nearest discrete value in the solution before being fixed.
`1`	If piecewise linear and general constraints should be kept in the problem with only the non-convex decisions (i.e. which part of a non-convex piecewise linear function or which variable attains a maximum) fixed. Otherwise all variables appearing in piecewise linear or general constraints will be fixed.

Example

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

```
p.read("myprob", "")
p.mipoptimize()
p.fixmipentities(1)
p.lpoptimize()
p.writeprtsol()
```

Further information

1. Because of tolerances, it is possible for e.g. a binary variable to be slightly fractional in the MIP solution, where it might have the value `0.999999` instead of being at exactly `1.0`. With `ifround = 0`, such a binary will be fixed at `0.999999`, but with `ifround = 1`, it will be fixed at `1.0`.
2. This command is useful for inspecting the reduced costs of the continuous variables in a problem after the MIP entities have been fixed. Sensitivity analysis can also be performed on the continuous variables in a MIP problem using `problem.rhssa` or `problem.objsa` after calling `fixmipentities`.

Related topics

`problem.mipoptimize`.

problem.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 `problem.attributes.rows` containing the values which are to be multiplied by the basis inverse. The transformed values appear in the array.

Example

To get the (unscaled) tableau column of structural variable number `jcol`, assuming that all arrays have been dimensioned, do the following:

```
y = [0,1,0,0]
p.ftran(y)
print("ftran result:", y)
```

Further information

If the problem is in a presolved state, the function will work with the basis for the presolved problem.

Related topics

`problem.btran`.

problem.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. Attributes can also be specified by id. In that case the keys for those attributes in a returned dictionary will be their ids.

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(rowstat, colstat)
```

Arguments

`rowstat` Array of length `problem.attributes.rows` to the basis status of the slack, surplus or artificial variable associated with each row. The status will be one of:

- `0` slack, surplus or artificial is non-basic at lower bound;
- `1` slack, surplus or artificial is basic;
- `2` slack or surplus is non-basic at upper bound.
- `3` slack or surplus is super-basic.

May be `None` if not required.

`colstat` Array of length `problem.attributes.cols` to hold the basis status of the columns in the constraint matrix. The status will be one of:

- `0` variable is non-basic at lower bound, or superbasic at zero if the variable has no lower bound;
- `1` variable is basic;
- `2` variable is non-basic at upper bound;
- `3` variable is super-basic.

May be `None` if not required.

Example

The following example minimizes a problem before saving the basis for later:

```
rstatus = []
cstatus = []
p.lpoptimize()
p.getbasis(rstatus, cstatus)
```

Related topics

`problem.getpresolvebasis`, `problem.loadbasis`, `problem.loadpresolvebasis`.

problem.getbasisval

Purpose

Returns the current basis status for a specific col or row.

Synopsis

```
rstatus, cstatus = problem.getbasisval(row=None, col=None)
```

Arguments

`row`	Row index to get the row basis status for.
`col`	Column index to get the col basis status for.
`rstatus`	The row basis status will be returned, or 0 if `row` was passed as `None`.
`cstatus`	The value of the col basis status, or 0 if `col==None`.

Related topics

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

problem.getccoef

Purpose

Retrieve a single nonlinear matrix coefficient as a formula in a string.

Synopsis

```
(factor, formula) = problem.getccoef(row, col, maxbytes)
```

Arguments

`row`	Integer holding the row index for the coefficient.
`col`	Integer holding the column index for the coefficient.
`maxbytes`	Maximum length of returned formula.

Return value

`factor`	The value of the constant factor multiplying the formula in the coefficient.
`formula`	String containing the formula, in the same format as used for input from a file.

Example

The following example displays the formula for the coefficient in row 2, column 3:

```
(factor, formula) = p.getccoef(2, 3, 60)
```

Further information

If the requested coefficient is constant, then `factor` will be set to 1.0 and the value will be formatted in `formula`.

If the length of the formula would exceed `maxbytes - 1`, the formula is truncated to the last token that will fit.

Related topics

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

problem.getcoef

Purpose

Returns a single coefficient in the constraint matrix.

Synopsis

```
coef = problem.getcoef(row, col)
```

Arguments

`row`	Row of the constraint matrix.
`col`	Column of the constraint matrix.

Further information

It is quite inefficient to get several coefficients with the `getcoef` function. It is better to use `getcols` or `getrows`.

Related topics

`problem.getcols`, `problem.getrows`.

problem.getcoefformula

Purpose

Retrieve a single nonlinear matrix coefficient as a formula split into tokens

Synopsis

```
(factor, tokencount, type, value) = problem.getcoefformula(row, col, parsed,
      maxtypes)
```

Arguments

`row`	The row index for the coefficient.
`col`	The column index for the coefficient.
`parsed`	Integer indicating whether the formula of the item is to be returned in internal unparsed format (`Parsed`=0) or parsed (reverse Polish) format (`Parsed`=1).
`maxtypes`	Maximum number of tokens to return, i.e. length of the `type` and `value` arrays.

Return value

`factor`	The value of the constant factor multiplying the formula in the coefficient.
`tokencount`	Number of tokens returned in `type` and `value`.
`type`	Array holding the token types for the formula.
`value`	Array of values corresponding to `type`.

Example

The following example displays the formula for the coefficient in row 2, column 3 in unparsed form:

```
(fac, tc, type, value) = p.getcoefformula(2, 3, 0, 10)
```

Further information

The `type` and `value` arrays are terminated by an `xslp_op_eof` token.

If the requested coefficient is constant, then `factor` will be set to 1.0 and the value will be returned with token type `xslp_op_con`.

Related topics

`problem.chgccoef`, `problem.chgnlcoef`, `problem.getccoef`

problem.getcoefs

Purpose

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

Synopsis

```
problem.getcoefs(rowind, colind)
```

Arguments

`rowind`	Row positions of the coefficients. May be `None` if not required.
`colind`	Column positions of the coefficients. May be `None` if not required.

Related topics

`problem.getccoef`, `problem.getcoefformula`

problem.getcolinfo

Purpose

Get current column information.

Synopsis

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

Arguments

`infotype`	Type of information (see below).
`colindex`	Column (i.e. `xpress.var` object, index, or name) whose information is to be handled.

Further information

If the data is not available, the type of the returned Info is set to `None`.

The following constants are provided for column information handling:

`xpress.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 (start, rowind, rowcoef, maxcoefs, first, last)
```

Arguments

`start`	Array which will be filled with the indices indicating the starting offsets in the `rowind` and `rowcoef` arrays for each requested column. It must be of length at least `last-first+2`. Column `i` starts at position `start[i]` in the `rowind` and `rowcoef` arrays, and has `start[i+1]-start[i]` elements in it. May be `None` if not required, but must be specified.
`rowind`	Array of length `maxcoefs` which will be filled with the rows of the nonzero coefficents for each column. May be `None` if not required, but must be specified.
`rowcoef`	Array of length `maxcoefs` which will be filled with the nonzero coefficient values. May be `None` if not required, but must be specified.
`maxcoefs`	The size of the `rowind` and `rowcoef` arrays. This is the maximum number of nonzero coefficients that the Optimizer is allowed to return.
`first`	First column in the range.
`last`	Last column in the range.

Example

The following examples retrieves the `start` vector of the problem:

```
p = xpress.problem()
p.read("example", "l")
start = []
p.getcols(start, rowind=None, rowcoef=None, maxcoefs=100, first=0, last=p.attr
```

Further information

It is possible to obtain just the number of elements in the range of columns by replacing `start`, `rowind` and `rowcoef` by `None`, as in the example. In this case, `maxcoefs` must be set to `0` to indicate that the length of arrays passed is zero. This is demonstrated in the example above.

Related topics

`problem.getrows`.

problem.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. Can also be used to retrieve objective controls.

Synopsis

```
c = problem.getControl(ctrl1, ctrl2, ..., objidx=None)
```

Arguments

`ctrl1,ctrl2,...` Names or numeric ids of the controls whose values to retrieve. If the `objidx` argument is provided, the control must be one of the following objective controls:

- `priority` the priority of the objective
- `weight` the weight of the objective
- `reltol` the relative tolerance of the objective
- `abstol` the absolute tolerance of the objective
- `rhs` the constant part of the objective

`objidx` (optional) Index of the objective whose control to retrieve.

Example

```
p = xpress.problem()
[...]
print("tolerance for feasibility and optimality: ",
  p.getControl('feastol'), p.getControl('miprelstop'))
all_ctrls = p.getControl()
ctrl_subset = p.getControl(['presolve', 'miprelstop', 'feastol'])
```

Further information

This function can be passed either a single control name, whose value will be returned, or a list of control names, in which case the return value is a dictionary associating each key in the list with its value. If no argument is provided, a dictionary containing all controls of the problem will be returned. Controls can also be specified by id. In that case the keys for those controls in a returned dictionary will be their ids.

Related topics

`problem.setControl`.

problem.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(cuttype, interp, delta, maxcuts, cutind, viol)
```

Arguments

`cuttype`	The user defined type of the cuts to be returned.
`interp`	Way in which the cut type is interpreted: `-1` get all cuts; `1` treat cut types as numbers; `2` treat cut types as bit maps - get cut if any bit matches any bit set in `cuttype`; `3` treat cut types as bit maps - get cut if all bits match those set in `cuttype`.
`delta`	Only those cuts with a signed violation greater than delta will be returned.
`maxcuts`	Maximum number of cuts to be returned.
`cutind`	Array of length `maxcuts` where the cuts will be returned.
`viol`	Array of length `maxcuts` where the values of the signed violations of the cuts will be returned.

Further information

1. The violated cuts can be obtained by setting the `delta` parameter to the maxcuts of the (signed) violation required. If unviolated cuts are required as well, `delta` may be set to `_MINUSINFINITY` which is defined in the library header file.
2. If the number of active cuts is greater than `maxcuts`, only `maxcuts` cuts will be returned. Otherwise only the existing cuts will be used to fill in the positions of `cutind`.
3. In case of a cut of type 'L', the violation equals the negative of the slack associated with the row of the cut. In case of a cut of type 'G', the violation equals the slack associated with the row of the cut. For cuts of type 'E', the violation equals the absolute value of the slack.
4. Please note that the violations returned are absolute violations, while feasibility is checked by the Optimizer in the scaled problem.

Related topics

`problem.delcpcuts`, `problem.getcpcuts`, `problem.getcutlist`, `problem.loadcuts`, `problem.getcutmap`, `problem.getcutslack`, 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 `colind` and `cutcoef` parameters. The columns and elements will be stored contiguously and the starting point of each cut will be returned in the region pointed to by the `start` parameter.

Synopsis

```
problem.getcpcuts(rowind, maxcoefs, cuttype, rowtype, start, colind,
      cutcoef, rhs)
```

Arguments

`rowind`	List containing the cuts.
`maxcoefs`	Maximum number of columns of the cuts to be returned.
`cuttype`	List where the cut types will be returned.
`rowtype`	Character list where the sense of the cuts (`L`, `G`, or `E`) will be returned.
`start`	Array containing the offsets into the `colind` and `cutcoef` arrays. The last element indicates the total number of elements.
`colind`	Array where the columns of the cuts will be returned.
`cutcoef`	Array where the coefficients will be returned.
`rhs`	Array where the right hand side elements for the cuts will be returned.

Related topics

`problem.getcpcutlist`, `problem.getcutlist`, 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(cuttype, interp, maxcuts, cutind)
```

Arguments

`cuttype` User defined type of the cuts to be returned. A value of `-1` indicates return all active cuts.

`interp` Way in which the cut type is interpreted:

- `-1` get all cuts;
- `1` treat cut types as numbers;
- `2` treat cut types as bit maps - get cut if any bit matches any bit set in `cuttype`;
- `3` treat cut types as bit maps - get cut if all bits match those set in `cuttype`.

`maxcuts` Maximum number of cuts to be retrieved.

`cutind` Array of length `maxcuts` where the cuts will be returned.

Further information

If the number of active cuts is greater than `maxcuts`, then `maxcuts` cuts will be returned. Otherwise only the positions corresponding to the number of active cuts will be filled in `cutind`.

Related topics

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

problem.getcutmap

Purpose

Returns in which rows a list of cutind are currently loaded into the Optimizer. This is useful for example to retrieve the duals associated with active cutind.

Synopsis

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

Arguments

`cutind`	Array with the cutind for which the row index is requested.
`cutmap`	Array where the rows are returned.

Further information

For cutind currently not loaded into the problem, a row index of `-1` is returned.

Related topics

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

problem.getcutslack

Purpose

Used to calculate the slack value of a cutind with respect to the current LP relaxation solution. The slack is calculated from the cutind itself, and might be requested for any cutind (even if it is not currently loaded into the problem).

Synopsis

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

Arguments

`cutind`	Cut object for which the slack is to be calculated.
`slack`	Value of the slack.

Related topics

`problem.getcpcutlist`, `problem.delcpcuts`, `problem.getcutlist`, `problem.loadcuts`, `problem.getcutmap`, `problem.getcpcuts`, Section "Working with the cutind 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(indices, prios, branchdirs, uppseudo, downpseudo)
```

Arguments

`indices` Array containing the column numbers (`0`, `1`, `2`,...) or negative values corresponding to special ordered sets (the first set numbered `-1`, the second numbered `-2`,...). May be `None` if not required.

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

`branchdirs` Character array with the branching direction for each column or set:

- `U` the entity is to be forced up;
- `D` the entity is to be forced down;
- `N` not specified.

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

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

Further information

The size of all lists is at most `MIPENTS`, obtainable from `problem.attributes.mipents`.

Related topics

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

problem.getdf

Purpose

Get a distribution factor

Synopsis

```
value = problem.getdf(col, row)
```

Arguments

`col`	The column (i.e. `xpress.var` object, index, or name) whose distribution factor is to be retrieved.
`row`	The row (i.e. `xpress.constraint` object, index, or name) from which the distribution factor is to be taken.
`value`	The value of the distribution factor.

Example

The following example retrieves the value of the distribution factor for column 282 in row 134 and changes it to be twice as large.

```
value = p.getdf(282,134)
value *= 2
p.chgdf(282,134,calue)
```

Further information

The *distribution factor* of a column in a row is the matrix coefficient of the corresponding delta vector in the row. Distribution factors are used in conventional recursion models, and are essentially normalized first-order derivatives. Xpress SLP can accept distribution factors instead of initial values, provided that the values of the variables involved can all be calculated after optimization using determining rows, or by a callback.

Related topics

`problem.adddfs`, `problem.chgdf`, `problem.loaddfs`

problem.getDual

Purpose

Return the dual for one or more constraints of the problem w.r.t. the solution found by `problem.optimize`; this only works on continuous optimization problems.

Synopsis

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

Arguments

`constraints` (optional) constraint objects whose duals will be returned. If none is provided, a list of duals for all constraints in the problem will be returned.

`d` A list of dual values if `*constraints` contains more than one constraint object, a single dual value otherwise.

Example

```
import xpress as xp
import numpy as np
x = xp.vars(10)
A = np.random.random((5,10))
b = np.random.random(5)
constr = xp.Dot(A,x) >= b
p = xp.problem(x, constr, xp.Sum(x))
p.optimize()
print("Duals of last two constraints:", p.getDual(constr[-2:]))
```

Related topics

`problem.getlpsol`, `problem.getSlack`, `problem.getRCost`.

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(ray)
```

Argument

`ray` Array of length `problem.attributes.rows` to hold the ray. May be `None` if not required.

Example

The following code tries to retrieve a dual ray:

```
if not p.hasdualray():
    print("Could not retrieve a dual ray")
else:
    dray = []
    p.getdualray(dray)
    print("dual ray:", dray)
```

Further information

1. It is possible to retrieve a dual ray only when, after solving an LP problem, the final status is `xpress.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.getgencons

Purpose

Returns the general constraints `y = f(x1, ..., xn, c1, ..., cm)` in a given range.

Synopsis

```
(ncol, nval) = problem.getgencons(contype, resultant, colstart, colind,
      maxcols, valstart, val, maxvals, first, last);
```

Arguments

`contype`	`None` if not required, otherwise a list which will be filled with the types of the general constraints: `xpress.gencons_max (0)` indicates a `maximum` constraint; `xpress.gencons_min (1)` indicates a `minimum` constraint; `xpress.gencons_and (2)` indicates an `and` constraint. `xpress.gencons_or (3)` indicates an `or` constraint; `xpress.gencons_abs (4)` indicates an `absolute value` constraint.
`resultant`	List/array which will be filled with the output variables `y`. May be `None` if not required.
`colstart`	List/array which will be filled with the start index of each general constraint in the `colind` array. May be `None` if not required.
`colind`	Integer array which will be filled with the indices of the input variables `xi`. May be `None` if not required.
`maxcols`	Maximum number of input columns to be retrieved.
`valstart`	Integer array of length at least `last-first+1` which will be filled with the start index of each general constraint in the `val` array. May be `None` if not required.
`val`	Integer array which will be filled with the constant values `ci`. May be `None` if not required.
`maxvals`	Maximum number of constant values to be retrieved.
`first`	First general constraint in the range.
`last`	Last general constraint in the range.
`ncol`	Number of values in the `colind` list if not `None`.
`nval`	Number of values in the `coef` list if not `None`.

Example

The following example retrieves all general constraints:

```
contype, resultant, colstart, colind, valstart, val = [], [], [], [], [], []
prob.getgencons(contype, resultant, colstart, colind, 1e9, valstart, val, 1e9,
```

Further information

It is possible to obtain just the number of input columns and/or constant values in the range of general constraints by calling this function with `maxcols` and `maxvals` set to `0`, in which case the required size for the arrays will be returned as a tuple with `ncols` and `nvals`.

