R interface reference manual

API reference manual

45.01

FICO® Xpress Optimization

FICO

©1983–2025 Fair Isaac Corporation. All rights reserved. This documentation is the property of Fair Isaac Corporation ("FICO"). Receipt or possession of this documentation does not convey rights to disclose, reproduce, make derivative works, use, or allow others to use it except solely for internal evaluation purposes to determine whether to purchase a license to the software described in this documentation, or as otherwise set forth in a written software license agreement between you and FICO (or a FICO affiliate). Use of this documentation and the software described in it must conform strictly to the foregoing permitted uses, and no other use is permitted.

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

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

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

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

Xpress Optimizer 45.01 (FICO® Xpress 9.7)

Deliverable Version: A

Last Revised: 29 July, 2025

Contents

Chapter 1

Introduction

The `xpress package` provides an R interface to the FICO Xpress optimizer. The interface consists of two parts:

- a thin wrapper for the C library. All the solver functionality is available through this wrapper.
- a set of convenience function that allow a more "R-like" experience when interacting with the solver.

1.1 Thin Wrapper API

The lowest level of the optimizer's R interface is a thin wrapper for the C library. People familiar with the C API of the solver will find that things look pretty similar to C. There are a few differences, though:

- In case of error, functions do not return non-zero status codes. Instead they raise an error by calling `stop()` with an appropriate error message.
- Functions that have more than one output do not return these outputs via output parameters but instead return a list that contains all the output arguments.
- Functions that don't have an explicit return value return the problem object. This way you can easily chain function calls, for example by using the `%>%` operator from the `magrittr` package.

Note that the C library does not operate with vectors and matrices like R does. Instead it works with arrays of indices and non-zero values. Indexing in these arrays is 0-based, so that the first variable, constraint, ... will have index 0. This is in contrast to R matrices and vectors for which indexing is 1-based. This must be kept in mind and taken care of when passing data to the optimizer and reading back results.

The convenience functions listed in the next section allow to directly exchange R matrices and vectors with the solver.

1.2 Creation and Deletion of Problems

In order to interact with the solver, a problem object is required. These problems are created by means of the `createprob()` function:

```
prob <- createprob()
```

Note that `createprob()` implicity calls `init("")` in order to initialize licensing if that did not happen yet.

Attached to the problem object are native resources, i.e., resources that are not managed by R. It is thus the user's responsibility to free the problem object when it is no longer needed. The object is freed using function `destroyprob()`. Good ways to make sure the object is deleted is to use `on.exit()` or `tryCatch`:

```
prob <- createprob()
# make sure problem is deleted when function exits
on.exit(destroyprob(prob))
# or alternatively, execute code in a tryCatch:
tryCatch({ ... },
         finally = { destroyprob(prob) })
```

1.3 Callbacks

The low-level API supports callback. There is one caveat, though: Since R is inherently single-threaded and cannot be called back into from other threads than the main thread, all callback invocations have to be routed through the main thread. The Xpress optimizer supports this by setting the `XPRS_CALLBACKFROMMASTERTHREAD` control to 1. This may result in some performance degradation, though.

A callback in the Xpress R interface is just a function or closure with the correct number of arguments. Please refer to the (C) reference documentation to learn what callbacks exist, how many arguments the functions require and what the callbacks can do.

A callback may return one of the following:

- `NULL` or `NA`. These values are just ignored.
- A length-one integer vector. If the corresponding C callback allows returning a value (usually in order to stop the search or indicate some other action) then this value is passed back to the optimizer, otherwise is is ignored.
- A list. If the corresponding C callback allows returning values then those values are read from the list and returned to the optimizer. The list may also contain a field "ret", which must be an integer and is treated the same as the single integer return value described above.

A warning is issued in case a callback returns something else or returns a list with unknown elements.

1.4 Convenience Functions

The FICO Xpress optimizer interface provides a set of convenience functions that are based on the thin wrapper described above but allow directly using R data structures like vectors and matrices directly.

Please refer to the documentation for functions `xprs_loadproblemdata` and `xprs_optimize` for further details. These functions allow using R matrices and vectors directly in order to solve optimization problems.

1.5 Installation

Please refer to the INSTALL.txt in the R directory of your Xpress installation for installation instructions.

1.6 Licensing

To run the Optimizer from the R interface it is necessary to have a valid licence file, "xpauth.xpr". Please

see the license chapter of the FICO Xpress Online Documentation for more information on licensing options.

The FICO Xpress licensing system is highly flexible and is easily configurable to cater for the user's requirements. The system can allow the Optimizer to be run on a specific machine, on a machine with a specific ethernet address or on a machine connected to an authorized hardware dongle.

On Unix systems it is necessary to set the `XPAUTH_PATH` environment variable to the full path to the license file. For ease of support it is recommended that the license file is placed in the bin directory within your Xpress installation and the `XPAUTH_PATH` environment variable is set accordingly before launching an R session , e.g.,

```
export XPAUTH_PATH=/opt/xpressmp/bin/xpauth.xpr
R
```

An alternative is to provide the path by calling the `init` function with the path to the license inside R:

```
library(xpress)
init("/opt/xpressmp/bin/xpauth.xpr")
```

Another alternative is to place the file "xpauth.xpr" into your current working directory before starting your R session, from where it will be picked up automatically.

On Windows operating systems the Optimizer searches for the license file in the directory containing the Xpress libraries, which are installed by default into the "C:\xpressmp\bin" folder. To avoid unnecessary licensing problems, it is recommended that the `XPAUTH_PATH` environment variable is not set on Windows.

1.7 A First LP Problem

For starters, we solve the following 2-variable model from R using the Xpress-R interface.

$$\begin{aligned} \min \quad & x_1 + x_2 \\ & 5x_1 + x_2 \geqslant 7 \\ & x_1 + 4x_2 \geqslant 9 \\ & x_1, x_2 \geqslant 0 \end{aligned}$$

```
suppressMessages(library(xpress))

# create a list object to hold all input
problemdata <- list()

# objective coefficients
problemdata$objcoef <- c(1, 1)

# row coefficients as a single matrix object
problemdata$A <- matrix(c(5, 1, 1, 4), nrow = 2, byrow = TRUE)

# right-hand side
problemdata$rhs <- c(7, 9)

# row sense
problemdata$rowtype <- c("G", "G")

# lower bounds (automatically 0 if not present)
problemdata$lb <- c(0, 0)

# upper bounds(automatically inferred if not present)
problemdata$ub <- c(Inf, Inf)
```

```
# names for writing to MPS/LP files
problemdata$colname <- c("x_1", "x_2")

# Problem Name displayed when the solver solves the problem.
problemdata$probname <- "FirstExample"
```

We now load everything into a new XPRSprob 'p'. `xprs_loadproblemdata` can be used to load all problem data at once into an existing or new problem object. In this case, we have no existing XPRSprob object. `xprs_loadproblemdata` creates one for us. For convenience and the use inside pipes, xprs_loadproblemdata returns the prob pointer.

```
prob <- xprs_loadproblemdata(problemdata = problemdata)
# You may also use the equivalent
# prob <- createprob()
# xprs_loadproblemdata(prob, problemdata = problemdata)
```

The newly created XPRSprob object supports the `print` and `summary` statements. Let's get an overview of the optimization problem loaded into prob

```
print(prob)
```

```
## XPRESS problem object FirstExample
##      2 rows         2 cols        4 elems
##      0 entities      0 sets        0 indicators
##      0 qelems        0 qcelems
##      0 gencons       0 pwlcons
```

We use `xprs_optimize` to solve the model. This again returns 'prob', which can be summarized using the base function `summary`

```
summary(xprs_optimize(prob))
```

```
##
## Objective value                     : 3.00000e+00
## Max primal violation      (abs/rel) : 0 / 0
## Max dual violation        (abs/rel) : 0 / 0
## LPSTATUS: 1
```

It may be useful for further processing to convert the solution into a data frame.

```
print(data.frame(Variable = problemdata$colname, Value = getsolution(prob)$x))
```

```
##   Variable Value
## 1      x_1     1
## 2      x_2     2
```

CHAPTER 2

Problem Creation Reference

- Mixed Integer Programs (MIP)
  - The Problem Data Representation
  - Working With R Matrices
  - Incremental Formulation
- Mixed Integer Quadratically Constrained Programs (MIQCQP)
  - The Problem Data Representation
  - Working With R Matrices
  - Incremental Formulation
- Problems With Special Constraints And Variables
  - Indicator Constraints
  - General Constraints
  - Piecewise Linear Constraints
  - Special Ordered Set Of Type 1 (SOS1)
  - Special Ordered Set Of Type 2 (SOS2)
  - Semi-Continuous Variables
  - Semi-continuous Integer Variables
  - Partial Integer Variables

There are two possibilities to input a problem into the FICO Xpress optimizer from R. The problem can be either incrementally formulated, for example by specifying each row separately, or by loading the entire problem at once. The FICO Xpress R interface provides a convenience function for loading an entire problem at once, which accepts R matrices as input for the various matrices that may describe the linear and quadratic constraint parts.

The `xprs_loadproblemdata` function accepts as input a list of named properties to declare all input at once. Internally, it uses the Xpress functions `XPRSloadmip`, `XPRSloadmiqp` etc. to translate the problem data into the internal formulation. The advantage of using a problem data object is that it lets the R user input matrices instead of sparse matrix representations for declaring linear and quadratic terms in the objective and constraints.

This page illustrates the two methods of formulating optimization problems for the FICO Xpress optimizer using an example of a mixed integer problem (MIP) and a mixed-integer quadratically constrained optimization problem (MIQCQP). As to the method that loads problem data at once, different ways of constructing matrices are discussed as well.

This page also shows the input of problems that contain some special constraints or variables, such as indicator constraints and semi-continuous variables, etc. It serves as a quick guide for the different elements of optimization problems and the interface functions to declare them.

2.1 Mixed Integer Programs (MIP)

We assume familiarity with the basic concepts of mathematical programming and its constraint notation. A mixed integer program is an optimization problem of the form

$$\begin{aligned} \min \quad & cx \\ \text{s.t.} \quad & Ax\{=,\leqslant,\geqslant\}b \\ & \ell \leqslant x \leqslant u \\ & x_j \in \{0,1\} \qquad \forall j \in \mathcal{B} \\ & x_j \in \mathbb{Z} \qquad\quad \forall j \in \mathcal{I} \end{aligned}$$

2.1.1 The Problem Data Representation

We use a simple MIP example to illustrate the two methods of loading problem data into the FICO Xpress optimizer.

$$\begin{aligned} \min \quad & 2x_1 + x_2 + 3x_3 \\ \text{s.t.} \quad & 3x_1 + 2x_3 \geqslant 5 \\ & 6x_2 - 7x_3 \geqslant 8 \\ & x_1 \in \{0,1\} \\ & x_2, x_3 \in \mathbb{Z} \\ & 0 \leqslant x_2 < \infty, 0 \leqslant x_3 < \infty \end{aligned}$$

The first way to input a problem into the FICO Xpress optimizer is to load the entire data at once using the function `xprs_loadproblemdata`. To realize this, we can write the constraint coefficients into an R matrix (in different ways, see next section), and then use the `xprs_loadproblemdata` function to load all data. Here we write the matrix in the example as a dense matrix and show the usage of `xprs_loadproblemdata`.

```
# create a list to store the problem data
problemdata <- list()

# write the constraint coefficients into a dense matrix
coefmatrix <- matrix(c(3, 0, 2, 0, 6,-7), nrow = 2, byrow = TRUE)
# load the matrix into the list as row coefficients
problemdata$A <- coefmatrix

# objective coefficients
problemdata$objcoef <- c(2, 1, 3)

# right-hand-side of the constraints
problemdata$rhs <- c(5, 8)

# row sense
problemdata$rowtype <- c("G", "G")

# specify all column types, where 'B' means 'binary' and 'I' means 'integer'
problemdata$coltype <- c('B', 'I', 'I')

#specify the indices of the binary and integer variables (use 0-based indexing)
problemdata$entind <- 0:2

# specify lower bounds and upper bounds for the columns
problemdata$lb <- c(0, 0, 0)
```

```
problemdata$ub <- c(1, Inf, Inf)

# specify the column names
problemdata$colname <- c("x_1", "x_2", "x_3")

# specify the row names
problemdata$rowname <- c("row1", "row2")

# problem name displayed when the solver solves the problem
problemdata$probname <- "MIPexample"

# load the problemdata into a problem 'p'
p <- xprs_loadproblemdata(problemdata = problemdata)
print(p)
# an alternative and equivalent way to do this is:
# p <- createprob()
# xprs_loadproblemdata(p, problemdata = problemdata)
```

```
## XPRESS problem object MIPexample
##      2 rows         3 cols        4 elems
##      3 entities      0 sets        0 indicators
##      0 qelems        0 qcelems
##      0 gencons       0 pwlcons
```

```
# use  `xprs_optimize ` to solve the problem and see the results
summary(xprs_optimize(p))
print(data.frame(Variable = problemdata$colname, Value = getsolution(p)$x))
```

```
##
## Final MIP objective                  :                   8
## Final MIP bound                      :                   8
## Solution time / primaldual integral  :        0s /     0.00%
## Number of solutions found / nodes    :         2 /        0
## MIPSTATUS: 6
##   Variable Value
## 1      x_1     1
## 2      x_2     3
## 3      x_3     1
```

2.1.2 Working With R Matrices

There are several ways to represent the matrices in R according to the nature of the matrices. For low-dimensional matrices with mostly nonzero elements, we can use a dense specification as shown above, where every entry is specified.

Most high-dimensional matrices occurring in optimization problems are usually very sparse, i.e., most of the coefficients are zero. This fact is widely used in sparse matrix representations, which only store the nonzero coefficients of each column of a matrix together with an index into the rows that correspond to those nonzero entries. Using a sparse representation for such matrices saves a lot of memory. As a rule of thumb, we create sparse matrices when the problem data is too large to store every single of the ROWS $\times$ COLS coefficients, or the matrix contains mostly 0 coefficients.

There are many ways to create sparse matrices. Here we still use the same MIP example to illustrate some possible methods to construct sparse matrices.

The first method only works for small-sized matrices because at creation we actually specify every element of the matrix.

We can always convert a sparse matrix back to a dense matrix:

```
# create a sparse matrix
coefmatrix_sparse1 <-
  Matrix(c(3, 0, 2, 0, 6,-7),
         nrow = 2,
```

```
         byrow = TRUE,
         sparse = TRUE)
coefmatrix_sparse1

# a sparse matrix can be converted to a dense matrix
coefmatrix_dense <- as.matrix(coefmatrix_sparse1)
print("The sparse matrix can be converted into a dense matrix:")
coefmatrix_dense
```

```
## 2 x 3 sparse Matrix of class "dgCMatrix"
##
## [1,] 3 .  2
## [2,] . 6 -7
## [1] "The sparse matrix can be converted into a dense matrix:"
##      [,1] [,2] [,3]
## [1,]    3    0    2
## [2,]    0    6   -7
```

In the second chunk, we first create a sparse matrix containing only 0's, and then assign the nonzero elements:

```
# first create a sparse matrix containing only 0 values
coefmatrix_sparse2 <- Matrix(0, nrow = 2, ncol = 3, byrow = TRUE, sparse = TRUE,
                             doDiag = FALSE)

# then assign the nonzero values
coefmatrix_sparse2[1, c(1, 3)] <- c(3, 2)
coefmatrix_sparse2[2, 2:3] <- c(6, -7)
coefmatrix_sparse2
```

```
## 2 x 3 sparse Matrix of class "dgCMatrix"
##
## [1,] 3 .  2
## [2,] . 6 -7
```

The third one, which creates a sparse matrix in triplet format:

```
# create a sparse matirx in triplet format
coefmatrix_sparse3 <-
  sparseMatrix(
    i = c(1, 1, 2, 2),
    j = c(1, 3, 2:3),
    x = c(3, 2, 6,-7),
    dims = c(2, 3)
  )
coefmatrix_sparse3
```

```
## 2 x 3 sparse Matrix of class "dgCMatrix"
##
## [1,] 3 .  2
## [2,] . 6 -7
```

2.1.3 Incremental Formulation

Another way to input problem data into Xpress optimizer is to add the problem data incrementally. We can use the convenience functions `xprs_addrow` and `xprs_addcol`, which are exclusive to the R interface, or `addrows` and `addcols` to realize this. One difference between these two pairs of functions is that, for `xprs_addrow` and `xprs_addcol`, row type, row name, column type and column name can be specified inside these functions, whereas for `addrows` and `addcols`, row type can be defined inside function `addrows`, the other three features need to be specified using `chgcoltype` and `addnames`. Another difference is that `xprs_addrow` and `xprs_addcol` only add a single row and a single column, while `addrows` and `addcols` can add multiple rows and columns.

Of course, it is possible to mix the use of these functions to formulate problems.

Firstly we show how to use `xprs_addcol` and `xprs_addrow` to incrementally add the data.

```
# firstly, create a new empty problem 'prob1'
prob1 <- createprob()

# add the columns
xprs_addcol(prob1, lb = 0, ub = 1, coltype = 'B', name = "x_1", objcoef = 2)
xprs_addcol(prob1, lb = 0, ub = Inf, coltype = 'I', name = "x_2", objcoef = 1)
xprs_addcol(prob1, lb = 0, ub = Inf, coltype = 'I', name = "x_3", objcoef = 3)

# add the rows
xprs_addrow(prob1,
       rowtype = "G",
       rhs = 5,
       name = "row1",
       colind = c(0, 2),
       rowcoef = c(3, 2),
       )

xprs_addrow(prob1,
       rowtype = "G",
       rhs = 8,
       name = "row2",
       colind = c(1, 2),
       rowcoef = c(6, -7),
       )

# specify the name of the problem
setprobname(prob1, "MIPexample")
```

```
# show the problem created
print(prob1)
```

```
## XPRESS problem object MIPexample
##      2 rows         3 cols        4 elems
##      3 entities      0 sets        0 indicators
##      0 qelems        0 qcelems
##      0 gencons       0 pwlcons
```

Next we show how to use `addrows` and `addcols` to incrementally add the problem data.

```
# firstly, create a new empty problem 'prob2'
prob2 <- createprob()

# add the columns
addcols(prob2,
        objcoef = c(2, 1, 3),
        start = c(0, 1, 2),
        rowind = NULL,
        rowcoef = NULL,
        lb = c(0, 0, 0),
        ub = c(1, Inf, Inf))

# specify the column types
chgcoltype(prob2, c(0, 1, 2), c('B', 'I', 'I'), ncols = 3)

# specify the column names
addnames(prob2, 2, c("x_1", "x_2", "x_3"), 0, 2)

# add the rows
addrows(prob2,
       rowtype = "G",
       rhs = 5,
       start = 0,
       colind = c(0, 2),
```

```
       rowcoef = c(3, 2)
       )
addrows(prob2,
       rowtype = "G",
       rhs = 8,
       start= 0,
       colind = c(1, 2),
       rowcoef = c(6, -7)
       )

# specify the row names
addnames(prob2, 1, c("row1", "row2"), 0, 1)

# specify the name of the problem
setprobname(prob2, "MIPexample")
```

```
# show the problem created
print(prob2)
```

```
## XPRESS problem object MIPexample
##      2 rows         3 cols        4 elems
##      3 entities      0 sets        0 indicators
##      0 qelems        0 qcelems
##      0 gencons       0 pwlcons
```

2.2 Mixed Integer Quadratically Constrained Programs (MIQCQP)

Mixed Integer Quadratically Constrained Programming (MIQCQP) problems are an extension of Mixed Integer Programming (MIP) problems where the objective function and constraints may include a second order polynomial.

$$
\begin{aligned}
\min \quad & \frac{1}{2} x^t Q_0 x + cx \\
\text{s.t.} \quad & Ax + Q(x)\{=, \leqslant, \geqslant\} b \\
& \ell \leqslant x \leqslant u \\
& x_j \in \{0, 1\} \qquad \forall j \in \mathcal{B} \\
& x_j \in \mathbb{Z} \qquad \forall j \in \mathcal{I}
\end{aligned}
$$

The function $Q : \mathbb{R}^n \to \mathbb{R}^m$, $Q(x) := (x^t Q_1 x, x^t Q_2 x, \ldots, x^t Q_m x)^t$ is shorthand for the quadratic parts of the constraints.

The sets $\mathcal{B}$ and $\mathcal{I}$ denote the subsets of the columns (of A) that are restricted to binary or integer values, respectively. There are also some more advanced column types such as semi-continuous, semi-integer, and partial integer columns that are explained below.

2.2.1 The Problem Data Representation

As for MIP, there are two possibilities to input a problem into the FICO Xpress optimizer: loading at once and incremental formulation. To illustrate this, we use the example below, which extends the MIP example by some quadratic terms.

$$\begin{aligned} \min \quad & x_1^2 + 2x_3^2 + 2x_1 + x_2 - 3x_3 \\ \text{s.t.} \quad & 3x_1 + 2x_3^2 \leqslant 5 \\ & x_1^2 - 7x_3 \leqslant 8 \\ & x_1 \in \{0, 1\} \\ & x_2, x_3 \in \mathbb{Z} \\ & 0 \leqslant x_2 < \infty, 0 \leqslant x_3 < \infty \end{aligned}$$

In this example, the matrix Q_0 in the objective is:

$$Q_0 = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 4 \end{bmatrix}$$

The matrix Q_1 in the first row is:

$$Q_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

The matrix Q_2 in the second row is:

$$Q_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

The first way we load the problem into the FICO Xpress optimizer at once is to use the function `xprs_loadproblemdata`.

```
qproblemdata <- list()

# objective coefficients
qproblemdata$objcoef <- c(2, 1, -3)

# write the constraint coefficients into a dense matrix
coefmatrix <- matrix(c(3, 0, 0, 0, 0,-7), nrow = 2, byrow = TRUE)
# load the matrix into the list as row coefficients
qproblemdata$A <- coefmatrix

# right-hand-side of the constraints
qproblemdata$rhs <- c(5, 8)

# row sense
qproblemdata$rowtype <- c("L", "L")

# specify all column types, where 'B' means 'binary' and 'I' means 'integer'
qproblemdata$coltype <- c('B', 'I', 'I')

#specify the indices of the binary and integer variables (use 0-based indexing)
qproblemdata$entind <- 0:2

# specify lower bounds and upper bounds for the columns
qproblemdata$lb <- c(0, 0, 0)
qproblemdata$ub <- c(1, Inf, Inf)

# specify the column names
qproblemdata$colname <- c("x_1", "x_2", "x_3")

# specify the row names
qproblemdata$rowname <- c("row1", "row2")
```

```
# problem Name displayed when the solver solves the problem
qproblemdata$probname <- "MIQCQPexample"


# # quadratic part in objective
# # 1.C-Style notation
# qproblemdata$nobjqcoef <- 2
# qproblemdata$objqcol1 <- c(0, 2)
# qproblemdata$objqcol2 <- c(0, 2)
# qproblemdata$objqcoef <- c(2, 4)

# 2.Matrix notation
qproblemdata$Qobj <-
  Matrix(
    c(2, 0, 0, 0, 0, 0, 0, 0, 4),
    nrow = 3,
    byrow = TRUE,
    sparse = TRUE,
    doDiag = FALSE
  )


# # quadratic part in constraints
# # 1.C-Style notation
# qproblemdata$nqrows <- 2
# qproblemdata$qrowind <- c(0, 1)
# qproblemdata$nrowqcoefs <- c(1, 1)
# qproblemdata$rowqcol1 <- c(2, 0)
# qproblemdata$rowqcol2 <- c(2, 0)
# qproblemdata$rowqcoef <- c(2, 1)

# 2.Matrix notation
qproblemdata$Qrowlist <- list()
qproblemdata$Qrowlist[[1]] <-
  Matrix(
    c(0, 0, 0, 0, 0, 0, 0, 0, 2),
    nrow = 3,
    byrow = TRUE,
    sparse = TRUE,
    doDiag = FALSE
  )
qproblemdata$Qrowlist[[2]] <-
  Matrix(
    c(1, 0, 0, 0, 0, 0, 0, 0, 0),
    nrow = 3,
    byrow = TRUE,
    sparse = TRUE,
    doDiag = FALSE
  )


# load the problemdata into a problem 'qcqp'
qcqp <- xprs_loadproblemdata(problemdata = qproblemdata)
```

```
## 'as(<dsCMatrix>, "dgTMatrix")' is deprecated.
## Use 'as(as(., "generalMatrix"), "TsparseMatrix")' instead.
## See help("Deprecated") and help("Matrix-deprecated").
```

```
print(qcqp)
# an alternative and equivalent way to do this is:
# p <- createprob()
# xprs_loadproblemdata(p, problemdata=qproblemdata)
```

```
## XPRESS problem object MIQCQPexample
##      2 rows         3 cols        2 elems
##      3 entities      0 sets        0 indicators
##      2 qelems        2 qcelems
##      0 gencons       0 pwlcons
```

```
# use  `xprs_optimize ` to solve the problem and see the results
summary(xprs_optimize(qcqp))
print(data.frame(Variable = qproblemdata$colname, Value = getsolution(qcqp)$x))
```

```
##
## Final MIP objective                  :                  -1
## Final MIP bound                      :                  -1
## Solution time / primaldual integral  :        0s /     0.00%
## Number of solutions found / nodes    :         1 /        1
## MIPSTATUS: 6
##   Variable Value
## 1      x_1     0
## 2      x_2     0
## 3      x_3     1
```

2.2.2 Working With R Matrices

Like for the linear part of the constraints (matrix A) and the quadratic objective matrix and the quadratic terms in constraints can be specified as matrices in sparse or dense representation.

In the above section, we specify all the matrices as dense matrices, and now we show how to construct them as sparse matrices.

The first way:

```
# create sparse matrix
qobjmat_sparse1 <-
  Matrix(
    c(2, 0, 0, 0, 0, 0, 0, 0, 4),
    nrow = 3,
    byrow = TRUE,
    sparse = TRUE
  )
qobjmat_sparse1

qrowmat1_sparse1 <-
  Matrix(
    c(0, 0, 0, 0, 0, 0, 0, 0, 2),
    nrow = 3,
    byrow = TRUE,
    sparse = TRUE
  )
qrowmat1_sparse1

qrowmat2_sparse1 <-
  Matrix(
    c(1, 0, 0, 0, 0, 0, 0, 0, 0),
    nrow = 3,
    byrow = TRUE,
    sparse = TRUE
  )
qrowmat2_sparse1
```

```
## 3 x 3 diagonal matrix of class "ddiMatrix"
##      [,1] [,2] [,3]
## [1,]    2    .    .
## [2,]    .    0    .
## [3,]    .    .    4
## 3 x 3 diagonal matrix of class "ddiMatrix"
##      [,1] [,2] [,3]
## [1,]    0    .    .
## [2,]    .    0    .
## [3,]    .    .    2
## 3 x 3 diagonal matrix of class "ddiMatrix"
##      [,1] [,2] [,3]
## [1,]    1    .    .
```

```
## [2,]    .    0    .
## [3,]    .    .    0
```

The second way:

```
# first create sparse matrices containing only 0 values
qobjmat_sparse2 <-
  Matrix(
    0,
    nrow = 3,
    ncol = 3,
    byrow = TRUE,
    sparse = TRUE,
    doDiag = FALSE
  )
qrowmat1_sparse2 <-
  Matrix(
    0,
    nrow = 3,
    ncol = 3,
    byrow = TRUE,
    sparse = TRUE,
    doDiag = FALSE
  )
qrowmat2_sparse2 <-
  Matrix(
    0,
    nrow = 3,
    ncol = 3,
    byrow = TRUE,
    sparse = TRUE,
    doDiag = FALSE
  )

# then assign the nonzero values
qobjmat_sparse2[1, 1] <- 2
qobjmat_sparse2[3, 3] <- 4

qrowmat1_sparse2[3, 3] <- 2

qrowmat2_sparse2[1, 1] <- 1

qobjmat_sparse2
qrowmat1_sparse2
qrowmat2_sparse2
```

```
## 3 x 3 sparse Matrix of class "dsCMatrix"
##
## [1,] 2 . .
## [2,] . . .
## [3,] . . 4
## 3 x 3 sparse Matrix of class "dsCMatrix"
##
## [1,] . . .
## [2,] . . .
## [3,] . . 2
## 3 x 3 sparse Matrix of class "dsCMatrix"
##
## [1,] 1 . .
## [2,] . . .
## [3,] . . .
```

The third way:

```
# create sparse matrices in triplet format
qobjmat_sparse3 <-
  sparseMatrix(
```

```
    i = c(1, 3),
    j = c(1, 3),
    x = c(2, 4),
    dims = c(3, 3)
  )
qobjmat_sparse3

qrowmat1_sparse3 <-
  sparseMatrix(
    i = 3,
    j = 3,
    x = 2,
    dims = c(3, 3)
  )
qrowmat1_sparse3

qrowmat2_sparse3 <-
  sparseMatrix(
    i = 1,
    j = 1,
    x = 1,
    dims = c(3, 3)
  )
qrowmat2_sparse3
```

```
## 3 x 3 sparse Matrix of class "dgCMatrix"
##
## [1,] 2 . .
## [2,] . . .
## [3,] . . 4
## 3 x 3 sparse Matrix of class "dgCMatrix"
##
## [1,] . . .
## [2,] . . .
## [3,] . . 2
## 3 x 3 sparse Matrix of class "dgCMatrix"
##
## [1,] 1 . .
## [2,] . . .
## [3,] . . .
```

2.2.3 Incremental Formulation

We can also formulate the MIQCQP example incrementally, and same as MIP, we can use the functions `xprs_addrow` and `xprs_addcol`, or `addrows` and `addcols` to realize this. The differences between these two pairs of functions are described in the MIP section. Here, we only show the usage of the convenience functions `xprs_addrow` and `xprs_addcol`.

As for the quadratic terms, we always use `chgmqobj` to add quadratic terms to the objective and `addqmatrix` to add quadratic terms to constraints.

```
# firstly, create a new empty problem 'qcqp1'
qcqp1 <- createprob()

# add the columns
xprs_addcol(qcqp1, lb = 0, ub = 1, coltype = 'B', name = "x_1", objcoef = 2)
xprs_addcol(qcqp1, lb = 0, ub = Inf, coltype = 'I', name = "x_2", objcoef = 1)
xprs_addcol(qcqp1, lb = 0, ub = Inf, coltype = 'I', name = "x_3", objcoef = -3)

# add the quadratic terms in objective
# since the quadratic matrix is assumed to be symmetric, so specifying the upper
# diagonal part of the matrix is enough, and the off-diagonal coefficients can be
# specified as they are.
chgmqobj(qcqp1, c(0, 2), c(0, 2), c(2, 4), 2)

# add the rows
```

```
xprs_addrow(qcqp1,
       rowtype = "L",
       rhs = 5,
       name = "row1",
       colind = 0,
       rowcoef = 3,
       )
xprs_addrow(qcqp1,
       rowtype = "L",
       rhs = 8,
       name = "row2",
       colind = 2,
       rowcoef = -7,
       )

# add quadratic terms in the constraints
addqmatrix(qcqp1,
           row = 0,
           rowqcol1 = 2,
           rowqcol2 = 2,
           rowqcoef = 2
           )
addqmatrix(qcqp1,
           row = 1,
           rowqcol1 = 0,
           rowqcol2 = 0,
           rowqcoef = 1
           )

# specify the name of the problem
setprobname(qcqp1, "MIQCQPexample")
```

```
# show the problem created
print(qcqp1)
```

```
## XPRESS problem object MIQCQPexample
##      2 rows         3 cols        2 elems
##      3 entities      0 sets        0 indicators
##      2 qelems        2 qcelems
##      0 gencons       0 pwlcons
```

```
# use  `xprs_optimize ` to solve the problem and see the results
summary(xprs_optimize(qcqp1))
print(data.frame(Variable=c("x_1", "x_2", "x_3"), Value=getsolution(qcqp1)$x))
```

```
##
## Final MIP objective                  :                  -1
## Final MIP bound                      :                  -1
## Solution time / primaldual integral  :        0s /     0.00%
## Number of solutions found / nodes    :         1 /        1
## MIPSTATUS: 6
##   Variable Value
## 1      x_1     0
## 2      x_2     0
## 3      x_3     1
```

2.3 Problems With Special Constraints And Variables

In this section, we display some examples that contain special constraints or variables and show how to input them into the FICO Xpress optimizer. To see how these special constraints or variables can have effect on the solutions, we use examples based on the MIP example so that we can compare the new solutions obtained with the MIP solution. For convenience, we create a function `load_MIP_example` to load the MIP example and we will use this function at the begining of each sub-section and then make adjustments on the MIP example incrementally (adding constraints or changing variable types, etc.).

```
load_MIP_example = function(){
  # firstly, create a new empty problem 'prob'
  prob <- createprob()

  # add the columns
  xprs_addcol(prob, lb = 0, ub = 1, coltype = 'B', name = "x_1", objcoef = 2)
  xprs_addcol(prob, lb = 0, ub = Inf, coltype = 'I', name = "x_2", objcoef = 1)
  xprs_addcol(prob, lb = 0, ub = Inf, coltype = 'I', name = "x_3", objcoef = 3)

  # add the rows
  xprs_addrow(prob,
              rowtype = "G",
              rhs = 5,
              name = "row1",
              colind = c(0, 2),
              rowcoef = c(3, 2),
              )

  xprs_addrow(prob,
              rowtype = "G",
              rhs = 8,
              name = "row2",
              colind = c(1, 2),
              rowcoef = c(6, -7),
              )
  # return 'prob'
  prob
}
```

Notice that for the constraints and special variables that will be discussed in this section, 'Indicator Constraints', 'General Constraints' and 'Piecewise Linear Constraints' cannot be specified when using the function `xprs_loadproblemdata` to load the problem at once. Instead, they need to be added using corresponding functions.

2.3.1 Indicator Constraints

Indicator constraints are constraints each with a specified associated binary 'controlling' variable where we assume the constraint must be satisfied when the binary variable is at a specified binary value; otherwise the constraint does not need to be satisfied.

We use an extension of the MIP example to show the formulation of problems with indicator constraints. Based on the MIP example, we add the constraint that if the binary variable x_1 takes value 1, then integer variable x_3 must be 0. Mathematically, we want to impose the implication:

$$x_1 = 1 \Rightarrow x_3 = 0$$

To add this constraint, we first add a new row $x_3 = 0$, and then use the function `setindicators` to specify the indicator constraint.

```
# firstly, load the MIP example to a problem 'probIndicator'
probIndicator <- load_MIP_example()

# add the new row x_3 = 0
xprs_addrow(probIndicator,
       rowtype = "E",
       rhs = 0,
       name = "row3",
       colind = 2,
       rowcoef = 1,
       )

# add the indicator implication between variable x_1 and the new row
setindicators(probIndicator,
```

```
  rowind = 2,
  colind = 0,
  complement = 1 # row is active when x_1 = 1
  )

# specify the name of the problem
setprobname(probIndicator, "IndicatorConstraintExample")
```

Now we solve this example. We see that the solution satisfies the indicator constraint, where x_1 takes 0 and thus x_3 can take nonzero value 3.

```
# show the problem created
print(probIndicator)
summary(xprs_optimize(probIndicator))
print(data.frame(
  Variable = c("x1", "x2", "x3"),
  Value = getsolution(probIndicator)$x
))
```

```
## XPRESS problem object IndicatorConstraintExample
##      3 rows         3 cols        5 elems
##      3 entities      0 sets        1 indicators
##      0 qelems        0 qcelems
##      0 gencons       0 pwlcons
##
## Final MIP objective                  :                  14
## Final MIP bound                      :                  14
## Solution time / primaldual integral  :        0s /     0.00%
## Number of solutions found / nodes    :         1 /        1
## MIPSTATUS: 6
##   Variable Value
## 1       x1     0
## 2       x2     5
## 3       x3     3
```

2.3.2 General Constraints

General constraints are specific type of MIP constraints to model `min`, `max`, `and`, `or`, and `absolute value` relationships between two or more variables.

Here we still use the MIP example and add a general constraint that x_3 takes the minimum value between 2 and x_2, mathematically, we require:

$$x_3 = \min\{x_2, 2\}$$

We use the function `addgencons` to add this general constraint:

```
# firstly, load the MIP example to a problem 'probGeneral'
probGeneral <- load_MIP_example()

# set the general constraint:
addgencons(probGeneral,
           contype = 1,
           resultant = 2,
           colstart = 0,
           colind = 1,
           valstart = 0,
           val = 2)

# specify the name of the problem
setprobname(probGeneral, "GeneralConstraintExample")
```

Now we solve this example. We see that the solution satisfies the general constraint, because x_3 takes

the value $\min\{x_2 = 4,\ 2\} = 2$.

```
# show the problem created
print(probGeneral)
summary(xprs_optimize(probGeneral))
print(data.frame(
  Variable = c("x1", "x2", "x3"),
  Value = getsolution(probGeneral)$x
))
```

```
## XPRESS problem object GeneralConstraintExample
##      2 rows         3 cols        4 elems
##      3 entities      0 sets        0 indicators
##      0 qelems        0 qcelems
##      1 gencons       0 pwlcons
##
## Final MIP objective                  :                  12
## Final MIP bound                      :                  12
## Solution time / primaldual integral  :        0s /     0.00%
## Number of solutions found / nodes    :         1 /        1
## MIPSTATUS: 6
##   Variable Value
## 1       x1     1
## 2       x2     4
## 3       x3     2
```

2.3.3 Piecewise Linear Constraints

Piecewise linear constraints are constraints that define a piecewise linear relationship between two variables. These are defined via a set of breakpoints with linearly interpolated values between and beyond them (with the slope before the first and after the last point continuing the slope between the first/last two points). The piece-wise linear functions are allowed to be discontinuous by defining multiple points with the same value of the input variable x, in which case the output variable y is allowed to take any value between the corresponding y-values of these breakpoints, while the first of them will define the slope before and the last will define the slope after this x-value.

Based on the MIP example, we add a piecewise linear constraint between x_2 and x_3: $x_2 = f(x_3)$, where:

$$x_2 = f(x_3) = \begin{cases} 2x_3, & \text{if } 0 \leqslant x_3 \leqslant 2 \\ \frac{1}{2}x_3 + 3, & \text{if } 2 < x_3 < \infty \end{cases}$$

This function can be defined using the breakpoints `(0, 0)`, `(2, 4)`, and `(4, 5)`. Note that the last breakpoint could also be replaced, e.g., by `(3, 4.5)`. We will use the function `addpwlcons` to add this piecewise linear constraint.

```
# firstly, load the MIP example to a problem 'probPWL'
probPWL <- load_MIP_example()

# set the piecewise linear constraint:
addpwlcons(probPWL,
           colind = 2,
           resultant = 1,
           start = 0,
           xval = c(0, 2, 4),
           yval = c(0, 4, 5))

# specify the name of the problem
setprobname(probPWL, "PWLExample")
```

Now we solve this example. We see that the solution satisfies the piecewise linear constraint, where $x_3 = 2$ and thus $x_2 = 2x_3 = 4$.

```
# show the problem created
print(probPWL)
summary(xprs_optimize(probPWL))
print(data.frame(
  Variable = c("x1", "x2", "x3"),
  Value = getsolution(probPWL)$x
))
```

```
## XPRESS problem object PWLExample
##      2 rows         3 cols        4 elems
##      3 entities      0 sets        0 indicators
##      0 qelems        0 qcelems
##      0 gencons       1 pwlcons
##
## Final MIP objective                  :                  12
## Final MIP bound                      :                  12
## Solution time / primaldual integral  :        0s /     0.00%
## Number of solutions found / nodes    :         1 /        1
## MIPSTATUS: 6
##   Variable Value
## 1       x1     1
## 2       x2     4
## 3       x3     2
```

2.3.4 Special Ordered Set Of Type 1 (SOS1)

SOS1 is a set of decision variables ordered by a set of specified continuous values (or reference values) of which at most one can take a nonzero value. Note that SOS constraints must be input in sparse row representation.

We illustrate the input of SOS1 by an extension of the MIP example. Here we only allow one of the three decision variables to assume a nonzero value by imposing the restriction

$$\text{SOS1}\,(x_1, x_2, x_3)$$

A mathematically equivalent formulation of this SOS1 restriction requires auxiliary binary variables. We introduce one auxiliary binary variables for each decision variable to indicate whether they are 0 or nonzero. We restrict the sum of these auxiliary binary variables to 1 to ensure only one variable takes nonzero value. Since x_1 itself is a binary variable, we only need two binary variables a_2 and a_3 for x_2 and x_3 such that:

$$x_j = 0 \Leftrightarrow a_j = 0 \quad \text{for } j \in \{2, 3\}$$

Note that this relationship can only be imposed through a linear constraint if the variable x_j has a finite upper bound.

Finally, we restrict the sum of these binary variables:

$$x_1 + a_2 + a_3 \leqslant 1$$

If we add this constraint to the original MIP example, the problem will be infeasible. So we change the row coefficient for x_3 in the second row "row2" from '-7' to '7', and keep everything else the same.

SOS-type constraints are called "Sets" in Xpress terminology. To add this SOS1 restriction, we use the function `addsets`.

```
# firstly, load the MIP example to a problem 'probSOS1'
probSOS1 <- load_MIP_example()

# change the row coefficient of x_3 in the second row from '-7' to 7
chgcoef(probSOS1, 1, 2, 7)
```

```
# add the SOS1 set
addsets(probSOS1,
        settype = '1',
        start = 0,
        colind = c(0, 1, 2),
        refval = c(1, 2, 3))

# specify the name of the problem
setprobname(probSOS1, "SOS1Example")
```

Now we solve this example. We see that the solution satisfies the SOS1 restriction, where only x_3 takes nonzero value 3.

```
# show the problem created
print(probSOS1)
summary(xprs_optimize(probSOS1))
print(data.frame(
  Variable = c("x1", "x2", "x3"),
  Value = getsolution(probSOS1)$x
))
```

```
## XPRESS problem object SOS1Example
##      2 rows         3 cols        4 elems
##      3 entities      1 sets        0 indicators
##      0 qelems        0 qcelems
##      0 gencons       0 pwlcons
##
## Final MIP objective                  :                    9
## Final MIP bound                      :                    9
## Solution time / primaldual integral  :        0s /     0.00%
## Number of solutions found / nodes    :         1 /        1
## MIPSTATUS: 6
##   Variable Value
## 1       x1     0
## 2       x2     0
## 3       x3     3
```

SOS1 can also be specified when loading the problem at once, so next we will show how to add SOS1 when using function `xprs_loadproblemdata`.

```
# create a list to store the problem data
pSOS1data <- list()

# write the constraint coefficients into a dense matrix
coefmatrix <- matrix(c(3, 0, 2, 0, 6, 7), nrow = 2, byrow = TRUE)
# load the matrix into the list as row coefficients
pSOS1data$A <- coefmatrix

# objective coefficients
pSOS1data$objcoef <- c(2, 1, 3)

# right-hand-side of the constraints
pSOS1data$rhs <- c(5, 8)

# row sense
pSOS1data$rowtype <- c("G", "G")

# specify all column types, where 'B' means 'binary' and 'I' means 'integer'
pSOS1data$coltype <- c('B', 'I', 'I')

#specify the indices of the binary and integer variables (use 0-based indexing)
pSOS1data$entind <- 0:2

# specify lower bounds and upper bounds for the columns
pSOS1data$lb <- c(0, 0, 0)
pSOS1data$ub <- c(1, Inf, Inf)
```

```
# specify the column names
pSOS1data$colname <- c("x_1", "x_2", "x_3")

# specify the row names
pSOS1data$rowname <- c("row1", "row2")

# problem Name displayed when the solver solves the problem
pSOS1data$probname <- "SOS1Example"

# specify SOS1
pSOS1data$settype <- '1'
pSOS1data$setstart <- c(0, 3)
pSOS1data$setind <- c(0, 1, 2)
pSOS1data$refval <- c(1, 2, 3)

# load the problemdata into a problem 'pSOS1'
pSOS1 <- xprs_loadproblemdata(problemdata=pSOS1data)
print(pSOS1)
```

```
## XPRESS problem object SOS1Example
##      2 rows         3 cols        4 elems
##      3 entities      1 sets        0 indicators
##      0 qelems        0 qcelems
##      0 gencons       0 pwlcons
```

We solve this example and we get the same solution as before.

```
# use  `xprs_optimize ` to solve the problem and see the results
summary(xprs_optimize(pSOS1))
print(data.frame(Variable = pSOS1data$colname, Value = getsolution(pSOS1)$x))
```

```
##
## Final MIP objective                  :                    9
## Final MIP bound                      :                    9
## Solution time / primaldual integral  :        0s /     0.00%
## Number of solutions found / nodes    :         1 /        1
## MIPSTATUS: 6
##   Variable Value
## 1      x_1     0
## 2      x_2     0
## 3      x_3     3
```

2.3.5 Special Ordered Set Of Type 2 (SOS2)

SOS2 is a set of variables ordered by a set of specified continuous values (or reference values) of which at most two can be nonzero, and if two are nonzero then they must be consecutive in their ordering. Note that SOS constraints must be input in sparse row representation.

We illustrate the input of SOS2 by an extension of the MIP example as well, here we only allow two consecutive decision variables to take nonzero values and we change the upper bounds of x_2 and x_3 both from infinity to 10.

As for SOS1, an alternative, mathematically equivalent representation of an SOS2 constraint, requires auxiliary binary variables a_2 and a_3, of which at most can be set to 1:

$$x_1 + a_2 + a_3 \leqslant 1$$

But this time, we set the constraints for each variable in this way:

$$\begin{aligned} x_1 &\leqslant x_1 \cdot 1 \\ x_2 &\leqslant (x_1 + a_2) \cdot 10 \\ x_3 &\leqslant (a_2 + a_3) \cdot 10 \end{aligned}$$

So for example, if the constraint enforces a_2 to be 1, then x_1 must be 0, and x_2 and x_3 are allowed to take nonzero values, which satisfy the SOS2 restriction.

To add this SOS2 restriction, we use again the function `addsets` with the appropriate set type.

```
# firstly, load the MIP example to a problem 'probSOS2'
probSOS2 <- load_MIP_example()

# change the upper bounds for both x_2 and x_3
chgbounds(probSOS2,
          colind = c(1, 2),
          bndtype = c('U', 'U'),
          bndval = c(10, 10))

# add the SOS2 set
addsets(probSOS2,
        settype = '2',
        start = 0,
        colind = c(0, 1, 2),
        refval = c(1, 2, 3))

# specify the name of the problem
setprobname(probSOS2, "SOS2Example")
```

Now we solve this example. We see that the solution satisfies the SOS2 restriction, where only x_2 and x_3 take nonzero values 5 and 3.

```
# show the problem created
print(probSOS2)
summary(xprs_optimize(probSOS2))
print(data.frame(
  Variable = c("x_1", "x_2", "x_3"),
  Value = getsolution(probSOS2)$x
))
```

```
## XPRESS problem object SOS2Example
##      2 rows         3 cols        4 elems
##      3 entities      1 sets        0 indicators
##      0 qelems        0 qcelems
##      0 gencons       0 pwlcons
##
## Final MIP objective                   :                   14
## Final MIP bound                       :                   14
## Solution time / primaldual integral   :        0s /     0.00%
## Number of solutions found / nodes     :         1 /        1
## MIPSTATUS: 6
##   Variable Value
## 1      x_1     0
## 2      x_2     5
## 3      x_3     3
```

SOS2 can also be specified when loading the problem at once, so next we will show how to add SOS2 when using function `xprs_loadproblemdata`.

```
# create a list to store the problem data
pSOS2data <- list()

# write the constraint coefficients into a dense matrix
coefmatrix <- matrix(c(3, 0, 2, 0, 6,-7), nrow = 2, byrow = TRUE)
# load the matrix into the list as row coefficients
pSOS2data$A <- coefmatrix

# objective coefficients
pSOS2data$objcoef <- c(2, 1, 3)

# right-hand-side of the constraints
```

```
pSOS2data$rhs <- c(5, 8)

# row sense
pSOS2data$rowtype <- c("G", "G")

# specify all column types, where 'B' means 'binary' and 'I' means 'integer'
pSOS2data$coltype <- c('B', 'I', 'I')

#specify the indices of the binary and integer variables (use 0-based indexing)
pSOS2data$entind <- 0:2

# specify lower bounds and upper bounds for the columns
pSOS2data$lb <- c(0, 1, 1)
pSOS2data$ub <- c(1, 10, 10)

# specify the column names
pSOS2data$colname <- c("x_1", "x_2", "x_3")

# specify the row names
pSOS2data$rowname <- c("row1", "row2")

# problem Name displayed when the solver solves the problem
pSOS2data$probname <- "SOS2Example"

# specify SOS2
pSOS2data$settype <- '2'
pSOS2data$setstart <- c(0, 3)
pSOS2data$setind <- c(0, 1, 2)
pSOS2data$refval <- c(1, 2, 3)

# load the problemdata into a problem 'p'
pSOS2 <- xprs_loadproblemdata(problemdata = pSOS2data)
print(pSOS2)
```

```
## XPRESS problem object SOS2Example
##      2 rows         3 cols        4 elems
##      3 entities      1 sets        0 indicators
##      0 qelems        0 qcelems
##      0 gencons       0 pwlcons
```

We solve this example and we get the same solution as before.

```
# use  `xprs_optimize ` to solve the problem and see the results
summary(xprs_optimize(pSOS2))
print(data.frame(
  Variable = c("x_1", "x_2", "x_3"),
  Value = getsolution(pSOS2)$x
))
```

```
##
## Final MIP objective                  :                  14
## Final MIP bound                      :                  14
## Solution time / primaldual integral  :        0s /     0.00%
## Number of solutions found / nodes    :         1 /        1
## MIPSTATUS: 6
##   Variable Value
## 1      x_1     0
## 2      x_2     5
## 3      x_3     3
```

2.3.6 Semi-Continuous Variables

Semi-continuous variables are decision variables that either have value 0, or a continuous value above a specified nonnegative limit. To show how we specify semi-continuous variables when formulating problems, we use the MIP example but change the variable type of x_3 from integer to semi-continuous,

and set the nonzero limit of x_3 as 2:

$$\begin{aligned} \min \quad & 2x_1 + x_2 + 3x_3 \\ \text{s.t.} \quad & 3x_1 + 2x_3 \geqslant 5 \\ & 6x_2 - 7x_3 \geqslant 8 \\ & x_1 \in \{0, 1\} \\ & x_2 \in \mathbb{Z}\,, 0 \leqslant x_2 < \infty \\ & x_3 \in \mathcal{S},\ x_3 = 0 \text{ or } 2 \leqslant x_3 < \infty \text{ and } x_3 \in \mathbb{R} \end{aligned}$$

Firstly we load the MIP example, then use `chgcoltype` to change the type of x_3 and use `chglblimit` to specify the nonzero limit of x_3.

```
# firstly, load the MIP example to a problem 'problemS'
problemS <- load_MIP_example()

# change the type of x_3
chgcoltype(problemS, 2, 'S')

# set the lower bound for semi-continuous variable x_3
chgglblimit(problemS, 2, 2)

# specify the name of the problem
setprobname(problemS, "MIPExampleWithS")
```

Notice that for setting a semi-continuous variable, we can directly declare its type as 'S' when adding the column. Or when we incrementally formulate the problem, we can declare a semi-continuous variable by function `xprs_addcol` or `adcols` and set 'coltype' as 'S'.

Now we solve this example. We see that the solution satisfies the semi-continuous restriction on x_3, whose value changed from 1 to 2.

```
# show the problem created
print(problemS)
summary(xprs_optimize(problemS))
print(data.frame(
  Variable = c("x_1", "x_2", "x_3"),
  Value = getsolution(problemS)$x
))
```

```
## XPRESS problem object MIPExampleWithS
##      2 rows         3 cols        4 elems
##      3 entities      0 sets        0 indicators
##      0 qelems        0 qcelems
##      0 gencons       0 pwlcons
##
## Final MIP objective                  :                  12
## Final MIP bound                      :                  12
## Solution time / primaldual integral  :        0s /     0.00%
## Number of solutions found / nodes    :         1 /        1
## MIPSTATUS: 6
##   Variable Value
## 1      x_1     1
## 2      x_2     4
## 3      x_3     2
```

More functions to interact with semi-continuous variables are: `chgbounds`, `chgcoef`, `chgobj`, `delcols` and `getcoltype`.

2.3.7 Semi-continuous Integer Variables

Semi-continuous integer variables are decision variables that either have value 0, or an integer value above a specified nonnegative limit. To show how we specify semi-continuous integer when formulating problems, we use the MIP example but change the variable type of x_3 from integer to semi-continuous integer, and set the nonzero limit of x_3 as 2:

$$\begin{aligned} \min \quad & 2x_1 + x_2 + 3x_3 \\ \text{s.t.} \quad & 3x_1 + 2x_3 \geqslant 5 \\ & 6x_2 - 7x_3 \geqslant 8 \\ & x_1 \in \{0, 1\} \\ & x_2 \in \mathbb{Z}\,, 0 \leqslant x_2 < \infty \\ & x_3 \in \mathcal{R},\ x_3 = 0 \text{ or } 2 \leqslant x_3 < \infty \text{ and } x_3 \in \mathbb{Z} \end{aligned}$$

Firstly, we load the MIP example, then use `chgcoltype` to change the type of x_3 and use `chglblimit` to specify the nonzero limit of x_3.

```
# firstly, load the MIP example to a problem 'problemR'
problemR <- load_MIP_example()

# change the type of x_3
chgcoltype(problemR, 2, 'R')

# set the lower bound for semi-continuous integer variable x_3
chgglblimit(problemR, 2, 2)

# specify the name of the problem
setprobname(problemR, "MIPExampleWithR")
```

Notice that for setting a semi-continuous integer variable, we can directly declare its type as 'R' when adding the column. Or when we incrementally formulate the problem, we can declare a semi-continuous variable by function `xprs_addcol` or `adcols` and set 'coltype' as 'R'.

Now we solve this example. We see that the solution satisfies the semi-continuous integer restriction on x_3, whose value changed from 1 to 2.

```
# show the problem created
print(problemR)
summary(xprs_optimize(problemR))
print(data.frame(
  Variable = c("x_1", "x_2", "x_3"),
  Value = getsolution(problemR)$x
))
```

```
## XPRESS problem object MIPExampleWithR
##      2 rows         3 cols        4 elems
##      3 entities      0 sets        0 indicators
##      0 qelems        0 qcelems
##      0 gencons       0 pwlcons
##
## Final MIP objective                  :                  12
## Final MIP bound                      :                  12
## Solution time / primaldual integral  :        0s /     0.00%
## Number of solutions found / nodes    :         1 /        1
## MIPSTATUS: 6
##   Variable Value
## 1      x_1     1
## 2      x_2     4
## 3      x_3     2
```

More functions to interact with semi-continuous integer variables are: `chgbounds`, `chgcoef`, `chgobj`, `delcols`, and `getcoltype`.

2.3.8 Partial Integer Variables

Partial integer variables are decision variables that have integer values below a specified limit and continuous values above the limit. To show how we specify partial integer variables when formulating problems, we use a modification of the MIP example where we change the variable type of x_2 from integer to partial integer, and set the specified limit of x_2 as 2:

$$\begin{aligned} \min \quad & 2x_1 + x_2 + 3x_3 \\ \text{s.t.} \quad & 3x_1 + 2x_3 \geqslant 5 \\ & 6x_2 - 7x_3 \geqslant 8 \\ & x_1 \in \{0, 1\} \\ & x_2 \in \mathcal{P}, \ x_2 \in \mathbb{Z} \ \text{ if } 0 \leqslant x_2 \leqslant 2, \text{ and } \ x_2 \in \mathbb{R} \text{ if } x_2 > 2 \\ & x_3 \in \mathbb{Z}, \ 0 \leqslant x_3 < \infty \end{aligned}$$

Firstly, we load the MIP example, then use `chgcoltype` to change the type of x_2 and use `chglblimit` to specify the nonzero limit of x_2.

```
# firstly, load the MIP example to a  problem 'problemP'
problemP <- load_MIP_example()

# change the type of x_2
chgcoltype(problemP, 1, 'P')

# set the specified limit for partial integer variable x_2
chgglblimit(problemP, 1, 2)

# specify the name of the problem
setprobname(problemP, "MIPExampleWithP")
```

Notice that for setting a partial integer variable, we can directly declare its type as 'P' when adding the column. Or when we incrementally formulate the problem, we can declare a partial integer variable by function `xprs_addcol` or `adcols` and set 'coltype' as 'P'.

Now we solve this example. We see that the solution satisfies the partial integer restriction on x_2, whose value changed from 3 to 2.5.

```
# show the problem created
print(problemP)
summary(xprs_optimize(problemP))
print(data.frame(
  Variable = c("x_1", "x_2", "x_3"),
  Value = getsolution(problemP)$x
))
```

```
## XPRESS problem object MIPExampleWithP
##      2 rows         3 cols        4 elems
##      3 entities      0 sets        0 indicators
##      0 qelems        0 qcelems
##      0 gencons       0 pwlcons
##
## Final MIP objective                  :                 7.5
## Final MIP bound                      :                 7.5
## Solution time / primaldual integral  :        0s /     0.00%
## Number of solutions found / nodes    :         3 /        1
## MIPSTATUS: 6
```

```
##   Variable Value
## 1      x_1   1.0
## 2      x_2   2.5
## 3      x_3   1.0
```

More functions to interact with partial integer variables are: `chgbounds`, `chgobj`, `chgcoef`, `delcols`, and `getcoltype`.

CHAPTER 3

R interface reference manual

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

addcbafterobjective

Purpose

Declares a callback which will be called after each objective in a multi-objective problem is solved.

Synopsis

```
addcbafterobjective(prob, cb, prio = 0)
```

Arguments

`prob` The problem pointer to which the callback is added.

`cb` The callback function to add.

`prio` The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The afterobjective callback supports the following input arguments (in this order):

`cbprob` The problem passed to the callback function, `afterobjective`.

The afterobjective callback does not have any output arguments.

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.

Synopsis

```
addcbbariteration(prob, cb, prio = 0)
```

Arguments

`prob` The problem pointer to which the callback is added.

`cb` The callback function to add.

`prio` The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The bariteration callback supports the following input arguments (in this order):

`cbprob` The problem passed to the callback function, `bariteration`.

The bariteration callback supports the following output arguments:

`ret` An integer that will interrupt the search if non-zero.

`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.

addcbbarlog

Purpose

Declares a barrier log callback function, called at each iteration during the interior point algorithm.

Synopsis

```
addcbbarlog(prob, cb, prio = 0)
```

Arguments

`prob`	The problem pointer to which the callback is added.
`cb`	The callback function to add.
`prio`	The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The barlog callback supports the following input arguments (in this order):

`cbprob`	The problem passed to the callback function, `barlog`.