Related topics

`problem.addgencons`, `problem.delgencons`, `xpress.And`, `xpress.Or`, `xpress.max`, `xpress.min`, `xpress.abs`.

problem.getmipentities

Purpose

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

Synopsis

```
problem.getmipentities(coltype, colind, limit, settype, start, setcols,
      refval)
```

Arguments

`coltype` Character array where the entity types will be returned. The types will be one of:
- `B` binary variables;
- `I` integer variables;
- `P` partial integer variables;
- `S` semi-continuous variables;
- `R` semi-continuous integer variables.

`colind` Array where the columns of the MIP entities will be returned.

`limit` Array where the limits for the partial integer variables and lower bounds for the semi-continuous and semi-continuous integer variables will be returned (any entries in the positions corresponding to binary and integer variables will be meaningless).

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

`start` Array where the offsets into the `setcols` and `refval` arrays indicating the start of the sets will be returned. This array must be of length `SETMEMBERS+1`: the final element contains the length of the `setcols` and `refval` arrays.

`setcols` Array of length `problem.attributes.setmembers` where the columns in each set will be returned.

`refval` Array of length `problem.attributes.setmembers` where the reference row entries for each member of the sets will be returned.

Example

The following obtains the SOS information:

```
settype = []
mstart = []
setcols = []
refval   = []
p.getmipentities(None, None, None, settype, mstart, setcols, refval)
```

Further information

All arguments may be `None` if not required.

Related topics

`problem.loadproblem`.

problem.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(iis, rowind, colind, contype, bndtype, duals, djs,
      isolationrows, isolationcols)
```

Arguments

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

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

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

`contype` Sense of rows in the IIS:
- `L` for less or equal row;
- `G` for greater or equal row.
- `E` for an equality row (for a non LP IIS);
- `1` for a SOS1 row;
- `2` for a SOS2 row;
- `I` for an indicator row.

Can be `None` if not required.

`bndtype` Sense of bound in the IIS:
- `U` for upper bound;
- `L` for lower bound.
- `F` for fixed columns (for a non LP IIS);
- `B` for a binary column;
- `I` for an integer column;
- `P` for a partial integer columns;
- `S` for a semi-continuous column;
- `R` for a semi-continuous integer column.

Can be `None` if not required.

`duals` The >dual multipliers associated with the rows. Can be `None` if not required.

`djs` The dual multipliers (reduced costs) associated with the bounds. Can be `None` if not required.

`isolationrows` The isolation status of the rows:
- `-1` if isolation information is not available for row (run iis isolations);
- `0` if row is not in isolation;
- `1` if row is in isolation.

Can be `None` if not required.

`isolationcols` The isolation status of the bounds:
- `-1` if isolation information is not available for column (run iisisolations);
- `0` if column is not in isolation;
- `1` if column is in isolation. Can be `None` if not required.

Example

This example first retrieves the size of IIS 1, then gets the detailed information for the IIS.

```
rowind = []
colind = []
contype = []
bndtype = []
duals = []
djs = []
```

```
isolationrows = []
isolationcols = []
p.getiisdata(1, rowind, colind, contype, bndtype,
             duals, djs, isolationrows, isolationcols)
```

Further information

1. IISs are numbered from 1 to `NUMIIS`. Index number 0 refers to the IIS approximation.
2. If `rowind` and `colind` both are `None`, only the `rownumber` and `colnumber` are returned.
3. The arrays may be `None` if not required. However, arrays `contype`, `duals` and `isolationrows` are only returned if `rowind` is not `None`. Similarly, arrays `bndtype`, `djs` and `isolationcols` are only returned if `colind` is not `None`.
4. For the initial IIS approximation (`iis` = 0) the number of rows and columns with a nonzero Lagrange multiplier (dual/reduced cost respectively) are returned. Please note that in such cases, it might be necessary to call `problem.iisstatus` to retrieve the necessary size of the return arrays.
5. If there are Special Ordered Sets in the IIS, their number is included in the `rowind` array.
6. For non-LP IISs, some column indices may appear more than once in the `colind` array, for example an integrality and a bound restriction for the same column.
7. Duals, reduced cost and isolation information is not available for nonlinear IIS problems, and for those the arrays are filled with zero values in case they are provided.

Related topics

`problem.iisall`, `problem.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(colind, complement, first, last)
```

Arguments

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

`complement` Array of length `last-first+1` where the indicator complement flags will be returned:

- `0` not an indicator constraint (in this case the corresponding entry in the `colind` array is ignored);
- `1` for indicator constraints with condition "`bin = 1`";
- `-1` for indicator constraints with condition "`bin = 0`";

`first` First row in the range.

`last` Last row in the range (inclusive).

Example

The following example retrieves information about three indicator constraints in the problem and prints a list of their indices.

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

Related topics

`problem.setindicators`.

problem.getinfeas

Purpose

Returns a list of infeasible primal and dual variables.

Synopsis

```
problem.getinfeas(x, slack, duals, djs)
```

Arguments

`x`	Array to store the primal infeasible variables. May be `None` if not required.
`slack`	Array to store the primal infeasible rows. May be `None` if not required.
`duals`	Array to store the dual infeasible rows. May be `None` if not required.
`djs`	Array to store the dual infeasible variables. May be `None` if not required.

Example

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

Further information

To find the infeasibilities in a previously saved solution, the solution must first be loaded into memory with the `problem.readbinsol` function.

Related topics

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

problem.getlastbarsol

Purpose

Obtains the last barrier solution values following optimization that used the barrier solver.

Synopsis

```
barsolstatus = problem.getastbarsol(x=None, slack=None, duals=None,
      djs=None);
```

Arguments

`x` Array of length `problem.attributes.cols` where the values of the primal variables will be returned. May be `None` if not required.

`slack` Array of length `problem.attributes.rows` where the values of the slack variables will be returned. May be `None` if not required.

`duals` Array of length `problem.attributes.rows` where the values of the dual variables ($c_B^T B^{-1}$) will be returned. May be `None` if not required.

`djs` Array of length `problem.attributes.cols` where the reduced cost for each variable ($c^T - c_B^T B^{-1} A$) will be returned. May be `None` if not required.

`barsolstatus` Status of the last barrier solve. Value matches that of the `lpstatus` attribute if the solve stopped immediately after the barrier.

Further information

1. If the barrier solver has not been used, `barsolstatus` will return `xpress.lp_unsolved`.
2. The barrier solution or the solution candidate is always available if the status is not `xpress.lp_unsolved`.
3. The last barrier solution is available until the next solve, and is not invalidated by otherwise working with the problem.

Related topics

`problem.getlpsol`

problem.getlasterror

Purpose

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

Synopsis

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

Example 1

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

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

Further information

The problem.getlasterror() function is an API wrapper for the `XPRSgetlasterror()` function in the Xpress C API. For this reason, errors that occur within the Xpress API are reported by getlasterror(). Errors that occur at the level of the Python interface are *not* reported by `getlasterror`. Both classes of errors can be handled with a `try/except` construct. In the two examples below, the first is an error that is detected by the Xpress API and propagated to a Python error, while the second is an incorrect statement for the Python module. They both trigger a RuntimeError exception.

Example 2

```
x = xpress.var()
try:
    p.addVariable(x)
    p.addVariable(x)
except RuntimeError as e:
    print(e)
```

Example 3

```
try:
    p.addVariable('John Cleese')
except RuntimeError as e:
    print(e)
```

Related topics

`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, duals, djs)
```

Arguments

`x`	Array to store the values of the primal variables. May be `None` if not required.
`slack`	Array to store the values of the slack variables. May be `None` if not required.
`duals`	Array to store the values of the dual variables ($c_B^T B^{-1}$). May be `None` if not required.
`djs`	Array to store the reduced cost for each variable ($c^T - c_B^T B^{-1} A$). May be `None` if not required.

Example

The following sequence of function calls will get the LP solution (`x`) at the top node of a MIP and the optimal MIP solution (`y`):

```
p.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 MIP callback the solution of the current node will be returned.
2. When an integer solution is found during a tree search, it is always set up as a solution to the current node; therefore the integer solution is available as the current node 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.getlpsolval

Purpose

Used to obtain a single LP solution value following optimization.

Synopsis

```
x, slack, dual, dj = problem.getlpsolval(col=None, row=None)
```

Arguments

`col`	Column of the variable for which to return the solution value.
`row`	Row of the constraint for which to return the solution value.
`x`	The returned value of the primal variable.
`slack`	The returned value of the slack variable.
`dual`	The returned value of the dual variable ($c_B^T B^{-1}$).
`dj`	The returned reduced cost for the variable ($c^T - c_B^T B^{-1} A$).

Further information

1. This function is currently not supported if the problem is in a presolved state.
2. If `col` or `row` are `None`, the corresponding output is set to `-xpress.infinity`.

Related topics

`problem.getlpsol`, `problem.getpresolvesol`, `problem.getmipsol`, `problem.writeprtsol`, `problem.writesol`.

problem.getmessagestatus

Purpose

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

Synopsis

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

Argument

`msgcode` The id number of the message. Refer to the Xpress Optimizer reference manual for a list of possible message numbers.

Further information

If a message is suppressed globally then the message will always have `status` return zero from `getmessagestatus`.

Related topics

`problem.setmessagestatus`.

problem.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 function calls 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.getmipsolval

Purpose

Used to obtain a single solution value of the last MIP solution that was found.

Synopsis

```
x, slack = problem.getmipsolval(col=None, row=None)
```

Arguments

`col`	Column index of the variable for which to return the solution value. May be `None`.
`row`	Row index of the constraint for which to return the solution value. May be `None`.
`x`	The returned value of the primal variable, or `-xpress.infinity` if `col` is `None`.
`slack`	The returned value of the slack variable, or `-xpress.infinity` if `row` is `None`.

Related topics

`problem.getmipsol`, `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(start, colind, objqcoef, maxcoefs, first, last)
```

Arguments

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

Further information

The objective function is of the form $c^T x + 0.5 x^T Q x$ where `Q` is positive semi-definite for minimization problems and negative semi-definite for maximization problems. If this is not the case the optimization algorithms may converge to a local optimum or may not converge at all. Note that only the upper or lower triangular part of the `Q` matrix is returned.

Related topics

`problem.chgmqobj`, `problem.chgqobj`, `problem.getqobj`.

problem.getobjn

Purpose

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

Synopsis

```
problem.getobjn(objidx, objcoef, first, last)
```

Arguments

`objidx`	Index of the objective function whose coefficients to return.
`objcoef`	Array of length `last-first+1` where the objective function coefficients are to be placed.
`first`	First column in the range.
`last`	Last column in the range.

Example

The following example retrieves the coefficients of the first five variables in the second objective function:

```
objcoef = []
p.getobjn(1, objcoef, 0, 4)
```

Related topics

`problem.getobj`.

problem.getnamelist

Purpose

Returns the names for the rows, columns or sets in a given range. The names will be returned in a list of Python strings.

Synopsis

```
names = problem.getnamelist(type, first, last);
```

Arguments

`type`	`1`	if row names are required;
	`2`	if column names are required.
	`3`	if set names are required.
`names`	A list containing all returned names.	
`first`	First row, column or set in the range. If `None`, it is taken as zero.	
`last`	Last row, column or set in the range. If `None`, it is taken as the penultimate element in the list defined by `type`.	

Example

The following example retrieves and outputs the row and column names for the current problem.

```
cols = prob.attributes.cols
rows = prob.attributes.rows

rnames = prob.getnamelist(1, 0, rows - 1)
cnames = prob.getnamelist(2, 0, cols - 1)

for k,v in enumerate(rnames):
    print("Row {0:4d}: {1}", k, v)

for k,v in enumerate(cnames):
    print("Column {0:4d}: {1}", k, v)
```

problem.getobj

Purpose

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

Synopsis

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

Arguments

`objcoef`	Array of length `last-first+1` where the objective function coefficients are to be placed.
`first`	First column in the range.
`last`	Last column in the range.

Example

The following example retrieves the objective function coefficients of the first five variables of the current problem:

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

Related topics

`problem.chgobj`.

problem.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 `problem.optimize` function is called:

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

Related topics

`problem.optimize`.

problem.getpivotorder

Purpose

Returns the pivot order of the basic variables.

Synopsis

```
problem.getpivotorder(pivotorder)
```

Argument

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

Example

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

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

Further information

Row indices are in the range `0` to `ROWS-1`, whilst columns are in the range `ROWS+SPAREROWS` to `ROWS+SPAREROWS+COLS-1`.

Related topics

`problem.getpivots`.

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

Synopsis

```
dobj, npiv = problem.getpivots(enter, outlist, x, maxpivots)
```

Arguments

`enter`	Index of the specified row or column to enter basis.
`outlist`	Array of length at least `maxpivots` to hold list of potential leaving variables. May be `None` if not required.
`x`	Array of length `problem.attributes.rows + problem.attributes.sparerows + problem.attributes.cols` to hold the values of all the variables that would result if `enter` entered the basis. May be `None` if not required.
`maxpivots`	Maximum number of potential leaving variables to return.

Example

The following retrieves a list of up to 5 potential leaving variables if variable 6 enters the basis:

```
outlist = []
x = []
obj, npiv = p.getpivots(2, outlist, x, 10)
```

Further information

1. If the variable `enter` enters the basis and the problem is degenerate then several basic variables are candidates for leaving the basis, and the number of potential candidates is returned enter `npiv`. A list of at most `maxpivots` of these candidates is returned enter `outlist` which must be at least `maxpivots` long. If variable `enter` were to be pivoted enter, then because the problem is degenerate, the resulting values of the objective function and all the variables do not depend on which of the candidates from `outlist` is chosen to leave the basis. The value of the objective is returned enter `dobj` and the values of the variables into `x`.
2. Row indices are enter the range `0` to `ROWS-1`, whilst columns are enter the range `ROWS+SPAREROWS` to `ROWS+SPAREROWS+COLS-1`.

Related topics

`problem.getpivotorder`.

problem.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 `problem.attributes.rows` to the basis status of the stack, surplus or artificial variable associated with each row. The status will be one of:

- `0` slack, surplus or artificial is non-basic at lower bound;
- `1` slack, surplus or artificial is basic;
- `2` slack or surplus is non-basic at upper bound.

May be `None` if not required.

`cstatus` Array of length `problem.attributes.cols` to hold the basis status of the columns in the constraint matrix. The status will be one of:

- `0` variable is non-basic at lower bound, or superbasic at zero if the variable has no lower bound;
- `1` variable is basic;
- `2` variable is at upper bound;
- `3` variable is super-basic.

May be `None` if not required.

Example

The following obtains and outputs basis information on a presolved problem prior to the tree search:

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

Related topics

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

problem.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, duals, djs)
```

Arguments

`x`	Array to store the values of the primal variables. May be `None` if not required.
`slack`	Array to store the values of the slack variables. May be `None` if not required.
`duals`	Array to store the values of the dual variables. May be `None` if not required.
`djs`	Array to store the reduced cost for each variable. May be `None` if not required.

Example

The following reads in a (Mixed) Integer Programming problem and displays the solution to the presolved problem following optimization of the linear relaxation:

```
p.read("MyProb", "")
p.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 MIP callback the solution of the current node will be returned.
4. When an integer solution is found during tree search, it is always set up as a solution to the current node; therefore the integer solution is available as the current node solution and can be retrieved with `getlpsol` and `problem.getpresolvesol`.

problem.getprimalray

Purpose

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

Synopsis

```
problem.getprimalray(ray)
```

Argument

`ray` Array of length `problem.attributes.cols` to hold the ray. May be `None` if not required.

Example

The following code tries to retrieve a primal ray:

```
if not p.hasprimalray():
    print("Could not retrieve a primal ray")
else:
    ray = []
    p.getprimalray(ray)
    print("primal ray:", ray)
```

Further information

1. It is possible to retrieve a primal ray only when, after solving an LP problem, the final status (`LPSTATUS`) is `xpress.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 call to problem.optimize. This function is deprecated. Instead, problem.attributes.solvestatus and problem.attributes.solstatus should be used.

Synopsis

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

Example

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

Related topics

problem.optimize, problem.getSolution, problem.getDual, problem.getSlack, problem.getRCost, problem.getProbStatusString.

problem.getProbStatusString

Purpose

Returns the string corresponding to the problem status before or after a call to problem.optimize. This function is deprecated. Instead, problem.attributes.solvestatus.name and problem.attributes.solstatus.name should be used.

Synopsis

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

Example

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

Related topics

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

problem.getpwlcons

Purpose

Returns the piecewise linear constraints `y = f(x)` in a given range.

Synopsis

```
npoints = problem.getpwlcons(colind, resultant, start, xval, yval,
      maxpoints, first, last);
```

Arguments

`colind`	Integer array which will be filled with the indices of the input variables `x`. It must be of length at least `last-first+1`. May be `None` if not required.
`resultant`	Integer array which will be filled with the indices of the output variables `y`. It must be of length at least `last-first+1`. May be `None` if not required.
`start`	Integer array which will be filled with the start indices of the different constraints in the breakpoint arrays. It must be of length at least `last-first+1`. The `x`-values of the breakpoints of piecewise linear constraint `i < last` will be given in `xval[start[i]]` to `xval[start[i+1]]`. May be `None` if not required.
`xval`	Array of length `maxpoints`which will be filled with the `x`-values of the breakpoints. May be `None` if not required.
`yval`	Array of length `maxpoints`which will be filled with the `y`-values of the breakpoints. May be `None` if not required.
`maxpoints`	Maximum number of breakpoints to be retrieved.
`first`	First piecewise linear constraint in the range.
`last`	Last piecewise linear constraint in the range.
`npoints`	The returned number of breakpoints in the `xval` and `yval` arrays. If the number of breakpoints is greater than `maxpoints`, then only `maxpoints` elements will be returned.

Example

The following example retrieves all variables and breakpoints in the first two piecewise linear constraints:

```
colind, resultant, start, xval, yval = [], [], [], [], []
npoints = prob.getpwlcons(prob, colind, resultant, start, xval, yval, 1e9, 0, 1
```

Further information

It is possible to obtain just the number of breakpoints in the range of piecewise linear constraints by calling this function with `maxpoints` set to `0`, in which case the required maxpoints for the breakpoint arrays will be returned in `npoints`.

Related topics

`problem.addpwlcons`, `problem.delpwlcons`, `xpress.pwl`.

problem.getqobj

Purpose

Returns a single quadratic objective function coefficient corresponding to the variable pair `(objqcol1, objqcol2)` of the Hessian matrix.

Synopsis

```
objqcoef = problem.getqobj(objqcol1, objqcol2)
```

Arguments

`objqcol1`	Column index for the first variable in the quadratic term.
`objqcol2`	Column index for the second variable in the quadratic term.

Example

The following returns the coefficient of the x_0^2 term in the objective function, placing it in the variable `value`:

```
print("diagonal coeff of the Hessian:",
  [p.getqobj(i,i) for i in range(p.attributes.cols)])
```

Further information

For example, if the objective function has the term $[3x_1x_2 + 3x_2x_1]/2$ the value retrieved by `getqobj` is `3.0` and if the objective function has the term $[6x_1^2]/2$ the value retrieved by `getqobj` is `6.0`.

Related topics

`problem.getmqobj`, `problem.chgqobj`, `problem.chgmqobj`.

problem.getqrowcoeff

Purpose

Returns a single quadratic constraint coefficient corresponding to the variable pair (`rowqcol1`, `rowqcol2`) of the Hessian of a given constraint.

Synopsis

```
coeff = problem.getqrowcoeff (row, rowqcol1, rowqcol2)
```

Arguments

`row`	The quadratic row where the coefficient is to be looked up.
`rowqcol1`	Column index for the first variable in the quadratic term.
`rowqcol2`	Column index for the second variable in the quadratic term.

Example

The following returns the coefficient of the $dist^2$ term in the constraint cons1:

```
print("diagonal coeff of dist:", p.getqrowcoeff(cons1, dist, dist)
```

Further information

The coefficient returned corresponds to the Hessian of the constraint. That means the for constraint `x + [x^2 + 6 xy] <= 10` `getqrowcoeff` would return `1` as the coefficient of x^2 and `3` as the coefficient of `xy`.

Related topics

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

problem.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(row, start, colind, rowqcoef, maxcoefs, first, last)
```

Arguments

`row`	Row (i.e. `xpress.constraint` object, index, or name) for which the quadratic coefficients are to be returned.
`start`	List to be filled with indices indicating the starting offsets in the `colind` and `dobjval` lists for each requested column. It must be length of at least `last-first+2`. Column `i` starts at position `start[i]` in the `mrwind` and `dmatval` arrays, and has `start[i+1]-start[i]` elements in it. May be `None` if maxcoefs is 0.
`colind`	Array of length `maxcoefs` which will be filled with the columns of the nonzero elements in the lower triangular part of Q. May be `None` if maxcoefs is 0.
`rowqcoef`	Array of length `maxcoefs` which will be filled with the nonzero element values. May be `None` if maxcoefs is 0.
`maxcoefs`	Maximum number of elements to be returned in `colind` and `rowqcoef`.
`first`	First column in the range.
`last`	Last column in the range.

Related topics

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

problem.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(row, rowqcol1, rowqcol2, rowqcoef)
```

Arguments

`row`	Row (i.e. `xpress.constraint` object, index, or name) for which the quadratic coefficients are to be returned.
`rowqcol1`	First index in the triplets. May be `None` if not required.
`rowqcol2`	Second index in the triplets. May be `None` if not required.
`rowqcoef`	Coefficients in the triplets. May be `None` if not required.

Further information

If a row index of `-1` is used, the function returns the quadratic coefficients for the objective function.

Related topics

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

problem.getqrows

Purpose

Returns a list of row objects that have quadratic coefficients.

Synopsis

```
problem.getqrows(rowind)
```

Argument

`rowind`	Array to contain the indices of rows with quadratic coefficients in them. May be `None` if not required.

Related topics

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

problem.getRCost

Purpose

Return the reduced cost of one or more variables of the problem w.r.t. the solution found by `problem.optimize`. This function only works on continuous optimization problems.

Synopsis

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

Arguments

`variables` (optional) variable objects whose reduced costs will be returned. If none is provided, a list of reduced costs of all variables in the problem will be returned.

`r` A list of reduced cost values if `*variables` contains more than one variable object, a single reduced cost value otherwise.

Example

```
import xpress as xp
import numpy as np
x = xp.vars(10, name='y')  # creates 10 variables named 'y(0)', 'y(1)', etc.
A = np.random.random((5,10))
b = np.random.random(5)
constr = xp.Dot(A,x) >= b
p = xp.problem(x, constr, xp.Sum(x))
p.optimize()
print("Reduced costs of first two variables:", p.getRCost(x[:2]))
print("Reduced costs of last two variables:", p.getRCost('y(8)', 'y(9)'))
```

Related topics

`problem.optimize`, `problem.getlpsol`, `problem.getSolution`, `problem.getDual`, `problem.getSlack`.

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`	Row (i.e. `xpress.constraint` object, index, or name) whose information is to be handled
`info`	Information to be retrieved

Further information

If the data is not available, the type of the returned `info` is set to `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(start, colind, colcoef, maxcoefs, first, last)
```

Arguments

`start`	Array which will be filled with the indices indicating the starting offsets in the `colind` and `colcoef` arrays for each requested row. It must be of length at least `last-first+2`. Column `i` starts at position `start[i]` in the `mrwind` and `colcoef` arrays, and has `start[i+1]-start[i]` elements in it. May be `None` if not required.
`colind`	Arrays which will be filled with at most `maxcoefs` column of the nonzero elements for each row. May be `None` if not required.
`colcoef`	Array which will be filled with at most `maxcoefs` nonzero element values. May be `None` if not required.
`maxcoefs`	Maximum number of elements to be retrieved.
`first`	First row in the range.
`last`	Last row in the range.

Related topics

`problem.getcols`, `problem.getrowtype`.

problem.getrowstatus

Purpose

Retrieve the status setting of a constraint

Synopsis

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

Arguments

`row`	The index of the matrix row whose data is to be obtained.
`status`	The status settings.

Example

This recovers the status of the rows of the matrix of the current problem and reports those which are flagged as enforced constraints.

```
m = p.getintattrib('rows')
for i in range(m):
  status = p.getrowstatus(i)
  if(Status & 0x800) print("Row {0} is enforced".format(i))
```

Further information

See the section on bitmap settings of the XSLP reference manual for details on the possible information in `Status`.

Related topics

`problem.chgrowstatus`

problem.getrowtype

Purpose

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

Synopsis

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

Arguments

`rowtype`	Character array of length `last-first+1` characters where the row types will be returned: `N` indicates a free constraint; `L` indicates a $\leq$ 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 four 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(row)
```

Arguments

`row`	The row (i.e. `xpress.constraint` object, index, or name) whose weight is to be retrieved.
`value`	The value of the weight.

Example

The following example gets the initial weight of row number 2.

```
value = p.getrowwt(2)
```

Further information

The initial row weight is used only when the augmented structure is created. After that, the current weighting can be accessed using `problem.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(x, slack, duals, djs)
```

Arguments

`x`	Array to store the primal infeasible variables. May be `None` if not required.
`slack`	Array to store the primal infeasible rows. May be `None` if not required.
`duals`	Array to store the dual infeasible rows. May be `None` if not required.
`djs`	Array to store the dual infeasible variables. May be `None` if not required.

Example

```
x = []
slack = []
duals = []
djs = []
p.getscaledinfeas(x, slack, duals, djs)
```

Related topics

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

problem.getSlack

Purpose

Return the slack for one or more constraints of the problem w.r.t. the solution found by `problem.optimize`. This function works both with continuous and mixed-integer optimization problems.

Synopsis

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

Arguments

`constraints` (optional) constraint objects whose slacks will be returned. If none is provided, a list of slacks for all constraints in the problem will be returned.

`s` A list of slack values if `*constraints` contains more than one constraint object, a single slack value otherwise.

Example

```
import xpress as xp
import numpy as np
x = xp.vars(10)
A = np.random.random((5,10))
b = np.random.random(5)
constr = xp.Dot(A,x) >= b
p = xp.problem(x, constr, xp.Sum(x))
p.optimize()
print("slack of 2nd and 3rd constraint:", p.getSlack(constr[1], constr[2]))
print("slack of first three constraints:", p.getSlack(constr[:3]))
```

Related topics

`problem.optimize`, `problem.getlpsol`, `problem.getmipsol`, `problem.getSolution`, `problem.getDual`, `problem.getRCost`

problem.getslpsol

Purpose

Obtain the solution values for the most recent SLP iteration

Synopsis

```
problem.getslpsol(x, slack, duals, djs)
```

Arguments

`x`	Array of length `problem.attributes.xslp_originalcols` to hold the values of the primal variables. May be `None` if not required.
`slack`	Array of length `problem.attributes.xslp_originalrows` to hold the values of the slack variables. May be `None` if not required.
`duals`	Array of length `problem.attributes.xslp_originalrows` to hold the values of the dual variables. May be `None` if not required.
`djs`	Array of length `problem.attributes.xslp_originalcols` to hold the recuded costs of the primal variables. May be `None` if not required.

Example

The following code fragment recovers the values and reduced costs of the primal variables from the most recent SLP iteration:

```
ncol = p.getintattrib(prob,xpress.xslp_originalcols)
val  = []
djs  = []
p.getslpsol(val,None,None,djs)
```

Further information

`getslpsol` can be called at any time after an SLP iteration has completed, and will return the same values even if the problem is subsequently changed. `getslpsol` returns values for the columns and rows originally in the problem and not for any augmentation rows or columns. To access the values of any augmentation columns or rows, use `getlpsol`; accessing the augmented solution is only recommended if `xslp_presolvelevel` indicates that the problem dimensions should not be changed in presolve.

problem.getSolution

Purpose

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

Synopsis

```
x = problem.getSolution(args=None, flatten=False)
```

Arguments

`args` (optional) specify indices, names, or objects whose solution value is requested. If `None`, it is assumed that all indices of the problem's variables are requested. Starting with version 8.8, `args` can contain expressions, both linear and nonlinear, and dictionaries thereof, in order to allow for more flexible evaluation of functions of the problem solution

`flatten` (optional) allows for backward compatibility with previous versions of the Xpress Python interface. Regardless of whether the passed object is a (nested) list, tuples, the returned value is a flattened list containing all requested values.

Example 1

Below are a few possible uses of the function. Note that one can specify variable names, variable indices, or variable objects, and embed them in lists, dictionaries, NumPy arrays, and tuples.

```
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 v
b = m.getSolution(range(4))         # Gets the value of v1 and x[0], x[1], x[2]
                                    # the first four variables of the problem
c = m.getSolution('Var1')           # Gets the value of v1 by its name
e = m.getSolution({1: x, 2: 0,
                   3: 'Var1'})      # Returns a dictionary containing the same
                                    # in the arguments and the values of the
                                    # variables/expressions passed
d = m.getSolution(v1 + 3*x)         # Gets the value of an expression under the
                                    # current solution
e = m.getSolution(np.array(x))      # Gets a NumPy array with the solution of x

y=xpress.var(name='var1')
x=xpress.var(name='var2')
[...]
p.optimize()
print("solution:",   p.getSolution())
print("x is",        p.getSolution(x))
print("first two var:",   p.getSolution([0,1]))
print("x and y are", p.getSolution(['var1', 'var2']))
```

Example 2

The next examples show how to use the `flatten` argument, which ensures that the returned value is a flattened list.

```
y=xpress.var(name='var1')
x=xpress.var(name='var2')
[...]
p.optimize()
print("x is",           p.getSolution([[x,y],[x,y]], flatten=True)) # will retu
print("first two var:", p.getSolution(0,1, flatten=True))  # will return the li
```

Further information

For efficiency reasons it is preferable that one call to `getSolution` is made, as the whole vector is obtained at each call and only the desired portion is returned.

The function `xpress.evaluate` is more flexible in that it allows more argument types. Apart from the case where the `args` argument contains indices and names of the variables, `getSolution` is equivalent to a call to `xpress.evaluate`.

Related topics

`xpress.evaluate`, `problem.getlpsol`, `problem.getmipsol`, `problem.getDual`, `problem.getSlack`, `problem.getRCost`.

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(tolset, tols)
```

Arguments

`tolset`	The index of the tolerance set.
`status`	The bit-map of status settings.
`tols`	Array of 9 double-precision values to hold the tolerances. May be `None` if not required.

Example

The following example retrieves the values for tolerance set 3 and prints those which are set:

```
tols = []
status = p.gettolset(3, tols)
for i in range(9):
  if status &(1<<i):
    print("Tolerance {0} = {1}".format(i,tols[i]))
```

Further information

If `tols` is `None`, then the corresponding information will not be returned.

If `tols` is not `None`, then a set of 9 values will always be returned. `status` indicates which of these values is active as follows. Bit `n` of `status` is set if `tols[n]` is active, where `n` is:

Entry / Bit	Tolerance	XSLP constant	XSLP bit constant
0	Closure tolerance (TC)	`xslp_TOLSET_TC`	`xslp_TOLSETBIT_TC`
1	Absolute delta tolerance (TA)	`xslp_TOLSET_TA`	`xslp_TOLSETBIT_TA`
2	Relative delta tolerance (RA)	`xslp_TOLSET_RA`	`xslp_TOLSETBIT_RA`
3	Absolute coefficient tolerance (TM)	`xslp_TOLSET_TM`	`xslp_TOLSETBIT_TM`
4	Relative coefficient tolerance (RM)	`xslp_TOLSET_RM`	`xslp_TOLSETBIT_RM`
5	Absolute impact tolerance (TI)	`xslp_TOLSET_TI`	`xslp_TOLSETBIT_TI`
6	Relative impact tolerance (RI)	`xslp_TOLSET_RI`	`xslp_TOLSETBIT_RI`
7	Absolute slack tolerance (TS)	`xslp_TOLSET_TS`	`xslp_TOLSETBIT_TS`
8	Relative slack tolerance (RS)	`xslp_TOLSET_RS`	`xslp_TOLSETBIT_RS`

The xslp_TOLSET constants can be used to access the corresponding entry in the value arrays, while the xslp_TOLSETBIT constants are used to set or retrieve which tolerance values are used for a given SLP variable.

Related topics

Related topics

`problem.addtolsets`, `problem.chgtolset`, `problem.deltolsets`, `problem.loadtolsets`

problem.getub

Purpose

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, initial, value, tolset,
      history, converged, vartype, delta, penaltydelta, updaterow,
      oldvalue) = problem.getvar(col)
```

Arguments

`col` The column (i.e. `xpress.var` object, index, or name).

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

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

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

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

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

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

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

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

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

`converged` An integer to receive the convergence status of the variable as defined in the "Convergence Criteria" section (The returned value will match the numbering of the tolerances). May be `None` if not required.

`vartype` An integer to receive the status settings (a bitmap defining the existence of certain properties for this variable). The following bits are defined:
`Bit 1:` Variable has a delta vector
`Bit 2:` Variable has an initial value
`Bit 14:` Variable is the reserved "=" column
Other bits are reserved for internal use. May be `None` if not required.

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

`penaltydelta` An integer to receive the index of the first penalty delta vector for the variable. The second penalty delta immediately follows the first. May be `None` if not required.

`updaterow` An integer to receive the index of the update row for the variable. May be `None` if not required.

`oldvalue` A double precision variable to receive the value of the variable at the previous SLP iteration. May be `None` if not required.

Example

The following example retrieves the current value, convergence history and status for column 3.

```
(a,b,c,d,e,value,g,history,converged,j,k,i,h,k,l) = p.getvar(3)
```

Further information

If `col` refers to a column which is not an SLP variable, then all the return values will indicate that there is

no corresponding data.

`detrow` will be set to -1 if there is no determining row.

`delta`, `penaltydelta` and `updaterow` will be set to -1 if there is no corresponding item.

Related topics

`problem.addvars`, `problem.chgvar`, `problem.delvars`, `problem.loadvars`

problem.getVariable

Purpose

Returns one or more variables of a problem corresponding to one or more indices passed as arguments. These variables are returned as Python objects and can be used to access and manipulate the problem.

Synopsis

```
x = problem.getVariable(index, first, last)
```

Arguments

`index`	(optional) Either an integer or a list of integers (not necessarily sorted) with the index/indices of all variables to be returned, all between 0 and `COLS - 1`
`first`	(optional) The first index of the variables to be returned. It must be between 0 and `COLS - 1`.
`last`	(optional) The last index of the variables to be returned. It must be between 0 and `COLS - 1`.

Further information

All arguments are optional. If neither of them is provided, the return value is a list with all variables of the problem. Otherwise, either `first` and `last` or just `index` can be passed.

Related topics

`problem.getConstraint`, `problem.getSOS`.

problem.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 function will stop if MAXIIS IISs have been found. By default the control MAXIIS is set to –1, in which case an IIS is found for each of the infeasibilities in the model.

2. The problem attribute NUMIIS allows the user to recover the number of IISs found in a particular search. Alternatively, the problem.iisstatus function may be used to retrieve the number of IISs found by the problem.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(mode)
```

Argument

`mode`	The IIS search mode:
`0`	stops after finding the initial infeasible subproblem;
`1`	find an IIS, emphasizing simplicity of the IIS;
`2`	find an IIS, emphasizing a quick result.

Example

This looks for the first IIS.

```
p.iisfirst(1)
```

Further information

1. A model may have several infeasibilities. Repairing a single IIS may not make the model feasible. For this reason the Optimizer can find an IIS for each of the infeasibilities in a model. For the generation of several independent IISs use functions `problem.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(iis)
```

Argument

`iis` 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 `iis` 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(nrows, ncols, suminfeas, numinfeas)
```

Arguments

`nrows`	Number of rows in the IISs.
`ncols`	Number of bounds in the IISs.
`suminfeas`	The sum of infeasibilities in the IISs after the first phase simplex.
`numinfeas`	The number of infeasible variables in the IISs after the first phase simplex.

Example

This example first retrieves the number of IISs found so far, and then retrieves their main properties. Note that the arrays have size `count+1`, since the first index is reserved for the initial infeasible subset.

```
rs = []
cs = []
ninf = []
p.iisstatus(rs, cs, numinfeas=ninf) # suminf is not of interest
```

Further information

1. The arrays are `0` based, index 0 corresponding to the initial infeasible subproblem.
2. The arrays may be `None` if not required.
3. For the initial infeasible problem (at position 0) the subproblem size is returned (which may be different from the number of bounds), while for the IISs the number of bounds is returned (usually much smaller than the number of columns in the IIS).
4. Note that the values in `suminfeas` and `numinfeas` heavily depend on the actual basis where the simplex has stopped.
5. `iiscount` is set to `-1` if the search for IISs has not yet started.

Related topics

`problem.getiisdata`, `problem.iisall`, `problem.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(iis, filename, filetype, flags)
```

Arguments

`iis`	The ordinal number of the IIS to be written.
`filename`	The name of the file to be created.
`filetype`	Type of file to be created:
`0`	creates an lp/mps file containing the IIS as a linear programming problem;
`1`	creates a comma separated (csv) file containing the description and supplementary information on the given IIS.
`flags`	Flags passed to the `problem.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:
    1. save the IIS using MPS hexadecimal format to eliminate rounding errors associated with conversion between internal and decimal representation.
    2. turn presolve off since the nature of an IIS makes it necessary that during their identification the presolve is turned off.
    3. use the primal simplex method to solve the problem.
2. Note that the original sense of the original objective function plays no role in an IIS.
3. Even though an attempt is made to identify the most infeasible IISs first by the `problem.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` function can be called from any callback.

problem.loadbasis

Purpose

Loads a basis as specified by the user.

Synopsis

```
problem.loadbasis(rowstat, colstat)
```

Arguments

`rowstat` Array of length `problem.attributes.rows` containing the basis status of the slack, surplus or artificial variable associated with each row. The status must be one of:

- `0` slack, surplus or artificial is non-basic at lower bound;
- `1` slack, surplus or artificial is basic;
- `2` slack or surplus is non-basic at upper bound.
- `3` slack or surplus is super-basic.

`colstat` Array of length `problem.attributes.cols` containing the basis status of each of the columns in the constraint matrix. The status must be one of:

- `0` variable is non-basic at lower bound or superbasic at zero if the variable has no lower bound;
- `1` variable is basic;
- `2` variable is at upper bound;
- `3` variable is super-basic.

Example

This example loads a problem and then reloads a (previously optimized) basis from a similar problem to speed up the optimization:

```
p.read("problem", "")
p.loadbasis(rstatus, cstatus)
p.lpoptimize("")
```

Further information

If the problem has been altered since saving an advanced basis, one can alter the basis as follows before loading it:

- Make new variables non-basic at their lower bound (`cstatus[icol]=0`), unless a variable has an infinite lower bound and a finite upper bound, in which case make the variable non-basic at its upper bound (`cstatus[icol]=2`);
- Make new constraints basic (`rstatus[jrow]=1`);
- Try not to delete basic variables, or non-basic constraints.

Related topics

`problem.getbasis`, `problem.getpresolvebasis`, `problem.loadpresolvebasis`.

problem.loadbranchdirs

Purpose

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

Synopsis

```
problem.loadbranchdirs(colind, dir)
```

Arguments

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

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

Related topics

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

problem.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 holding the start position in the arrays `Type` and `Value` of the formula for the coefficients. `fstart[nSLPCoef]` should be set to the next position after the end of the last formula.
`parsed`	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 `fstart[j]`. The tokens follow the rules for parsed or unparsed formulae as indicated by the setting of `Parsed`. The formula must be terminated with an `xslp_op_eof` token. If several coefficients share the same formula, they can have the same value in `fstart`. For possible token types and values see the chapter on Formula Parsing in the SLP reference manual.

The `loadcoefs` function loads items into the SLP problem. Any existing items of the same type are deleted first. The corresponding `addcoefs` function adds or replace items leaving other items of the same type unchanged.

Related topics

`problem.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(coltype, interp, cutind)
```

Arguments

`coltype` Cut type.

`interp` The way in which the cut type is interpreted:

- `-1` load all cuts;
- `1` treat cut types as numbers;
- `2` treat cut types as bit maps - load cut if any bit matches any bit set in `coltype`;
- `3` treat cut types as bit maps - `0` load cut if all bits match those set in `coltype`.

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

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 tree search. These are rows that must be satisfied for any integer solution, but will not be loaded into the active set of constraints until required.

Synopsis

```
problem.loaddelayedrows(rowind)
```

Argument

`rowind` An array of rows (i.e. `xpress.constraint` objects, indices, or names) to treat as delayed rows.

Example

This sets the first six matrix rows as delayed rows in the MIP problem `prob`.

```
p.loaddelayedrows([0,1,2,3,4,5])
p.mipoptimize("")
```

Further information

Delayed rows must be set up before solving the problem. Any delayed rows will be removed from the problem after presolve and added to a special pool. A delayed row will be added back into the active matrix only when such a row is violated by an integer solution found by the Optimizer.

Related topics

`problem.loadmodelcuts`.

problem.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(colind, priority, dir, uppseudo, downpseudo)
```

Arguments

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

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

`dir` Character array specifying the branching direction for each column or set:

- `U` the entity is to be forced up;
- `D` the entity is to be forced down;
- `N` not specified.

May be `None` if not required.

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

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

Related topics

`problem.getdirs`, `problem.loadpresolvedirs`, `problem.readdirs`.

problem.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, duals, djs)
```

Arguments

`x`	Optional: Array of length `problem.attributes.cols` (for the original problem and not the presolve problem) containing the values of the variables.
`slack`	Optional: double array of length `problem.attributes.rows` containing the values of slack variables.
`duals`	Optional: double array of length `problem.attributes.rows` containing the values of duals variables.
`djs`	Optional: double array of length `problem.attributes.cols` containing the values of reduced costs.

Example

This example loads a problem and loads a solution for the problem.

```
p.read("problem", "")
status = p.loadlpsol(x, None, duals, None)
```

Further information

1. At least one of variables `x` and duals variables `duals` must be provided.
2. When variables `x` is `None`, the variables will be set to their bounds.
3. When slack variables `slack` is `None`, it will be computed from variables `x`. If slacks are provided, variables cannot be omitted.
4. When duals variables `duals` is `None`, both duals variables and reduced costs will be set to zero.
5. When reduced costs `djs` is `None`, it will be computed from duals variables `duals`. If reduced costs are provided, duals variables cannot be omitted.

Related topics

`problem.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(x)
```

Argument

`x` Array of length `problem.attributes.cols` (for the original problem and not the presolve problem) containing the values of the variables.

Example

This example loads a problem and then loads a solution found previously for the problem to help speed up the MIP search:

```
p.read("problem", "")
status = p.loadmipsol(x)
p.mipoptimize("")
```

Further information

1. When a solution is loaded before a MIP solve, the solution is simply placed in temporary storage until the MIP solve is started. Only after the MIP solve has commenced and any presolve has been applied, will the loaded solution be checked and possibly accepted as a new incumbent integer solution. There are no checks performed on the solution before the MIP solve and the returned status in `problem.loadmipsol` will always be `0` for accepted.
2. Solutions can be loaded during a MIP solve using the optnode callback function. Any solution loaded this way is immediately checked and the returned status will be one of the values `0` through `3`.
3. Loaded solution values will automatically be adjusted to fit within the current problem bounds.

Related topics

`problem.getmipsol`, `problem.addcboptnode`.

problem.loadmodelcuts

Purpose

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

Synopsis

```
problem.loadmodelcuts(rowind)
```

Argument

`rowind` An array of rows (i.e. `xpress.constraint` objects, indices, or names) to be treated as cuts.

Example

This sets the first six matrix rows as model cuts in the MIP problem `myprob`.

```
p.loadmodelcuts([0,1,2,3,4,5])
p.mipoptimize("")
```

Further information

1. During presolve the model cuts are removed from the problem and added to an internal cut pool. During the tree search, the Optimizer will regularly check this cut pool for any violated model cuts and add those that cuts off a node LP solution.
2. The model cuts must be "true" model cuts, in the sense that they are redundant at the optimal MIP solution. The Optimizer does not guarantee to add all violated model cuts, so they must not be required to define the optimal MIP solution.

problem.loadpresolvebasis

Purpose

Loads a presolved basis from the user's areas.

Synopsis

```
problem.loadpresolvebasis(rowstat, colstat)
```

Arguments

`rowstat` Array containing the basis status of the slack, surplus or artificial variable associated with each row. The status must be one of:

- `0` slack, surplus or artificial is non-basic at lower bound;
- `1` slack, surplus or artificial is basic;
- `2` slack or surplus is non-basic at upper bound.

`colstat` Array containing the basis status of each of the columns in the matrix. The status must be one of:

- `0` variable is non-basic at lower bound or superbasic at zero if the variable has no lower bound;
- `1` variable is basic;
- `2` variable is at upper bound;
- `3` variable is super-basic.

Example

The following example saves the presolved basis for one problem, loading it into another:

```
p1 = xpress.problem()
p2 = xpress.problem()

p1.read("myprob", "")
p1.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(colind, priority, dir, uppseudo, downpseudo)
```

Arguments

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

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

`dir` Character array specifying the branching direction for each column or set:

- `U` the entity is to be forced up;
- `D` the entity is to be forced down;
- `N` not specified.

May be `None` if not required.

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

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

Example

The following loads priority directives for column `0` in the problem:

```
p.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, rowtype, rhs, rng, objcoef, start, collen,
       rowind, rowcoef, lb, ub, objqcol1, objqcol2, objqcoef, qrowind,
       nrowqcoefs, rowqcol1, rowqcol2, rowqcoef, coltype, entind, limit,
       settype, setstart, setind, refval, colnames, rownames)
```

Arguments

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

`rowtype` Character array containing the row types:

- `L` indicates a <= constraint;
- `E` indicates an = constraint;
- `G` indicates a >= constraint;
- `R` indicates a range constraint;
- `N` indicates a nonbinding constraint.

`rhs` Array containing the right hand side coefficients of the rows. The right hand side value for a range row gives the **upper** bound on the row.

`rng` Array containing the range values for range rows. Values for all other rows will be ignored. May be `None` if there are no ranged constraints. The lower bound on a range row is the right hand side value minus the range value. The sign of the range value is ignored - the absolute value is used in all cases.

`objcoef` Array containing the objective function coefficients.

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

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

`rowind` Array containing the row indices for the nonzero elements in each column. If the indices are input contiguously, with the columns in ascending order, the length of the `rowind` is `start[ncol-1]+collen[ncol-1]` or, if `collen` is `None`, `start[ncol]`.

`rowcoef` Array containing the nonzero element values; length as for `rowind`.

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

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

`objqcol1` (optional) Array with the first variable in each quadratic term.

`objqcol2` (optional) Array with the second variable in each quadratic term.

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

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

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

`rowqcol1` (optional) Array with a number of elements equal to the sum of the elements in

nrowqcoefs (i.e. the total number of quadratic matrix elements in all the constraints). It contains the first column indices of the quadratic matrices. Indices for the first matrix are listed from 0 to nrowqcoefs[0]−1, for the second matrix from nrowqcoefs[0] to nrowqcoefs[0]+ nrowqcoefs[1]−1, etc.

rowqcol2 (optional) Array containing the second index for the quadratic constraint matrices.

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

coltype Character array containing the entity types:
- B binary variables;
- I integer variables;
- P partial integer variables;
- S semi-continuous variables;
- R semi-continuous integer variables.

entind (optional) Array containing the variables of the MIP entities.

limit (optional) Array containing the integer limits for the partial integer variables and lower bounds for semi-continuous and semi-continuous integer variables (any entries in the positions corresponding to binary and integer variables will be ignored). May be None if not required.

settype (optional) Character array of length equal to the number of sets specified, problem.attributes.nsets, and specifies the set types:
- 1 SOS1 type sets;
- 2 SOS2 type sets.

May be None if not required.

setstart (optional) Array containing the offsets in the setind and refval arrays indicating the start of the sets. This array is of length nsets+1, the last member containing the offset where set nsets+1 would start. May be None if not required.

setind (optional) Array of length setstart[nsets]−1 containing the columns in each set. May be None if not required.

refval (optional) Array of length setstart[nsets]−1 containing the reference row entries for each member of the sets. May be None if not required.

colnames (optional) Array of containing the column names for all variables added.

rownames (optional) Array of containing the row names for all constraints added.

Further information

1. The objective function is of the form $c^T x + 1/2\ x^T Q x$ where Q is positive semi-definite for minimization problems and negative semi-definite for maximization problems. If this is not the case the optimization algorithms may converge to a local optimum or may not converge at all. Note that only the upper or lower triangular part of the Q matrix is specified.
2. All Q matrices in the constraints must be positive semi-definite. Note that only the upper or lower triangular part of the Q matrix is specified for constraints as well.
3. If indices are specified, both row and column indices are from 0 to rows−1 and 0 to cols−1 respectively.
4. Semi-continuous lower bounds are taken from the dlim array. If this is None then they are given a default value of 1.0. If a semi-continuous variable has a positive lower bound then this will be used as the semi-continuous lower bound and the lower bound on the variable will be set to zero.

Related topics

problem.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(rowind, colind)
```

Arguments

`rowind`	Array containing the rows to be marked. May be `None` if not required.
`colind`	Array containing the columns to be marked. May be `None` if not required.

Example

This sets the first six rows and the first four columns to not be removed during presolve.

```
p.read("myprob", "")
p.loadsecurevecs(rowind=[0,1,2,3,4,5], colind=[0,1,2,3])
p.mipoptimize("")
```

problem.loadtolsets

Purpose

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 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 (note that Bit numbering begins at zero):
`Bit 1` Variable has a delta vector;
`Bit 2` Variable has an initial value;
`Bit 14` Variable is the reserved "=" column;
May be `None` if not required.

`detrow` Integer array holding the index of the determining row for each SLP variable (a negative value means there is no determining row)
May be `None` if not required.

`seqnum` Integer array holding the index sequence number for cascading for each SLP variable (a zero value means there is no pre-defined order for this variable)
May be `None` if not required.

`tolindex` Integer array holding the index of the tolerance set for each SLP variable (a zero value means the default tolerances are used)
May be `None` if not required.

`initvalue` Double array holding the initial value for each SLP variable (use the `VarType` bit map to indicate if a value is being provided)
May be `None` if not required.

`stepbound` Double array holding the initial step bound size for each SLP variable (a zero value means that no initial step bound size has been specified). If a value of `xpress.infinity` is used for a value in `StepBound`, the delta will never have step bounds applied, and will almost always be regarded as converged.
May be `None` if not required.

Example

The following example loads two SLP variables into the problem. They correspond to columns 23 and 25 of the underlying LP problem. Column 25 has an initial value of 1.42; column 23 has no specific initial value

```
colindex  = [23,25]
vartype   = [0,4]
initvalue = [0,1.42]

p.loadvars(colindex, vartype, None, None, None, initvalue, None)
```

`InitValue` is not set for the first variable, because it is not used (`VarType` = 0). Bit 1 of `VarType` is set for the second variable to indicate that the initial value has been set. The arrays for determining rows, sequence numbers, tolerance sets and step bounds are not used at all, and so have been passed to the function as `None`.

Further information

The `loadvars` functions load items into the SLP problem. Any existing items of the same type are deleted first. The corresponding `addvars` functions add or replace items leaving other items of the same type unchanged.

Related topics

problem.addvars, problem.chgvar, problem.delvars, problem.getvar

problem.lpoptimize

Purpose

This 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 MIP entities in the problem will be ignored.

Synopsis

```
problem.lpoptimize(flags)
```

Argument

`flags` (optional) Flags to pass to `lpoptimize`. The default is `""` or `None`, in which case the algorithm used is determined by the DEFAULTALG control. If the argument includes:

- `b` the model will be solved using the Newton barrier method;
- `p` the model will be solved using the primal simplex algorithm;
- `d` the model will be solved using the dual simplex algorithm;
- `n` (lower case `N`), the network part of the model will be identified and solved using the network simplex algorithm;

Further information

1. The algorithm used to optimize is determined by the DEFAULTALG control if no flags are provided. By default, the dual simplex is used for linear problems and the barrier is used for non-linear problems.
2. The `d` and `p` flags can be used with the `n` flag to complete the solution of the model with either the dual or primal algorithms once the network algorithm has solved the network part of the model.
3. The `b` flag cannot be used with the `n` flag.

Related topics

`problem.mipoptimize`, Chapter 4 of the Xpress Optimizer reference manual.

problem.mipoptimize

Purpose

This function begins a tree search for the optimal MIP solution. The direction of optimization is given by OBJSENSE. The status of the problem when the function completes can be checked using MIPSTATUS.

Synopsis