The barlog callback supports the following output arguments:

`ret`	An integer that will interrupt the search if non-zero.

addcbbeforeobjective

Purpose

Declares a callback which will be called before each objective in a multi-objective problem is solved.

Synopsis

```
addcbbeforeobjective(prob, cb, prio = 0)
```

Arguments

`prob` The problem pointer to which the callback is added.

`cb` The callback function to add.

`prio` The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The beforeobjective callback supports the following input arguments (in this order):

`cbprob` The problem passed to the callback function, `beforeobjective`.

The beforeobjective callback does not have any output arguments.

addcbchecktime

Purpose

Declares a callback function which is called every time the Optimizer checks if the time limit has been reached.

Synopsis

```
addcbchecktime(prob, cb, prio = 0)
```

Arguments

`prob` The problem pointer to which the callback is added.

`cb` The callback function to add.

`prio` The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The checktime callback supports the following input arguments (in this order):

`cbprob` The problem passed to the callback function, `checktime`.

The checktime callback supports the following output arguments:

`ret` An integer that will interrupt the search if non-zero.

addcbchgbranchobject

Purpose

Declares a callback function that will be called after the selection of a MIP entity to branch on.

Synopsis

```
addcbchgbranchobject(prob, cb, prio = 0)
```

Arguments

`prob` The problem pointer to which the callback is added.

`cb` The callback function to add.

`prio` The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The chgbranchobject callback supports the following input arguments (in this order):

`cbprob` The problem passed to the callback function, `chgbranchobject`.

`branch` The candidate branching data selected by the Optimizer. Will be `NULL` if no candidates exist.

The chgbranchobject callback supports the following output arguments:

`ret` An integer that will interrupt the search if non-zero.

`newbranch` Optional new branching data to replace the Optimizer's selection.

addcbcomputerestart

Purpose

Declares a callback to be called when a solve executed in compute mode needs to be restarted.

Synopsis

```
addcbcomputerestart(prob, cb, prio = 0)
```

Arguments

`prob` The problem pointer to which the callback is added.

`cb` The callback function to add.

`prio` The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The computerestart callback supports the following input arguments (in this order):

`cbprob` The problem passed to the callback function, `computerestart`.

The computerestart callback does not have any output arguments.

addcbcutlog

Purpose

Declares a cut log callback function, called each time the cut log is printed.

Synopsis

```
addcbcutlog(prob, cb, prio = 0)
```

Arguments

`prob`	The problem pointer to which the callback is added.
`cb`	The callback function to add.
`prio`	The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The cutlog callback supports the following input arguments (in this order):

`cbprob`	The problem passed to the callback function, `cutlog`.

The cutlog callback supports the following output arguments:

`ret`	An integer that will interrupt the search if non-zero.

addcbcutround

Purpose

Declares a callback function that is called when the Optimizer could separate cutting planes during the branch and bound search.

Synopsis

```
addcbcutround(prob, cb, prio = 0)
```

Arguments

`prob` The problem pointer to which the callback is added.

`cb` The callback function to add.

`prio` The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The cutround callback supports the following input arguments (in this order):

`cbprob` The problem passed to the callback function, `cutround`.

`ifxpresscuts` An integer set to 1 if the Optimizer will apply a round of cuts after this callback. 0 otherwise.

`action` An integer return value that specifies the action the Optimizer should take:

`-1`	Continue unchanged. The default action.
`0`	No further rounds of cuts should be applied on this node.
`1`	The Optimizer should apply one more round of cutting, regardless of the value of `ifxpresscuts`
`2`	The Optimizer should process any changes applied during this callback and fire the callback again, but skip any Optimizer cutting.

The cutround callback supports the following output arguments:

`ret` An integer that will interrupt the search if non-zero.

`action` An integer return value that specifies the action the Optimizer should take:

`-1`	Continue unchanged. The default action.
`0`	No further rounds of cuts should be applied on this node.
`1`	The Optimizer should apply one more round of cutting, regardless of the value of `ifxpresscuts`
`2`	The Optimizer should process any changes applied during this callback and fire the callback again, but skip any Optimizer cutting.

addcbdestroymt

Purpose

Declares a destroy MIP thread callback function, called every time a MIP thread is destroyed by the parallel MIP code.

Synopsis

```
addcbdestroymt(prob, cb, prio = 0)
```

Arguments

`prob` The problem pointer to which the callback is added.

`cb` The callback function to add.

`prio` The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The destroymt callback supports the following input arguments (in this order):

`cbprob` The thread problem passed to the callback function.

The destroymt callback does not have any output arguments.

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

```
addcbgapnotify(prob, cb, prio = 0)
```

Arguments

`prob` The problem pointer to which the callback is added.

`cb` The callback function to add.

`prio` The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The gapnotify callback supports the following input arguments (in this order):

`cbprob` The current problem.

`relgapnotifytarget` 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.

`absgapnotifytarget` 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.

`absgapnotifyobjtarget` 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.

`absgapnotifyboundtarget` 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.

The gapnotify callback supports the following output arguments:

`ret` An integer that will interrupt the search if non-zero.

`relgapnotifytarget` The value the MIPRELGAPNOTIFY control will be set to after this callback.

`absgapnotifytarget` The value the MIPABSGAPNOTIFY control will be set to after this callback.

`absgapnotifyobjtarget` The value the MIPABSGAPNOTIFYOBJ control will be set to after this callback.

`absgapnotifyboundtarget` The value the MIPABSGAPNOTIFYBOUND control will be set to after this callback.

addcbgloballog

Purpose

Declares a MIP log callback function, called each time the MIP log is printed.

Synopsis

```
addcbgloballog(prob, cb, prio = 0)
```

Further information

This function is deprecated and will be removed from future releases. Please use `addcbmiplog`

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. The callback is also invoked if the current node gets cut off, i.e., if its objective is proven to exceed the current primal bound.

Synopsis

```
addcbinfnode(prob, cb, prio = 0)
```

Arguments

`prob` The problem pointer to which the callback is added.

`cb` The callback function to add.

`prio` The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The infnode callback supports the following input arguments (in this order):

`cbprob` The problem passed to the callback function, `infnode`.

The infnode callback does not have any output arguments.

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.

Synopsis

```
addcbintsol(prob, cb, prio = 0)
```

Arguments

`prob` The problem pointer to which the callback is added.

`cb` The callback function to add.

`prio` The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The intsol callback supports the following input arguments (in this order):

`cbprob` The problem passed to the callback function, `intsol`.

The intsol callback does not have any output arguments.

addcblplog

Purpose

Declares a simplex log callback function which is called after every `LPLOG` iterations of the simplex algorithm.

Synopsis

```
addcblplog(prob, cb, prio = 0)
```

Arguments

`prob`	The problem pointer to which the callback is added.
`cb`	The callback function to add.
`prio`	The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The lplog callback supports the following input arguments (in this order):

`cbprob`	The problem passed to the callback function, `lplog`.

The lplog callback supports the following output arguments:

`ret`	An integer that will interrupt the search if non-zero.

addcbmessage

Purpose

Declares an output callback function, called every time a text line relating to the given XPRSprob is output by the Optimizer.

Synopsis

```
addcbmessage(prob, cb, prio = 0)
```

Arguments

`prob` The problem pointer to which the callback is added.

`cb` The callback function to add.

`prio` The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The message callback supports the following input arguments (in this order):

`cbprob` The problem passed to the callback function.

`msg` A null terminated character array (string) containing the message, which may simply be a new line. The total number of bytes (including `NUL` terminator) will not exceed `MAXMESSAGELENGTH`. If a message needs to be truncated to meet this limit, the last four bytes in `msg` are set to "...0".

`msgtype` Indicates the type of output message:

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

The message callback does not have any output arguments.

addcbmiplog

Purpose

Declares a MIP log callback function, called each time the MIP log is printed.

Synopsis

```
addcbmiplog(prob, cb, prio = 0)
```

Arguments

`prob` The problem pointer to which the callback is added.

`cb` The callback function to add.

`prio` The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The miplog callback supports the following input arguments (in this order):

`cbprob` The problem passed to the callback function, `miplog`.

The miplog callback supports the following output arguments:

`ret` An integer that will interrupt the search if non-zero.

addcbmipthread

Purpose

Declares a MIP thread callback function, called every time a MIP worker problem is created by the parallel MIP code.

Synopsis

```
addcbmipthread(prob, cb, prio = 0)
```

Arguments

`prob` The problem pointer to which the callback is added.

`cb` The callback function to add.

`prio` The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The mipthread callback supports the following input arguments (in this order):

`cbprob` The problem passed to the callback function.

`threadprob` The problem pointer for the MIP thread

The mipthread callback does not have any output arguments.

addcbnewnode

Purpose

Declares a callback function that will be called every time a new node is created during the branch and bound search.

Synopsis

```
addcbnewnode(prob, cb, prio = 0)
```

Arguments

`prob` The problem pointer to which the callback is added.

`cb` The callback function to add.

`prio` The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The newnode callback supports the following input arguments (in this order):

`cbprob` The problem passed to the callback function, `newnode`.

`parentnode` Unique identifier for the parent of the new node.

`node` 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`.

The newnode callback does not have any output arguments.

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.

Synopsis

```
addcbnodecutoff(prob, cb, prio = 0)
```

Arguments

`prob` The problem pointer to which the callback is added.

`cb` The callback function to add.

`prio` The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The nodecutoff callback supports the following input arguments (in this order):

`cbprob` The problem passed to the callback function, `nodecutoff`.

`node` The node id of the node that is cut off. This id cannot be queried from the CURRENTNODE attribute as for other callbacks since this callback is not invoked in the context of the node being cutoff.

The nodecutoff callback does not have any output arguments.

addcbnodelpsolved

Purpose

Declares a callback function, called during the branch and bound search, after the LP relaxation has been solved for the current node, but before any internal cuts and heuristics have been applied.

Synopsis

```
addcbnodelpsolved(prob, cb, prio = 0)
```

Arguments

`prob`	The problem pointer to which the callback is added.
`cb`	The callback function to add.
`prio`	The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The nodelpsolved callback supports the following input arguments (in this order):

`cbprob`	The problem passed to the callback function, `nodelpsolved`.

The nodelpsolved callback does not have any output arguments.

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.

Synopsis

```
addcboptnode(prob, cb, prio = 0)
```

Arguments

`prob` The problem pointer to which the callback is added.

`cb` The callback function to add.

`prio` The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The optnode callback supports the following input arguments (in this order):

`cbprob` The problem passed to the callback function, `optnode`.

The optnode callback supports the following output arguments:

`ret` An integer that will interrupt the search if non-zero.

`infeasible` The feasibility status.

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.

Synopsis

```
addcbpreintsol(prob, cb, prio = 0)
```

Arguments

`prob` The problem pointer to which the callback is added.

`cb` The callback function to add.

`prio` The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The preintsol callback supports the following input arguments (in this order):

`cbprob` The problem passed to the callback function, `preintsol`.

`soltype` The type of MIP solution that has been found:

- `0` The continuous relaxation solution to the current node of the tree search, which has been found to be integer feasible.
- `1` A MIP solution found by a heuristic.
- `2` A MIP solution provided by the user.

`cutoff` The new cutoff value that the Optimizer will use if the solution is accepted. If the user changes `cutoff`, the new value will be used instead. The cutoff value will not be updated if the solution is rejected.

The preintsol callback supports the following output arguments:

`ret` An integer that will interrupt the search if non-zero.

`reject` Set this to 1 if the solution should be rejected.

`cutoff` The new cutoff value that the Optimizer will use if the solution is accepted.

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.

Synopsis

```
addcbprenode(prob, cb, prio = 0)
```

Arguments

`prob` The problem pointer to which the callback is added.

`cb` The callback function to add.

`prio` The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The prenode callback supports the following input arguments (in this order):

`cbprob` The problem passed to the callback function, `prenode`.

The prenode callback supports the following output arguments:

`ret` An integer that will interrupt the search if non-zero.

`infeasible` The feasibility status.

addcbpresolve

Purpose

Declares a callback to be called after presolve has been performed.

Synopsis

```
addcbpresolve(prob, cb, prio = 0)
```

Arguments

`prob` The problem pointer to which the callback is added.

`cb` The callback function to add.

`prio` The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The presolve callback supports the following input arguments (in this order):

`cbprob` The problem passed to the callback function.

The presolve callback does not have any output arguments.

addcbusersolnotify

Purpose

Declares a callback function to be called each time a solution added by XPRSaddmipsol has been processed.

Synopsis

```
addcbusersolnotify(prob, cb, prio = 0)
```

Arguments

`prob` The problem pointer to which the callback is added.

`cb` The callback function to add.

`prio` The priority of the callback. If multiple callbacks are registered for the same event then they are invoked according to their priority ordering.

Return value

Always returns 0 (zero).

Further information

Note the general callback conventions:

- a callback function may return either an integer or a list.
- if the C callback function has an integer return type then the value returned to the optimizer is the integer value returned or the list's "ret" element (if it exists).
- if the C callback function has output arguments then these arguments are taken from the list that is returned by the callback function.
- the R callback function will only receive the input arguments of the C callback

The usersolnotify callback supports the following input arguments (in this order):

`cbprob` The problem passed to the callback function, `usersolnotify`.

`solname` The string name assigned to the solution when it was loaded into the Optimizer using addmipsol.

The usersolnotify callback does not have any output arguments.

addcols

Purpose

Adds columns to the optimizer matrix.

Synopsis

```
addcols(
prob,
objcoef,
start,
rowind,
rowcoef,
lb,
ub,
ncols = x_max_vec_length(objcoef, lb, ub),
ncoefs = x_max_vec_length(rowind, rowcoef)
)
```

Arguments

`prob`	The current problem.
`objcoef`	Double array of length `ncols` containing the objective function coefficients of the new columns.
`start`	Integer array of length `ncols` containing the offsets in the `rowind` and `rowcoef` arrays of the start of the elements for each column.
`rowind`	Integer array of length `ncoefs` containing the row indices for the elements in each column.
`rowcoef`	Double array of length `ncoefs` containing the element values.
`lb`	Double array of length `ncols` containing the lower bounds on the added columns.
`ub`	Double array of length `ncols` containing the upper bounds on the added columns.
`ncols`	Number of new columns.
`ncoefs`	Number of new nonzeros in the added columns.

Return value

The input argument `prob`.

addcuts

Purpose

Adds cuts directly to the matrix at the current node.
The cuts will automatically be added to the cut pool. Cuts added to a node will automatically be inherited on any descendant node, unless explicitly deleted with a call to delcuts.

Synopsis

```
addcuts(
prob,
cuttype,
rowtype,
rhs,
start,
colind,
cutcoef,
ncuts = x_max_vec_length(cuttype, rowtype, rhs)
)
```

Arguments

`prob`	The current problem.
`cuttype`	Integer array of length `ncuts` containing the user assigned cut types.
`rowtype`	Character array of length `ncuts` containing the row types: `L` indicates a <= row; `G` indicates a >= row; `E` indicates an = row.
`rhs`	Double array of length `ncuts` containing the right hand side elements for the cuts.
`start`	Integer array containing offset into the `colind` and `cutcoef` arrays indicating the start of each cut.
`colind`	Integer array of length `start[ncuts]` containing the column indices in the cuts.
`cutcoef`	Double array of length `start[ncuts]` containing the matrix values for the cuts.
`ncuts`	Number of cuts to add.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

addgencons

Purpose

Adds one or more general constraints to the problem.
Each general constraint `y = f(x1, ..., xn, c1, ..., cn)` consists of one or more (input) columns xi, zero or more constant values ci and a resultant (output column) y, different from all xi. General constraints include `maximum` and `minimum` (arbitrary number of input columns of any type and arbitrary number of input values, at least one total), `and` and `or` (at least one binary input column, no constant values, binary resultant) and `absolute value` (exactly one input column of arbitrary type, no constant values).

Synopsis

```
addgencons(
prob,
contype,
resultant,
colstart,
colind,
valstart,
val,
ncons = x_max_vec_length(contype, resultant, colstart, valstart),
ncols = x_max_vec_length(colind),
nvals = x_max_vec_length(val)
)
```

Arguments

`prob`	The current problem.
`contype`	Integer array of length `ncons` containing the types of the general constraints: `_GENCONS_MAX (0)` indicates a `maximum` constraint; `_GENCONS_MIN (1)` indicates a `minimum` constraint; `_GENCONS_AND (2)` indicates an `and` constraint. `_GENCONS_OR (3)` indicates an `or` constraint; `_GENCONS_ABS (4)` indicates an `absolute value` constraint.
`resultant`	Integer array of length `ncons` containing the indices of the output variables of the general constraints.
`colstart`	Integer array of length `ncons` containing the start index of each general constraint in the `colind` array.
`colind`	Integer array of length `ncols` containing the input variables in all general constraints.
`valstart`	Integer array of length `ncons` containing the start index of each general constraint in the `val` array (may be `NULL` if `ncoefs = 0`).
`val`	Double array of length `nvals` containing the constant values in all general constraints (may be `NULL` if `ncoefs = 0`).
`ncons`	The number of general constraints to add.
`ncols`	The total number of input variables in general constraints that should be added.
`nvals`	The total number of constant values in general constraints that should be added.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

addmanagedcuts

Purpose

Adds cuts to the Optimizer's internal cut pool from within the cutround callback set by addcbcutround. The cuts will be added to an internal pool of cuts managed by the Optimizer. The Optimizer will use internal priorities to dynamically load violated cuts from this pool into branch-and-bound node problems and remove inactive cuts. Cuts can be either local or global. Cuts flagged as local are assumed to be valid only for the the current node of the branch-and-bound search or any of its descendants. Global cuts are assumed to be valid for the whole problem and might be used on any node of the branch-and-bound search tree. The cuts should be formulated in the original space of variables and will automatically be presolved.

Synopsis

```
addmanagedcuts(
prob,
globalvalid,
rowtype,
rhs,
start,
colind,
cutcoef,
ncuts = x_max_vec_length(rowtype, rhs)
)
```

Arguments

`prob`	The current problem.
`globalvalid`	Nonzero if the cuts should be assumed to be valid for the whole problem.
`rowtype`	Character array of length `ncuts` containing the row types: `L` indicates a <= row; `G` indicates a >= row; `E` indicates an = row.
`rhs`	Double array of length `ncuts` containing the right hand side elements for the cuts.
`start`	Integer array containing offset into the `colind` and `cutcoef` arrays indicating the start of each cut.
`colind`	Integer array of length `start[ncuts]` containing the column indices in the cuts.
`cutcoef`	Double array of length `start[ncuts]` containing the matrix values for the cuts.
`ncuts`	Number of cuts to add.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

addmipsol

Purpose

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

Synopsis

```
addmipsol(
prob,
solval,
colind,
name,
length = x_max_vec_length(solval, colind)
)
```

Arguments

`prob`	The current problem.
`solval`	Double array of length `length` containing solution values.
`colind`	Optional integer array of length `length` containing the column indices for the solution values provided in `solval`.
`name`	An optional name to associate with the solution.
`length`	Number of columns for which a value is provided.

Return value

The input argument `prob`.

addnames

Purpose

When a model is loaded, the rows, columns, sets, piecewise linear and general constraints of the model may not have names associated with them.
This may not be important as the rows, columns, sets, piecewise linear and general constraints can be referred to by their sequence numbers. However, if you wish row, column, set, piecewise linear and general constraint names to appear in the ASCII solutions files, the names for a range of rows/columns/... can be added with `addnames`.

Synopsis

```
addnames(prob, type, names, first, last)
```

Arguments

`prob` The current problem.

`type` `_NAMES_ROW (=1)` for row names;
`_NAMES_COLUMN (=2)` for column names;
`_NAMES_SET (=3)` for set names;
`_NAMES_PWLCONS (=4)` for piecewise linear constraint names;
`_NAMES_GENCONS (=5)` for general constraint names;
`_NAMES_OBJECTIVE (=6)` for objective names.

`names` Character buffer containing the null-terminated string names.

`first` Start of the range of rows, columns, sets, piecewise linear constraints, general constraints or objectives.

`last` End of the range of rows, columns, sets, piecewise linear constraints, general constraints or objectives.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

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

```
addobj(
prob,
colind,
objcoef,
priority,
weight,
ncols = x_max_vec_length(colind, objcoef)
)
```

Arguments

`prob`	The current problem.
`colind`	Integer array of length `ncols` containing the indices of the columns whose objective coefficients will change.
`objcoef`	Double array of length `ncols` giving the new objective function coefficients.
`priority`	The priority for the objective function.
`weight`	The weight for the objective function.
`ncols`	Number of objective function coefficient elements to add.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

addpwlcons

Purpose

Adds one or more piecewise linear constraints to the problem.
Each piecewise linear constraint `y = f(x)` consists of an (input) column x, a (different) resultant (output column) y and a piecewise linear function f. The piecewise linear function f is described by at least two breakpoints, which are given as combinations of x- and y-values. Discontinuous piecewise linear functions are supported, in this case both the left and right limit at a given point need to be entered as breakpoints. To differentiate between left and right limit, the breakpoints need to be given as a list with non-decreasing x-values.

Synopsis

```
addpwlcons(
prob,
colind,
resultant,
start,
xval,
yval,
npwls = x_max_vec_length(colind, resultant, start),
npoints = x_max_vec_length(xval, yval)
)
```

Arguments

`prob`	The current problem.
`colind`	Integer array of length `npwls` containing the indices of the input variables x of the piecewise linear functions.
`resultant`	Integer array of length `npwls` containing the indices of the output variables y of the piecewise linear functions.
`start`	Integer array of length `npwls` containing the start index of each piecewise linear constraint in the `xval` and `yval` arrays.
`xval`	Double array of length `npoints` containing the x-values of the breakpoints.
`yval`	Double array of length `npoints` containing the y-values of the breakpoints.
`npwls`	The number of piecewise linear constraints to add.
`npoints`	The total number of breakpoints of all piecewise linear constraints that should be added.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

addqmatrix

Purpose

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

Synopsis

```
addqmatrix(
prob,
row,
rowqcol1,
rowqcol2,
rowqcoef,
ncoefs = x_max_vec_length(rowqcol1, rowqcol2, rowqcoef)
)
```

Arguments

`prob`	The current problem.
`row`	Index of the row where the quadratic matrix is to be added.
`rowqcol1`	First index in the triplets.
`rowqcol2`	Second index in the triplets.
`rowqcoef`	Coefficients in the triplets.
`ncoefs`	Number of triplets used to define the quadratic matrix.

Return value

The input argument `prob`.

addrows

Purpose

Adds rows to the optimizer matrix.

Synopsis

```
addrows(
prob,
rowtype,
rhs,
start,
colind,
rowcoef,
rng = NULL,
nrows = x_max_vec_length(rowtype, rhs),
ncoefs = x_max_vec_length(colind, rowcoef)
)
```

Arguments

`prob`	The current problem.
`rowtype`	Character array of length nrows containing the row types: `L` indicates a <= row; `G` indicates >= row; `E` indicates an = row. `R` indicates a range constraint; `N` indicates a nonbinding constraint.
`rhs`	Double array of length `nrows` containing the right hand side elements.
`start`	Integer array of length `nrows` containing the offsets in the `colind` and `rowcoef` arrays of the start of the elements for each row.
`colind`	Integer array of length `ncoefs` containing the (contiguous) column indices for the elements in each row.
`rowcoef`	Double array of length `ncoefs` containing the (contiguous) element values.
`rng`	Double array of length `nrows` containing the row range elements.
`nrows`	Number of new rows.
`ncoefs`	Number of new nonzeros in the added rows.

Return value

The input argument `prob`.

addsetnames

Purpose

Deprecated Use addnames instead.
When a model with MIP entities is loaded, any special ordered sets may not have names associated with them. If you wish names to appear in the ASCII solutions files, the names for a range of sets can be added with this function.

Synopsis

```
addsetnames(prob, names, first, last)
```

Arguments

`prob`	The current problem.
`names`	Character buffer containing the null-terminated string names.
`first`	Start of the range of sets.
`last`	End of the range of sets.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

addsets

Purpose

Allows sets to be added to the problem after passing it to the Optimizer using the input routines.

Synopsis

```
addsets(
prob,
settype,
start,
colind,
refval,
nsets = x_max_vec_length(settype),
nelems = x_max_vec_length(colind, refval)
)
```

Arguments

`prob`	The current problem.
`settype`	Character array of length nsets containing the set types: `1` indicates a SOS1; `2` indicates a SOS2;
`start`	Integer array of length `nsets` containing the offsets in the `colind` and `refval` arrays of the start of the elements for each set.
`colind`	Integer array of length `nelems` containing the (contiguous) column indices for the elements in each set.
`refval`	Double array of length `nelems` containing the (contiguous) reference values.
`nsets`	Number of new sets.
`nelems`	Number of new nonzeros in the added sets.

Return value

The input argument `prob`.

alter

Purpose

Deprecated To change matrix coefficients, use chgmcoef.
To change column bounds, use chgbounds. To change constraint right-hand side values, use chgrhs. To change constraint types, use chgrowtype. To change objective coefficients, use chgobj. Alters or changes matrix elements, right hand sides and constraint senses in the current problem.

Synopsis

```
alter(prob, filename)
```

Arguments

`prob`	The current problem.
`filename`	A string of up to MAXPROBNAMELENGTH characters specifying the file to be read.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

basisstability

Purpose

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

Synopsis

```
basisstability(prob, type, norm, scaled)
```

Arguments

`prob`	The current problem.	
`type`	`0`	Condition number of the basis.
	`1`	Stability measure for the solution relative to the current basis.
	`2`	Stability measure for the duals relative to the current basis.
	`3`	Stability measure for the right hand side relative to the current basis.
	`4`	Stability measure for the basic part of the objective relative to the current basis.
`norm`	`0`	Use the infinity norm.
	`1`	Use the 1 norm.
	`2`	Use the Euclidian norm for vectors, and the Frobenius norm for matrices.
`scaled`	If the stability values are to be calculated in the scaled, or the unscaled matrix.	

Return value

The calculated value.

beginlicensing

Purpose

Wraps callable C library function XPRSbeginlicensing.
Please refer to the OEM guide for details.

Synopsis

```
beginlicensing()
```

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

```
bndsa(prob, colind, ncols = x_max_vec_length(colind))
```

Arguments

`prob`	The current problem.
`colind`	Integer array of length `ncols` containing the indices of the columns whose bounds' ranges are required.
`ncols`	Number of variables whose sensitivity is sought.

Return value

A list with the following elements:

`lblower`	The variable lower bound lower range values.
`lbupper`	The variable lower bound upper range values.
`ublower`	The variable upper bound lower range values.
`ubupper`	The variable upper bound upper range values.

Further information

Please refer to the C documentation for more details.

bo_addbounds

Purpose

Adds new bounds to a branch of a user branching object.

Synopsis

```
bo_addbounds(
bo,
branch,
bndtype,
colind,
bndval,
nbounds = x_max_vec_length(bndtype, colind, bndval)
)
```

Arguments

`bo`	The user branching object to modify.
`branch`	The number of the branch to add the new bounds for.
`bndtype`	Character array of length `nbounds` indicating the type of bounds to add: `L` Lower bound. `U` Upper bound.
`colind`	Integer array of length `nbounds` containing the column indices for the new bounds.
`bndval`	Double array of length `nbounds` giving the bound values.
`nbounds`	Number of new bounds to add.

Return value

The input argument `bo`.

bo_addbranches

Purpose

Adds new, empty branches to a user defined branching object.

Synopsis

```
bo_addbranches(bo, nbranches)
```

Arguments

`bo`	The user branching object to modify.
`nbranches`	Number of new branches to create.

Return value

The input argument `bo`.

bo_addcuts

Purpose

Adds stored user cuts as new constraints to a branch of a user branching object.

Synopsis

```
bo_addcuts(bo, branch, ncuts, cutind = NULL)
```

Arguments

`bo`	The user branching object to modify.
`branch`	The number of the branch to add the cuts for.
`ncuts`	Number of cuts to add.
`cutind`	Array of length `ncuts` containing the pointers to user cuts that should be added to the branch.

Return value

The input argument `bo`.

bo_addrows

Purpose

Adds new constraints to a branch of a user branching object.

Synopsis

```
bo_addrows(
bo,
branch,
rowtype,
rhs,
start,
colind,
rowcoef,
nrows = x_max_vec_length(rowtype, rhs, start),
ncoefs = x_max_vec_length(colind, rowcoef)
)
```

Arguments

`bo`	The user branching object to modify.
`branch`	The number of the branch to add the new constraints for.
`rowtype`	Character array of length `nrows` indicating the type of constraints to add: `L` Less than type. `G` Greater than type. `E` Equality type.
`rhs`	Double array of length `nrows` containing the right hand side values.
`start`	Integer array of length `nrows` containing the offsets of the `colind` and `rowcoef` arrays of the start of the non zero coefficients in the new constraints.
`colind`	Integer array of length `ncoefs` containing the column indices for the non zero coefficients.
`rowcoef`	Double array of length `ncoefs` containing the non zero coefficient values.
`nrows`	Number of new constraints to add.
`ncoefs`	Number of non-zero coefficients in all new constraints.

Return value

The input argument `bo`.

bo_create

Purpose

Creates a new user defined branching object for the Optimizer to branch on.
This function should be called only from within one of the callback functions set by addcboptnode, addcbchgbranchobject, or addcbpreintsol (only if the `soltype` is 0).