```
problem.mipoptimize(flags)
```

Argument

`flags` (optional) Flags to pass to problem.mipoptimize, which specifies how to solve the initial continuous problem where the MIP entities are relaxed. If the argument includes:

- `b` the initial continuous relaxation will be solved using the Newton barrier method;
- `p` the initial continuous relaxation will be solved using the primal simplex algorithm;
- `d` the initial continuous relaxation will be solved using the dual simplex algorithm;
- `n` the network part of the initial continuous relaxation will be identified and solved using the network simplex algorithm;
- `l` 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 tree search finishing before solving the continuous relaxation to optimality.
3. If the function returns without having completed the search for an optimal solution, the search can be resumed from where it stopped by calling problem.mipoptimize again.
4. The algorithm used to reoptimize the continuous relaxations during the tree search is given by DEFAULTALG. The default is to use the dual simplex algorithm.

Related topics

problem.mipoptimize.

problem.msaddcustompreset

Purpose

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.
`count`	Maximum number of jobs to be added to the multistart pool.
`ivcols`	Indices of the variables for which to set an initial value. May be `None` if nIVs is zero.
`ivvalues`	Initial values for the variables for which to set an initial value. May be `None` if nIVs is zero.
`control`	Python dictionary with control strings as keys and numbers as values. Note that only numerical controls are allowed.
`job_object`	Job-specific user context object to be passed to the multistart callbacks.

Further information

This function allows for repeatedly calling the same multistart preset (e.g. initial values) using different basic controls.

Related topics

`problem.msaddpreset`, `problem.msaddjob`, `problem.msclear`

problem.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 object to be passed to the multistart callbacks.

Further information

Adds a mutistart job, applying the specified initial point and option combinations on top of the base problem, i.e. the options and initial values specified to the function is applied on top of the existing settigns.

This function allows for loading empty template jobs, that can then be identified using the pJobObject variable.

Related topics

`problem.msaddpreset`, `problem.msaddcustompreset`, `problem.msclear`

problem.msaddpreset

Purpose

Loads a preset of jobs into the multistart job pool.

Synopsis

```
problem.msaddpreset(description, preset, maxjobs, data)
```

Arguments

`description` Text description of the preset. Used for messaging, may be `None` if not required.

`preset` Which preset to load.

`maxjobs` Maximum number of jobs to be added to the multistart pool.

`data` Job-specific user context object to be passed to the multistart callbacks.

Further information

The following presets are defined:

`msset_initialvalues`: generate `count` number of random base points.

`msset_solvers`: load all solvers.

`msset_slp_basic`: load the most typical SLP tuning settings. A maximum of `count` jobs are loaded.

`msset_slp_extended`: load a comprehensive set of SLP tuning settings. A maximum of `count` jobs are loaded.

`msset_knitro_basic`: load the most typical Knitro tuning settings. A maximum of `count` jobs are loaded.

`msset_knitro_extended`: load a comprehensive set of Knitro tuning settings. A maximum of `count` jobs are loaded.

`msset_initialfiltered`: generate `count` number of random base points, filtered by a merit function centred on initial feasibility.

See `xslp_MSMAXBOUNDRANGE` for controlling the range in which initial values are generated.

Related topics

`problem.msaddjob`, `problem.msaddcustompreset`, `problem.msclear`

problem.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.nlpoptimize

Purpose

Solves an SLP problem

Synopsis

```
problem.nlpoptimize(flags)
```

Argument

`flags` Flags affecting the solve. See the SLP reference manual for their meaning

problem.optimize

Purpose

This function begins a search for the optimal solution of the problem. The direction of optimization is given by OBJSENSE.

Synopsis

```
solvestatus, solstatus = problem.optimize(flags)
```

Arguments

`flags` Flags to pass to `problem.optimize`. The default is `""`, in which case the algorithm is determined automatically. If the argument includes:

- `x` the problem will be solved using the global solver;
- `s` the problem will be solved using XSLP or Knitro;
- `g` the branch and bound search will be performed.

`solvestatus` The solve status after termination. Takes the same values as SOLVESTATUS

`solstatus` The solution status after termination. Takes the same values as SOLSTATUS

Further information

1. If no flags are provided, the optimization will take any given constraints into account, including integrality and nonlinearities. Nonlinear problems will be solved to global optimality if the GLOBALSOLVE control is 1.
2. Any additional flags not listed above will be treated in the same way as for problem.lpoptimize, problem.mipoptimize and problem.nlpoptimize, depending on the type of optimization performed. The DEFAULTALG control will also behave in the same way as for these functions.

Related topics

problem.lpoptimize, problem.mipoptimize, problem.nlpoptimize

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(colind, lower, upper)
```

Arguments

`colind`	Array containing the columns (i.e. `xpress.var` objects, indices, or names) whose objective function coefficients sensitivity ranges are required.
`lower`	Array of the same size as `mindex` where the objective function lower range values are to be returned.
`upper`	Array of the same size as `mindex` where the objective function upper range values are to be returned.

Example

Here we obtain the objective function ranges for the three columns: `2`, `6` and `8`:

```
l = []
u = []
p.objsa([2,8,6], l, u)
```

After which `l` and `u` contain:

```
l = [5, 3.8, 5.7]
u = [7, 5.2, 1e+20]
```

Meaning that the current basis remains optimal when $5.0 \leq C_2 \leq 7.0$, $3.8 \leq C_8 \leq 5.2$ 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.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.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.optimize("")
```

Further information

If bit 1 of `xslp_presolve` is not set, no nonlinear presolve will be performed. Otherwise, the presolve will be performed in accordance with the bit settings. `problem.presolve` is called automatically by `problem.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(rowtype, origcolind, origrowcoef,
      origrhs, maxcoefs, colind, rowcoef)
```

Arguments

`rowtype`	The type of the row: `L` indicates a $\leq$ row; `G` indicates a $\geq$ row.
`origcolind`	Array containing the columns (i.e. `xpress.var` objects, indices, or names) of the row to presolve.
`origrowcoef`	Array containing the nonzero coefficients of the row to presolve.
`origrhs`	The right-hand side constant of the row to presolve.
`maxcoefs`	Maximum number of elements to return in the `colind` and `rowcoef` arrays.
`colind`	Array which will be filled with the columns of the presolved row.
`rowcoef`	Array which will be filled with the coefficients of the presolved row.

Example

Adding the row $2x_1 + x_2 \leq 1$ 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 nonzero coefficients in rows that will be presolved, it is possible to protect these individual columns through the `problem.loadsecurevecs` function. This way the Optimizer is left free to apply all possible reductions to the remaining columns.

Related topics

`problem.addcuts`, `problem.loadsecurevecs`, `problem.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.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.optimize("", "")
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 function calls:

```
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 tree search.

Synopsis

```
problem.readdirs(filename)
```

Argument

`filename` A string of up to 200 characters containing the file name from which the directives are to be read. If omitted (or `None`), the default *problem_name* is used with a `.dir` extension.

Example

The following example reads in directives from the file `dirfile.dir` for use with the problem, `prob2`:

```
p.read("prob2","")
p.readdirs("dirfile")
p.mipoptimize("")
```

Further information

1. Directives cannot be read in after a model has been presolved, so unless presolve has been disabled by setting `PRESOLVE` to `0`, this function must be called before `problem.mipoptimize`.
2. Directives can be given relating to priorities, forced branching directions, pseudo costs and model cuts. There is a priority value associated with each MIP entity. The *lower* the number, the *more* likely the entity is to be selected for branching; the *higher*, the *less* likely. By default, all MIP entities have a priority value of `500` which can be altered with a priority entry in the directives file. In general, it is advantageous for the entity's priority to reflect its relative importance in the model. Priority entries with values in excess of `1000` are illegal and are ignored. A full description of the directives file format may be found in the Xpress Optimizer reference manual.
3. By default, `problem.mipoptimize` will explore the branch expected to yield the best integer solution from each node, irrespective of whether this forces the MIP entity up or down. This can be overridden with an `UP` or `DN` entry in the directives file, which forces `mipoptimize` to branch up first or down first on the specified entity.
4. Pseudo-costs are estimates of the unit cost of forcing an entity up or down. By default `mipoptimize` uses dual information to calculate estimates of the unit up and down costs and these are added to the default pseudo costs which are set to the `PSEUDOCOST` control. The default pseudo costs can be overridden by a `PU` or `PD` entry in the directives file.
5. If model cuts are used, then the specified constraints are removed from the problem and added to the Optimizer cut pool, and only put back in the problem when they are violated by an LP solution at one of the nodes in the tree search.
6. If creating a directives file by hand, wild cards can be used to specify several vectors at once, for example `PR x1* 2` will give all MIP entities whose names start with `x1` a priority of `2`.

Related topics

`problem.loaddirs`.

problem.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 MIP entities, or otherwise loads it as an LP (barrier in case of quadratic problems) solution into the problem.

Further information

1. When `readslxsol` is called before a MIP solve, the loaded solutions will not be checked before calling `problem.mipoptimize`. By default, only the last MIP solution read from the `.slx` file will be stored. Use the `a` flag to store all MIP solutions read from the file.
2. When using the `a` flag, read solutions will be queued similarly to the user of the `problem.addmipsol` function. Each name string given by the `NAME` field in the `.slx` file will be associated with the corresponding solution. Any registered `usersolnotify` callback will be fired when the solution has been checked, and will include the read name string as one of its arguments.
3. Refer to the Appendix of the Xpress Optimizer reference manual on Log and File Formats for a description of the ASCII Solution (.slx) file format.

Related topics

`problem.readbinsol`, `problem.writeslxsol`, `problem.writebinsol`, `problem.readbinsol`, `problem.addmipsol`, `problem.addcbusersolnotify`.

problem.refinemipsol

Purpose

Runs the MIP solution refiner.

Synopsis

```
refinestatus = problem.refinemipsol(options, flags, solution, refined)
```

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` 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 MIP 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.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(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all bariteration callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all barrier iteration callbacks with the function `callback` will be removed.

Related topics

`problem.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(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all barrier log callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all barrier log callbacks with the function `callback` will be removed.

Related topics

`problem.addcbbarlog`.

problem.removecbchecktime

Purpose

Removes a callback function previously added by `problem.addcbchecktime`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbchecktime(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None`, then all checktime callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all checktime callbacks with the function pointer `callback` will be removed.

Related topics

`problem.addcbchecktime`

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(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all branch object callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If None, the object value will not be checked and all branch object callbacks with the function `callback` will be removed.

Related topics

`problem.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(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all cut log callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all cut log callbacks with the function `callback` will be removed.

Related topics

`problem.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(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all thread destruction callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all thread destruction callbacks with the function `callback` will be removed.

Related topics

`problem.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(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all `gapnotify` callback functions added with the given user-defined value will be removed.
`data`	The user-defined object that the callback was added with. If `None` then the object will not be checked and all the `gapnotify` callbacks with the function `callback` will be removed.

Related topics

`problem.addcbgapnotify`.

problem.removecbmiplog

Purpose

Removes a MIP log callback function previously added by `addcbmiplog`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbmiplog(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all MIP log callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all MIP log callbacks with the function `callback` will be removed.

Example

The following code sets and removes a callback function:

```
prob.controls.miplog = 3
prob.addcbmiplog(mipLog, None, 0)
prob.mipoptimize("")
prob.removecbmiplog(mipLog, None)
```

Related topics

`problem.addcbmiplog`.

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(callback, data)
```

Arguments

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

`data` The object value that the callback was added with. If `None`, then the object value will not be checked and all user infeasible node callbacks with the function `callback` will be removed.

Related topics

`problem.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(callback, data)
```

Arguments

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

`data` The object value that the callback was added with. If `None`, then the object value will not be checked and all integer solution callbacks with the function `callback` will be removed.

Related topics

`problem.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(callback, data)
```

Arguments

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

`data` The object value that the callback was added with. If `None`, then the object value will not be checked and all lplog callbacks with the function `callback` will be removed.

Example

The following code sets and removes a callback function:

```
prob.controls.lplog = 10
prob.addcblplog(lpLog, None, 0)
prob.readprob("problem", "")
prob.lpoptimize("")
prob.removecblplog(lpLog, None)
```

Related topics

`problem.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(callback, data)
```

Arguments

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

`data` The object value that the callback was added with. If `None`, then the object value will not be checked and all message callbacks with the function `callback` will be removed.

Further information

The Xpress Python API registers a message callback that prints messages to `stdout`. This callback cannot be removed explicity but can be disabled using `xpress.setOutputEnabled`.

Related topics

`problem.addcbmessage`. `xpress.setOutputEnabled`.

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(callback, data)
```

Arguments

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

`data` The object value that the callback was added with. If `None`, then the object value will not be checked and all variable branching callbacks with the function `callback` will be removed.

Related topics

`problem.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(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all separation callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all separation callbacks with the function `callback` will be removed.

Related topics

`problem.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(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all node-cutoff callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all node-cutoff callbacks with the function `callback` will be removed.

Related topics

`problem.addcbnodecutoff`.

problem.removecbnodelpsolved

Purpose

Removes a node lp solved callback function previously added by `addcbnodelpsolved`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
problem.removecbnodelpsolved(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all lp solved callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all lp solved callbacks with the function `callback` will be removed.

Related topics

`problem.addcbnodelpsolved`.

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(callback, data)
```

Arguments

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

`data` The object value that the callback was added with. If `None`, then the object value will not be checked and all node-optimal callbacks with the function `callback` will be removed.

Related topics

`problem.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(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all user infeasible node callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all user infeasible node callbacks with the function `callback` will be removed.

Related topics

`problem.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(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all preprocess node callback functions added with the given user-defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all preprocess node callbacks with the function `callback` will be removed.

Related topics

`problem.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(callback, data)
```

Arguments

`callback`	The callback function to remove. If `None` then all user solution notification callback functions added with the given user defined object value will be removed.
`data`	The object value that the callback was added with. If `None`, then the object value will not be checked and all integer solution callbacks with the function `callback` will be removed.

Related topics

`problem.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(penalty, phase2, flags, lepref, gepref,
      lbpref, ubpref, delta)
```

Arguments

`penalty` The type of penalties created from the preferences:
- `c` each penalty is the reciprocal of the preference (default);
- `s` the penalties are placed in the scaled problem.

`phase2` Controls the second phase of optimization:
- `o` use the objective sense of the original problem (default);
- `x` maximize the relaxed problem using the original objective;
- `f` skip optimization regarding the original objective;
- `n` minimize the relaxed problem using the original objective;
- `i` if the relaxation is infeasible, generate an irreducible infeasible subset for the analys of the problem;
- `a` if the relaxation is infeasible, generate all irreducible infeasible subsets for the analys of the problem.

`flags` Specifies if the tree search should be done:
- `g` do the tree search (default);
- `l` solve as a linear model ignoring the discreteness of variables.

`lepref` Preference for relaxing the less or equal side of row.

`gepref` Preference for relaxing the greater or equal side of a row.

`lbpref` Preferences for relaxing lower bounds.

`ubpref` Preferences for relaxing upper bounds.

`delta` The relaxation multiplier in the second phase -1. A positive value means a relative relaxation by multiplying the first phase objective with (`delta-1`), while a negative value means an absolute relaxation, by adding `abs(delta)` to the first phase objective.

Further information

1. A row or bound is relaxed by introducing a new nonnegative variable that will contain the infeasibility of the row or bound. Suppose for example that row $a^T x = b$ is relaxed from below. Then a new variable (infeasibility breaker) s>=0 is added to the row, which becomes $a^T x + s = b$. Observe that $a^T x$ may now take smaller values than b. To minimize such violations, the weighted sum of these new variables is minimized.
2. A preference of 0 results in the row or bound not being relaxed.
3. A negative preference indicates that a quadratic penalty cost should be applied. This can specified on a per constraint side or bound basis.
4. Note that the set of preferences are scaling independent.
5. If a feasible solution is identified for the relaxed problem, with a sum of violations p, then the sum of violations is restricted to be no greater than (`1+delta`)`p`, and the problem is optimized with respect to the original objective function. A nonzero delta increases the freedom of the original problem.
6. Note that on some problems, slight modifications of delta may affect the value of the original objective drastically.
7. Note that because of their special associated modeling properties, binary and semi-continuous variables are not relaxed.
8. The default algorithm for the first phase is the simplex algorithm, since the primal problem can be efficiently warm started in case of the extended problem. These may be altered by setting the value of control `DEFAULTALG`.
9. If `penalty` is set such that each penalty is the reciprocal of the preference, the following rules are applied while introducing the auxiliary variables:

Preference	Affects	Relaxation	Cost if pref.>0	Cost if pref.<0
`lepref`	`= rows`	$a^T x$ `- aux_var = b`	`1/lepref*aux_var`	`1/lepref*aux_var`^{2}
`lepref`	`<= rows`	$a^T x$ `- aux_var <= b`	`1/lepref*aux_var`	`1/lepref*aux_var`^{2}
`gepref`	`= rows`	$a^T x$ `+ aux_var = b`	`1/gepref*aux_var`	`1/gepref*aux_var`^{2}
`gepref`	`>= rows`	$a^T x$ `+ aux_var >= b`	`1/gepref*aux_var`	`1/gepref*aux_var`^{2}
`ubpref`	`upper bounds`	x_i `- aux_var <= u`	`1/ubpref*aux_var`	`1/ubpref*aux_var`^{2}
`lbpref`	`lower bounds`	x_i `+ aux_var >= l`	`1/lbpref*aux_var`	`1/lbpref*aux_var`^{2}

10. If an irreducible infeasible set (IIS) has been identified, the generated IIS(s) are accesible through the IIS retrieval functions, see `NUMIIS` and `problem.getiisdata`.

Related topics

`problem.repairweightedinfeas`.

problem.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(lepref, gepref, lbpref, ubpref,
      phase2, delta, flags)
```

Arguments

`lepref` Array of size `ROWS` containing the preferences for relaxing the less or equal side of row.