Synopsis

```
bo_create(prob, isoriginal)
```

Arguments

`prob`	The problem structure that the branching object should be created for.
`isoriginal`	If the branching object will be set up for the original matrix, which determines how column indices are interpreted when adding bounds and rows to the object:
	`0` Column indices should refer to the current (presolved) node problem.
	`1` Column indices should refer to the original matrix.

Return value

The new object.

Further information

Please refer to the C documentation for more details.

bo_destroy

Purpose

Frees all memory for a user branching object, when the object was not stored with the Optimizer.

Synopsis

```
bo_destroy(bo)
```

Argument

`bo` The user branching object to free.

Return value

The input argument `bo`.

bo_getbounds

Purpose

Returns the bounds for a branch of a user branching object.

Synopsis

```
bo_getbounds(bo, branch)
```

Arguments

`bo`	The branching object to inspect.
`branch`	The number of the branch to get the bounds for.

Return value

A list with the following elements:

`nbounds`	The number of bounds for the given branch.
`bndtype`	The types of bounds:
	`L` Lower bound. Allowed to be NULL.
	`U` Upper bound.
`colind`	The column indices.
`bndval`	The bound values.

bo_getbranches

Purpose

Returns the number of branches of a branching object.

Synopsis

```
bo_getbranches(bo)
```

Argument

`bo` The user branching object to inspect.

Return value

The number of branches.

bo_getid

Purpose

Returns the unique identifier assigned to a branching object.

Synopsis

```
bo_getid(bo)
```

Argument

`bo` A branching object.

Return value

The identifier.

bo_getlasterror

Purpose

Returns the last error encountered during a call to the given branch object.

Synopsis

```
bo_getlasterror(bo)
```

Argument

`bo` The branch object.

Return value

A list with the following elements:

`msgcode` The error code.

`msg` The last error message relating to the given branching object.

bo_getrows

Purpose

Returns the constraints for a branch of a user branching object.

Synopsis

```
bo_getrows(bo, branch)
```

Arguments

`bo`	The user branching object to inspect.
`branch`	The number of the branch to get the constraints from.

Return value

A list with the following elements:

`nrows`	The number of rows.
`ncoefs`	The number of non zero coefficients in the constraints.
`rowtype`	The types of the rows:
	`L` Less than type.
	`G` Greater than type.
	`E` Equality type.
`rhs`	The right hand side values.
`start`	The offsets of the `colind` and `rowcoef` arrays of the start of the non zero coefficients in the returned constraints.
`colind`	The column indices for the non zero coefficients.
`rowcoef`	The non zero coefficient values.

bo_setpreferredbranch

Purpose

Specifies which of the child nodes corresponding to the branches of the object should be explored first.

Synopsis

```
bo_setpreferredbranch(bo, branch)
```

Arguments

`bo`	The user branching object.
`branch`	The number of the branch to mark as preferred.

Return value

The input argument `bo`.

bo_setpriority

Purpose

Sets the priority value of a user branching object.

Synopsis

```
bo_setpriority(bo, priority)
```

Arguments

`bo`	The user branching object.
`priority`	The new priority value to assign to the branching object, which must be a number from 0 to 1000.

Return value

The input argument `bo`.

bo_store

Purpose

Adds a new user branching object to the Optimizer's list of candidates for branching.
This function is available only through the callback functions set by addcboptnode or addcbpreintsol (if the `soltype` is 0).

Synopsis

```
bo_store(bo)
```

Argument

`bo` The new user branching object to store.

Return value

The returned status from checking the provided branching object:

`0`	The object was accepted successfully. The object was not added to the candidate list if a non zero status is returned.
`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.

Further information

Please refer to the C documentation for more details.

bo_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

```
bo_validate(bo)
```

Argument

`bo` A branching object.

Return value

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.

Further information

Please refer to the C documentation for more details.

calcobjective

Purpose

Calculates the objective value of a given solution.

Synopsis

```
calcobjective(prob, solution)
```

Arguments

`prob`	The current problem.
`solution`	Double array of length COLS that holds the solution.

Return value

The calculated objective value.

calcobjn

Purpose

Calculates the objective value of the given objective function in a multi-objective problem.

Synopsis

```
calcobjn(prob, objidx, solution)
```

Arguments

`prob`	The current problem.
`objidx`	Index of the objective function to calculate.
`solution`	Double array of length `COLS` that holds the solution, or `NULL` to use the current solution.

Return value

The calculated objective value.

calcreducedcosts

Purpose

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

Synopsis

```
calcreducedcosts(prob, duals, solution)
```

Arguments

`prob`	The current problem.
`duals`	Double array of length ROWS that holds the dual solution to calculate the reduced costs for.
`solution`	Optional double array of length COLS that holds the primal solution.

Return value

The calculated reduced costs.

calcslacks

Purpose

Calculates the row slack values for a given solution.

Synopsis

```
calcslacks(prob, solution)
```

Arguments

`prob`	The current problem.
`solution`	Double array of length COLS that holds the solution to calculate the slacks for.

Return value

The calculated row slacks.

calcsolinfo

Purpose

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

Synopsis

```
calcsolinfo(prob, solution, duals, property)
```

Arguments

`prob` The current problem.

`solution` Double array of length COLS that holds the solution.

`duals` Double array of length ROWS that holds the dual solution.

`property` Defined the property to be calculated.

- `_SOLINFO_ABSPRIMALINFEAS` the calculated maximum absolute primal infeasibility is returned.
- `_SOLINFO_RELPRIMALINFEAS` the calculated maximum relative primal infeasibility is returned.
- `_SOLINFO_ABSDUALINFEAS` the calculated maximum absolute dual infeasibility is returned.
- `_SOLINFO_RELDUALINFEAS` the calculated maximum relative dual infeasibility is returned.
- `_SOLINFO_MAXMIPFRACTIONAL` the calculated maximum absolute MIP fractionality or SOS infeasibility.
- `_SOLINFO_ABSMIPINFEAS` the calculated maximum absolute MIP infeasibility (including delayed rows, indicators, general and piecewise linear constraints) is returned.
- `_SOLINFO_RELMIPINFEAS` the calculated maximum relative MIP infeasibility (including delayed rows, indicators, general and piecewise linear constraints) is returned.

Return value

The calculated value.

chgbounds

Purpose

Used to change the bounds on columns in the matrix.

Synopsis

```
chgbounds(
prob,
colind,
bndtype,
bndval,
nbounds = x_max_vec_length(colind, bndtype, bndval)
)
```

Arguments

`prob`	The current problem.
`colind`	Integer array of size `nbounds` containing the indices of the columns on which the bounds will change.
`bndtype`	Character array of length `nbounds` 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`	Double array of length `nbounds` giving the new bound values.
`nbounds`	Number of bounds to change.

Return value

The input argument `prob`.

chgcoef

Purpose

Used to change a single coefficient in the matrix.
If the coefficient does not already exist, a new coefficient will be added to the matrix. If many coefficients are being added to a row of the matrix, it may be more efficient to delete the old row of the matrix and add a new row.

Synopsis

```
chgcoef(prob, row, col, coef)
```

Arguments

`prob`	The current problem.
`row`	Row index for the coefficient.
`col`	Column index for the coefficient.
`coef`	New value for the coefficient.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

chgcoltype

Purpose

Used to change the type of a specified set of columns in the matrix.

Synopsis

```
chgcoltype(prob, colind, coltype, ncols = x_max_vec_length(colind,
      coltype))
```

Arguments

`prob` The current problem.

`colind` Integer array of length `ncols` containing the indices of the columns.

`coltype` Character array of length `ncols` giving the new column types:

- `C` indicates a continuous column;
- `B` indicates a binary column;
- `I` indicates an integer column.
- `S` indicates a semicontinuous column. The semicontinuous lower bound will be set to `1.0`.
- `R` indicates a semiinteger column. The semiinteger lower bound will be set to `1.0`.
- `P` indicates a partial integer column. The partial integer limit will be set to `1.0`.

`ncols` Number of columns to change.

Return value

The input argument `prob`.

chgglblimit

Purpose

Used to change semi-continuous or semi-integer lower bounds, or upper limits on partial integers.

Synopsis

```
chgglblimit(prob, colind, limit, ncols = x_max_vec_length(colind, limit))
```

Arguments

`prob`	The current problem.
`colind`	Integer array of size `ncols` containing the indices of the semi-continuous, semi-integer or partial integer columns that should have their limits changed.
`limit`	Double array of length `ncols` giving the new limit values.
`ncols`	Number of column limits to change.

Return value

The input argument `prob`.

chgmcoef

Purpose

Used to change multiple coefficients in the matrix.
If any coefficient does not already exist, it will be added to the matrix. If many coefficients are being added to a row of the matrix, it may be more efficient to delete the old row of the matrix and add a new one.

Synopsis

```
chgmcoef(
prob,
rowind,
colind,
rowcoef,
ncoefs = x_max_vec_length(rowind, colind, rowcoef)
)
```

Arguments

`prob`	The current problem.
`rowind`	Integer array of length `ncoefs` containing the row indices of the coefficients to be changed.
`colind`	Integer array of length `ncoefs` containing the column indices of the coefficients to be changed.
`rowcoef`	Double array of length `ncoefs` containing the new coefficient values.
`ncoefs`	Number of new coefficients.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

chgmqobj

Purpose

Used to change multiple quadratic coefficients in the objective function.
If any of the coefficients does not exist already, new coefficients will be added to the objective function.

Synopsis

```
chgmqobj(
prob,
objqcol1,
objqcol2,
objqcoef,
ncoefs = x_max_vec_length(objqcol1, objqcol2, objqcoef)
)
```

Arguments

`prob`	The current problem.
`objqcol1`	Integer array of size `ncol` containing the column index of the first variable in each quadratic term.
`objqcol2`	Integer array of size `ncol` containing the column index of the second variable in each quadratic term.
`objqcoef`	New values for the coefficients.
`ncoefs`	The number of coefficients to change.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

chgobj

Purpose

Used to change the objective function coefficients.

Synopsis

```
chgobj(prob, colind, objcoef, ncols = x_max_vec_length(colind, objcoef))
```

Arguments

`prob`	The current problem.
`colind`	Integer array of length `ncols` containing the indices of the columns whose objective coefficients will change.
`objcoef`	Double array of length `ncols` giving the new objective function coefficients.
`ncols`	Number of objective function coefficient elements to change.

Return value

The input argument `prob`.

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

```
chgobjn(
prob,
objidx,
colind,
objcoef,
ncols = x_max_vec_length(colind, objcoef)
)
```

Arguments

`prob`	The current problem.
`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.
`objcoef`	Double array of length `ncols` giving the new objective function coefficients.
`ncols`	Number of objective function coefficient elements to change.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

chgobjsense

Purpose

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

Synopsis

```
chgobjsense(prob, objsense)
```

Arguments

`prob`	The current problem.
`objsense`	`OBJ_MINIMIZE` to change into a minimization, or `OBJ_MAXIMIZE` to change into maximization problem.

Return value

The input argument `prob`.

chgqobj

Purpose

Used to change a single quadratic coefficient in the objective function corresponding to the variable pair `(objqcol1,objqcol2)` of the Hessian matrix.

Synopsis

```
chgqobj(prob, objqcol1, objqcol2, objqcoef)
```

Arguments

`prob`	The current problem.
`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.

Return value

The input argument `prob`.

chgqrowcoeff

Purpose

Changes a single quadratic coefficient in a row.

Synopsis

```
chgqrowcoeff(prob, row, rowqcol1, rowqcol2, rowqcoef)
```

Arguments

`prob`	The current problem.
`row`	Index of the row 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.

Return value

The input argument `prob`.

chgrhs

Purpose

Used to change righthand side values of the problem.

Synopsis

```
chgrhs(prob, rowind, rhs, nrows = x_max_vec_length(rowind, rhs))
```

Arguments

`prob`	The current problem.
`rowind`	Integer array of length `nrows` containing the indices of the rows on which the right hand side values will change.
`rhs`	Double array of length `nrows` giving the right hand side values.
`nrows`	Number of right hand side values to change.

Return value

The input argument `prob`.

chgrhsrange

Purpose

Used to change the range for a row of the problem matrix.

Synopsis

```
chgrhsrange(prob, rowind, rng, nrows = x_max_vec_length(rowind, rng))
```

Arguments

`prob`	The current problem.
`rowind`	Integer array of length `nrows` containing the indices of the rows on which the range elements will change.
`rng`	Double array of length `nrows` giving the range values.
`nrows`	Number of range elements to change.

Return value

The input argument `prob`.

chgrowtype

Purpose

Used to change the type of a row in the matrix.

Synopsis

```
chgrowtype(prob, rowind, rowtype, nrows = x_max_vec_length(rowind,
      rowtype))
```

Arguments

`prob`	The current problem.
`rowind`	Integer array of length `nrows` containing the indices of the rows.
`rowtype`	Character array of length `nrows` giving the new row types:
	`L` indicates a <= row;
	`E` indicates an = row;
	`G` indicates a >= row;
	`R` indicates a range row;
	`N` indicates a free row.
`nrows`	Number of rows to change.

Return value

The input argument `prob`.

chk_min_length

Purpose

Check that an array has at least a certain length
Returns TRUE if the array has the required length, or if the array is optional and NULL

Synopsis

```
chk_min_length(arr, expected_len, is_optional = FALSE)
```

clearrowflags

Purpose

Clears extra information attached to a range of rows.

Synopsis

```
clearrowflags(prob, flags, first, last)
```

Arguments

`prob`	The current problem
`flags`	Int array of length `last-first+1` including type of extra information to remove (see below)
`first`	First row index to be checked
`last`	Last row index to be checked

Return value

The input argument `prob`.

copycallbacks

Purpose

Copies callback functions defined for one problem to another.

Synopsis

```
copycallbacks(dest, src)
```

Arguments

`dest` The problem to which the callbacks are copied.

`src` The problem from which the callbacks are copied.

Return value

The input argument `dest`.

copycontrols

Purpose

Copies controls defined for one problem to another.

Synopsis

```
copycontrols(dest, src)
```

Arguments

`dest`	The problem to which the controls are copied.
`src`	The problem from which the controls are copied.

Return value

The input argument `dest`.

copyprob

Purpose

Copies information defined for one problem to another.

Synopsis

```
copyprob(dest, src, name)
```

Arguments

`dest`	The new problem pointer to which information is copied.
`src`	The old problem pointer from which information is copied.
`name`	A string of up to 1024 characters including `NULL` terminator containing the name for the problem copy.

Return value

The input argument `dest`.

createprob

Purpose

Sets up a new problem within the Optimizer.

Synopsis

```
createprob()
```

Return value

The new 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

```
crossoverlpsol(prob)
```

Argument

`prob` The current problem.

Return value

The status. The status is one of:

`0`	The crossover is successful.
`1`	The crossover is not performed because the problem has no solution.

Further information

Please refer to the C documentation for more details.

delcols

Purpose

Delete columns from a matrix.

Synopsis

```
delcols(prob, colind, ncols = x_max_vec_length(colind))
```

Arguments

`prob`	The current problem.
`colind`	Integer array of length `ncols` containing the columns to delete.
`ncols`	Number of columns to delete.

Return value

The input argument `prob`.

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 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

```
delcpcuts(prob, ncuts, cutind = NULL, cuttype = 0, interp = -1)
```

Arguments

`prob`	The current problem.
`ncuts`	The number of cuts to delete.
`cutind`	Array containing pointers to the cuts which are to be deleted.
`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-vectors (compare Section) - delete if any bit matches any bit set in `cuttype`;
`3`	treat cut types as bit-vectors (compare Section) - delete if all bits match those set in `cuttype`.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

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 addcuts or 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

```
delcuts(
prob,
basis,
cutind = NULL,
cuttype = 0,
interp = -1,
delta = -Inf,
ncuts = -1
)
```

Arguments

`prob`	The current problem.
`basis`	Ensures the basis will be valid if set to `1`.
`cutind`	Array containing pointers to the cuts which are to 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-vectors (compare Section) - delete if any bit matches any bit set in `cuttype`;
	`3` treat cut types as bit-vectors (compare Section) - delete if all bits match those set in `cuttype`.
`delta`	Only delete cuts with an absolute slack value greater than `delta`.
`ncuts`	Number of cuts to drop if a list of cuts is provided.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

delgencons

Purpose

Delete general constraints from a problem.

Synopsis

```
delgencons(prob, conind, ncons = x_max_vec_length(conind))
```

Arguments

`prob`	The current problem.
`conind`	An integer array of length `ncons` containing the general constraints to delete.
`ncons`	Number of general constraints to delete.

Return value

The input argument `prob`.

delindicators

Purpose

Delete indicator constraints.
This turns the specified rows into normal rows (not controlled by indicator variables).

Synopsis

```
delindicators(prob, first, last)
```

Arguments

`prob`	The current problem.
`first`	First row in the range.
`last`	Last row in the range (inclusive).

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

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

```
delobj(prob, objidx)
```

Arguments

`prob`	The current problem.
`objidx`	Index of the objective to remove.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

delpwlcons

Purpose

Delete piecewise linear constraints from a problem.

Synopsis

```
delpwlcons(prob, pwlind, npwls = x_max_vec_length(pwlind))
```

Arguments

`prob`	The current problem.
`pwlind`	An integer array of length `npwls` containing the piecewise linear constraints to delete.
`npwls`	Number of piecewise linear constraints to delete.

Return value

The input argument `prob`.

delqmatrix

Purpose

Deletes the quadratic part of a row or of the objective function.

Synopsis

```
delqmatrix(prob, row)
```

Arguments

`prob`	The current problem.
`row`	Index of row from which the quadratic part is to be deleted.

Return value

The input argument `prob`.

delrows

Purpose

Delete rows from a matrix.

Synopsis

```
delrows(prob, rowind, nrows = x_max_vec_length(rowind))
```

Arguments

`prob`	The current problem.
`rowind`	An integer array of length `nrows` containing the rows to delete.
`nrows`	Number of rows to delete.

Return value

The input argument `prob`.

delsets

Purpose

Delete sets from a problem.

Synopsis

```
delsets(prob, setind, nsets = x_max_vec_length(setind))
```

Arguments

`prob`	The current problem.
`setind`	An integer array of length `nsets` containing the sets to delete.
`nsets`	Number of sets to delete.

Return value

The input argument `prob`.

destroyprob

Purpose

Removes a given problem and frees any memory associated with it following manipulation and optimization.

Synopsis

```
destroyprob(prob)
```

Argument

`prob` The problem to be destroyed.

Return value

The input argument `prob`.

dumpcontrols

Purpose

Displays the list of controls and their current value for those controls that have been set to a non default value.

Synopsis

```
dumpcontrols(prob)
```

Argument

`prob` The problem for which controls are dumped.

Return value

The input argument `prob`.

endlicensing

Purpose

Wraps callable C library function XPRSendlicensing.
Please refer to the OEM guide for details.

Synopsis

```
endlicensing()
```

estimaterowdualranges

Purpose

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

Synopsis

```
estimaterowdualranges(prob, nrows, rowind, iterlim)
```

Arguments

`prob`	The current problem.
`nrows`	The number of rows to analyze.
`rowind`	Row indices to analyze.
`iterlim`	Effort limit expressed as simplex iterations per row.

Return value

A list with the following elements:

`mindual`	Estimated lower bounds on the possible dual ranges.
`maxdual`	Estimated upper bounds on the possible dual ranges.

featurequery

Purpose

Checks if the provided feature is available in the current license used by the optimizer.

Synopsis

```
featurequery(feature)
```

Argument

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

Return value

Return status of the check, a value of 1 indicates the feature is available.

fixglobals

Purpose

Fixes all the MIP entities to the values of the last found MIP solution.

Synopsis

```
fixglobals(prob, options)
```

Further information

This function is deprecated and will be removed from future releases. Please use `fixmipentities`

fixmipentities

Purpose

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

Synopsis

```
fixmipentities(prob, options)
```

Arguments

`prob` The current problem.

`options` Options how to fix the MIP entities.

`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.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

free

Purpose

Frees any allocated memory and closes all open files.

Synopsis

```
free()
```

ge_getcomputeallowed

Purpose

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

Synopsis

```
ge_getcomputeallowed()
```

Return value

The value. Value will equal one of the following constants.

`_ALLOW_COMPUTE_ALWAYS` Always allow solves to be sent to Compute.

`_ALLOW_COMPUTE_NEVER` Never allow solves to be sent to Compute.

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

ge_getdebugmode

Purpose

Wraps callable C library function XPRS_ge_getdebugmode.

Synopsis

```
ge_getdebugmode()
```

ge_getlasterror

Purpose

Returns the last error encountered during a call to the Xpress global environment.

Synopsis

```
ge_getlasterror()
```

Return value

A list with the following elements:

`msgcode`	The error code.
`msg`	The last error message relating to the global environment.

ge_getsafemode

Purpose

Wraps callable C library function XPRS_ge_getsafemode.

Synopsis

```
ge_getsafemode()
```

ge_setarchconsistency

Purpose

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

Synopsis

```
ge_setarchconsistency(consistent)
```

Argument

`consistent`	Whether to force the same execution path:	
	`0`	Do not force the same execution path (default behavior);
	`1`	Force the same execution path.

Return value

Always returns 0 (zero).

ge_setcomputeallowed

Purpose

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

Synopsis

```
ge_setcomputeallowed(allow)
```

Argument

`allow` Whether the Insight Compute interface may be used; must be one of the following constants:

- `_ALLOW_COMPUTE_ALWAYS` Always allow solves to be sent to Compute.
- `_ALLOW_COMPUTE_NEVER` Never allow solves to be sent to Compute.
- `_ALLOW_COMPUTE_DEFAULT` Allow solves to be sent to Compute only from non-OEM applications.

Return value

Always returns 0 (zero).

ge_setdebugmode

Purpose

Wraps callable C library function XPRS_ge_setdebugmode.

Synopsis

```
ge_setdebugmode(debugmode)
```

ge_setsafemode

Purpose

Wraps callable C library function XPRS_ge_setsafemode.

Synopsis

```
ge_setsafemode(safemode)
```

getattribinfo

Purpose

Accesses the id number and the type information of an attribute given its name.
An attribute name may be for example `ROWS`. Names are case-insensitive and may or may not have the _ prefix. The id number is the constant used to identify the attribute for calls to functions such as getintattrib. The type information returned will be one of the below integer constants defined in the `xprs.h` header file. The function will return an id number of 0 and a type value of `TYPE_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

```
getattribinfo(prob, name)
```

Arguments

`prob` The current problem.

`name` The name of the attribute to be queried.

Return value

A list with the following elements:

`id` The id number.

`type` The type id.

Further information

Please refer to the C documentation for more details.

getbanner

Purpose

Returns the banner and copyright message.

Synopsis

```
getbanner()
```

Return value

The null terminated banner string.

getbarnumstability

Purpose

Wraps callable C library function XPRSgetbarnumstability.

Synopsis

```
getbarnumstability(prob)
```

getbasis

Purpose

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

Synopsis

```
getbasis(prob, rowstat = TRUE, colstat = TRUE)
```

Arguments

`prob` The current problem.

`rowstat` Flag to indicate whether the output list should contain `rowstat`.

`colstat` Flag to indicate whether the output list should contain `colstat`.

Return value

A list with the following elements:

`rowstat` Basis status of the slack, surplus or artificial variable associated with each row. The status will be one of:

- `_NONBASIC_LOWER (0)` slack, surplus or artificial is non-basic at lower bound; May be `NULL` if not required.
- `_BASIC (1)` slack, surplus or artificial is basic;
- `_NONBASIC_UPPER (2)` slack or surplus is non-basic at upper bound.
- `_SUPERBASIC (3)` slack or surplus is super-basic.

`colstat` The basis status of the columns in the constraint matrix. The status will be one of:

- `_NONBASIC_LOWER (0)` variable is non-basic at lower bound, or superbasic at zero if the variable has no lower bound; May be `NULL` if not required.
- `_BASIC (1)` variable is basic;
- `_NONBASIC_UPPER (2)` variable is non-basic at upper bound;
- `_SUPERBASIC (3)` variable is super-basic.

getbasisval

Purpose

Returns the current basis status for a specific column or row.

Synopsis

```
getbasisval(prob, row, col)
```

Arguments

`prob`	The current problem.
`row`	Row index to get the row basis status for.
`col`	Column index to get the column basis status for.

Return value

A list with the following elements:

`rowstat`	The row basis status.
`colstat`	The column basis status.

getcallbackduals

Purpose

Returns the dual values from the solution associated with the current callback.

Synopsis

```
getcallbackduals(
prob,
first = 0,
last = getintattrib(prob, xpress:::INPUTROWS) - 1
)
```

Arguments

`prob` The current problem.

`first` First row whose dual value to return.

`last` Last row whose dual value to return.

Return value

A list with the following elements:

`available` This variable will be set to 1 if a dual solution is available.

`duals` The values of the dual variables.

getcallbackpresolveduals

Purpose

Returns the dual values from the solution to the presolved problem associated with the current callback.

Synopsis

```
getcallbackpresolveduals(prob, first, last)
```

Arguments

`prob`	The current problem.
`first`	First row whose dual value to return.
`last`	Last row whose dual value to return.

Return value

A list with the following elements:

`available`	This variable will be set to 1 if a dual solution is available.
`duals`	The values of the dual variables.

getcallbackpresolveredcosts

Purpose

Returns the reduced costs from the solution to the presolved problem associated with the current callback.

Synopsis

```
getcallbackpresolveredcosts(prob, first, last)
```

Arguments

`prob` The current problem.

`first` First column whose reduced cost to return.

`last` Last column whose reduced cost to return.

Return value

A list with the following elements:

`available` This variable will be set to 1 if a dual solution is available.

`djs` The reduced costs of the variables.

getcallbackpresolveslacks

Purpose

Returns the slack values from the solution to the presolved problem associated with the current callback.

Synopsis

```
getcallbackpresolveslacks(prob, first, last)
```

Arguments

`prob` The current problem.

`first` First row whose slack value to return.

`last` Last row whose slack value to return.

Return value

A list with the following elements:

`available` This variable will be set to 1 if a solution is available.

`slacks` The values of the slack variables.

getcallbackpresolvesolution

Purpose

Returns the solution to the presolved problem associated with the current callback.

Synopsis

```
getcallbackpresolvesolution(prob, first, last)
```

Arguments

`prob`	The current problem.
`first`	First column in the solution to return.
`last`	Last column in the solution to return.

Return value

A list with the following elements:

`available`	This variable will be set to 1 if a solution is available.
`x`	The values of the primal variables.

getcallbackredcosts

Purpose

Returns the reduced costs from the solution associated with the current callback.

Synopsis

```
getcallbackredcosts(
prob,
first = 0,
last = getintattrib(prob, xpress:::INPUTCOLS) - 1
)
```

Arguments

`prob`	The current problem.
`first`	First column whose reduced cost to return.
`last`	Last column whose reduced cost to return.

Return value

A list with the following elements:

`available`	This variable will be set to 1 if a dual solution is available.
`djs`	The reduced costs of the variables.

getcallbackslacks

Purpose

Returns the slack values from the solution associated with the current callback.

Synopsis

```
getcallbackslacks(
prob,
first = 0,
last = getintattrib(prob, xpress:::INPUTROWS) - 1
)
```

Arguments

`prob` The current problem.

`first` First row whose slack value to return.

`last` Last row whose slack value to return.

Return value

A list with the following elements:

`available` This variable will be set to 1 if a solution is available.

`slacks` The values of the slack variables.

getcallbacksolution

Purpose

Returns the primal values from the solution associated with the current callback.

Synopsis

```
getcallbacksolution(
prob,
first = 0,
last = getintattrib(prob, xpress:::INPUTCOLS) - 1
)
```

Arguments

`prob` The current problem.

`first` First column in the solution to return.

`last` Last column in the solution to return.

Return value

A list with the following elements:

`available` This variable will be set to 1 if a solution is available.

`x` The values of the primal variables.

getcheckedmode

Purpose

You can use this function to interrogate whether checking and validation of all Optimizer function calls is enabled for the current process.
Checking and validation is enabled by default but can be disabled by setcheckedmode.

Synopsis

```
getcheckedmode()
```

Return value

0 if checking and validation of Optimizer function calls is disabled for the current process, non-zero otherwise.

Further information

Please refer to the C documentation for more details.

getcoef

Purpose

Returns a single coefficient in the constraint matrix.

Synopsis

```
getcoef(prob, row, col)
```

Arguments

`prob`	The current problem.
`row`	Row of the constraint matrix.
`col`	Column of the constraint matrix.

Return value

The coefficient.

getcols

Purpose

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

Synopsis

```
getcols(prob, first, last)
```

Arguments

`prob`	The current problem.
`first`	First column in the range.
`last`	Last column in the range.

Return value

A list with the following elements:

`start`	The indices indicating the starting offsets in the `rowind` and `rowcoef` arrays for each requested column
`rowind`	The row indices of the nonzero coefficents for each column.
`rowcoef`	The nonzero coefficient values.

getcoltype

Purpose

Returns the column types for the columns in a given range.

Synopsis

```
getcoltype(prob, first, last)
```

Arguments

`prob` The current problem.

`first` First column in the range.

`last` Last column in the range.

Return value

The column types:

`C`	indicates a continuous variable;
`I`	indicates an integer variable;
`B`	indicates a binary variable;
`S`	indicates a semi-continuous variable;
`R`	indicates a semi-continuous integer variable;
`P`	indicates a partial integer variable.

getcontrolinfo

Purpose

Accesses the id number and the type information of a control given its name.
A control name may be for example `PRESOLVE`. Names are case-insensitive and may or may not have the _ prefix. The id number is the constant used to identify the control for calls to functions such as getintcontrol. The function will return an id number of `0` and a type value of `TYPE_NOTDEFINED` if the name is not recognized as a control name. Note that this will occur if the name is an attribute name and not a control name.