`gepref` Array of size `ROWS` containing the preferences for relaxing the greater or equal side of a row.

`lbpref` Array of size `COLS` containing the preferences for relaxing lower bounds.

`ubpref` Array of size `COLS` containing preferences for relaxing upper bounds.

`phase2` Controls the second phase of optimization:

- `o` use the objective sense of the original problem (default);
- `x` maximize the relaxed problem using the original objective;
- `f` skip optimization regarding the original objective;
- `n` minimize the relaxed problem using the original objective;
- `i` if the relaxation is infeasible, generate an irreducible infeasible subset for the analys of the problem;
- `a` if the relaxation is infeasible, generate all irreducible infeasible subsets for the analys of the problem.

`delta` The relaxation multiplier in the second phase -1.

`flags` Specifies flags to be passed to the Optimizer.

Further information

1. A row or bound is relaxed by introducing a new nonnegative variable that will contain the infeasibility of the row or bound. Suppose for example that row $a^T x$ = b is relaxed from below. Then a new variable ('infeasibility breaker') s>=0 is added to the row, which becomes $a^T x$ +s = b. Observe that $a^T x$ may now take smaller values than b. To minimize such violations, the weighted sum of these new variables is minimized.
2. A preference of 0 results in the row or bound not being relaxed. The higher the preference, the more willing the modeller is to relax a given row or bound.
3. The weight of each infeasibility breaker in the objective minimizing the violations is 1/p, where `p` is the preference associated with the infeasibility breaker. Thus the higher the preference is, the lower a penalty is associated with the infeasibility breaker while minimizing the violations.
4. If a feasible solution is identified for the relaxed problem, with a sum of violations `p`, then the sum of violations is restricted to be no greater than `(1+delta)p`, and the problem is optimized with respect to the original objective function. A nonzero delta increases the freedom of the original problem.
5. Note that on some problems, slight modifications of delta may affect the value of the original objective drastically.
6. Note that because of their special associated modeling properties, binary and semi-continuous variables are not relaxed.
7. If `pflags` is set such that each penalty is the reciprocal of the preference, the following rules are applied while introducing the auxiliary variables:

Pref. array	Affects	Relaxation	Cost if pref.>0	Cost if pref.<0
`lepref`	`= rows`	$a^T x$ `- aux_var = b`	`1/lrp*aux_var`	$1/lrp*aux\_var^2$
`lepref`	`<= rows`	$a^T x$ `- aux_var <= b`	`1/lrp*aux_var`	$1/lrp*aux\_var^2$
`gepref`	`= rows`	$a^T x$ `+ aux_var = b`	`1/grp*aux_var`	$1/grp*aux\_var^2$
`gepref`	`>= rows`	$a^T x$ `+ aux_var >= b`	`1/grp*aux_var`	$1/grp*aux\_var^2$
`ubpref`	`upper bounds`	x_i `- aux_var <= u`	`1/ubp*aux_var`	$1/ubp*aux\_var^2$
`lbpref`	`lower bounds`	x_i `+ aux_var >= l`	`1/lbp*aux_var`	$1/lbp*aux\_var^2$

8. If an irreducible infeasible set (IIS) has been identified, the generated IIS(s) are accesible through the IIS retrieval functions, see `NUMIIS` and `problem.getiisdata`.

Related topics

`problem.repairinfeas`, `problem.repairweightedinfeasbounds`.

problem.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(lepref, gepref, lbpref, ubpref,
       lerelax, gerelax, lbrelax, ubrelax, phase2, delta, flags)
```

Arguments

`lepref` Array of size `ROWS` containing the preferences for relaxing the less or equal side of row.

`gepref` Array of size `ROWS` containing the preferences for relaxing the greater or equal side of a row.

`lbpref` Array of size `COLS` containing the preferences for relaxing lower bounds.

`ubpref` Array of size `COLS` containing preferences for relaxing upper bounds.

`lerelax` Array of size `ROWS` containing the upper bounds on the amount the less or equal side of a row can be relaxed.

`gerelax` Array of size `ROWS` containing the upper bounds on the amount the greater or equal side of a row can be relaxed.

`lbrelax` Array of size `COLS` containing the upper bounds on the amount the lower bounds can be relaxed.

`ubrelax` Array of size `COLS` containing the upper bounds on the amount the upper bounds can be relaxed.

`phase2` Controls the second phase of optimization:

- `o` use the objective sense of the original problem (default);
- `x` maximize the relaxed problem using the original objective;
- `f` skip optimization regarding the original objective;
- `n` minimize the relaxed problem using the original objective;
- `i` if the relaxation is infeasible, generate an irreducible infeasible subset for the analys of the problem;
- `a` if the relaxation is infeasible, generate all irreducible infeasible subsets for the analys of the problem.

`delta` The relaxation multiplier in the second phase -1.

`flags` Specifies flags to be passed to the Optimizer.

Further information

1. A row or bound is relaxed by introducing a new nonnegative variable that will contain the infeasibility of the row or bound. Suppose for example that row $a^T x = b$ is relaxed from below. Then a new variable ('infeasibility breaker') s>=0 is added to the row, which becomes $a^T x + s = b$. Observe that $a^T x$ may now take smaller values than b. To minimize such violations, the weighted sum of these new variables is minimized.
2. A preference of 0 results in the row or bound not being relaxed. The higher the preference, the more willing the modeller is to relax a given row or bound.
3. A negative preference indicates that a quadratic penalty cost should be applied. This can specified on a per constraint side or bound basis.
4. If a feasible solution is identified for the relaxed problem, with a sum of violations `p`, then the sum of violations is restricted to be no greater than `(1+delta)p`, and the problem is optimized with respect to the original objective function. A nonzero delta increases the freedom of the original problem.
5. Note that on some problems, slight modifications of delta may affect the value of the original objective drastically.
6. Note that because of their special associated modeling properties, binary and semi-continuous variables are not relaxed.
7. Given any row `j` with preferences `lrp=lepref[j]` and `grp=gepref[j]`, or variable `i` with bound preferences `ubp=ubpref[i]` and `lbp=lbpref[i]`, the following rules are applied while introducing the auxiliary variables:

Preference	Affects	Relaxation	Cost if pref.>0	Cost if pref.<0
`lrp`	`= rows`	$a^T x$ `- aux_var = b`	`1/lrp*aux_var`	`1/lrp*aux_var`^{2}
`lrp`	`<= rows`	$a^T x$ `- aux_var <= b`	`1/lrp*aux_var`	`1/lrp*aux_var`^{2}
`grp`	`= rows`	$a^T x$ `+ aux_var = b`	`1/grp*aux_var`	`1/grp*aux_var`^{2}
`grp`	`>= rows`	$a^T x$ `+ aux_var >= b`	`1/grp*aux_var`	`1/grp*aux_var`^{2}
`ubp`	`upper bounds`	x_i `- aux_var <= u`	`1/ubp*aux_var`	`1/ubp*aux_var`^{2}
`lbp`	`lower bounds`	x_i `+ aux_var >= l`	`1/lbp*aux_var`	`1/lbp*aux_var`^{2}

8. Only positive bounds are applied; a zero or negative bound is ignored and the amount of relaxation allowed for the corresponding row or bound is not limited. The effect of a zero bound on a row or bound would be equivalent with not relaxing it, and can be achieved by setting its preference array value to zero instead, or not including it in the preference arrays.
9. If an irreducible infeasible set (IIS) has been identified, the generated IIS(s) are accesible through the IIS retrieval functions, see `NUMIIS` and `problem.getiisdata`.

Related topics

`problem.repairinfeas`.

problem.reset

Purpose

Clears all information regarding an optimization problem and returns it to the same status as it would be after creation (i.e. after the instruction `p = xpress.problem()`).

Synopsis

```
problem.reset()
```

Example

```
p = xpress.problem()
p.read("problem0", "l")
p.optimize()
x0 = p.getSolution()
p.reset()
p.read("problem1", "")
p.optimize()
x1 = p.getSolution()
```

Related topics

`problem.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 tree 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(rowind, lower, upper)
```

Arguments

`rowind`	Array containing the rows (i.e. `xpress.constraint` objects, indices, or names) whose RHS coefficients sensitivity ranges are required.
`lower`	Array where the RHS lower range values are to be returned.
`upper`	Array where the RHS upper range values are to be returned.

Example

Here we obtain the RHS function ranges for the three columns: `2`, `6` and `8`:

```
l = []
u = []
p.rhssa([2,8,6], l, u)
```

After which lower and upper contain:

```
l = [5, 3.8, 5.7]
u = [7, 5.2, 1e+20]
```

Meaning that the current basis remains optimal when $5.0 \leq \text{rhs}_2$, $3.8 \leq \text{rhs}_8 \leq 5.2$ 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(filename=None)
```

Argument

`filename` A file name on which to write the problem.

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 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(rowscale, colscale)
```

Arguments

`rowscale` Array of size `ROWS` containing the exponents of the powers of `2` with which to scale the rows, or `None` if not required.

`colscale` 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 `rowscale` and `colscale` 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 MIP entity.

Synopsis

```
problem.setbranchbounds(bounds)
```

Argument

`bounds` 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 MIP entity..

Synopsis

```
problem.setbranchcuts(cutind)
```

Argument

`cutind` Array containing 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(callback, data)
value = callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called at the end of the cascading process. `callback` returns an integer value. The return value is noted by Xpress SLP but it has no effect on the optimization.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbcascadeend`.
`data`	User-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Example

The following example sets up a callback to be executed at the end of the cascading process which checks if any of the values have been changed significantly:

```
csol = [1,2,3,4]
p.setcbcascadeend(CBCascEnd, csol)
```

A suitable callback function might resemble this:

```
def CBCascEnd(prob, obj):
  for iCol in range(prob.controls.cols):
    (a,b,c,s,d,e,f,value,g,h,i,j,k,l,m,n) = prob.getvar(iCol)
    if abs(value - obj[iCol]) > .01:
      print("Col {0} changed from {1} to {2}".format(iCol, obj[iCol], value)
  return 0
```

The `obj` argument is used here to hold the original solution values.

Further information

This callback can be used at the end of the cascading, when all the solution values have been recalculated.

Related topics

`problem.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(callback, data)
retval = callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called at the start of the cascading process. `callback` returns an integer value. If the return value is nonzero, the cascading process will be omitted for the current SLP iteration, but the optimization will continue.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbcascadestart`.
`data`	A user-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Further information

This callback can be used at the start of the cascading, before any of the solution values have been recalculated.

Related topics

`problem.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(callback, data)
retval = callback(my_prob, my_object, colindex)
```

Arguments

`callback`	The function to be called after each column has been cascaded. `callback` returns an integer value. If the return value is nonzero, the cascading process will be omitted for the remaining variables during the current SLP iteration, but the optimization will continue.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `problem.setcbcascadevar`.
`colindex`	The number of the column which has been cascaded.
`data`	User-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Example

The following example sets up a callback to be executed after each variable has been cascaded:

```
obj = []
p.setcbcascadevar(CBCascVar, obj)
```

The following sample callback function resets the value of the variable if the cascaded value is of the opposite sign to the original value:

```
def CBCascVar(myprob, obj, iCol):
(a,b,c,d,e,f,value,g,h,i,j,k,l,m,n) = myprob.getvar(iCol)
  if value * obj[iCol] < 0:
    p.chgvar(col=ColNum, value=obj[iCol])
  return 0
```

The `data` argument is used here to hold the array `cSol` which we assume has been populated with the original solution values.

Further information

This callback can be used after each variable has been cascaded and its new value has been calculated.

Related topics

`problem.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(callback, data)
retval = callback(my_prob, my_object, colindex)
```

Arguments

`callback`	The function to be called after cascading a column was not successful. `callback` returns an integer value. If the return value is nonzero, the cascading process will be omitted for the remaining variables during the current SLP iteration, but the optimization will continue.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbcascadevarfail`.
`colindex`	The number of the column which has been cascaded.
`data`	A user-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Further information

This callback can be used to provide user defined updates for SLP variables having a determining row that were not successfully cascaded due to the determining row being close to singular around the current values. This callback will always be called in place of the cascadevar callback in such cases, and in no situation will both the cascadevar and the cascadevarfail callback be called in the same iteration for the same variable.

Related topics

`problem.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(callback, data)
retval = callback(my_prob, my_object, rowindex, colindex)
```

Arguments

`callback`	The function to be called when an evaluation fails.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbcoefevalerror`.
`rowindex`	The row position of the coefficient.
`colindex`	The column position of the coefficient.
`data`	A user-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Further information

This callback can be used to capture when an evaluation of a coefficient fails. The callback is called only once for each coefficient.

Related topics

`problem.printevalinfo`

problem.setcbconstruct

Purpose

Set a user callback to be called during the Xpress SLP augmentation process

Synopsis

```
problem.setcbconstruct(callback, data)
retval = callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called during problem augmentation. `callback` returns an integer value. See below for an explanation of the values.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbconstruct`.
`data`	A user-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Example

The following example sets up a callback to be executed during the Xpress SLP problem augmentation:

```
value = []
p.setcbconstruct(CBConstruct, value)
```

The following sample callback function sets values for the variables the first time the function is called and returns to `problem.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 is None:
    n = myprob.attributes.cols
    cValue = n * [0]
    # initialize with values (not shown here)
    for i in range(n):
    # store into SLP structures
    myprob.chgvar(col=i, value=cValue[i])
    # set Object non-None to indicate we have processed data
    obj = cValue
    return -1
  else:
    obj = None
  return 0
```

Further information

This callback can be used during the problem augmentation, generally (although not exclusively) to change the initial values for the variables.

The following return codes are accepted:

`0`	Normal return: augmentation continues
`-1`	Return to recalculate matrix values
`-2`	Return to recalculate row weights and matrix entries
`other`	Error return: augmentation terminates, `problem.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(callback, data)
callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called when the SLP problem is about to be destroyed. `callback` returns an integer value. At present the return value is ignored.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbdestroy`.
`data`	A user-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Example

The following example sets up a callback to be executed before the SLP problem is destroyed:

```
p.setcbdestroy(CBDestroy, cSol)
```

The following sample callback function frees the memory associated with the user-defined object:

```
def CBDestroy(myprob, Obj):
  if Obj is not None:
    Obj.inuse = 0
  return 0
```

The `Obj` argument is used here to hold the array `cSol` which we assume was assigned using one of the `malloc` functions.

Further information

This callback can be used when the problem is about to be destroyed to free any user-defined resources which were allocated during the life of the problem.

problem.setcbdrcol

Purpose

Set a user callback used to override the update of variables with small determining column

Synopsis

```
problem.setcbdrcol(callback, data)
newvalue = callback(my_prob, my_object, colindex, drcolindex, drcolvalue,
       vlb, vub)
```

Arguments

`callback`	The function to be called after each column has been cascaded. `callback` returns an integer value. If the return value is positive, it will indicate that the value has been fixed, and cascading should be omitted for the variable. A negative value indicates that a previously fixed value has been relaxed. If no action is taken, a 0 return value should be used.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbcascadevar`.
`ColIndex`	The column (i.e. `xpress.var` object, index, or name) for which the determining columns is checked.
`DrColIndex`	The index of the determining column for the column that is being updated.
`DrColValue`	The value of the determining column in the current SLP iteration.
`NewValue`	Used to return the new value for column `ColIndex`, should it need to be updated, in which case the callback must return a positive value to indicate that this value should be used.
`VLB`	The original lower bound of column `ColIndex`. The callback provides this value as a reference, should the bound be updated or changed during the solution process.
`VUB`	The original upper bound of column `ColIndex`. The callback provides this value as a reference, should the bound be updated or changed during the solution process.
`data`	A user-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Further information

If set, this callback is called as part of the cascading procedure. Please see the chapter on cascading of the SLP Reference Manual for more information.

Related topics

`xslp_DRCOLTOL`, `problem.cascade`, `problem.setcbcascadeend`, `problem.setcbcascadestart`

problem.setcbintsol

Purpose

Set a user callback to be called during MISLP when an integer solution is obtained

Synopsis

```
problem.setcbintsol(callback, data)
callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called when an integer solution is obtained.
`data`	A user-defined object, which can be used for any purpose. `data` is passed to `callback` as `my_object`.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbintsol`.

Example

The following example sets up a callback to be executed whenever an integer solution is found during MISLP:

```
cSol = []
p.setcbintsol(CBIntSol, cSol)
```

The following sample callback function saves the solution values for the integer solution just found:

```
def CBIntSol(prob, cSol):
  prob.getmipsol(x=cSol, None, None, None)}
```

Related topics

`problem.setcboptnode`, `problem.setcbprenode`

problem.setcbiterend

Purpose

Set a user callback to be called at the end of each SLP iteration

Synopsis

```
problem.setcbiterend(callback, data)
retval = callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called at the end of each SLP iteration. `callback` returns an integer value. If the return value is nonzero, the SLP iterations will stop.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbiterend`.
`data`	User-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Example

The following example sets up a callback to be executed at the end of each SLP iteration. It records the number of LP iterations in the latest optimization and stops if there were fewer than 10:

```
p.setcbiterend(CBIterEnd, None)
```

A suitable callback function might resemble this:

```
def CBIterEnd(MyProb, Obj):
  niter = MyProb.attributes.simplexiter
  return (niter < 10)
```

The `Obj` argument is not used here, and so is passed as `None`.

Further information

This callback can be used at the end of each SLP iteration to carry out any further processing and/or stop any further SLP iterations.

Related topics

`problem.setcbiterstart`, `problem.setcbitervar`

problem.setcbiterstart

Purpose

Set a user callback to be called at the start of each SLP iteration

Synopsis

```
problem.setcbiterstart(callback, data)
retval = callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called at the start of each SLP iteration. `callback` returns an integer value. If the return value is nonzero, the SLP iterations will stop.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbiterstart`.
`data`	User-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Example

The following example sets up a callback to be executed at the start of the optimization to save to save the values of the variables from the previous iteration:

```
p.setcbiterstart(CBIterStart, cSol)
```

A suitable callback function might resemble this:

```
def CBIterStart(MyProb, Obj):
  niter = MyProb.attributes.xslp_iter
  if nIter == 0:
    return 0 # no previous solution
  Obj = []
  MyProb.getlpsol(Obj, None, None, None)
  return 0
```

The `Obj` argument is used here to hold the array `cSol` which we populate with the solution values.