Synopsis

```
getcontrolinfo(prob, name)
```

Arguments

`prob`	The current problem.
`name`	The name of the control to be queried.

Return value

A list with the following elements:

`id`	The id number.
`type`	The type information.

Further information

Please refer to the C documentation for more details.

getcpcutlist

Purpose

Returns a list of cut indices from the cut pool.

Synopsis

```
getcpcutlist(prob, cuttype = 0, interp = -1, delta = -Inf)
```

Arguments

`prob`	The current problem.
`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-vectors (compare Section) - get cut if any bit matches any bit set in `cuttype`;
	`3` treat cut types as bit-vectors (compare Section) - get cut if all bits match those set in `cuttype`.
`delta`	Only those cuts with a signed violation greater than delta will be returned.

Return value

A list with the following elements:

`ncuts`	The number of cuts of type `cuttype` in the cut pool.
`cutind`	The pointers to the cuts .
`viol`	The values of the signed violations of the cuts.

getcpcuts

Purpose

Returns cuts from the cut pool.
A list of cut pointers in the array `rowind` 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

```
getcpcuts(prob, rowind, ncuts = x_max_vec_length(rowind))
```

Arguments

`prob`	The current problem.
`rowind`	Array of length `ncuts` containing the pointers to the cuts.
`ncuts`	Number of cuts to be returned.

Return value

A list with the following elements:

`cuttype`	The cut types.
`rowtype`	The sense of the cuts (`L`, `G`, or `E`).
`start`	The offsets into the `colind` and `cutcoef` arrays.
`colind`	The column indices of the cuts.
`cutcoef`	The matrix values.
`rhs`	The right hand side elements for the cuts.

Further information

Please refer to the C documentation for more details.

getcutlist

Purpose

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

Synopsis

```
getcutlist(prob, cuttype = 0, interp = -1)
```

Arguments

`prob`	The current problem.	
`cuttype`	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-vectors (compare Section) - get cut if any bit matches any bit set in `cuttype`;
	`3`	treat cut types as bit-vectors (compare Section) - get cut if all bits match those set in `cuttype`.

Return value

A list with the following elements:

`ncuts`	The number of active cuts of type `cuttype`.
`cutind`	The pointers to the cuts .

getcutmap

Purpose

Used to return in which rows a list of cuts are currently loaded into the Optimizer. This is useful for example to retrieve the duals associated with active cuts.

Synopsis

```
getcutmap(prob, cutind, ncuts = x_max_vec_length(cutind))
```

Arguments

`prob`	The current problem.
`cutind`	Pointer array to the cuts for which the row index is requested.
`ncuts`	Number of cuts in the cutind array.

Return value

The row indices.

Further information

Please refer to the C documentation for more details.

getcutslack

Purpose

Used to calculate the slack value of a cut with respect to the current LP relaxation solution.
The slack is calculated from the cut itself, and might be requested for any cut (even if it is not currently loaded into the problem).

Synopsis

```
getcutslack(prob, cutind)
```

Arguments

`prob`	The current problem.
`cutind`	Pointer of the cut for which the slack is to be calculated.

Return value

The slack.

Further information

Please refer to the C documentation for more details.

getdaysleft

Purpose

Returns the number of days left until the license expires.

Synopsis

```
getdaysleft()
```

Return value

The number of days.

getdblattrib

Purpose

Enables users to retrieve the values of various double problem attributes. Problem attributes are set during loading and optimization of a problem.

Synopsis

```
getdblattrib(prob, attrib)
```

Arguments

`prob`	The current problem.
`attrib`	Problem attribute whose value is to be returned.

Return value

The value of the problem attribute.

Further information

Please refer to the C documentation for more details.

getdblcontrol

Purpose

Retrieves the value of a given double control parameter.

Synopsis

```
getdblcontrol(prob, control)
```

Arguments

`prob`	The current problem.
`control`	Control parameter whose value is to be returned.

Return value

The control value.

getdirs

Purpose

Used to return the directives that have been loaded into a matrix.
Priorities, forced branching directions and pseudo costs can be returned. If called after presolve, `getdirs` will get the directives for the presolved problem.

Synopsis

```
getdirs(prob)
```

Argument

`prob` The current problem.

Return value

A list with the following elements:

`ndir` The number of directives.

`indices` The column numbers (`0`, `1`, `2`,...) or negative values corresponding to special ordered sets (the first set numbered `-1`, the second numbered `-2`,...).

`prios` The priorities for the columns and sets, where columns/sets with smallest priority will be branched on first.

`branchdirs` Character array of length `ndir` specifying the branching direction for each column or set:

- `U` the entity is to be forced up; May be NULL if not required.
- `D` the entity is to be forced down;
- `N` not specified.

`uppseudo` The up pseudo costs for the columns and sets.

`downpseudo` The down pseudo costs for the columns and sets.

Further information

Please refer to the C documentation for more details.

getdualray

Purpose

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

Synopsis

```
getdualray(prob)
```

Argument

`prob` The current problem.

Return value

A list with the following elements:

`ray`	The ray.
`hasray`	This variable will be set to 1 if the Optimizer is able to return a dual ray, 0 otherwise.

getduals

Purpose

Returns the dual values from the incumbent solution during or after optimization of a continuous problem with optimize, lpoptimize or `nlpoptimize`.

Synopsis

```
getduals(prob, first = 0, last = getintattrib(prob, xpress:::INPUTROWS) -
      1)
```

Arguments

`prob` The current problem.

`first` First row in the dual solution.

`last` Last row in the dual solution.

Return value

A list with the following elements:

`status` Information about the dual solution returned.

`duals` The values of the dual variables.

getgencons

Purpose

Returns the general constraints `y = f(x1, ..., xn, c1, ..., cm)` in a given range.

Synopsis

```
getgencons(prob, maxcols, maxvals, first, last)
```

Arguments

`prob`	The current problem.
`maxcols`	Maximum number of input columns to be retrieved.
`maxvals`	Maximum number of constant values to be retrieved.
`first`	First general constraint in the range.
`last`	Last general constraint in the range.

Return value

A list with the following elements:

`contype`	Integer array of length at least `last-first+1` which will be filled with the types of the general constraints: `_GENCONS_MAX (0)` indicates a `maximum` constraint; `_GENCONS_MIN (1)` indicates a `minimum` constraint; `_GENCONS_AND (2)` indicates an `and` constraint. `_GENCONS_OR (3)` indicates an `or` constraint; `_GENCONS_ABS (4)` indicates an `absolute value` constraint.
`resultant`	The indices of the output variables `y`
`colstart`	The start index of each general constraint in the `colind` array.
`colind`	The indices of the input variables `xi`
`ncols`	The number of input columns in the `colind` array.
`valstart`	The start index of each general constraint in the `val` array.
`val`	The constant values `ci`
`nvals`	The number of constant values in the `val` array.

getglobal

Purpose

Retrieves integer and entity information about a problem.

Synopsis

```
getglobal(prob)
```

Further information

This function is deprecated and will be removed from future releases. Please use `getmipentities`

getiisdata

Purpose

Returns information for an Irreducible Infeasible Set: size, variables and constraints (row and column vectors), and conflicting sides of the variables.
For pure linear problems there is also information on duals, reduced costs and isolations.

Synopsis

```
getiisdata(prob, iis)
```

Arguments

`prob` The current problem.

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

Return value

A list with the following elements:

`nrows` The number of rows in the IIS.

`ncols` The number of bounds in the IIS.

`rowind` Indices of rows in the IIS.

`colind` Indices of bounds (columns) in the IIS.

`contype` Sense of rows in the IIS:

`L`	for less or equal row; Can be NULL if not required.
`G`	for greater or equal row.
`E`	for an equality row (for a non LP IIS);
`1`	for a SOS1 row;
`2`	for a SOS2 row;
`W`	for a piecewise linear constraint;
`X`	for a general constraint;
`I`	for an indicator row.

`bndtype` Sense of bound in the IIS:

`U`	for upper bound; Can be NULL if not required.
`L`	for lower bound.
`F`	for fixed columns (for a non LP IIS);
`B`	for a binary column;
`I`	for an integer column;
`P`	for a partial integer columns;
`S`	for a semi-continuous column;
`R`	for a semi-continuous integer column.

`duals` The dual multipliers associated with the rows.

`djs` The dual multipliers (reduced costs) associated with the bounds.

`isolationrows` The isolation status of the rows:

`-1`	if isolation information is not available for row (run iis isolations); Can be NULL if not required.
`0`	if row is not in isolation;
`1`	if row is in isolation.

`isolationcols` The isolation status of the bounds:

`-1`	if isolation information is not available for column (run iis isolations); Can be NULL if not required.

0	if column is not in isolation;
1	if column is in isolation.

Further information

Please refer to the C documentation for more details.

getindex

Purpose

Returns the index for a specified row or column name.

Synopsis

```
getindex(prob, type, name)
```

Arguments

`prob` The current problem.

`type` `_NAMES_ROW (=1)` if row index is required;
`_NAMES_COLUMN (=2)` if column index is required;
`_NAMES_SET (=3)` if set index is required;
`_NAMES_PWLCONS (=4)` if piecewise linear constraint index is required;
`_NAMES_GENCONS (=5)` if general constraint index is required;
`_NAMES_OBJECTIVE (=6)` if objective index is required;
`_NAMES_USERFUNC (=7)` if user function index is required;
`_NAMES_INTERNALFUNC (=8)` if an internal function index is required.

`name` Null terminated string.

Return value

The row or column index number.

getindicators

Purpose

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

Synopsis

```
getindicators(prob, first, last)
```

Arguments

`prob` The current problem.

`first` First row in the range.

`last` Last row in the range (inclusive).

Return value

A list with the following elements:

`colind` Column indices of the indicator variables.

`complement` The indicator complement flags:

`0`	not an indicator constraint (in this case the corresponding entry in the `colind` array is ignored); May be `NULL`.
`1`	for indicator constraints with condition "`bin = 1`";
`-1`	for indicator constraints with condition "`bin = 0`".

getinfeas

Purpose

Returns a list of infeasible primal and dual variables.

Synopsis

```
getinfeas(prob)
```

Argument

`prob` The current problem.

Return value

A list with the following elements:

`nprimalcols` The number of primal infeasible variables.

`nprimalrows` The number of primal infeasible rows.

`ndualrows` The number of dual infeasible rows.

`ndualcols` The number of dual infeasible variables.

`x` The primal infeasible variables.

`slack` The primal infeasible rows.

`duals` The dual infeasible rows.

`djs` The dual infeasible variables.

getintattrib

Purpose

Enables users to recover the values of various integer problem attributes. Problem attributes are set during loading and optimization of a problem.

Synopsis

```
getintattrib(prob, attrib)
```

Arguments

`prob`	The current problem.
`attrib`	Problem attribute whose value is to be returned.

Return value

The value of the problem attribute.

Further information

Please refer to the C documentation for more details.

getintcontrol

Purpose

Enables users to recover the values of various integer control parameters

Synopsis

```
getintcontrol(prob, control)
```

Arguments

`prob`	The current problem.
`control`	Control parameter whose value is to be returned.

Return value

The value of the control.

getlastbarsol

Purpose

Used to obtain the last barrier solution values following optimization that used the barrier solver.

Synopsis

```
getlastbarsol(prob, x = TRUE, slack = TRUE, duals = TRUE, djs = TRUE)
```

Arguments

`prob`	The current problem.
`x`	Flag to indicate whether the output list should contain `x`.
`slack`	Flag to indicate whether the output list should contain `slack`.
`duals`	Flag to indicate whether the output list should contain `duals`.
`djs`	Flag to indicate whether the output list should contain `djs`.

Return value

A list with the following elements:

`x`	The values of the primal variables.
`slack`	The values of the slack variables.
`duals`	The values of the dual variables (cBTB-1).
`djs`	The reduced cost for each variable (cT-cBTB-1A).
`status`	Status of the last barrier solve.

getlasterror

Purpose

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

Synopsis

```
getlasterror(prob)
```

Argument

`prob` The current problem.

Return value

The last error message.

getlb

Purpose

Returns the lower bounds for the columns in a given range.

Synopsis

```
getlb(prob, first, last)
```

Arguments

`prob`	The current problem.
`first`	First column in the range.
`last`	Last column in the range.

Return value

Lower bounds.

getlicerrmsg

Purpose

Retrieves an error message describing the last licensing error, if any occurred.

Synopsis

```
getlicerrmsg(maxbytes = 2048)
```

Argument

`maxbytes` Length of the buffer.

Return value

The error message.

getlpsol

Purpose

Used to obtain the LP solution values following optimization.

Synopsis

```
getlpsol(prob, x = TRUE, slack = TRUE, duals = TRUE, djs = TRUE)
```

Arguments

`prob`	The current problem.
`x`	Flag to indicate whether the output list should contain `x`.
`slack`	Flag to indicate whether the output list should contain `slack`.
`duals`	Flag to indicate whether the output list should contain `duals`.
`djs`	Flag to indicate whether the output list should contain `djs`.

Return value

A list with the following elements:

`x`	The values of the primal variables.
`slack`	The values of the slack variables.
`duals`	The values of the dual variables (cBTB-1).
`djs`	The reduced cost for each variable (cT-cBTB-1A).

getlpsolval

Purpose

Deprecated Use getsolution or getcallbacksolution and related functions instead.
Used to obtain a single LP solution value following optimization.

Synopsis

```
getlpsolval(prob, col, row)
```

Arguments

`prob`	The current problem.
`col`	Column index of the variable for which to return the solution value.
`row`	Row index of the constraint for which to return the solution value.

Return value

A list with the following elements:

`x`	The primal variable.
`slack`	The slack variable.
`dual`	The dual variable (cBTB-1).
`dj`	Reduced costs for the variable (cT-cBTB-1A).

Further information

Please refer to the C documentation for more details.

getmessagestatus

Purpose

Retrieves the current suppression status of a message.

Synopsis

```
getmessagestatus(prob, msgcode)
```

Arguments

`prob`	The problem to check for the suppression status of the message error code.
`msgcode`	The id number of the message.

Return value

Non-zero if the message is not suppressed; `0` otherwise.

getmipentities

Purpose

Retrieves integr and entity information about a problem.
It must be called before mipoptimize if the presolve option is used.

Synopsis

```
getmipentities(prob)
```

Argument

`prob` The current problem.

Return value

A list with the following elements:

`nentities` The number of binary, integer, semi-continuous, semi-continuous integer and partial integer entities.

`nsets` The number of SOS1 and SOS2 sets.

`coltype` The entity types. The types will be one of:

`B`	binary variables;
`I`	integer variables;
`P`	partial integer variables;
`S`	semi-continuous variables;
`R`	semi-continuous integer variables.

`colind` The column indices of the MIP entities.

`limit` The limits for the partial integer variables and lower bounds for the semi-continuous and semi-continuous integer variables (any entries in the positions corresponding to binary and integer variables will be meaningless).

`settype` The set types. The set types will be one of:

`1`	SOS1 type sets;
`2`	SOS2 type sets.

`start` The offsets into the `setcols` and `refval` arrays indicating the start of the sets.

`setcols` The columns in each set.

`refval` The reference row entries for each member of the sets.

Further information

Please refer to the C documentation for more details.

getmipsol

Purpose

Deprecated Use getsolution and getslacks instead.
Used to obtain the solution values of the last MIP solution that was found.

Synopsis

```
getmipsol(prob, x = TRUE, slack = TRUE)
```

Arguments

`prob`	The current problem.
`x`	Flag to indicate whether the output list should contain `x`.
`slack`	Flag to indicate whether the output list should contain `slack`.

Return value

A list with the following elements:

`x`	The values of the primal variables.
`slack`	The values of the slack variables.

Further information

Please refer to the C documentation for more details.

getmipsolval

Purpose

Deprecated Use getsolution and getslacks instead.
Used to obtain a single solution value of the last MIP solution that was found.

Synopsis

```
getmipsolval(prob, col, row)
```

Arguments

`prob`	The current problem.
`col`	Column index of the variable for which to return the solution value.
`row`	Row index of the constraint for which to return the solution value.

Return value

A list with the following elements:

`x`	The primal variable.
`slack`	The slack variable.

Further information

Please refer to the C documentation for more details.

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

```
getmqobj(prob, first, last)
```

Arguments

`prob`	The current problem.
`first`	First column in the range.
`last`	Last column in the range.

Return value

A list with the following elements:

`start`	Indices indicating the starting offsets in the `colind` and `objqcoef` arrays for each requested column
`colind`	The column indices of the nonzero elements in the lower triangular part of `Q`.
`objqcoef`	The nonzero element values.

Further information

Please refer to the C documentation for more details.

getnamelist

Purpose

Returns the names for the rows, columns, sets, piecewise linear constraints, general constraints or objectives in a given range.
The names will be returned in a character buffer, with no trailing whitespace and with each name being separated by a NULL character.

Synopsis

```
getnamelist(prob, type, first, last)
```

Arguments

`prob` The current problem.

`type` `_NAMES_ROW (=1)` if row names are required;
`_NAMES_COLUMN (=2)` if column names are required;
`_NAMES_SET (=3)` if set names are required;
`_NAMES_PWLCONS (=4)` if piecewise linear constraint names are required;
`_NAMES_GENCONS (=5)` if general constraint names are required;
`_NAMES_OBJECTIVE (=6)` if objective function names are required.

`first` First row, column, set, piecewise linear or general constraint in the range.

`last` Last row, column, set, piecewise linear or general constraint in the range.

Return value

The names.

Further information

Please refer to the C documentation for more details.

getnamelistobject

Purpose

Deprecated The names list API is scheduled for removal.
Returns the `namelist` object for the rows, columns or sets of a problem. The names stored in this object can be queried using the `_nml_` functions.

Synopsis

```
getnamelistobject(prob, type)
```

Arguments

`prob` The current problem.

`type` `_NAMES_ROW (=1)` if the row name list is required;
`_NAMES_COLUMN (=2)` if the column name list is required;
`_NAMES_SET (=3)` if the set name list is required;
`_NAMES_PWLCONS (=4)` if piecewise linear constraint name list is required;
`_NAMES_GENCONS (=5)` if general constraint name list is required;
`_NAMES_OBJECTIVE (=6)` if objective name list is required.

Return value

The name list contained by the problem.

Further information

Please refer to the C documentation for more details.

getnames

Purpose

Deprecated Use getnamelist instead.
Returns the names for the rows, columns, sets, piecewise linear constraints, general constraints or objectives in a given range. The names will be returned in a character buffer, each name being separated by a null character.

Synopsis

```
getnames(prob, type, first, last)
```

Arguments

`prob` The current problem.

`type` `_NAMES_ROW (=1)` if row names are required;
`_NAMES_COLUMN (=2)` if column names are required;
`_NAMES_SET (=3)` if set names are required;
`_NAMES_PWLCONS (=4)` if piecewise linear constraint names are required;
`_NAMES_GENCONS (=5)` if general constraint names are required;
`_NAMES_OBJECTIVE (=6)` if objective function names are required;

`first` First row, column, set, piecewise linear or general constraint in the range.

`last` Last row, column, set, piecewise linear or general constraint in the range.

Return value

The names.

Further information

Please refer to the C documentation for more details.

getobj

Purpose

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

Synopsis

```
getobj(prob, first, last)
```

Arguments

`prob` The current problem.

`first` First column in the range.

`last` Last column in the range.

Return value

Objective function coefficients.

getobjdblattrib

Purpose

Retrieves the value of a given double attribute associated with a multi-objective solve.
When solving a multi-objective problem, several objectives might be optimized in sequence. After each solve, the problem attributes are captured so that they can be queried afterwards.

Synopsis

```
getobjdblattrib(prob, solveidx, attrib)
```

Arguments

`prob`	The current problem.
`solveidx`	Index of the solve to query.
`attrib`	Problem attribute whose value is to be returned.

Return value

Attribute value.

Further information

Please refer to the C documentation for more details.

getobjdblcontrol

Purpose

Retrieves the value of a given double control parameter associated with an objective function. These parameters control how the objective is treated during multi-objective optimization.

Synopsis

```
getobjdblcontrol(prob, objidx, control)
```

Arguments

`prob` The current problem.

`objidx` Index of the objective to query.

`control` Control parameter whose value is to be returned. Must be one of:

- `_OBJECTIVE_WEIGHT` get the weight of the given objective;
- `_OBJECTIVE_ABSTOL` get the absolute tolerance of the given objective;
- `_OBJECTIVE_RELTOL` get the relative tolerance of the given objective;
- `_OBJECTIVE_RHS` get the constant term of the given objective.

Return value

The control value.

Further information

Please refer to the C documentation for more details.

getobjintattrib

Purpose

Retrieves the value of a given integer attribute associated with a multi-objective solve.
When solving a multi-objective problem, several objectives might be optimized in sequence. After each solve, the problem attributes are captured so that they can be queried afterwards.

Synopsis

```
getobjintattrib(prob, solveidx, attrib)
```

Arguments

`prob`	The current problem.
`solveidx`	Index of the solve to query.
`attrib`	Problem attribute whose value is to be returned.

Return value

Attribute value.

Further information

Please refer to the C documentation for more details.

getobjintcontrol

Purpose

Retrieves the value of a given integer control parameter associated with an objective.
These parameters control how the objective is treated during multi-objective optimization.

Synopsis

```
getobjintcontrol(prob, objidx, control)
```

Arguments

`prob`	The current problem.
`objidx`	Index of the objective to query.
`control`	Control parameter whose value is to be returned. Must be one of:
	`_OBJECTIVE_PRIORITY` get the priority of the given objective.

Return value

The control value.

Further information

Please refer to the C documentation for more details.

getobjn

Purpose

For a given objective function, returns the objective coefficients for the columns in a given range.

Synopsis

```
getobjn(prob, objidx, first, last)
```

Arguments

`prob`	The current problem.
`objidx`	Index of the objective function whose coefficients to return.
`first`	First column in the range.
`last`	Last column in the range.

Return value

Objective function coefficients.

getpivotorder

Purpose

Returns the pivot order of the basic variables.

Synopsis

```
getpivotorder(prob)
```

Argument

`prob` The current problem.

Return value

The pivot order.

getpivots

Purpose

Returns a list of potential leaving variables if a specified variable enters the basis.

Synopsis

```
getpivots(prob, enter)
```

Arguments

`prob`	The current problem.
`enter`	Index of the specified row or column to enter basis.

Return value

A list with the following elements:

`outlist`	List of potential leaving variables.
`x`	The values of all the variables that would result if `enter` entered the basis.
`objval`	The objective function value that would result if `enter` entered the basis.
`npivots`	The actual number of potential leaving variables.

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

```
getpresolvebasis(prob, rowstat = TRUE, colstat = TRUE)
```

Arguments

`prob`	The current problem.
`rowstat`	Flag to indicate whether the output list should contain `rowstat`.
`colstat`	Flag to indicate whether the output list should contain `colstat`.

Return value

A list with the following elements:

`rowstat` Basis status of the stack, surplus or artificial variable associated with each row. The status will be one of:

- `_NONBASIC_LOWER (0)` slack, surplus or artificial is non-basic at lower bound; May be `NULL` if not required.
- `_BASIC (1)` slack, surplus or artificial is basic;
- `_NONBASIC_UPPER (2)` slack or surplus is non-basic at upper bound.

`colstat` The basis status of the columns in the constraint matrix. The status will be one of:

- `_NONBASIC_LOWER (0)` variable is non-basic at lower bound, or superbasic at zero if the variable has no lower bound; May be `NULL` if not required.
- `_BASIC (1)` variable is basic;
- `_NONBASIC_UPPER (2)` variable is at upper bound;
- `_SUPERBASIC (3)` variable is super-basic.

Further information

Please refer to the C documentation for more details.

getpresolvemap

Purpose

Returns the mapping of the row and column numbers from the presolve problem back to the original problem.

Synopsis

```
getpresolvemap(prob)
```

Argument

`prob` The current problem.

Return value

A list with the following elements:

`rowmap` The row maps.

`colmap` The column maps.

getpresolvesol

Purpose

Returns the solution for the presolved problem from memory.

Synopsis

```
getpresolvesol(prob, x = TRUE, slack = TRUE, duals = TRUE, djs = TRUE)
```

Arguments

`prob`	The current problem.
`x`	Flag to indicate whether the output list should contain `x`.
`slack`	Flag to indicate whether the output list should contain `slack`.
`duals`	Flag to indicate whether the output list should contain `duals`.
`djs`	Flag to indicate whether the output list should contain `djs`.

Return value

A list with the following elements:

`x`	The values of the primal variables.
`slack`	The values of the slack variables.
`duals`	The values of the dual variables.
`djs`	The reduced cost for each variable.

getprimalray

Purpose

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

Synopsis

```
getprimalray(prob)
```

Argument

`prob` The current problem.

Return value

A list with the following elements:

`ray`	The ray.
`hasray`	This variable will be set to 1 if the Optimizer is able to return a primal ray, 0 otherwise.

getprobname

Purpose

Returns the current problem name.

Synopsis

```
getprobname(prob)
```

Argument

`prob` The current problem.

Return value

A buffer of MAXPROBNAMELENGTH+1 bytes to contain the current problem name.

getpwlcons

Purpose

Returns the piecewise linear constraints `y = f(x)` in a given range.

Synopsis

```
getpwlcons(prob, maxpoints, first, last)
```

Arguments

`prob`	The current problem.
`maxpoints`	Maximum number of breakpoints to be retrieved.
`first`	First piecewise linear constraint in the range.
`last`	Last piecewise linear constraint in the range.

Return value

A list with the following elements:

`colind`	The indices of the input variables `x`
`resultant`	The indices of the output variables `y`
`start`	The start indices of the different constraints in the breakpoint arrays
`xval`	The `x`-values of the breakpoints.
`yval`	The `y`-values of the breakpoints.
`npoints`	The number of breakpoints in the selected constraints.

getqobj

Purpose

Returns a single quadratic objective function coefficient corresponding to the variable pair `(objqcol1, objqcol2)` of the Hessian matrix.

Synopsis

```
getqobj(prob, objqcol1, objqcol2)
```

Arguments

`prob`	The current problem.
`objqcol1`	Column index for the first variable in the quadratic term.
`objqcol2`	Column index for the second variable in the quadratic term.

Return value

The objective function coefficient.

getqrowcoeff

Purpose

Returns a single quadratic constraint coefficient corresponding to the variable pair (`rowqcol1`, `rowqcol2`) of the Hessian of a given constraint.

Synopsis

```
getqrowcoeff(prob, row, rowqcol1, rowqcol2)
```

Arguments

`prob`	The current problem.
`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.

Return value

The objective function coefficient.

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

```
getqrowqmatrix(prob, row, first, last)
```

Arguments

`prob`	The current problem.
`row`	Index of the row for which the quadratic coefficients are to be returned.
`first`	First column in the range.
`last`	Last column in the range.

Return value

A list with the following elements:

`start`	Indices indicating the starting offsets in the `colind` and `rowqcoef` arrays for each requested column
`colind`	The column indices of the nonzero elements in the lower triangular part of Q.
`rowqcoef`	The nonzero element values.

Further information

Please refer to the C documentation for more details.

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

```
getqrowqmatrixtriplets(prob, row)
```

Arguments

`prob`	The current problem.
`row`	Index of the row for which the quadratic coefficients are to be returned.

Return value

A list with the following elements:

`ncoefs`	Argument used to return the number of quadratic coefficients in the row.
`rowqcol1`	First index in the triplets.
`rowqcol2`	Second index in the triplets.
`rowqcoef`	Coefficients in the triplets.

Further information

Please refer to the C documentation for more details.

getqrows

Purpose

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

Synopsis

```
getqrows(prob)
```

Argument

`prob` The current problem.

Return value

A list with the following elements:

`nrows` Used to return the number of quadratic constraints in the matrix.

`rowind` The indices of rows with quadratic coefficients in them.

getredcosts

Purpose

Returns the reduced costs from the incumbent solution during or after optimization of a continuous problem with optimize, lpoptimize or `nlpoptimize`.

Synopsis

```
getredcosts(prob, first = 0, last = getintattrib(prob, xpress:::INPUTCOLS)
      - 1)
```

Arguments

`prob`	The current problem.
`first`	First column in the reduced costs.
`last`	Last column in the reduced costs.

Return value

A list with the following elements:

`status`	Information about the reduced costs returned.
`djs`	The reduced costs for the variables.

getrhs

Purpose

Returns the right hand side elements for the rows in a given range.

Synopsis

```
getrhs(prob, first, last)
```

Arguments

`prob`	The current problem.
`first`	First row in the range.
`last`	Last row in the range.

Return value

Right hand side elements.

getrhsrange

Purpose

Returns the right hand side range values for the rows in a given range.

Synopsis

```
getrhsrange(prob, first, last)
```

Arguments

`prob` The current problem.

`first` First row in the range.

`last` Last row in the range.

Return value

Right hand side range values.

getrowflags

Purpose

Retrieve if a range of rows have been set up as special rows.

Synopsis

```
getrowflags(prob, first, last)
```

Arguments

`prob`	The current problem
`first`	First row index to be checked
`last`	Last row index to be checked

Return value

Int array of length `last-first+1` where type of information (see below) will be returned

getrows

Purpose

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

Synopsis

```
getrows(prob, first, last)
```

Arguments

`prob`	The current problem.
`first`	First row in the range.
`last`	Last row in the range.

Return value

A list with the following elements:

`start`	The indices indicating the starting offsets in the `colind` and `colcoef` arrays for each requested row
`colind`	The column indices of the nonzero elements for each row.
`colcoef`	The nonzero element values.

getrowtype

Purpose

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

Synopsis

```
getrowtype(prob, first, last)
```

Arguments

`prob`	The current problem.
`first`	First row in the range.
`last`	Last row in the range.

Return value

The row types:

`N`	indicates a free constraint;
`L`	indicates a <= constraint;
`E`	indicates an = constraint;
`G`	indicates a >= constraint;
`R`	indicates a range constraint.

getscale

Purpose

Returns the the current scaling of the matrix.

Synopsis

```
getscale(prob)
```

Argument

`prob` The current problem.

Return value

A list with the following elements:

`rowscale` Integer array of size ROWS that will contain the powers of `2` with which the rows are currently scaled.

`colscale` Integer array of size COLS that will contain the powers of `2` with which the columns are currently scaled.

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

```
getscaledinfeas(prob)
```

Argument

`prob` The current problem.

Return value

A list with the following elements:

`nprimalcols`	Number of primal infeasible variables.
`nprimalrows`	Number of primal infeasible rows.
`ndualrows`	Number of dual infeasible rows.
`ndualcols`	Number of dual infeasible variables.
`x`	The primal infeasible variables.
`slack`	The primal infeasible rows.
`duals`	The dual infeasible rows.
`djs`	The dual infeasible variables.

Further information

Please refer to the C documentation for more details.

getslacks

Purpose

Returns the slack values from the incumbent solution during or after optimization with optimize, mipoptimize, lpoptimize or `nlpoptimize`.

Synopsis

```
getslacks(prob, first = 0, last = getintattrib(prob, xpress:::INPUTROWS) -
      1)
```

Arguments

`prob` The current problem.

`first` First row in the slacks.

`last` Last row in the slacks.

Return value

A list with the following elements:

`status` Information about the slacks returned.

`slacks` The value of the slack variables.

getsolution

Purpose

Returns the incumbent solution during or after optimization with optimize, mipoptimize, lpoptimize or `nlpoptimize`.

Synopsis

```
getsolution(prob, first = 0, last = getintattrib(prob, xpress:::INPUTCOLS)
      - 1)
```

Arguments

`prob`	The current problem.
`first`	First column in the solution.
`last`	Last column in the solution.

Return value

A list with the following elements:

`status`	Information about the solution returned.
`x`	The value of the primal variables.

getstrattrib

Purpose

Get string attribute.
This is the same as `getstringattrib`.

Synopsis

```
getstrattrib(prob, index)
```

getstrcontrol

Purpose

Get string control.
This is the same as `getstringcontrol`.

Synopsis

```
getstrcontrol(prob, index)
```

getstringattrib

Purpose

Enables users to recover the values of various string problem attributes. Problem attributes are set during loading and optimization of a problem.

Synopsis

```
getstringattrib(prob, attrib)
```

Arguments

`prob`	The current problem.
`attrib`	Problem attribute whose value is to be returned.

Return value

The value of the attribute.

Further information

Please refer to the C documentation for more details.

getstringcontrol

Purpose

Returns the value of a given string control parameters.

Synopsis

```
getstringcontrol(prob, control)
```

Arguments

`prob`	The current problem.
`control`	Control parameter whose value is to be returned.

Return value

The value of the control.

getub

Purpose

Returns the upper bounds for the columns in a given range.

Synopsis

```
getub(prob, first, last)
```

Arguments

`prob`	The current problem.
`first`	First column in the range.
`last`	Last column in the range.

Return value

Upper bounds.

getunbvec

Purpose

Returns the index vector which causes the primal simplex or dual simplex algorithm to determine that a matrix is primal or dual unbounded respectively.

Synopsis

```
getunbvec(prob)
```

Argument

`prob` The current problem.

Return value

The vector causing the problem to be detected as being primal or dual unbounded.

getversion

Purpose

Returns the full Optimizer version number in the form 15.10.03, where 15 is the major release, 10 is the minor release, and 03 is the build number.

Synopsis

```
getversion()
```

Return value

The version string.

getversionnumbers

Purpose

Returns the Optimizer version numbers split into major, minor, and build number.

Synopsis

```
getversionnumbers()
```

Return value

A list with the following elements:

`major`	Pointer to integer to receive the major version number.
`minor`	Pointer to integer to receive the minor version number.
`build`	Pointer to integer to receive the build number.

handlectrlc

Purpose

Handle Ctrl-C signals.
The function installs or removes a Ctrl-C handler that interacts with `prob`. Such a handler will invoke `XPRSinterrupt()` on `prob` if Ctrl-C is pressed. Note that there can be only one such handler at any time. If `handle` is `NULL` then a new handler is installed and the function returns an object that represents the previously installed handler. If `handle` is not `NULL` then the function assumes that the object was returned by a previous call to this function. In that case the previous handler is returned.

Synopsis

```
handlectrlc(prob, handle)
```

Arguments

`prob` The problem object with which to handler should interact.

`handle` `NULL` to install a new handler, non-`NULL` to restore a previous handler.

Return value

A description of the previous handler if a new handler was installed, `NULL` otherwise.

handleintr

Purpose

Install or remove a handler for interruption.
This function installs or removes a function that is invoked periodically by the solver to check whether R has requested an interruption. If such a request is detected then the solver will interrupt itself at the next opportunitiy and will return gracefully to R.

Synopsis

```
handleintr(prob, set)
```

Arguments

`prob`	The problem into which the handler should be installed or from which it should be removed.
`set`	`TRUE` to install a handler, `FALSE` to remove it.

Return value

Always returns 0 (zero).

Further information

Attention! Checking for an interrupt may result in performance degradation, that is why interrupt checking is not enabled by default. Do not call this function while a solve is in progress, i.e., do not call it from a callback.

iisall

Purpose

Performs an automated search for independent Irreducible Infeasible Sets (IIS) in an infeasible problem.

Synopsis

```
iisall(prob)
```

Argument

`prob` The current problem.

Return value

The input argument `prob`.

iisclear

Purpose

Resets the search for Irreducible Infeasible Sets (IIS).

Synopsis

```
iisclear(prob)
```

Argument

`prob` The current problem.

Return value

The input argument `prob`.

iisfirst

Purpose

Initiates a search for an Irreducible Infeasible Set (IIS) in an infeasible problem.

Synopsis

```
iisfirst(prob, mode)
```

Arguments

`prob` The current problem.

`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.

Return value

The status after the search:

`0`	success;
`1`	feasible problem;
`2`	error;
`3`	timeout or interruption.

iisisolations

Purpose

Performs the isolation identification procedure for an Irreducible Infeasible Set (IIS).
This function applies only to linear problems.

Synopsis

```
iisisolations(prob, iis)
```

Arguments

`prob` The current problem.

`iis` The number of the IIS identified by either iisfirst (IIS), iisnext (IIS `-n`) or iisall (IIS `-a`) in which the isolations should be identified.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

iisnext

Purpose

Continues the search for further Irreducible Infeasible Sets (IIS), or calls iisfirst (IIS) if no IIS has been identified yet.

Synopsis

```
iisnext(prob)
```

Argument

`prob` The current problem.

Return value

The status after the search:

0	success;
1	no more IIS could be found, or problem is feasible if no iisfirst call preceded;
2	on error (when the function returns nonzero).

iisprint

Purpose

Prints a given Irreducible Infeasible Set (IIS) in the log.
If 0 is passed as the IIS number parameter, the initial infeasible subproblem is printed.

Synopsis

```
iisprint(prob, iis)
```

Arguments

`prob`	The current problem.
`iis`	The ordinal number of the IIS to be printed.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

iisstatus

Purpose

Returns statistics on the Irreducible Infeasible Sets (IIS) found so far by iisfirst (IIS), iisnext (IIS `-n`) or iisall (IIS `-a`).

Synopsis

```
iisstatus(prob)
```

Argument

`prob` The current problem.

Return value

A list with the following elements:

`niis`	The number of IISs found so far.
`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.

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

```
iiswrite(prob, iis, filetype, filename = NULL, flags = "lp")
```

Arguments

`prob`	The current problem.
`iis`	The ordinal number of the IIS to be written.
`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.
`filename`	The name of the file to be created.
`flags`	Flags passed to the writeprob function.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

init

Purpose

Initializes the Optimizer library.
This must be called before any other library routines.

Synopsis

```
init(path = "")
```

Argument

`path` The directory where the FICO Xpress license file is located.

Return value

Always returns 0 (zero).

Further information

Please refer to the C documentation for more details.

interrupt

Purpose

Interrupts the Optimizer algorithms.

Synopsis

```
interrupt(prob, reason = 9)
```

Arguments

`prob` The current problem.

`reason` The reason for stopping. Possible reasons are:

- `_STOP_NONE` do not stop;
- `_STOP_TIMELIMIT` time limit hit;
- `_STOP_WORKLIMIT` work limit hit;
- `_STOP_CTRLC` control C hit;
- `_STOP_NODELIMIT` node limit hit;
- `_STOP_ITERLIMIT` iteration limit hit;
- `_STOP_MIPGAP` MIP gap is sufficiently small;
- `_STOP_SOLLIMIT` solution limit hit;
- `_STOP_USER` user interrupt;
- `>= 1000` user defined value.

Return value

The input argument `prob`.

license

Purpose

Wraps callable C library function XPRSlicense.
Please refer to the OEM guide for details.

Synopsis

```
license(i, c)
```

loadbasis

Purpose

Loads a basis from the user's areas.

Synopsis

```
loadbasis(prob, rowstat, colstat)
```

Arguments

`prob` The current problem.

`rowstat` Integer array of length ROWS containing the basis status of the slack, surplus or artificial variable associated with each row. The status must be one of:

- `_NONBASIC_LOWER (0)` slack, surplus or artificial is non-basic at lower bound;
- `_BASIC (1)` slack, surplus or artificial is basic;
- `_NONBASIC_UPPER (2)` slack or surplus is non-basic at upper bound.
- `_SUPERBASIC (3)` slack or surplus is super-basic.

`colstat` Integer array of length COLS containing the basis status of each of the columns in the constraint matrix. The status must be one of:

- `_NONBASIC_LOWER (0)` variable is non-basic at lower bound or superbasic at zero if the variable has no lower bound;
- `_BASIC (1)` variable is basic;
- `_NONBASIC_UPPER (2)` variable is at upper bound;
- `_SUPERBASIC (3)` variable is super-basic.

Return value

The input argument `prob`.

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

```
loadbranchdirs(prob, colind, dir = NULL, ncols = x_max_vec_length(colind))
```

Arguments

`prob`	The current problem.
`colind`	Integer array of length `ncols` containing the column numbers.
`dir`	Integer array of length `ncols` containing either 0 or 1 for the entities given in `colind`.
`ncols`	Number of directives.

Return value

The input argument `prob`.

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 delcuts.

Synopsis

```
loadcuts(
prob,
cutind,
cuttype = 0,
interp = -1,
ncuts = x_max_vec_length(cutind)
)
```

Arguments

`prob`	The current problem.
`cutind`	Array of length `ncuts` containing pointers to the cuts to be loaded into the matrix.
`cuttype`	Cut type.
`interp`	The way in which the cut type is interpreted:

`-1`	load all cuts;
`1`	treat cut types as numbers;
`2`	treat cut types as bit-vectors (compare Section) - load cut if any bit matches any bit set in `cuttype`;
`3`	treat cut types as bit-vectors (compare Section) - `0` load cut if all bits match those set in `cuttype`.

`ncuts`	Number of cuts to load.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

loaddelayedrows

Purpose

Specifies that a set of rows in the matrix will be treated as delayed rows during a tree search. These are rows that must be satisfied for any integer solution, but will not be loaded into the active set of constraints until required.

Synopsis

```
loaddelayedrows(prob, rowind, nrows = x_max_vec_length(rowind))
```

Arguments

`prob`	The current problem.
`rowind`	An array of row indices to treat as delayed rows.
`nrows`	The number of delayed rows.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

loaddirs

Purpose

Loads directives into the matrix.

Synopsis

```
loaddirs(
prob,
colind,
dir,
priority = NULL,
uppseudo = NULL,
downpseudo = NULL,
ndirs = x_max_vec_length(colind, dir)
)
```

Arguments

`prob`	The current problem.
`colind`	Integer array of length `ndirs` containing the column numbers.
`dir`	Character array of length `ndirs` specifying the branching direction for each column or set: `U` the entity is to be forced up; May be `NULL` if not required. `D` the entity is to be forced down; `N` not specified.
`priority`	Integer array of length `ndirs` containing the priorities for the columns or sets.
`uppseudo`	Double array of length `ndirs` containing the up pseudo costs for the columns or sets.
`downpseudo`	Double array of length `ndirs` containing the down pseudo costs for the columns or sets.
`ndirs`	Number of directives.

Return value

The input argument `prob`.

loadglobal

Purpose

Used to load a MIP problem into the Optimizer data structures.

Synopsis

```
loadglobal(
prob,
probname,
rowtype,
rhs,
rng,
objcoef,
start,
collen,
rowind,
rowcoef,
lb,
ub,
coltype,
entind,
limit,
settype,
setstart,
setind,
refval,
ncols = x_max_vec_length(objcoef, lb, ub),
nrows = x_max_vec_length(rowtype, rhs, rng),
nentities = x_max_vec_length(coltype, entind, limit),
nsets = x_max_vec_length(settype)
)
```

Further information

This function is deprecated and will be removed from future releases. Please use `loadmip`

loadlp

Purpose

Enables the user to pass a matrix directly to the Optimizer, rather than reading the matrix from a file.

Synopsis

```
loadlp(
prob,
probname,
rowtype,
rhs,
rng,
objcoef,
start,
collen,
rowind,
rowcoef,
lb,
ub,
ncols = x_max_vec_length(objcoef, lb, ub),
nrows = x_max_vec_length(rowtype, rhs, rng)
)
```

Arguments

`prob`	The current problem.
`probname`	A string of up to MAXPROBNAMELENGTH characters containing a names for the problem.
`rowtype`	Character array of length `nrows` 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`	Double array of length `nrows` containing the right hand side coefficients of the rows.
`rng`	Double array of length `nrows` containing the range values for range rows.
`objcoef`	Double array of length `ncols` containing the objective function coefficients.
`start`	Integer array containing the offsets in the `rowind` and `rowcoef` arrays of the start of the elements for each column.
`collen`	Integer array of length `ncols` containing the number of nonzero elements in each column.
`rowind`	Integer array containing the row indices for the nonzero elements in each column.
`rowcoef`	Double array containing the nonzero element values; length as for `rowind`.
`lb`	Double array of length `ncols` containing the lower bounds on the columns.
`ub`	Double array of length `ncols` containing the upper bounds on the columns.
`ncols`	Number of structural columns in the matrix.
`nrows`	Number of rows in the matrix (not including the objective).

Return value

The input argument `prob`.

loadlpsol

Purpose

Loads an LP solution for the problem into the Optimizer.

Synopsis

```
loadlpsol(prob, x = NULL, slack = NULL, duals = NULL, djs = NULL)
```

Arguments

`prob`	The current problem.
`x`	Optional: Double array of length COLS (for the original problem and not the presolve problem) containing the values of the variables.
`slack`	Optional: double array of length ROWS containing the values of slack variables.
`duals`	Optional: double array of length ROWS containing the values of dual variables.
`djs`	Optional: double array of length COLS containing the values of reduced costs.

Return value

The status. The status is one of:

`0`	Solution is loaded.
`1`	Solution is not loaded because the problem is in presolved status.

loadmip

Purpose

Used to load a MIP problem into the Optimizer data structures.
Integer, binary, partial integer, semi-continuous and semi-continuous integer variables can be defined, together with sets of type 1 and 2. The reference row values for the set members are passed as an array rather than specifying a reference row.

Synopsis

```
loadmip(
prob,
probname,
rowtype,
rhs,
rng,
objcoef,
start,
collen,
rowind,
rowcoef,
lb,
ub,
coltype,
entind,
limit,
settype,
setstart,
setind,
refval,
ncols = x_max_vec_length(objcoef, lb, ub),
nrows = x_max_vec_length(rowtype, rhs, rng),
nentities = x_max_vec_length(coltype, entind, limit),
nsets = x_max_vec_length(settype)
)
```

Arguments

`prob`	The current problem.
`probname`	A string of up to MAXPROBNAMELENGTH characters containing a name for the problem.
`rowtype`	Character array of length `nrows` 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`	Double array of length `nrows` containing the right hand side coefficients.
`rng`	Double array of length `nrows` containing the range values for range rows.
`objcoef`	Double array of length `ncols` containing the objective function coefficients.
`start`	Integer array containing the offsets in the `rowind` and `rowcoef` arrays of the start of the elements for each column.
`collen`	Integer array of length `ncols` containing the number of nonzero elements in each column.
`rowind`	Integer arrays containing the row indices for the nonzero elements in each column.

`rowcoef` Double array containing the nonzero element values length as for `rowind`.

`lb` Double array of length `ncols` containing the lower bounds on the columns.

`ub` Double array of length `ncols` containing the upper bounds on the columns.

`coltype` Character array of length `nentities` containing the entity types:

- `B` binary variables;
- `I` integer variables;
- `P` partial integer variables;
- `S` semi-continuous variables;
- `R` semi-continuous integer variables.

`entind` Integer array of length `nentities` containing the column indices of the MIP entities.

`limit` Double array of length `nentities` 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).

`settype` Character array of length `nsets` containing the set types:

- `1` SOS1 type sets; May be `NULL` if not required.
- `2` SOS2 type sets.

`setstart` Integer array containing the offsets in the `setind` and `refval` arrays indicating the start of the sets.

`setind` Integer array of length `setstart[nsets]-1` containing the columns in each set.

`refval` Double array of length `setstart[nsets]-1` containing the reference row entries for each member of the sets.

`ncols` Number of structural columns in the matrix.

`nrows` Number of rows in the matrix not (including the objective row).

`nentities` Number of binary, integer, semi-continuous, semi-continuous integer and partial integer entities.

`nsets` Number of SOS1 and SOS2 sets.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

loadmipsol

Purpose

Loads a starting MIP solution for the problem into the Optimizer.

Synopsis

```
loadmipsol(prob, x)
```

Arguments

`prob`	The current problem.
`x`	Double array of length COLS (for the original problem and not the presolve problem) containing the values of the variables.

Return value

The status. The status is one of:

`-1`	Solution rejected because an error occurred;
`0`	Solution accepted.

loadmiqcqp

Purpose

Used to load a mixed integer quadratic problem with quadratic constraints into the Optimizer data structure.
Such a problem may have quadratic terms in its objective function as well as in its constraints. Integer, binary, partial integer, semi-continuous and semi-continuous integer variables can be defined, together with sets of type 1 and 2. The reference row values for the set members are passed as an array rather than specifying a reference row.

Synopsis

```
loadmiqcqp(
prob,
probname,
rowtype,
rhs,
rng,
objcoef,
start,
collen,
rowind,
rowcoef,
lb,
ub,
objqcol1,
objqcol2,
objqcoef,
qrowind,
nrowqcoefs,
rowqcol1,
rowqcol2,
rowqcoef,
nentities,
nsets,
coltype,
entind,
limit,
settype,
setstart,
setind,
refval,
ncols = x_max_vec_length(objcoef, lb, ub),
nrows = x_max_vec_length(rowtype, rhs, rng),
nobjqcoefs = x_max_vec_length(objqcol1, objqcol2, objqcoef),
nqrows = x_max_vec_length(qrowind, nrowqcoefs)
)
```

Arguments

`prob`	The current problem.
`probname`	A string of up to MAXPROBNAMELENGTH characters containing a name for the problem.
`rowtype`	Character array of length `nrows` containing the row types: `L` indicates a <= constraint (use this one for quadratic constraints as well); `E` indicates an = constraint;

	`G` indicates a >= constraint;
	`R` indicates a range constraint;
	`N` indicates a nonbinding constraint.
`rhs`	Double array of length `nrows` containing the right hand side coefficients of the rows.
`rng`	Double array of length `nrows` containing the range values for range rows.
`objcoef`	Double array of length `ncols` containing the objective function coefficients.
`start`	Integer array containing the offsets in the `rowind` and `rowcoef` arrays of the start of the elements for each column.
`collen`	Integer array of length `ncols` containing the number of nonzero elements in each column.
`rowind`	Integer array containing the row indices for the nonzero elements in each column.
`rowcoef`	Double array containing the nonzero element values; length as for `rowind`.
`lb`	Double array of length `ncols` containing the lower bounds on the columns.
`ub`	Double array of length `ncols` containing the upper bounds on the columns.
`objqcol1`	Integer array of size `nobjqcoefs` containing the column index of the first variable in each quadratic term.
`objqcol2`	Integer array of size `nobjqcoefs` containing the column index of the second variable in each quadratic term.
`objqcoef`	Double array of size `nobjqcoefs` containing the quadratic coefficients.
`qrowind`	Integer array of size `nqrows`, containing the indices of rows with quadratic matrices in them.
`nrowqcoefs`	Integer array of size `nqrows`, containing the number of nonzeros in each quadratic constraint matrix.
`rowqcol1`	Integer array of size `nqcelem`, where `nqcelem` equals the sum of the elements in `nrowqcoefs` (i.e. the total number of quadratic matrix elements in all the constraints).
`rowqcol2`	Integer array of size `nqcelem`, containing the second index for the quadratic constraint matrices.
`rowqcoef`	Integer array of size `nqcelem`, containing the coefficients for the quadratic constraint matrices.
`nentities`	Number of binary, integer, semi-continuous, semi-continuous integer and partial integer entities.
`nsets`	Number of SOS1 and SOS2 sets.
`coltype`	Character array of length `nentities` containing the entity types:
	`B` binary variables;
	`I` integer variables;
	`P` partial integer variables;
	`S` semi-continuous variables;
	`R` semi-continuous integer variables.
`entind`	Integer array of length `nentities` containing the column indices of the MIP entities.
`limit`	Double array of length `nentities` 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).
`settype`	Character array of length `nsets` containing the set types:
	`1` SOS1 type sets; May be `NULL` if not required.
	`2` SOS2 type sets.
`setstart`	Integer array containing the offsets in the `setind` and `refval` arrays indicating the start of the sets.

`setind`	Integer array of length `setstart[nsets]-1` containing the columns in each set.
`refval`	Double array of length `setstart[nsets]-1` containing the reference row entries for each member of the sets.
`ncols`	Number of structural columns in the matrix.
`nrows`	Number of rows in the matrix (not including the objective row).
`nobjqcoefs`	Number of quadratic terms.
`nqrows`	Number of rows containing quadratic matrices.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

loadmiqp

Purpose

Used to load a MIQP problem, hence a MIP with quadratic objective coefficients, into the Optimizer data structures.
Integer, binary, partial integer, semi-continuous and semi-continuous integer variables can be defined, together with sets of type 1 and 2. The reference row values for the set members are passed as an array rather than specifying a reference row.

Synopsis

```
loadmiqp(
prob,
probname,
rowtype,
rhs,
rng,
objcoef,
start,
collen,
rowind,
rowcoef,
lb,
ub,
objqcol1,
objqcol2,
objqcoef,
nentities,
nsets,
coltype,
entind,
limit,
settype,
setstart,
setind,
refval,
ncols = x_max_vec_length(objcoef, lb, ub),
nrows = x_max_vec_length(rowtype, rhs, rng),
nobjqcoefs = x_max_vec_length(objqcol1, objqcol2, objqcoef)
)
```

Arguments

`prob`	The current problem.
`probname`	A string of up to MAXPROBNAMELENGTH characters containing a name for the problem.
`rowtype`	Character array of length `nrows` containing the row type:
	`L` indicates a <= constraint;
	`E` indicates an = constraint;
	`G` indicates a >= constraint;
	`R` indicates a range constraint;
	`N` indicates a nonbinding constraint.
`rhs`	Double array of length `nrows` containing the right hand side coefficients.
`rng`	Double array of length `nrows` containing the range values for range rows.
`objcoef`	Double array of length `ncols` containing the objective function coefficients.

`start`	Integer array containing the offsets in the `rowind` and `rowcoef` arrays of the start of the elements for each column.
`collen`	Integer array of length `ncols` containing the number of nonzero elements in each column.
`rowind`	Integer arrays containing the row indices for the nonzero elements in each column.
`rowcoef`	Double array containing the nonzero element values length as for `rowind`.
`lb`	Double array of length `ncols` containing the lower bounds on the columns.
`ub`	Double array of length `ncols` containing the upper bounds on the columns.
`objqcol1`	Integer array of size `nobjqcoefs` containing the column index of the first variable in each quadratic term.
`objqcol2`	Integer array of size `nobjqcoefs` containing the column index of the second variable in each quadratic term.
`objqcoef`	Double array of size `nobjqcoefs` containing the quadratic coefficients.
`nentities`	Number of binary, integer, semi-continuous, semi-continuous integer and partial integer entities.
`nsets`	Number of SOS1 and SOS2 sets.
`coltype`	Character array of length `nentities` containing the entity types: `B` binary variables; `I` integer variables; `P` partial integer variables; `S` semi-continuous variables; `R` semi-continuous integers.
`entind`	Integer array of length `nentities` containing the column indices of the MIP entities.
`limit`	Double array of length `nentities` 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).
`settype`	Character array of length `nsets` containing: `1` SOS1 type sets; May be `NULL` if not required. `2` SOS2 type sets.
`setstart`	Integer array containing the offsets in the `setind` and `refval` arrays indicating the start of the sets.
`setind`	Integer array of length `setstart[nsets]-1` containing the columns in each set.
`refval`	Double array of length `setstart[nsets]-1` containing the reference row entries for each member of the sets.
`ncols`	Number of structural columns in the matrix.
`nrows`	Number of rows in the matrix (not including the objective).
`nobjqcoefs`	Number of quadratic terms.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

loadmodelcuts

Purpose

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

Synopsis

```
loadmodelcuts(prob, rowind, nrows = x_max_vec_length(rowind))
```

Arguments

`prob`	The current problem.
`rowind`	An array of row indices to be treated as cuts.
`nrows`	The number of model cuts.

Return value

The input argument `prob`.

loadpresolvebasis

Purpose

Loads a presolved basis from the user's areas.

Synopsis

```
loadpresolvebasis(prob, rowstat, colstat)
```

Arguments

`prob` The current problem.

`rowstat` Integer array of length ROWS containing the basis status of the slack, surplus or artificial variable associated with each row. The status must be one of:

`_NONBASIC_LOWER (0)` slack, surplus or artificial is non-basic at lower bound;

`_BASIC (1)` slack, surplus or artificial is basic;

`_NONBASIC_UPPER (2)` slack or surplus is non-basic at upper bound.

`colstat` Integer array of length COLS containing the basis status of each of the columns in the matrix. The status must be one of:

`_NONBASIC_LOWER (0)` variable is non-basic at lower bound or superbasic at zero if the variable has no lower bound;

`_BASIC (1)` variable is basic;

`_NONBASIC_UPPER (2)` variable is at upper bound;

`_SUPERBASIC (3)` variable is super-basic.

Return value

The input argument `prob`.

loadpresolvedirs

Purpose

Loads directives into the presolved matrix.

Synopsis

```
loadpresolvedirs(
prob,
colind,
priority = NULL,
dir = NULL,
uppseudo = NULL,
downpseudo = NULL,
ndirs = x_max_vec_length(colind)
)
```

Arguments

`prob`	The current problem.
`colind`	Integer array of length `ndirs` containing the column numbers.
`priority`	Integer array of length `ndirs` containing the priorities for the columns or sets.
`dir`	Character array of length `ndirs` specifying the branching direction for each column or set: `U` the entity is to be forced up; May be `NULL` if not required. `D` the entity is to be forced down; `N` not specified.
`uppseudo`	Double array of length `ndirs` containing the up pseudo costs for the columns or sets.
`downpseudo`	Double array of length `ndirs` containing the down pseudo costs for the columns or sets.
`ndirs`	Number of directives.

Return value

The input argument `prob`.

loadqcqp

Purpose

Used to load a quadratic problem with quadratic side constraints into the Optimizer data structure. Such a problem may have quadratic terms in its objective function as well as in its constraints.

Synopsis

```
loadqcqp(
prob,
probname,
rowtype,
rhs,
rng,
objcoef,
start,
collen,
rowind,
rowcoef,
lb,
ub,
objqcol1,
objqcol2,
objqcoef,
qrowind,
nrowqcoef,
rowqcol1,
rowqcol2,
rowqcoef,
ncols = x_max_vec_length(objcoef, lb, ub),
nrows = x_max_vec_length(rowtype, rhs, rng),
nobjqcoefs = x_max_vec_length(objqcol1, objqcol2, objqcoef),
nqrows = x_max_vec_length(qrowind, nrowqcoef)
)
```

Arguments

`prob`	The current problem.
`probname`	A string of up to MAXPROBNAMELENGTH characters containing a name for the problem.
`rowtype`	Character array of length `nrows` containing the row types: `L` indicates a <= constraint (use this one for quadratic constraints as well); `E` indicates an = constraint; `G` indicates a >= constraint; `R` indicates a range constraint; `N` indicates a nonbinding constraint.
`rhs`	Double array of length `nrows` containing the right hand side coefficients of the rows.
`rng`	Double array of length `nrows` containing the range values for range rows.
`objcoef`	Double array of length `ncols` containing the objective function coefficients.
`start`	Integer array containing the offsets in the `rowind` and `rowcoef` arrays of the start of the elements for each column.
`collen`	Integer array of length `ncols` containing the number of nonzero elements in each column.
`rowind`	Integer array containing the row indices for the nonzero elements in each column.