Further information

This callback can be used at the start of each SLP iteration before the optimization begins.

Related topics

`problem.setcbiterend`, `problem.setcbitervar`

problem.setcbitervar

Purpose

Set a user callback to be called after each column has been tested for convergence

Synopsis

```
problem.setcbitervar(callback, data)
retval = callback(my_prob, my_object, colindex)
```

Arguments

`callback` The function to be called after each column has been tested for convergence. `callback` returns an integer value. The return value is interpreted as a convergence status. The possible values are:

`< 0`	The variable has not converged;
`0`	The convergence status of the variable is unchanged;
`1 to 10`	The column has converged on a system-defined convergence criterion (these values should not normally be returned);
`> 10`	The variable has converged on user criteria.

`my_prob` The problem passed to the callback function.

`my_object` The user-defined object passed as `data` to `setcbitervar`.

`ColIndex` The number of the column which has been tested for convergence.

`data` A user-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Example

The following example sets up a callback to be executed after each variable has been tested for convergence. The user object `Important` is an integer array which has already been set up and holds a flag for each variable indicating whether it is important that it converges.

```
Obj = None
p.setcbitervar(CBIterVar, Obj)
```

The following sample callback function tests if the variable is already converged. If not, then it checks if the variable is important. If it is not important, the function returns a convergence status of 99.

```
def CBIterVar(MyProb, Obj, iCol):
  (a,b,c,d,e,f,g,h,i,converged,j,k,l,m,n) = MyProb.getvar(iCol)
  if converged:
    return 0
  if Obj[iCol]:
    return 99
  return -1
```

The `object` argument is used here to hold the array `Important`.

Further information

This callback can be used after each variable has been checked for convergence, and allows the convergence status to be reset if required.

Related topics

`problem.setcbiterend`, `problem.setcbiterstart`

problem.setcbmessage

Purpose

Set a user callback to be called whenever Xpress Nonlinear outputs a line of text

Synopsis

```
problem.setcbmessage(callback, data)
callback(my_prob, my_object, msg, msgtype)
```

Arguments

`callback` The function to be called whenever Xpress Nonlinear outputs a line of text. `callback` does not return a value.

`my_prob` The problem passed to the callback function.

`my_object` The user-defined object passed as `data` to `setcbmessage`.

`msg` String to be output.

`msgtype` Type of message. The following are system-defined:

`1`	Information message
`3`	Warning message
`4`	Error message

A negative value indicates that the Optimizer is about to finish and any buffers should be flushed at this time. User-defined values are also possible for `msgtype`.

`data` A user-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Example

The following example creates a log file into which all messages are placed. System messages are also printed on standard output:

```
log = ''
p.setcbmessage(CBMessage, log)
```

A suitable callback function could resemble the following:

```
def CBMessage(Obj, msg, msgtype):
  if msgtype < 0:
    print(log)
    log = ''
    return
  if msgtype >= 1 and msgtype <= 4:
    print(msg)
  else:
    log += msg + ';'
```

Further information

If a user message callback is defined, screen output is automatically disabled.

Output can be directed into a log file by using `problem.setlogfile`.

Also, because of Python's garbage collection functions, it is advised to explicitly delete a problem at the end of its use (with the `del` statement) if a message callback was set for that problem using `setcbmessage`. def CBMessage(Obj, msg, msgtype): pass

p = xp.problem() p.setcbmessage(CBMessage, None) [...] del p

Related topics

`problem.setlogfile`

problem.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(callback, data)
status = callback(my_prob, my_object, job_object, description)
```

Arguments

`callback` The function to be called when a new multistart job is created

`my_prob` The problem passed to the callback function.

`my_object` The user-defined object passed as `data` to `setcbmsjobend`.

`job_object` Job specific user-defined object, as specified in by the multistart job creating API functions.

`description` The description of the problem as specified in by the multistart job creating API functions.

`status` User return status variable:
0 - use the default evaluation of the finished job
1 - disregard the result and continue
2 - stop the multistart search

Further information

The multistart pool is dynamic, and this callback can be used to load new multistart jobs using the normal API functions.

Related topics

`problem.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(callback, data)
status = callback(my_prob, my_object, job_object, description)
```

Arguments

`callback` The function to be called when a new multistart job is created;

`my_prob` The problem passed to the callback function.

`my_object` The user-defined object passed as `data` to `setcbmsjobstart`.

`job_object` Job specific user-defined object, as specified in by the multistart job creating API functions.

`description` The description of the problem as specified in by the multistart job creating API functions.

`status` User return status variable:
0 - normal return, solve the job,
1 - disregard this job and continue,
2 - Stop multistart.

Further information

All mulit-start jobs operation on an independent copy of the original problem, and any modification to the problem is allowed, including structural changes. Please note however, that any modification will be carried over to the base problem, should a modified problem be declared the winner prob.

Related topics

`problem.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(callback, data)
callback(my_prob, my_object, job_object, description)
```

Arguments

`callback` The function to be called when a new multistart job is created

`my_prob` The problem passed to the callback function.

`my_object` The user-defined object passed as `data` to `setcbmswinner`.

`job_object` Job specific user-defined object, as specified in by the multistart job creating API functions.

`description` The description of the problem as specified in by the multistart job creating API functions.

Further information

The multistart pool is dynamic, and this callback can be used to load new multistart jobs using the normal API functions.

Related topics

`problem.setcbmsjobstart`, `problem.setcbmsjobend`

problem.setcboptnode

Purpose

Set a user callback to be called during MISLP when an optimal SLP solution is obtained at a node

Synopsis

```
problem.setcboptnode(callback, data)
infeas = callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called when an optimal SLP solution is obtained at a node. It must returns an integer value. If the return value is nonzero, or if the feasibility flag is set nonzero, then further processing of the node will be terminated (it is declared infeasible).
`data`	The user-defined object passed as `my_object` to `setcboptnode`.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed to `setcboptnode`.
`infeas`	Integer containing the feasibility flag. If nonzero, the node is declared infeasible.

Example

The following example defines a callback function to be run at each node when an SLP optimal solution is found. If there are significant penalty errors in the solution, the node is declared infeasible.

```
p.setcboptnode(CBOptNode, None)
```

A suitable callback function might resemble the following:

```
def CBOptNode(prob, data) {
  total = prob.attributes.xslp_errorcosts
  objval = prob.attributes.xslp_objval
  if abs(total) > abs(objval) * 0.001 and abs(total) > 1:
    return 1
  else:
    return 0
```

Further information

If a node is declared infeasible from the callback function, the cost of exploring the node further will be avoided.

This callback must be used in place of `setcboptnode` when optimizing with MISLP.

Related topics

`problem.setcbprenode`, `problem.setcbslpnode`

problem.setcbprenode

Purpose

Set a user callback to be called during MISLP after the set-up of the SLP problem to be solved at a node, but before SLP optimization

Synopsis

```
problem.setcbprenode(callback, data)
feas = callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called after the set-up of the SLP problem to be solved at a node. `callback` returns an integer value. If the return value is nonzero, then further processing of the node will be terminated (it is declared infeasible).
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbprenode`.
`feas`	feasibility flag. If `callback` return a nonzero, the node is declared infeasible.

Example

The following example sets up a callback function to be executed at each node before the SLP optimization starts. The array `IntList` contains a list of integer variables, and the function prints the bounds on these variables.

```
IntList = [...]
prob.setcbprenode(CBPreNode, IntList)
```

A suitable callback function might resemble the following:

```
def CBPreNode(myProb, intlist):
  for i in intlist:
    LO,UP = [],[]
    myProb.getlb(LO,i,i);
    myProb.getub(UP,i,i);
    lb,ub = LO[0], UP[0]
    if lb > 0 or ub < xp.infinity:
      print("Col {0}: {1} <= {2}".format(i,lb,ub)
  return 0
```

Further information

If a node can be identified as infeasible by the callback function, then the initial optimization at the current node is avoided, as well as further exploration of the node.

Related topics

problem.setcboptnode, problem.setcbslpnode

problem.setcbpreupdatelinearization

Purpose

Set a user callback to be called before the linearization is updated

Synopsis

```
problem.setcbpreupdatelinearization(callback, data)
ifRepeat = callback(my_prob, my_object, when)
```

Arguments

`callback`	The function to be called before the linearization is updated. If `callback` returns `True`, another call to the callback will be scheduled. If it returns `False`, a final call with `when == -1` will be made.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbpreupdatelinearization`.
`when`	Indicates the call number, starting at 1. A value of -1 indicates the end of the linearization update.

Further information

When the linearization is updated, all user functions are evaluated and their derivatives calculated at the current base point. In some models, it is cheaper to compute the derivatives for all user functions at the same time, thereby avoiding repeated calculations for each function. This callback is intended to be used in such cases.

During each SLP iteration, the callback is invoked repeatedly, with `when` indicating the current call number (starting from 1), until the callback indicates that no further calls are needed, by returning `False`. Between each callback invocation, the solver evaluates the user functions without requesting derivatives. After the callback has returned `False`, the user functions are evaluated one more time, this time requesting derivatives, and then finally the callback is called with `when == -1`, marking the end of the linearization update. The only time derivatives will be requested outside of this sequence is during KKT validation. This can be disabled during the solve by clearing the `XSLP_CONVERGEBIT_VALIDATION_K` bit in `XSLP_CONVERGENCEOPS`, ensuring that derivatives can always be precomputed.

One way that this callback can be used to precompute derivatives for user functions is as follows:

1. On each SLP iteration, the callback is first called with `when = 1`. This is a signal that derivatives will be needed soon. The callback sets a flag to indicate that user functions should capture their input values, and returns `True` to request another call.
2. When the callback returns, the user functions are evaluated without requesting derivatives. Each user function captures its input values somewhere, and returns the correct function value.
3. The callback is called again, with `when == 2`. The callback now computes derivates for all user functions using the captured input values. The callback clears the flag so that user functions no longer capture their input values, and returns `False` to indicate that no further calls are needed.
4. When the callback returns, the user functions are evaluated again. Derivatives are requested, and the user functions return the precomputed derivative values.
5. The callback is invoked one more time for this iteration with `when == -1`, marking the end of the linearization update. User functions should behave normally from this point.

problem.setcbslpend

Purpose

Set a user callback to be called at the end of the SLP optimization

Synopsis

```
problem.setcbslpend(callback, data)
callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called at the end of the SLP optimization. `callback` returns an integer value. If the return value is nonzero, the optimization will return an error code and the "User Return Code" error will be set.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbslpend`.
`data`	A user-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Example

The following example sets up a callback to be executed at the end of the SLP optimization. It frees the memory allocated to the object created when the optimization began:

```
ObjData = None
p.setcbslpend(CBSlpEnd, ObjData)
```

A suitable callback function might resemble this:

```
def CBSlpEnd(MyProb, Obj):
  if Obj is not None:
    Obj = []
  return 0
```

Further information

This callback can be used at the end of the SLP optimization to carry out any further processing or housekeeping before the optimization function returns.

Related topics

`problem.setcbslpstart`

problem.setcbslpnode

Purpose

Set a user callback to be called during MISLP after the SLP optimization at each node.

Synopsis

```
problem.setcbslpnode(callback, data)
(retval, infeas) = callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called after the set-up of the SLP problem to be solved at a node. `callback` returns an integer value. If the return value is nonzero, or if the feasibility flag is set nonzero, then further processing of the node will be terminated (it is declared infeasible).
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbslpnode`.
`infeas`	An integer containing the feasibility flag. If `callback` sets the flag nonzero, the node is declared infeasible.
`data`	A user-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Example

The following example sets up a callback function to be executed at each node after the SLP optimization finishes. If the solution value is worse than a target value (referenced through the user object), the node is cut off (it is declared infeasible).

```
objtarget = []
p.setcbslpnode(CBSLPNode, objtarget)
```

A suitable callback function might resemble the following:

```
def CBSLPNode(my_prob, my_obj):
  lpval = my_prob.attributes.lpobjval
  return (0, (lpval < my_obj))
```

Further information

If a node can be cut off by the callback function, then further exploration of the node is avoided.

problem.setcbslpstart

Purpose

Set a user callback to be called at the start of the SLP optimization

Synopsis

```
problem.setcbslpstart(callback, data)
retval = callback(my_prob, my_object)
```

Arguments

`callback`	The function to be called at the start of the SLP optimization. `callback` returns an integer value. If the return value is nonzero, the optimization will not be carried out.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `data` to `setcbslpstart`.
`data`	User-defined object, which can be used for any purpose by the function. `data` is passed to `callback` as `my_object`.

Example

The following example sets up a callback to be executed at the start of the SLP optimization:

```
Objdata = []
p.setcbslpstart(CBSlpStart, Objdata)
```

A suitable callback function might resemble this:

```
def CBSlpStart(object):
  object.append(1)
  return 0
```

Further information

This callback can be used at the start of the SLP optimization to carry out any housekeeping before the optimization actually starts. Note that a nonzero return code from the callback will terminate the optimization immediately.

Related topics

`problem.setcbslpend`

problem.setControl

Purpose

Sets one or more controls of a problem. Can also be used to set objective controls.

Synopsis

```
problem.setControl(control, value, objidx=None)
problem.setControl(dict, objidx=None)
```

Arguments

`control` Name or numeric id of the control whose value to change. If the `objidx` argument is provided, the control must be one of the following objective controls:

- `priority` the priority of the objective
- `weight` the weight of the objective
- `reltol` the relative tolerance of the objective
- `abstol` the absolute tolerance of the objective
- `rhs` the constant part of the objective

`value` Value to which to set the control.

`dict` A dictionary mapping names or numeric ids of controls to values to which to set them.

`objidx` (optional) Index of the objective whose control to modify.

Example

```
p = xpress.problem()
p.setControl('miprelstop', 1e-4)
p.setControl({'feastol': 1e-4, 'presolve': 0})
```

Further information

1. As mentioned in the previous chapter, there is an alternative way to set and retrieve controls. It works by querying the data structure `controls` of each problem or, if one wants to set a control to be used by all problems defined subsequently, the global control object `xpress.controls`.
2. This function can be used in two ways depending on whether one wants to set one or more controls. In the first case, the arguments form a pair (string, value) where the first element is the lower-case name of a control (see the Xpress Optimizer reference manual for a complete list of controls). In the second case, the argument is a Python *dictionary* whose keys are control name string and whose values are the value of the control. Instead of control names it is also possible to use their numeric ids.

Related topics

`problem.getControl`.

problem.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. It must be called before the problem is read with `problem.read` or loaded with `problem.loadproblem`.

Synopsis

```
problem.setdefaults()
```

Example

The following turns off presolve to solve a problem, before resetting the control defaults, reading it and solving it again:

```
p.controls.presolve = 0
p.mipoptimize("")
p.writeprtsol()
p.setdefaults()
p.read()
p.mipoptimize("")
```

Related topics

`xpress.setdefaultcontrol`, `xpress.setdefaults`, `problem.setdefaults`.

problem.setindicators

Purpose

Specifies that a set of rows in the problem will be treated as indicator constraints during a tree search. An indicator constraint is made of a `condition` and a `linear inequality`. The `condition` is of the type "`bin = value`", where `bin` is a binary variable and `value` is either 0 or 1. The `linear inequality` is any linear row in the problem with type <= (`L`) or >= (`G`). During tree search, a row configured as an indicator constraint is enforced only when condition holds, that is only if the indicator variable `bin` has the specified value.

Synopsis

```
problem.setindicators(rowind, colind, complement)
```

Arguments

`rowind` Array containing the rows (i.e. `xpress.constraint` objects, indices, or names) that define the linear inequality part for the indicator constraints.

`colind` Array containing the columns (i.e. `xpress.var` objects, indices, or names) of the indicator variables.

`complement` Array with the complement flags:

- `0` not an indicator constraint (in this case the corresponding entry in the `colind` array is ignored);
- `1` for indicator constraints with condition "`bin = 1`";
- `-1` for indicator constraints with condition "`bin = 0`";

Example

This sets the first two matrix rows as indicator rows in the MIP problem prob; the first row controlled by condition `x4=1` and the second row controlled by condition `x5=0` (assuming `x4` and `x5` correspond to columns indices 4 and 5).

```
p.setindicators([0,1],[4,5],[1,-1])
p.mipoptimize("")
```

Further information

Indicator rows must be set up before solving the problem. Any indicator row will be removed from the problem after presolve and added to a special pool. An indicator row will be added back into the active matrix only when its associated condition holds. An indicator variable can be used in multiple indicator rows and can also appear in normal rows and in the objective function.

Related topics

`problem.getindicators`.

problem.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(msgcode, status)
```

Arguments

`msgcode`	The id number of the message. Refer to the Section 9 of the Xpress Optimizer reference manual for a list of possible message numbers.
`status`	Nonzero if the message is not suppressed; `0` otherwise.