`rowcoef`	Double array containing the nonzero element values; length as for `rowind`.
`lb`	Double array of length `ncols` containing the lower bounds on the columns.
`ub`	Double array of length `ncols` containing the upper bounds on the columns.
`objqcol1`	Integer array of size `nobjqcoefs` containing the column index of the first variable in each quadratic term.
`objqcol2`	Integer array of size `nobjqcoefs` containing the column index of the second variable in each quadratic term.
`objqcoef`	Double array of size `nobjqcoefs` containing the quadratic coefficients.
`qrowind`	Integer array of size `nqrows`, containing the indices of rows with quadratic matrices in them.
`nrowqcoef`	Integer array of size `nqrows`, containing the number of nonzeros in each quadratic constraint matrix.
`rowqcol1`	Integer array of size `nqcelem`, where `nqcelem` equals the sum of the elements in `nrowqcoef` (i.e. the total number of quadratic matrix elements in all the constraints).
`rowqcol2`	Integer array of size `nqcelem`, containing the second index for the quadratic constraint matrices.
`rowqcoef`	Integer array of size `nqcelem`, containing the coefficients for the quadratic constraint matrices.
`ncols`	Number of structural columns in the matrix.
`nrows`	Number of rows in the matrix (not including the objective row).
`nobjqcoefs`	Number of quadratic terms.
`nqrows`	Number of rows containing quadratic matrices.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

loadqcqpglobal

Purpose

Used to load a mixed integer quadratic problem with quadratic constraints into the Optimizer data structure.

Synopsis

```
loadqcqpglobal(
prob,
probname,
rowtype,
rhs,
rng,
objcoef,
start,
collen,
rowind,
rowcoef,
lb,
ub,
objqcol1,
objqcol2,
objqcoef,
qrowind,
nrowqcoefs,
rowqcol1,
rowqcol2,
rowqcoef,
nentities,
nsets,
coltype,
entind,
limit,
settype,
setstart,
setind,
refval,
ncols = x_max_vec_length(objcoef, lb, ub),
nrows = x_max_vec_length(rowtype, rhs, rng),
nobjqcoefs = x_max_vec_length(objqcol1, objqcol2, objqcoef),
nqrows = x_max_vec_length(qrowind, nrowqcoefs)
)
```

Further information

This function is deprecated and will be removed from future releases. Please use `loadmiqcqp`

loadqglobal

Purpose

Used to load a MIQP problem, hence a MIP with quadratic objective coefficients, into the Optimizer data structures.

Synopsis

```
loadqglobal(
prob,
probname,
rowtype,
rhs,
rng,
objcoef,
start,
collen,
rowind,
rowcoef,
lb,
ub,
objqcol1,
objqcol2,
objqcoef,
nentities,
nsets,
coltype,
entind,
limit,
settype,
setstart,
setind,
refval,
ncols = x_max_vec_length(objcoef, lb, ub),
nrows = x_max_vec_length(rowtype, rhs, rng),
nobjqcoefs = x_max_vec_length(objqcol1, objqcol2, objqcoef)
)
```

Further information

This function is deprecated and will be removed from future releases. Please use `loadmiqp`

loadqp

Purpose

Used to load a quadratic problem into the Optimizer data structure.
Such a problem may have quadratic terms in its objective function, although not in its constraints.

Synopsis

```
loadqp(
prob,
probname,
rowtype,
rhs,
rng,
objcoef,
start,
collen,
rowind,
rowcoef,
lb,
ub,
objqcol1,
objqcol2,
objqcoef,
ncols = x_max_vec_length(objcoef, lb, ub),
nrows = x_max_vec_length(rowtype, rhs, rng),
nobjqcoefs = x_max_vec_length(objqcol1, objqcol2, objqcoef)
)
```

Arguments

`prob`	The current problem.
`probname`	A string of up to MAXPROBNAMELENGTH characters containing a names for the problem.
`rowtype`	Character array of length `nrows` 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`	Double array of length `nrows` containing the right hand side coefficients of the rows.
`rng`	Double array of length `nrows` containing the range values for range rows.
`objcoef`	Double array of length `ncols` containing the objective function coefficients.
`start`	Integer array containing the offsets in the `rowind` and `rowcoef` arrays of the start of the elements for each column.
`collen`	Integer array of length `ncols` containing the number of nonzero elements in each column.
`rowind`	Integer array containing the row indices for the nonzero elements in each column.
`rowcoef`	Double array containing the nonzero element values; length as for `rowind`.
`lb`	Double array of length `ncols` containing the lower bounds on the columns.
`ub`	Double array of length `ncols` containing the upper bounds on the columns.
`objqcol1`	Integer array of size `nobjqcoefs` containing the column index of the first variable in each quadratic term.

`objqcol2`	Integer array of size `nobjqcoefs` containing the column index of the second variable in each quadratic term.
`objqcoef`	Double array of size `nobjqcoefs` containing the quadratic coefficients.
`ncols`	Number of structural columns in the matrix.
`nrows`	Number of rows in the matrix (not including the objective row).
`nobjqcoefs`	Number of quadratic terms.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

loadsecurevecs

Purpose

Allows the user to mark rows and columns in order to prevent the presolve removing these rows and columns from the matrix.

Synopsis

```
loadsecurevecs(
prob,
rowind,
colind,
nrows = x_max_vec_length(rowind),
ncols = x_max_vec_length(colind)
)
```

Arguments

`prob`	The current problem.
`rowind`	Integer array of length `nrows` containing the rows to be marked.
`colind`	Integer array of length `ncols` containing the columns to be marked.
`nrows`	Number of rows to be marked.
`ncols`	Number of columns to be marked.

Return value

The input argument `prob`.

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

```
lpoptimize(prob, flags = "")
```

Arguments

`prob` The current problem.

`flags` Flags to pass to `lpoptimize` (`LPOPTIMIZE`). The default is `""` or `NULL`, in which case the algorithm used is determined by the DEFAULTALG control. If the argument includes:

- `b` the problem will be solved using the Newton barrier method, or the Hybrid gradient method if BARALG is set to 4;
- `p` the problem will be solved using the primal simplex algorithm;
- `d` the problem will be solved using the dual simplex algorithm;
- `n` (lower case `N`), the network part of the problem will be identified and solved using the network simplex algorithm;

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

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

```
mipoptimize(prob, flags = "")
```

Arguments

`prob` The current problem.

`flags` Flags to pass to mipoptimize (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 (or the hybrid gradient method if BARALG is set to 4);
- `p` the initial continuous relaxation will be solved using the primal simplex algorithm;
- `d` the initial continuous relaxation will be solved using the dual simplex algorithm;
- `n` the network part of the initial continuous relaxation will be identified and solved using the network simplex algorithm;
- `l` stop after having solved the initial continous relaxation.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

nlpaddformulas

Purpose

Add non-linear formulas to the SLP problem.

Synopsis

```
nlpaddformulas(
prob,
rowind,
formulastart,
parsed,
type,
value,
ncoefs = x_max_vec_length(rowind)
)
```

Arguments

`prob`	The current SLP problem.
`rowind`	Integer array holding index of row for the coefficient.
`formulastart`	Integer array of length `ncoefs+1` holding the start position in the arrays `type` and `value` of the formula for the coefficients.
`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`.
`ncoefs`	Number of non-linear coefficients to be added.

Return value

The input argument `prob`.

nlpchgformula

Purpose

Add or replace a single matrix formula using a parsed or unparsed formula

Synopsis

```
nlpchgformula(prob, row, parsed, type, value)
```

Arguments

`prob`	The current SLP problem.
`row`	The index of the matrix row for the coefficient.
`parsed`	Integer indicating the 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 description and formula for each item.
`value`	Array of values corresponding to the types in `type`.

Return value

The input argument `prob`.

nlpchgformulastr

Purpose

Add or replace a single matrix formula using a character string for the formula.

Synopsis

```
nlpchgformulastr(prob, row, formula)
```

Arguments

`prob`	The current problem.
`row`	The index of the matrix row for the coefficient.
`formula`	Character string holding the formula with the tokens separated by spaces.

Return value

The input argument `prob`.

nlpdelformulas

Purpose

Delete nonlinear formulas from the current problem

Synopsis

```
nlpdelformulas(prob, rowind, nformulas = x_max_vec_length(rowind))
```

Arguments

`prob`	The current SLP problem.
`rowind`	Row indices of the SLP nonlinear formulas to delete.
`nformulas`	Number of SLP nonlinear formulas to delete.

Return value

The input argument `prob`.

nlpevaluateformula

Purpose

Evaluate a formula using the current values of the variables

Synopsis

```
nlpevaluateformula(prob, parsed, type, values)
```

Arguments

`prob`	The current SLP problem.
`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`	Double array of values corresponding to `type`.

Return value

Address of a double precision value to receive the result of the calculation.

nlpgetformula

Purpose

Retrieve a single matrix formula as a formula split into tokens.

Synopsis

```
nlpgetformula(prob, row, parsed, maxtypes)
```

Arguments

`prob`	The current SLP problem.
`row`	Integer holding the row index for the formula.
`parsed`	Integer indicating whether the formula of the row is to be returned in internal unparsed format (`parsed`=0) or parsed (reverse Polish) format (`parsed`=1).
`maxtypes`	Maximum number of tokens to return, i.e., the length of the type and value arrays.

Return value

A list with the following elements:

`ntypes`	Will be set to the length of the formula, including the `XSLP_EOF` token.
`type`	Integer array to hold the token types for the formula.
`value`	Double array of values corresponding to `type`.

nlpgetformularows

Purpose

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

Synopsis

```
nlpgetformularows(prob)
```

Argument

`prob` The current SLP problem.

Return value

A list with the following elements:

`nformulas` Integer used to return the total number of nonlinear formulas in the problem.

`rowind` Integer array used for returning the row positions of the nonlinear formulas.

nlpgetformulastr

Purpose

Retrieve a single matrix formula in a character string.

Synopsis

```
nlpgetformulastr(prob, row)
```

Arguments

`prob`	The current SLP problem.
`row`	Integer holding the row index for the formula.

Return value

Character buffer in which the formula will be placed in the same format as used for input from a file.

nlploadformulas

Purpose

Load non-linear formulas into the SLP problem

Synopsis

```
nlploadformulas(
prob,
rowind,
formulastart,
parsed,
type,
value,
nnlpcoefs = x_max_vec_length(rowind)
)
```

Arguments

`prob`	The current SLP problem.
`rowind`	Integer array holding index of row for the coefficient.
`formulastart`	Integer array of length `nnlpcoefs+1` holding the start position in the arrays `type` and `value` of the formula for the coefficients.
`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`.
`nnlpcoefs`	Number of non-linear coefficients to be loaded.

Return value

The input argument `prob`.

nlpoptimize

Purpose

Maximize or minimize an SLP problem

Synopsis

```
nlpoptimize(prob, flags = "")
```

Arguments

`prob` The current SLP problem.

`flags` These have the same meaning as for `maxim` and `minim`.

Return value

The input argument `prob`.

nlppostsolve

Purpose

Restores the problem to its pre-solve state

Synopsis

```
nlppostsolve(prob)
```

Argument

`prob` The current SLP problem.

Return value

The input argument `prob`.

nlpprintevalinfo

Purpose

Print a summary of any evaluation errors that may have occurred during solving a problem

Synopsis

```
nlpprintevalinfo(prob)
```

Argument

`prob` The current SLP problem.

Return value

The input argument `prob`.

nlpsetcurrentiv

Purpose

Transfer the current solution to initial values

Synopsis

```
nlpsetcurrentiv(prob)
```

Argument

`prob` The current SLP problem.

Return value

The input argument `prob`.

nlpsetfunctionerror

Purpose

Set the function error flag for the problem

Synopsis

```
nlpsetfunctionerror(prob)
```

Argument

`prob` The current SLP problem.

Return value

The input argument `prob`.

nlpsetinitval

Purpose

Set the initial value of a variable

Synopsis

```
nlpsetinitval(prob, colind, initial, nvars = x_max_vec_length(colind,
      initial))
```

Arguments

`prob`	The current SLP problem.
`colind`	Array of length `nvars` with index of the column for which the initial value is provided.
`initial`	Array of length `nvars` with the initial value.
`nvars`	Number of variables for which the initial value is to be set.

Return value

The input argument `prob`.

nlpvalidate

Purpose

Validate the feasibility of constraints in a converged solution

Synopsis

```
nlpvalidate(prob)
```

Argument

`prob` The current SLP problem.

Return value

The input argument `prob`.

nlpvalidatekkt

Purpose

Validates the first order optimality conditions also known as the Karush-Kuhn-Tucker (KKT) conditions versus the currect solution

Synopsis

```
nlpvalidatekkt(prob, mode, respectbasis, updatemult, violtarget)
```

Arguments

`prob` The current SLP problem.

`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.

Return value

The input argument `prob`.

nlpvalidaterow

Purpose

Prints an extensive analysis on a given constraint of the SLP problem

Synopsis

```
nlpvalidaterow(prob, row)
```

Arguments

`prob`	The current SLP problem.
`row`	The index of the row to be analyzed

Return value

The input argument `prob`.

nlpvalidatevector

Purpose

Validate the feasibility of constraints for a given solution

Synopsis

```
nlpvalidatevector(prob, solution = NULL)
```

Arguments

`prob`	The current SLP problem.
`solution`	A vector of length `COLS` containing the solution vector to be checked.

Return value

A list with the following elements:

`suminf`	The sum of infeasibility.
`sumscaledinf`	The sum of scaled (relative) infeasibility.
`objval`	The net objective.

nml_addnames

Purpose

Deprecated The names list API is scheduled for removal.
The `_nml_*` functions provide a simple, generic interface to lists of names, which may be names of rows/columns on a problem or may be a list of arbitrary names provided by the user. Use the `_nml_addnames` to add names to a name list, or modify existing names on a namelist.

Synopsis

```
nml_addnames(nml, names, first, last)
```

Arguments

`nml`	The name list to which you want to add names.
`names`	Character buffer containing the null-terminated string names.
`first`	The index of the first name to add/replace.
`last`	The index of the last name to add/replace.

Return value

The input argument `nml`.

Further information

Please refer to the C documentation for more details.

nml_copynames

Purpose

Deprecated The names list API is scheduled for removal.
The `_nml_*` functions provide a simple, generic interface to lists of names, which may be names of rows/columns on a problem or may be a list of arbitrary names provided by the user. `_nml_copynames` allows you to copy all the names from one name list to another. As name lists representing row/column names cannot be modified, `_nml_copynames` will be most often used to copy such names to a namelist where they can be modified, for some later use.

Synopsis

```
nml_copynames(dest, src)
```

Arguments

`dest`	The namelist object to copy names to.
`src`	The namelist object from which all the names should be copied.

Return value

The input argument `dest`.

Further information

Please refer to the C documentation for more details.

nml_create

Purpose

Deprecated The names list API is scheduled for removal.
The `_nml_*` functions provide a simple, generic interface to lists of names, which may be names of rows/columns on a problem or may be a list of arbitrary names provided by the user. `_nml_create` will create a new namelist to which the user can add, remove and otherwise modify names.

Synopsis

```
nml_create()
```

Return value

The new namelist.

Further information

Please refer to the C documentation for more details.

nml_destroy

Purpose

Deprecated The names list API is scheduled for removal.
Destroys a namelist and frees any memory associated with it. Note you need only destroy namelists created by `_nml_destroy` - those returned by getnamelistobject are automatically destroyed when you destroy the problem object.

Synopsis

```
nml_destroy(nml)
```

Argument

`nml` The namelist to be destroyed.

Return value

The input argument `nml`.

Further information

Please refer to the C documentation for more details.

nml_findname

Purpose

Deprecated The names list API is scheduled for removal.
The `_nml_*` functions provide a simple, generic interface to lists of names, which may be names of rows/columns on a problem or may be a list of arbitrary names provided by the user. `_nml_findname` returns the index of the given name in the given name list.

Synopsis

```
nml_findname(nml, name)
```

Arguments

`nml`	The namelist in which to look for the name.
`name`	Null-terminated string containing the name for which to search.

Return value

The index of the name is returned, or in which -1 if the name is not found in the namelist.

Further information

Please refer to the C documentation for more details.

nml_getlasterror

Purpose

Deprecated The names list API is scheduled for removal.
Returns the last error encountered during a call to a namelist object.

Synopsis

```
nml_getlasterror(nml)
```

Argument

`nml` The namelist object.

Return value

A list with the following elements:

`msgcode` The error code.

`msg` The last error message relating to this namelist.

Further information

Please refer to the C documentation for more details.

nml_getmaxnamelen

Purpose

Deprecated The names list API is scheduled for removal.
The `_nml_*` functions provide a simple, generic interface to lists of names, which may be names of rows/columns on a problem or may be a list of arbitrary names provided by the user.
`_nml_getmaxnamelen` returns the length of the longest name in the namelist.

Synopsis

```
nml_getmaxnamelen(nml)
```

Argument

`nml` The namelist object.

Return value

The length of the longest name.

Further information

Please refer to the C documentation for more details.

nml_getnamecount

Purpose

Deprecated The names list API is scheduled for removal.
The _nml_* functions provide a simple, generic interface to lists of names, which may be names of rows/columns on a problem or may be a list of arbitrary names provided by the user.
_nlm_getnamecount returns the number of names in the namelist.

Synopsis

```
nml_getnamecount(nml)
```

Argument

`nml` The namelist object.

Return value

The number of names.

Further information

Please refer to the C documentation for more details.

nml_getnames

Purpose

Deprecated The names list API is scheduled for removal.
The `_nml_*` functions provide a simple, generic interface to lists of names, which may be names of rows/columns on a problem or may be a list of arbitrary names provided by the user. The `_nml_getnames` function returns some of the names held in the name list. The names shall be returned in a character buffer, and with each name being separated by a `NULL` character.

Synopsis

```
nml_getnames(nml, first, last, pad = 0)
```

Arguments

`nml`	The namelist object.
`first`	The index of the first name in the namelist to return.
`last`	The index of the last name in the namelist to return.
`pad`	The minimum length of each name.

Return value

The names.

Further information

Please refer to the C documentation for more details.

nml_removenames

Purpose

Deprecated The names list API is scheduled for removal.
The `_nml_*` functions provide a simple, generic interface to lists of names, which may be names of rows/columns on a problem or may be a list of arbitrary names provided by the user.
`_nml_removenames` will remove the specified names from the name list. Any subsequent names will be moved down to replace the removed names.

Synopsis

```
nml_removenames(nml, first, last)
```

Arguments

`nml`	The name list from which you want to remove names.
`first`	The index of the first name to remove.
`last`	The index of the last name to remove.

Return value

The input argument `nml`.

Further information

Please refer to the C documentation for more details.

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

```
objsa(prob, colind, ncols = x_max_vec_length(colind))
```

Arguments

`prob`	The current problem.
`colind`	Integer array of length `ncols` containing the indices of the columns whose objective function coefficients sensitivity ranges are required.
`ncols`	Number of objective function coefficients whose sensitivity is sought.

Return value

A list with the following elements:

`lower`	The objective function lower range values.
`upper`	The objective function upper range values.

Further information

Please refer to the C documentation for more details.

optimize

Purpose

This function begins a search for the optimal solution of the problem.
The direction of optimization is given by OBJSENSE.

Synopsis

```
optimize(prob, flags = "")
```

Arguments

`prob` The current problem.

`flags` Flags to pass to `optimize` (`OPTIMIZE`). The default is `""` or `NULL`. If the argument includes:

`s`	solve the problem to local optimality;
`x`	solve the problem to global optimality;
`l`	if a branch and bound search is necessary to solve the problem, stop after solving the root node.

Return value

A list with the following elements:

`solvestatus` The solve status after termination.

`solstatus` The solution status after termination.

Further information

Please refer to the C documentation for more details.

pivot

Purpose

Performs a simplex pivot by bringing variable `enter` into the basis and removing `leave`.

Synopsis

```
pivot(prob, enter, leave)
```

Arguments

`prob`	The current problem.
`enter`	Index of row or column to enter basis.
`leave`	Index of row or column to leave basis.

Return value

The input argument `prob`.

postsolve

Purpose

Postsolve the current matrix when it is in a presolved state.

Synopsis

```
postsolve(prob)
```

Argument

`prob` The current problem.

Return value

The input argument `prob`.

postsolvesol

Purpose

Postsolves a primal solution formulated in the presolved space into the corresponding solution formulated in the original space.
The problem itself is unchanged.

Synopsis

```
postsolvesol(prob, prex)
```

Arguments

`prob` The current problem.

`prex` Double array of length COLS with the values of the primal variables in the presolved space.

Return value

The values of the primal variables.

Further information

Please refer to the C documentation for more details.

presolverow

Purpose

Presolves a row formulated in terms of the original variables such that it can be added to a presolved matrix.

Synopsis

```
presolverow(
prob,
rowtype,
origcolind,
origrowcoef,
origrhs,
norigcoefs = x_max_vec_length(origcolind, origrowcoef)
)
```

Arguments

`prob`	The current problem.
`rowtype`	The type of the row: `L` indicates a <= row; `G` indicates a >= row.
`origcolind`	Integer array of length `norigcoefs` containing the column indices of the row to presolve.
`origrowcoef`	Double array of length `norigcoefs` containing the non-zero coefficients of the row to presolve.
`origrhs`	The right-hand side constant of the row to presolve.
`norigcoefs`	Number of elements in the `origcolind` and `origrowcoef` arrays.

Return value

A list with the following elements:

`ncoefs`	The number of elements in the `colind` and `rowcoef` arrays.
`colind`	The column indices of the presolved row
`rowcoef`	The coefficients of the presolved row
`rhs`	The presolved right-hand side.
`status`	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.

print.XPRSboundsRef

Purpose

Print an XPRESS bound reference.

Synopsis

```
print.XPRSboundsRef(bnd, ...)
```

Argument

`bnd` The bound reference to print.

print.XPRSbranchobject

Purpose

Print an XPRESS branching object.

Synopsis

```
print.XPRSbranchobject(obj, ...)
```

Argument

`objcoef` The branching object to be printed.

print.XPRScut

Purpose

Print an XPRESS cut.

Synopsis

```
print.XPRScut(cut, ...)
```

Argument

`cut` The cut to print.

print.XPRSnamelist

Purpose

Print an XPRESS name list.

Synopsis

```
print.XPRSnamelist(namelist, ...)
```

Argument

`namelist` The name list to print.

print.XPRSprob

Purpose

Print an XPRESS problem.

Synopsis

```
print.XPRSprob(prob, ...)
```

Argument

`prob` The problem to be printed

print.XPRSvoid

Purpose

Print an external generic pointer.

Synopsis

```
print.XPRSvoid(ptr, ...)
```

Argument

`ptr` The pointer to print

problemdata_validation_list

Purpose

List of input names allowed for problemdata.
We allow either Matrix Style or C API Style for the inputs.

Synopsis

```
problemdata_validation_list
```

Further information

This is a list of lists, which splits all acceptable arguments into C-Style Xpress and Matrix-style The validation code checks that only names from this list appear, and that the mutually exclusive alternatives are not mixed

readbasis

Purpose

Instructs the Optimizer to read in a previously saved basis from a file.

Synopsis

```
readbasis(prob, filename = "", flags = "")
```

Arguments

`prob` The current problem.

`filename` A string of up to MAXPROBNAMELENGTH characters containing the file name from which the basis is to be read.

`flags` Flags to pass to `readbasis` (`READBASIS`): CPLEX compatibility; (no effect, kept for compatibility);

- `n` input basis file containing basic solution values;
- `t` input a compact advanced form of the basis;
- `v` use the provided filename verbatim, without appending the `.bss` extension;
- `z` read a compressed input file.

Return value

The input argument `prob`.

readbinsol

Purpose

Reads a solution from a binary solution file.

Synopsis

```
readbinsol(prob, filename = "", flags = "")
```

Arguments

`prob`	The current problem.
`filename`	A string of up to MAXPROBNAMELENGTH characters containing the file name from which the solution is to be read.
`flags`	Flags to pass to `readbinsol` (`READBINSOL`):

`m`	load the solution as a solution for the MIP;
`x`	load the solution as a solution for the LP;
`v`	use the provided filename verbatim, without appending the `.sol` extension;
`z`	read a compressed input file.

Return value

The input argument `prob`.

readdirs

Purpose

Reads a directives file to help direct the tree search.

Synopsis

```
readdirs(prob, filename = NULL)
```

Arguments

`prob`	The current problem.
`filename`	A string of up to MAXPROBNAMELENGTH characters containing the file name from which the directives are to be read.

Return value

The input argument `prob`.

readprob

Purpose

Reads an (X)MPS or LP format matrix from file.

Synopsis

```
readprob(prob, filename, flags = "")
```

Arguments

`prob`	The current problem.
`filename`	The path and file name from which the problem is to be read.
`flags`	Flags to be passed:
	`l` only `filename.lp` is searched for;
	`v` use the provided filename verbatim, without appending the `.mps`, `.mat` or `.lp` extension;
	`z` read a compressed input file.

Return value

The input argument `prob`.

readslxsol

Purpose

Reads an ASCII solution file `.slx` created by the writeslxsol function.

Synopsis

```
readslxsol(prob, filename = "", flags = "")
```

Arguments

`prob`	The current problem.
`filename`	A string of up to MAXPROBNAMELENGTH characters containing the file name to which the solution is to be read.
`flags`	Flags to pass to `readslxsol` (`READSLXSOL`): non-breaking-whitespace conversion;
	`l` read the solution as an LP solution in case of a MIP problem;
	`m` read the solution as a solution for the MIP problem;
	`a` read multiple MIP solutions from the `.slx` file and add them to the MIP problem;
	`v` use the provided filename verbatim, without appending the `.slx` extension;
	`z` read a compressed input file.

Return value

The input argument `prob`.

refinemipsol

Purpose

Deprecated Please use REFINEOPS instead.
Executes the MIP solution refiner.

Synopsis

```
refinemipsol(prob, options, solution, flags = "")
```

Arguments

`prob`	The current problem.
`options`	Refinement options:
	`0` Reducing MIP fractionality is priority (If bit 10 of REFINEOPS is set, will switch to other mode if unsuccessful).
	`1` Reducing LP infeasibility is priority.
`solution`	The MIP solution to refine.
`flags`	Flags passed to any optimization calls during refinement.

Return value

A list with the following elements:

`refined`	The refined MIP solution in case of success
`status`	Refinement results:
	`0` An error has occurred
	`1` The solution has been refined
	`2` Current solution meets target criteria
	`3` Solution cannot be refined
	`5` The solution has been refined, but MIP fractionality could not be reduced.

Further information

Please refer to the C documentation for more details.

removecbafterobjective

Purpose

Remove all `afterobjective` callbacks.
Removes any callback function registered for `afterobjective` events from `prob`.

Synopsis

```
removecbafterobjective(prob)
```

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecbbariteration

Purpose

Remove all `bariteration` callbacks.
Removes any callback function registered for `bariteration` events from `prob`.

Synopsis

```
removecbbariteration(prob)
```

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecbbarlog

Purpose

Remove all `barlog` callbacks.
Removes any callback function registered for `barlog` events from `prob`.

Synopsis

```
removecbbarlog(prob)
```

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecbbeforeobjective

Purpose

Remove all `beforeobjective` callbacks.
Removes any callback function registered for `beforeobjective` events from `prob`.

Synopsis

```
removecbbeforeobjective(prob)
```

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecbchecktime

Purpose

Remove all `checktime` callbacks.
Removes any callback function registered for `checktime` events from `prob`.

Synopsis

`removecbchecktime(prob)`

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecbchgbranchobject

Purpose

Remove all `chgbranchobject` callbacks.
Removes any callback function registered for `chgbranchobject` events from `prob`.

Synopsis

`removecbchgbranchobject(prob)`

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecbcomputerestart

Purpose

Remove all `computerestart` callbacks.
Removes any callback function registered for `computerestart` events from `prob`.

Synopsis

```
removecbcomputerestart(prob)
```

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecbcutlog

Purpose

Remove all `cutlog` callbacks.
Removes any callback function registered for `cutlog` events from `prob`.

Synopsis

```
removecbcutlog(prob)
```

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecbcutround

Purpose

Remove all `cutround` callbacks.
Removes any callback function registered for `cutround` events from `prob`.

Synopsis

```
removecbcutround(prob)
```

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecbdestroymt

Purpose

Remove all `destroymt` callbacks.
Removes any callback function registered for `destroymt` events from `prob`.

Synopsis

`removecbdestroymt(prob)`

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecbgapnotify

Purpose

Remove all `gapnotify` callbacks.
Removes any callback function registered for `gapnotify` events from `prob`.

Synopsis

`removecbgapnotify(prob)`

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecbgloballog

Purpose

Remove all `miplog` callbacks.

Synopsis

```
removecbgloballog(prob)
```

Further information

This function is deprecated and will be removed from future releases. Please use `removecbmiplog`

removecbinfnode

Purpose

Remove all `infnode` callbacks.
Removes any callback function registered for `infnode` events from `prob`.

Synopsis

```
removecbinfnode(prob)
```

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecbintsol

Purpose

Remove all `intsol` callbacks.
Removes any callback function registered for `intsol` events from `prob`.

Synopsis

```
removecbintsol(prob)
```

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecblplog

Purpose

Remove all `lplog` callbacks.
Removes any callback function registered for `lplog` events from `prob`.