Example

Attempting to optimize a problem that has no matrix loaded gives error 91. The following code uses `setmessagestatus` to suppress the error message:

```
p = xpress.problem()
p.setmessagestatus(91, 0)
p.lpoptimize("")
```

Further information

If a message is suppressed globally then the message can only be enabled for any problem once the global suppression is removed with a call to `setmessagestatus` with `prob` passed as `None`.

Related topics

`problem.getmessagestatus`.

problem.setObjective

Purpose

Sets the objective function of the problem.

Synopsis

```
problem.setObjective(expr, sense=None, objidx=0, priority=None, weight=None,
       abstol=None, reltol=None)
```

Arguments

`expr`	An expression involving variables which have been added to the problem prior to this call. An error will be returned if any variable in the objective was not already added to the problem via `addVariable`.
`sense`	(optional) Either `xpress.minimize` or `xpress.maximize` (by default the sense will be left unchanged).
`objidx`	(optional) Index of the objective to modify.
`priority`	(optional) Priority for the new objective (only relevant for multi-objective problems).
`weight`	(optional) Weight for the new objective (only relevant for multi-objective problems).
`abstol`	(optional) Absolute tolerance for the new objective (only relevant for multi-objective problems).
`reltol`	(optional) Relative tolerance for the new objective (only relevant for multi-objective problems).

Example

The following example sets the objective function of the problem to $[2x_1^2 + 3x_1x_2 + 5x_2^2 + 4x_1 + 4]$:

```
x1 = xpress.var()
x2 = xpress.var()
p = xpress.problem()
p.addVariable(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.addObjective`, `problem.addobj`, `problem.chgobjn`, `problem.delobj`.

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.storebounds

Purpose

Stores bounds for node separation using user separate callback function.

Synopsis

```
mindex = problem.storebounds(colind, bndtype, bndval)
```

Arguments

`colind` Array containing the columns (i.e. `xpress.var` objects, indices, or names).

`bndtype` Array containing the bounds types:
- `U` indicates an upper bound;
- `L` indicates a lower bound.

`bndval` 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(nodups, cuttype, rowtype, rhs, start, cutind, colind,
      cutcoef)
```

Arguments

`nodups`
- `0` do not exclude duplicates from the cut pool;
- `1` duplicates are to be excluded from the cut pool;
- `2` duplicates are to be excluded from the cut pool, ignoring cut type.

`cuttype` Array containing the cut types. The cut types can be any integer and are used to identify the cuts.

`rowtype` Character array containing the row types:
- `L` indicates a $\leq$ row;
- `E` indicates an = row;
- `G` indicates a $\geq$ row.

`rhs` Array containing the right hand side elements for the cuts.

`start` Array containing offsets into the `colind` and `cutcoef` arrays indicating the start of each cut. This array is of length `ncuts+1` where `ncuts` is the length of `rhs`, with the last element `start[ncuts]` being where cut `ncuts+1` would start.

`cutind` Array where the cuts will be returned.

`colind` Array containing the columns in the cuts.

`cutcoef` Array containing the matrix values for the cuts.

Further information

1. `storecuts` can be used to eliminate duplicate cuts. If the `nodups` parameter is set to `1`, the cut pool will be checked for duplicate cuts with a cut type identical to the cuts being added. If a duplicate cut is found the new cut will only be added if its right hand side value makes the cut stronger. If the cut in the pool is weaker than the added cut it will be removed unless it has been applied to an active node of the tree. If `nodups` is set to `2` the same test is carried out on all cuts, ignoring the cut type.

2. `storecuts` returns a list of the cuts added to the cut pool in the `cutind` array. If the cut is not added to the cut pool because a stronger cut exits a None will be returned. The `cutind` array can be passed directly to `problem.loadcuts` 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 `colind` and `cutcoef` arrays passed to `storecuts`. The starting point of each cut must be stored in the `start` array. To determine the length of the final cut the `start` array must be of length `ncuts+1` with the last element of this array containing where the cut `ncuts+1` would start.

Related topics

`problem.loadcuts`, `problem.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(colind, bndtype, bndval, iterlim, objval, status)
```

Arguments

`colind` Array containing the columns (i.e. `xpress.var` objects, indices, or names) on which the bounds will change.

`bndtype` Character array indicating the type of bound to change:
- `U` indicates change the upper bound;
- `L` indicates change the lower bound;
- `B` indicates change both bounds, i.e. fix the column.

`bndval` Array giving the new bound values.

`iterlim` Maximum number of LP iterations to perform for each bound change.

`objval` Objective value of each LP after performing the strong branching iterations.

`status` Status of each LP after performing the strong branching iterations, as detailed for the `LPSTATUS` attribute.

Example

Suppose that the current LP relaxation has two integer columns (columns `0` and `1` which are fractionals at 0.3 and 1.5, respectively, and we want to perform strong branching in order to choose which to branch on. This could be done in the following way:

```
objval = []
status = []
p.strongbranch([0,0,1,0], ['','','',''], [1,0,2,1],
               1000, objval, status)
```

Further information

Prior to calling strongbranch, the current LP problem must have been solved to optimality and an optimal basis must be available.

problem.strongbranchcb

Purpose

Performs strong branching iterations on all specified bound changes. For each candidate bound change, `strongbranchcb` performs dual simplex iterations starting from the current optimal solution of the base LP, and returns both the status and objective value reached after these iterations.

Synopsis

```
problem.strongbranchcb(colind, bndtype, bndval, iterlim, objval, status,
       callback, data)
ret = callback(my_prob, my_object, ibnd)
```

Arguments

`colind` Array containing the columns (i.e. `xpress.var` objects, indices, or names) on which the bounds will change.

`bndtype` Character array indicating the type of bound to change:
- `U` indicates change the upper bound;
- `L` indicates change the lower bound;
- `B` indicates change both bounds, i.e. fix the column.

`bndval` Array giving the new bound values.

`iterlim` Maximum number of LP iterations to perform for each bound change.

`objval` Objective value of each LP after performing the strong branching iterations.

`status` Status of each LP after performing the strong branching iterations, as detailed for the `LPSTATUS` attribute.

`callback` Function to be called after each strong branch has been reoptimized.

`data` The user-defined object passed as `my_object` to `callback`.

`ibnd` The index of bound for which `callback` is called.

Further information

Prior to calling `strongbranchcb`, the current LP problem must have been solved to optimality and an optimal basis must be available.

`strongbranchcb` is an extension to `problem.strongbranch`. If identical input arguments are provided both will return identical results, the difference being that for the case of `PRSstrongbranchcb` the `sbnodecb` function is called at the end of each LP reoptimization. For each branch optimized, the LP can be interrogated: the LP status of the branch is available through checking `LPSTATUS`, and the objective function value is available through `LPOBJVAL`. It is possible to access the full current LP solution by using `problem.getlpsol`.

problem.tune

Purpose

Begin a tuner session for the current problem. The tuner will solve the problem multiple times while evaluating a list of control settings and promising combinations of them. When finished, the tuner will select and set the best control setting on the problem. Note that the direction of optimization is given by `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.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

`validate` checks the feasibility of a converged solution against relative and absolute tolerances for each constraint. The left hand side and the right hand side of the constraint are calculated using the converged solution values. If the calculated values imply that the constraint is infeasible, then the difference (*D*) is tested against the absolute and relative validation tolerances.
If $D < XSLP\_VALIDATIONTOL\_A$
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(mode, respectbasis, updatemult, violtarget)
```

Arguments

`mode` The calculation mode can be:

- `0` recalculate the reduced costs at the current solution using the current dual solution.
- `1` minimize the sum of KKT violations by adjusting the dual solution.
- `2` perform both.

`respectbasis` The following ways are defined to assess if a constraint is active:

- `0` evaluate the recalculated slack activity versus `xslp_ECFTOL_R`.
- `1` use the basis status of the slack in the linearized problem if available.
- `2` use both.

`updatemult` The calculated values can be:

- `0` only used to calculate the `xslp_validationindex_k` measure.
- `1` used to update the current dual solution and reduced costs.

`violtarget` When calculating the best KKT multipliers, it is possible to enforce an even distribution of reduced costs violations by enforcing a bound on them.

Further information

The bounds enforced by `violtarget` are automatically relaxed if the desired accuracy cannot be achieved.

problem.validaterow

Purpose

Prints an extensive analysis on a given constraint of the SLP problem

Synopsis

```
problem.validate(row)
```

Argument

`row` The row (i.e. `xpress.constraint` object, index, or name) to be analyzed.

Further information

The analysis will include the readable format of the original constraint and the augmented constraint. For infeasible constraints, the absolute and relative infeasibility is calculated. Variables in the constraints are listed including their value in the solution of the last linearization, the internal value (e.g. cascaded), reduced cost, step bound and convergence status. Scaling analysis is also provided.

problem.validatevector

Purpose

Validate the feasibility of constraints for a given solution

Synopsis

```
(suminf, sumscaledinf, obj) = problem.validate(solution)
```

Arguments

`solution` A vector of length `xpress.attributes.cols` containing the solution vector to be checked.

`suminf` The sum of infeasibilities.

`sumscaledinf` The sum of scaled (relative) infeasibilities.

`obj` The net objective.

Further information

`validatevector` works the same way as `problem.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.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. For example:

```
p.read("myprob", "")
p.lpoptimize("")
p.writebasis("", "")
```

This reads in a problem file, maximizes the LP and saves the basis. Loading a basis for a MIP problem can disable some MIP presolve operations which can result in a large increase in solution times so it is generally not recommended.

Further information

1. The `t` flag is only useful for later input to a similar problem using the `t` flag with `problem.readbasis`.
2. If the Newton barrier algorithm has been used for optimization then crossover must have been performed before there is a valid basis. This basis can then only be used for restarting the simplex (primal or dual) algorithm.
3. `writebasis` will output the basis for the original problem even if the problem has been presolved.

Related topics

`problem.getbasis`, `problem.readbasis`.

problem.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 tree search directives from the current problem to a directives file.

Synopsis

```
problem.writedirs(filename)
```

Argument

`filename` A string of up to 200 characters containing the file name to which the directives should be written. If omitted (or `None`), the default *problem_name* is used with a `.dir` extension.

Further information

If the problem has been presolved, only the directives for columns in the presolved problem will be written to file.

Related topics

`problem.loaddirs`.

problem.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 function is not as readily useful as that produced by `problem.writesol`. The main purpose of `writeprtsol` is to create a file that can be sent directly to a printer. The format of this fixed format ASCII file is described in the Xpress Optimizer reference manual.
2. To create a prt file for a previously saved solution, the solution must first be loaded with the `problem.readbinsol` function.

Related topics

`problem.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 MIP entities, or otherwise saves the LP(barrier in case of quadratic problems) solution of the problem.

Related topics

`problem.readslxsol`, `problem.writeprtsol`, `problem.writebinsol`, `problem.readbinsol`.

problem.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 solution saved;
- `p` outputs in full precision;
- `q` only outputs vectors with nonzero optimum value;
- `x` output the current LP solution instead of the MIP solution.

Example

In this example the basis status is output (along with the sequence number) following optimization:

```
p.read("prob1", "")
p.lpoptimize("")
p.writesol("", "sb")
```

Further information

1. The function produces two readable files: `filename.hdr` (the solution header file) and `filename.asc` (the CSV foramt solution file). The header file contains summary information, all in one line. The ASCII file contains one line of information for each row and column in the problem. Any fields appearing in the `.asc` file will be in the order the flags are described above. The order that the flags are specified by the user is irrelevant.

2. Additionally, the mask control `OUTPUTMASK` may be used to control which names are reported to the ASCII file. Only vectors whose names match `OUTPUTMASK` are output. `OUTPUTMASK` is set by default to "`????????`", so that all vectors are output.

Related topics

`problem.getlpsol`, `problem.getmipsol`, `problem.writeprtsol`.

problem.getOutputEnabled

Purpose

Returns True if Optimizer messages will be written to the Python output stream, False otherwise.

Synopsis

```
enabled = problem.getOutputEnabled()
```

Related topics

`problem.setOutputEnabled`, `xpress.getOutputEnabled`, `xpress.setOutputEnabled`.

problem.setOutputEnabled

Purpose

Enables or disables writing Optimizer messages to the Python output stream.

Synopsis

```
problem.setOutputEnabled(enabled)
```

Argument

`enabled` True if Optimizer messages should be written to the Python output stream, False otherwise.

Related topics

`problem.getOutputEnabled`, `xpress.getOutputEnabled`, `xpress.setOutputEnabled`.

7.13 Xpress branch object methods

branchobj.addbounds

Purpose

Adds new bounds to a branch of a user branching object.

Synopsis

```
branchobj.addbounds(branch, bndtype, colind, bndval)
```

Arguments

`branch` The number of the branch to add the new bounds for. This branch must already have been created using `branchobj.addbranches`. Branches are indexed starting from zero.

`bndtype` Character array indicating the type of bounds to add:

- `L` Lower bound.
- `U` Upper bound.

`colind` Array containing the columns for the new bounds.

`bndval` Array giving the bound values.

branchobj.addbranches

Purpose

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(branch, cutind)
```

Arguments

`branch`	The number of the branch to add the cuts for. This branch must already have been created using `branchobj.addbranches`. Branches are indexed starting from zero.
`cutind`	Array containing the user cuts that should be added to the branch.

Related topics

`branchobj.addrows`.

branchobj.addrows

Purpose

Adds new constraints to a branch of a user branching object.

Synopsis

```
branchobj.addrows(branch, rowtype, rhs, start, colind, rowcoef)
```

Arguments

`branch` The number of the branch to add the new constraints for. This branch must already have been created using `branchobj.addbranches`. Branches are indexed starting from zero.

`rowtype` Character array indicating the type of constraints to add:
- `L` Less than type.
- `G` Greater than type.
- `E` Equality type.

`rhs` Array containing the right-hand side values.

`start` Array containing the offsets of the `colind` and `dval` arrays of the start of the non zero coefficients in the new constraints.

`colind` Array containing the columns for the non zero coefficients.

`rowcoef` Array containing the nonzero coefficient values.

Example

The following function will create a branching object that branches on constraints $x_1 + x_2 \geq 1$ 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(branch, maxbounds, bndtype, colind, bndval)
```

Arguments

`branch`	The number of the branch to get the bounds for.
`maxbounds`	Maximum number of bounds to return.
`bndtype`	Character array of length `maxbounds` where the types of bounds twill be returned: `L` Lower bound. `U` Upper bound.
`colind`	Array of length `maxbounds` where the columns will be returned.
`bndval`	Array of length `maxbounds` where the bound values will be returned.

Related topics

`branchobj.addbounds`.

branchobj.getbranches

Purpose

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(branch, maxrows, maxcoefs, rowtype, rhs, start, colind,
      rowcoef)
```

Arguments

`branch`	The number of the branch to get the constraints from.
`maxrows`	Maximum number of rows to return.
`maxcoefs`	Maximum number of non zero coefficients to return.
`rowtype`	Character array of length `maxrows` where the types of the rows will be returned: `L` Less than type. `G` Greater than type. `E` Equality type.
`rhs`	Array of length `maxrows` where the right hand side values will be returned.
`start`	Array of length `maxrows` which will be filled with the offsets of the `colind` and `rowcoef` arrays of the start of the non zero coefficients in the returned constraints.
`colind`	Array of length `maxcoefs` which will be filled with the column indices for the non zero coefficients.
`rowcoef`	Array of length `maxcoefs` which will be filled with the non zero coefficient values.

Related topics

`branchobj.addrows`.

branchobj.setpreferredbranch

Purpose

Specifies which of the child nodes corresponding to the branches of the object should be explored first.

Synopsis

```
branchobj.setpreferredbranch(branch)
```

Argument

`branch`	The number of the branch to mark as preferred.

branchobj.setpriority

Purpose

Sets the priority value of a user branching object.

Synopsis

```
branchobj.setpriority(priority)
```

Argument

`priority` The new priority value to assign to the branching object, which must be a number from 0 to 1000. User branching objects are created with a default priority value of 500.

Further information

1. A candidate branching object with lowest priority number will always be selected for branching before an object with a higher number.
2. Priority values must be an integer from 0 to 1000. User branching objects and MIP entities are by default assigned a priority value of 500. Special branching objects, such as those arising from structural branches or split disjunctions are assigned a priority value of 400.

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 and a link to the Customer Self Service Portal (online support) on the Product Support home page (www.fico.com/en/product-support).

The FICO Customer Self Service Portal is a secure web portal that is available 24 hours a day, 7 days a week from the Product Support home page. The portal allows you to open, review, update, and close cases, as well as find solutions to common problems in the FICO Knowledge Base.

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 home page 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.

Please include your contact information (name, company, email address, and optionally, your phone number) so we may reach you if we have questions.

Sales and maintenance

If you need information on other Xpress Optimization products, or you need to discuss maintenance contracts or other sales-related items, contact FICO by:

- Phone: +1 (408) 535-1500 or +44 207 940 8718
- Web: http://www.fico.com/optimization and use the available contact forms

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.

FICO Community

The FICO Community is a great resource to find the experts and information you need to collaborate, support your business, and solve common business challenges. You can get informal technical support, build relationships with local and remote professionals, and improve your business practices. For additional information, visit the FICO Community (community.fico.com/welcome).

About FICO

FICO (NYSE:FICO) powers decisions that help people and businesses around the world prosper. Founded in 1956 and based in Silicon Valley, the company is a pioneer in the use of predictive analytics and data science to improve operational decisions. FICO holds more than 165 US and foreign patents on technologies that increase profitability, customer satisfaction, and growth for businesses in financial services, telecommunications, health care, retail, and many other industries. Using FICO solutions, businesses in more than 100 countries do everything from protecting 2.6 billion payment cards from fraud, to helping people get credit, to ensuring that millions of airplanes and rental cars are in the right place at the right time. Learn more at www.fico.com.

Index

X