Synopsis

```
removecblplog(prob)
```

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecbmessage

Purpose

Remove all `message` callbacks.
Removes any callback function registered for `message` events from `prob`.

Synopsis

```
removecbmessage(prob)
```

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecbmiplog

Purpose

Remove all `miplog` callbacks.
Removes any callback function registered for `miplog` events from `prob`.

Synopsis

```
removecbmiplog(prob)
```

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecbmipthread

Purpose

Remove all `mipthread` callbacks.
Removes any callback function registered for `mipthread` events from `prob`.

Synopsis

```
removecbmipthread(prob)
```

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecbnewnode

Purpose

Remove all `newnode` callbacks.
Removes any callback function registered for `newnode` events from `prob`.

Synopsis

```
removecbnewnode(prob)
```

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecbnodecutoff

Purpose

Remove all `nodecutoff` callbacks.
Removes any callback function registered for `nodecutoff` events from `prob`.

Synopsis

```
removecbnodecutoff(prob)
```

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecbnodelpsolved

Purpose

Remove all `nodelpsolved` callbacks.
Removes any callback function registered for `nodelpsolved` events from `prob`.

Synopsis

```
removecbnodelpsolved(prob)
```

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecboptnode

Purpose

Remove all `optnode` callbacks.
Removes any callback function registered for `optnode` events from `prob`.

Synopsis

```
removecboptnode(prob)
```

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecbpreintsol

Purpose

Remove all `preintsol` callbacks.
Removes any callback function registered for `preintsol` events from `prob`.

Synopsis

```
removecbpreintsol(prob)
```

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecbprenode

Purpose

Remove all `prenode` callbacks.
Removes any callback function registered for `prenode` events from `prob`.

Synopsis

```
removecbprenode(prob)
```

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecbpresolve

Purpose

Remove all `presolve` callbacks.
Removes any callback function registered for `presolve` events from `prob`.

Synopsis

```
removecbpresolve(prob)
```

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

removecbusersolnotify

Purpose

Remove all `usersolnotify` callbacks.
Removes any callback function registered for `usersolnotify` events from `prob`.

Synopsis

```
removecbusersolnotify(prob)
```

Argument

`prob` The problem pointer from which callbacks are removed.

Return value

Always returns 0 (zero).

repairinfeas

Purpose

Provides a simplified interface for repairweightedinfeas.

Synopsis

```
repairinfeas(
prob,
penalty,
phase2,
flags,
lepref,
gepref,
lbpref,
ubpref,
delta
)
```

Arguments

`prob`	The current problem.	
`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 flags to be passed to optimize.	
`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.	

Return value

The status after the relaxation:

`0`	relaxed optimum found;
`1`	relaxed problem is infeasible;
`2`	relaxed problem is unbounded;
`3`	solution of the relaxed problem regarding the original objective is nonoptimal;
`4`	error (when return code is nonzero);
`5`	numerical instability;
`6`	analysis of an infeasible relaxation was performed, but the relaxation is feasible.

repairweightedinfeas

Purpose

By relaxing a set of selected constraints and bounds of an infeasible problem, it attempts to identify a 'solution' that violates the selected set of constraints and bounds minimally, while satisfying all other constraints and bounds.
Among such solution candidates, it selects one that is optimal regarding the original objective function.
For the console version, see REPAIRINFEAS.

Synopsis

```
repairweightedinfeas(
prob,
lepref,
gepref,
lbpref,
ubpref,
phase2,
delta,
flags
)
```

Arguments

`prob` The current problem.

`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 optimize.

Return value

The status after the relaxation:

- `0` relaxed optimum found;
- `1` relaxed problem is infeasible;
- `2` relaxed problem is unbounded;
- `3` solution of the relaxed problem regarding the original objective is nonoptimal;
- `4` error (when return code is nonzero);
- `5` numerical instability;
- `6` analysis of an infeasible relaxation was performed, but the relaxation is feasible.

Further information

Please refer to the C documentation for more details.

repairweightedinfeasbounds

Purpose

An extended version of repairweightedinfeas that allows for bounding the level of relaxation allowed.

Synopsis

```
repairweightedinfeasbounds(
prob,
lepref,
gepref,
lbpref,
ubpref,
lerelax,
gerelax,
lbrelax,
ubrelax,
phase2,
delta,
flags
)
```

Arguments

`prob`	The current problem.
`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 optimize.

Return value

The status after the relaxation:

`0`	relaxed optimum found;

1 relaxed problem is infeasible;
2 relaxed problem is unbounded;
3 solution of the relaxed problem regarding the original objective is nonoptimal;
4 error (when return code is nonzero);
5 numerical instability;
6 analysis of an infeasible relaxation was performed, but the relaxation is feasible.

restore

Purpose

Restores the Optimizer's data structures from a file created by save (SAVE).
Optimization may then recommence from the point at which the file was created.

Synopsis

```
restore(prob, probname = "", flags = "")
```

Arguments

`prob`	The current problem.
`probname`	A string of up to MAXPROBNAMELENGTH characters containing the problem name.
`flags`	Additional flags force (no effect, kept for compatibility);
	`h` Do not restore hardware information from the file;
	`v` use the provided filename verbatim, without appending the `.svf` extension.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

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

```
rhssa(prob, rowind, nrows = x_max_vec_length(rowind))
```

Arguments

`prob`	The current problem.
`rowind`	Integer array of length `nrows` containing the indices of the rows whose RHS coefficients sensitivity ranges are required.
`nrows`	The number of RHS coefficients for which sensitivity ranges are required.

Return value

A list with the following elements:

`lower`	The RHS lower range values.
`upper`	The RHS upper range values.

Further information

Please refer to the C documentation for more details.

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

```
save(prob)
```

Argument

`prob` The current problem.

Return value

The input argument `prob`.

saveas

Purpose

Saves the current data structures, i.e. matrices, control settings and problem attribute settings to file and terminates the run so that optimization can be resumed later.

Synopsis

```
saveas(prob, sSaveFileName = "")
```

Arguments

`prob`	The current problem.
`sSaveFileName`	The name of the file (without .svf) to save to.

Return value

The input argument `prob`.

scale

Purpose

Re-scales the current matrix.

Synopsis

```
scale(prob, rowscale, colscale)
```

Arguments

`prob`	The current problem.
`rowscale`	Integer array of size ROWS containing the powers of `2` with which to scale the rows, or `NULL` if not required.
`colscale`	Integer array of size COLS containing the powers of `2` with which to scale the columns, or `NULL` if not required.

Return value

The input argument `prob`.

setcheckedmode

Purpose

You can use this function to disable some of the checking and validation of function calls and function call parameters for calls to the Xpress Optimizer API.
This checking is relatively lightweight but disabling it can improve performance in cases where non-intensive Xpress Optimizer functions are called repeatedly in a short space of time. Please note: after disabling function call checking and validation, invalid usage of Xpress Optimizer functions may not be detected and may cause the Xpress Optimizer process to behave unexpectedly or crash. It is not recommended that you disable function call checking and validation during application development.

Synopsis

```
setcheckedmode(checkedmode)
```

Argument

`checkedmode` Pass as 0 to disable much of the validation for all Xpress function calls from the current process.

Return value

Always returns 0 (zero).

Further information

Please refer to the C documentation for more details.

setdblcontrol

Purpose

Sets the value of a given double control parameter.

Synopsis

```
setdblcontrol(prob, control, value)
```

Arguments

`prob`	The current problem.
`control`	Control parameter whose value is to be set.
`value`	Value to which the control parameter is to be set.

Return value

The input argument `prob`.

setdefaultcontrol

Purpose

Sets a single control to its default value.

Synopsis

```
setdefaultcontrol(prob, control)
```

Arguments

`prob`	The current problem.
`control`	Integer, double or string control parameter whose default value is to be set.

Return value

The input argument `prob`.

setdefaults

Purpose

Sets all controls to their default values.
Must be called before the problem is read or loaded by readprob, loadmip, loadlp, loadmiqp, loadqp.

Synopsis

```
setdefaults(prob)
```

Argument

`prob` The current problem.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

setindicators

Purpose

Specifies that a set of rows in the matrix will be treated as indicator constraints during a tree search. An indicator constraint is made of a `condition` and a `constraint`. The `condition` is of the type "bin = value", where `bin` is a binary variable and `value` is either 0 or 1. The `constraint` is any matrix row (may be linear, quadratic or general nonlinear). During tree search, a row configured as an indicator constraint is enforced only when condition holds, that is only if the indicator variable `bin` has the specified value. Note that every row may only get assigned a single indicator variable and term. If a row needs to be activated by multiple different terms, the row needs to be duplicated so that each term can be assigned to a distinct row. If the indicator variable should be changed, the old term needs to be deleted first (by calling delindicators or by calling this function with a comps argument of 0) before assigning a new one.

Synopsis

```
setindicators(
prob,
rowind,
colind,
complement,
nrows = x_max_vec_length(rowind, colind, complement)
)
```

Arguments

`prob`	The current problem.
`rowind`	Integer array of length `nrows` containing the indices of the rows that define the constraint part for the indicator constraints.
`colind`	Integer array of length `nrows` containing the column indices of the indicator variables.
`complement`	Integer array of length `nrows` 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`".
`nrows`	The number of indicator constraints.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

setintcontrol

Purpose

Sets the value of a given integer control parameter.

Synopsis

```
setintcontrol(prob, control, value)
```

Arguments

`prob`	The current problem.
`control`	Control parameter whose value is to be set.
`value`	Value to which the control parameter is to be set.

Return value

The input argument `prob`.

setlogfile

Purpose

This directs all Optimizer output to a log file.

Synopsis

```
setlogfile(prob, filename)
```

Arguments

`prob`	The current problem.
`filename`	A string of up to MAXPROBNAMELENGTH characters containing the file name to which all logging output should be written.

Return value

The input argument `prob`.

setmessage

Purpose

Set message verbosity for optimizer.
The function registers a `message` callback with `prob`. All enabled messages are printed using the `cat` function.

Synopsis

```
setmessage(prob, info, warn = TRUE, err = TRUE)
```

Arguments

`prob`	The problem object for which messages should be enabled.
`info`	`TRUE` to enable info messages, `FALSE` to disable them.
`warn`	`TRUE` to enable warning messages, `FALSE` to disable them.
`err`	`TRUE` to enable error messages, `FALSE` to disable them.

Return value

Always returns `prob`.

setmessagestatus

Purpose

Manages suppression of messages.

Synopsis

```
setmessagestatus(prob, msgcode, status)
```

Arguments

`prob`	The problem for which message `msgcode` is to have its suppression status changed; pass `NULL` if the message should have the status apply globally to all problems.
`msgcode`	The id number of the message.
`status`	Non-zero if the message is not suppressed; `0` otherwise. If a value for `status` is not supplied in the command-line call then the console Optimizer prints the value of the suppression status to screen i.e., non-zero if the message is not suppressed; `0` otherwise.

Return value

The input argument `prob`.

setobjdblcontrol

Purpose

Sets the value of a given double control parameter associated with an objective.
These parameters control how the objective is treated during multi-objective optimization.

Synopsis

```
setobjdblcontrol(prob, objidx, control, value)
```

Arguments

`prob` The current problem.

`objidx` Index of the objective to modify.

`control` Control parameter whose value is to be modified. Must be one of:

- `_OBJECTIVE_WEIGHT` set the weight of the given objective;
- `_OBJECTIVE_ABSTOL` set the absolute tolerance of the given objective;
- `_OBJECTIVE_RELTOL` set the relative tolerance of the given objective;
- `_OBJECTIVE_RHS` set the constant term of the given objective.

`value` Value to which the control parameter is to be set.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

setobjintcontrol

Purpose

Sets the value of a given integer control parameter associated with an objective.
These parameters control how the objective is treated during multi-objective optimization.

Synopsis

```
setobjintcontrol(prob, objidx, control, value)
```

Arguments

`prob`	The current problem.
`objidx`	Index of the objective to modify.
`control`	Control parameter whose value is to be modified. Must be one of:
	`_OBJECTIVE_PRIORITY` set the priority of the given objective.
`value`	Value to which the control parameter is to be set.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

setoutput

Purpose

Convenience function to redirect optimizer messages.
The function can be used to redirect optimizer messages to `stdin` or `stderr`. The values for `info`, `warn`, `err` can be 0 (zero) to suppress the respective messages, 1 (one) to redirect the respective messages to `stdout`, 2 to redirect them to `stderr` and 3 to redirect them to both. Any previous redirection set with this function is removed.

Synopsis

```
setoutput(prob, info = 1L, warn = 2L, err = 2L)
```

Arguments

`prob`	The problem pointer for which messages should be redirected.
`info`	Specifies redirection of info messages.
`warn`	Specifies redirection of warning messages.
`err`	Specifies redurection of error messages.

Return value

Always returns `prob`.

setprobname

Purpose

Sets the current default problem name.
This command is rarely used.

Synopsis

```
setprobname(prob, probname)
```

Arguments

`prob`	The current problem.
`probname`	A string of up to MAXPROBNAMELENGTH characters containing the problem name.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

setstrcontrol

Purpose

Used to set the value of a given string control parameter.

Synopsis

```
setstrcontrol(prob, control, value)
```

Arguments

`prob`	The current problem.
`control`	Control parameter whose value is to be set.
`value`	A string containing the value to which the control is to be set.

Return value

The input argument `prob`.

slpcascade

Purpose

Re-calculate consistent values for SLP variables based on the current values of the remaining variables.

Synopsis

```
slpcascade(prob)
```

Argument

`prob` The current SLP problem.

Return value

The input argument `prob`.

slpcascadeorder

Purpose

Establish a re-calculation sequence for SLP variables with determining rows.

Synopsis

```
slpcascadeorder(prob)
```

Argument

`prob` The current SLP problem.

Return value

The input argument `prob`.

slpchgcascadenlimit

Purpose

Set a variable specific cascade iteration limit

Synopsis

```
slpchgcascadenlimit(prob, col, limit)
```

Arguments

`prob`	The current SLP problem.
`col`	The index of the column corresponding to the SLP variable for which the cascading limit is to be imposed.
`limit`	The new cascading iteration limit.

Return value

The input argument `prob`.

slpchgdeltatype

Purpose

Changes the type of the delta assigned to a nonlinear variable

Synopsis

```
slpchgdeltatype(
prob,
varind,
deltatypes,
values,
nvars = x_max_vec_length(varind, deltatypes, values)
)
```

Arguments

`prob` The current SLP problem.

`varind` Indices of the variables to change the deltas for.

`deltatypes` Type of the delta variable:

- `0 (XSLP_DELTA_CONT)` Differentiable variable, default.
- `1 (XSLP_DELTA_SEMICONT)` Variable where a minimum perturbation size given in `values` may be required before a significant change in the problem is achieved.
- `2 (XSLP_DELTA_INTEGER)` Variable defined over the grid size given in `values`.
- `3 (XSLP_DELTA_EXPLORE)` 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.

`nvars` The number of SLP variables to change the delta type for.

Return value

The input argument `prob`.

slpchgrowstatus

Purpose

Change the status setting of a constraint

Synopsis

```
slpchgrowstatus(prob, row)
```

Arguments

`prob` The current SLP problem.

`row` The index of the matrix row to be changed.

Return value

Address of an integer holding a bitmap with the new status settings. If the status is to be changed, always get the current status first (use XSLPgetrowstatus) 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).

slpchgrowwt

Purpose

Set or change the initial penalty error weight for a row

Synopsis

```
slpchgrowwt(prob, row, weight)
```

Arguments

`prob`	The current SLP problem.
`row`	The index of the row whose weight is to be set or changed.
`weight`	Address of a double precision variable holding the new value of the weight.

Return value

The input argument `prob`.

slpconstruct

Purpose

Create the full augmented SLP matrix and data structures, ready for optimization

Synopsis

```
slpconstruct(prob)
```

Argument

`prob` The current SLP problem.

Return value

The input argument `prob`.

slpfixpenalties

Purpose

Fixe the values of the error vectors

Synopsis

```
slpfixpenalties(prob)
```

Argument

`prob` The current SLP problem.

Return value

Return status after fixing the penalty variables: 0 is successful, nonzero otherwise.

slpgetrowstatus

Purpose

Retrieve the status setting of a constraint

Synopsis

```
slpgetrowstatus(prob, row)
```

Arguments

`prob` The current SLP problem.

`row` The index of the matrix row whose data is to be obtained.

Return value

Address of an integer holding a bitmap to receive the status settings.

slpgetrowwt

Purpose

Get the initial penalty error weight for a row

Synopsis

```
slpgetrowwt(prob, row)
```

Arguments

`prob`	The current SLP problem.
`row`	The index of the row whose weight is to be retrieved.

Return value

Address of a double precision variable to receive the value of the weight.

slpreinitialize

Purpose

Reset the SLP problem to match a just augmented system

Synopsis

```
slpreinitialize(prob)
```

Argument

`prob` The current SLP problem.

Return value

The input argument `prob`.

slpsetdetrow

Purpose

Set the determining row of a variable

Synopsis

```
slpsetdetrow(prob, colind, rowind, nvars = x_max_vec_length(colind,
       rowind))
```

Arguments

`prob`	The current SLP problem.
`colind`	Array of length `nvars` with the index of the column for which the determining row is set.
`rowind`	Array of length `nvars` with the index of the determining row.
`nvars`	The number of variables for which determining rows are set.

Return value

The input argument `prob`.

slpunconstruct

Purpose

Removes the augmentation and returns the problem to its pre-linearization state

Synopsis

```
slpunconstruct(prob)
```

Argument

`prob` The current SLP problem.

Return value

The input argument `prob`.

slpupdatelinearization

Purpose

Updates the current linearization

Synopsis

```
slpupdatelinearization(prob)
```

Argument

`prob` The current SLP problem.

Return value

The input argument `prob`.

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 loadcuts before they become active.

Synopsis

```
storecuts(
prob,
nodups,
cuttype,
rowtype,
rhs,
start,
colind,
cutcoef,
ncuts = x_max_vec_length(cuttype, rowtype, rhs)
)
```

Arguments

`prob`	The current problem.	
`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`	Integer array of length `ncuts` containing the cut types.	
`rowtype`	Character array of length `ncuts` containing the row types:	
	`L`	indicates a <= row;
	`E`	indicates an = row;
	`G`	indicates a >= row.
`rhs`	Double array of length `ncuts` containing the right hand side elements for the cuts.	
`start`	Integer array containing offsets into the `colind` and `dmtval` arrays indicating the start of each cut.	
`colind`	Integer array of length `start[ncuts]` containing the column indices in the cuts.	
`cutcoef`	Double array of length `start[ncuts]` containing the matrix values for the cuts.	
`ncuts`	Number of cuts to add.	

Return value

The pointers to the cuts .

Further information

Please refer to the C documentation for more details.

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

```
strongbranch(prob, nbounds, colind, bndtype, bndval, iterlim)
```

Arguments

`prob` The current problem.

`nbounds` Number of bound changes to try.

`colind` Integer array of size `nbounds` containing the indices of the columns on which the bounds will change.

`bndtype` Character array of length `nbounds` 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` Double array of length `nbounds` giving the new bound values.

`iterlim` Maximum number of LP iterations to perform for each bound change.

Return value

A list with the following elements:

`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.

Further information

Please refer to the C documentation for more details.

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

```
strongbranchcb(prob, nbounds, colind, bndtype, bndval, iterlim, callback)
```

Arguments

`prob`	The current problem.
`nbounds`	Number of bound changes to try.
`colind`	Integer array of size `nbounds` containing the indices of the columns on which the bounds will change.
`bndtype`	Character array of length `nbounds` 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`	Double array of length `nbounds` giving the new bound values.
`iterlim`	Maximum number of LP iterations to perform for each bound change.
`callback`	Function to be called after each strong branch has been reoptimized.

Return value

A list with the following elements:

`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.

Further information

Please refer to the C documentation for more details.

summary.XPRSprob

Purpose

Print a summary for an XPRESS problem.

Synopsis

```
summary.XPRSprob(prob, ...)
```

Argument

`prob` The problem for which the summary should be printed.

tune

Purpose

This function begins a tuner session for the current problem.
The tuner will solve the problem multiple times while evaluating a list of control settings and promising combinations of them. When finished, the tuner will select and set the best control setting on the problem. Note that the direction of optimization is given by OBJSENSE.

Synopsis

```
tune(prob, flags = "")
```

Arguments

`prob` The current problem.

`flags` Flags to pass to tune, which specify whether to tune the current problem as an LP or a MIP problem, and the algorithm for solving the LP problem or the initial LP relaxation of the MIP. The flags are optional. If the argument includes:

`l`	will tune the problem as an LP (mutually exclusive with flag `g`);
`g`	will tune the problem as a MIP (mutually exclusive with flag `l`);
`x`	will tune the problem as a Global Optimization problem with Xpress Global;
`d`	will use the dual simplex method;
`p`	will use the primal simplex method;
`b`	will use the barrier method;
`n`	will use the network simplex method.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

tuneprobsetfile

Purpose

This function begins a tuner session for a set of problems.
The tuner will solve the problems multiple times while evaluating a list of control settings and promising combinations of them. When finished, the tuner will select and set the best control setting on the problems.

Synopsis

```
tuneprobsetfile(prob, setfile, ifmip, sense)
```

Arguments

`prob`	The current problem.	
`setfile`	A plain text file which contains a list of problem filenames.	
`ifmip`	`-1`	to automatically determine whether to solve the problem set as LP or MIP;
	`0`	to force the tuner to tune the problem set as LP;
	`1`	to force the tuner to tune the problem set as MIP.
`sense`	`0`	to automatically determine the sense of each problem;
	`1`	to force the tuner to minimize each problem;
	`-1`	to force the tuner to maximize each problem.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

tunerreadmethod

Purpose

This function loads a user defined tuner method from the given file.

Synopsis

```
tunerreadmethod(prob, methodfile)
```

Arguments

`prob`	The current problem.
`methodfile`	The method file name, from which the tuner can load a user-defined tuner method.

Return value

The input argument `prob`.

tunerwritemethod

Purpose

This function writes the current tuner method to a given file or prints it to the console.

Synopsis

```
tunerwritemethod(prob, methodfile)
```

Arguments

`prob`	The current problem.
`methodfile`	The method file name, to which the tuner will write the current tuner method.

Return value

The input argument `prob`.

unloadprob

Purpose

Deprecated Call loadlp with all `NULL` arguments to reset the problem to an empty problem. Unloads and frees all memory associated with the current problem. It also invalidates the current problem (as opposed to reading in an empty problem).

Synopsis

```
unloadprob(prob)
```

Argument

`prob` The current problem.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

writebasis

Purpose

Writes the current basis to a file for later input into the Optimizer.

Synopsis

```
writebasis(prob, filename = "", flags = "")
```

Arguments

`prob`	The current problem.
`filename`	A string of up to MAXPROBNAMELENGTH characters containing the file name from which the basis is to be written.
`flags`	Flags to pass to `writebasis` (`WRITEBASIS`): scrambled vector names; output values in hexadecimal; output in a format compatible with CPLEX;

`i`	output the internal presolved basis;
`t`	output a compact advanced form of the basis;
`n`	output basis file containing current solution values;
`h`	output values in single precision;
`p`	output values in full precision (obsolete as this is now default behavior);
`v`	use the provided filename verbatim, without appending the `.bss` extension;
`z`	compress the output file.

Return value

The input argument `prob`.

writebinsol

Purpose

Writes the current MIP or LP solution to a binary solution file for later input into the Optimizer.

Synopsis

```
writebinsol(prob, filename = "", flags = "")
```

Arguments

`prob`	The current problem.
`filename`	A string of up to MAXPROBNAMELENGTH characters containing the file name to which the solution is to be written.
`flags`	Flags to pass to `writebinsol` (`WRITEBINSOL`):

`m`	output the MIP solution;
`x`	output the LP solution;
`v`	use the provided filename verbatim, without appending the `.sol` extension;
`z`	compress the output file.

Return value

The input argument `prob`.

writedirs

Purpose

Writes the tree search directives from the current problem to a directives file.

Synopsis

```
writedirs(prob, filename = NULL)
```

Arguments

`prob`	The current problem.
`filename`	A string of up to MAXPROBNAMELENGTH characters containing the file name to which the directives should be written.

Return value

The input argument `prob`.

writeprob

Purpose

Writes the current problem to an MPS or LP file.

Synopsis

```
writeprob(prob, filename = "", flags = "")
```

Arguments

`prob`	The current problem.
`filename`	A string of up to MAXPROBNAMELENGTH characters to contain the file name to which the problem is to be written.
`flags`	Flags, which can be one or more of the following: output in a format compatible with CPLEX;

`o`	one element per line;
`n`	output the scaled problem;
`s`	scrambled vector names;
`l`	output in LP format;
`p`	output values in full precision (obsolete as this is now default behavior);
`t`	omit the Xpress header in LP or MPS format;
`v`	use the provided filename verbatim, without appending the `.mps` or `.lp` extension;
`z`	compress the output file.

Return value

The input argument `prob`.

writeprtsol

Purpose

Writes the current solution to a fixed format ASCII file, problem_name `.prt`.

Synopsis

```
writeprtsol(prob, filename = "", flags = "")
```

Arguments

`prob`	The current problem.
`filename`	A string of up to MAXPROBNAMELENGTH characters containing the file name to which the solution is to be written.
`flags`	Flags for `writeprtsol` (`WRITEPRTSOL`) are: print the solution to the screen (via the message callback) instead of writing to a file;
	`x` write the LP solution instead of the current MIP solution;
	`v` use the provided filename verbatim, without appending the `.prt` extension;
	`z` write a compressed output file;
	`s` include classical sensitivity analysis.

Return value

The input argument `prob`.

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 readslxsol function.

Synopsis

```
writeslxsol(prob, filename = "", flags = "")
```

Arguments

`prob`	The current problem.
`filename`	A string of up to MAXPROBNAMELENGTH characters containing the file name to which the solution is to be written.
`flags`	Flags to pass to `writeslxsol` (`WRITESLXSOL`):

`l`	write the LP solution in case of a MIP problem;
`m`	write the MIP solution;
`p`	use full precision for numerical values (obsolete as this is now default behavior);
`s`	including slack variables;
`d`	LP solution only: including dual variables;
`r`	LP solution only: including reduced cost;
`v`	use the provided filename verbatim, without appending the `.slx` extension;
`z`	compress the output file.

Return value

The input argument `prob`.

Further information

Please refer to the C documentation for more details.

writesol

Purpose

Writes the current solution to a CSV format ASCII file, problem_name . `asc` (and `.hdr`).

Synopsis

```
writesol(prob, filename = "", flags = "")
```

Arguments

`prob`	The current problem.
`filename`	A string of up to MAXPROBNAMELENGTH characters containing the file name to which the solution is to be written.
`flags`	Flags to control which optional fields are output:

`s`	sequence number; If no flags are specified, all fields are output.Additional flags:
`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).
`p`	outputs in full precision;
`q`	only outputs vectors with nonzero optimum value;
`x`	output the current LP solution instead of the MIP solution;
`z`	compress the output file.

Return value

The input argument `prob`.

x_max_vec_length

Purpose

returns the maximum length of all input vectors

Synopsis

```
x_max_vec_length(...)
```

Argument

`...` an arbitrary list of vectors, some of which may be NULL

Return value

maximum length of all input vectors. Note that NULL has a length of 0.

Example

```
x_max_vec_length(1:3, 1:4)
# 4
x_max_vec_length(1:3, 1:4, 1:2)
4
x_max_vec_length(1:3, 1:4, NULL)
# 4
```

x_to_column_major_matrix

Purpose

Conversion to sparse matrix in column major form
convert an input matrix into a column major form with slots i, p, and x

Synopsis

```
x_to_column_major_matrix(mat)
```

Argument

`mat` matrix object, either dense or sparse

Return value

converted mat in sparse column major form

x_to_sparse_triplet_matrix

Purpose

Conversion to sparse triplet matrix
convert an input matrix into a sparse triplet matrix with slots i, j, and x

Synopsis

```
x_to_sparse_triplet_matrix(mat)
```

Argument

`mat` matrix object, either dense or sparse

Return value

converted mat in sparse triplet form

x_validate_length

Purpose

Validates that all non-NULL input vectors hold at least n elements

Synopsis

```
x_validate_length(n, ...)
```

Arguments

`n`	an integer length
`...`	an arbitrary list of vectors, some of which may be NULL

Return value

logical: TRUE if all input vectors have length() at least n

Example

```
x_validate_length(3, 1:2, 1:3)
# FALSE
x_validate_length(2, 1:2, 1:3)
# TRUE
x_validate_length(2, 1:2, 1:3, 1:4, 1:5)
# TRUE
x_validate_length(4, 1:8, 1:3, 1:4, 1:5)
# FALSE
x_validate_length(4, 1:8, NULL, 1:4, 1:5)
# TRUE
```

x_validate_problemdata

Purpose

Validate the problem data
Run a couple of sanity checks on the problem data to verify that no two descriptions (C-style or R style) are given. If problemdata violates any condition, an error is thrown.

Synopsis

```
x_validate_problemdata(prob, problemdata)
```

Arguments

`prob`	prob an XPRSprob object
`problemdata`	list object with attributes describing the input optimization problem.

xprs_add_names_type

Purpose

adds a type of names (rows, columns etc) to the optimizer

Synopsis

```
xprs_add_names_type(prob, charvec, itype)
```

Arguments

`prob`	XPRSprob pointer
`charvec`	character vector or NULL, in which case, the function adds nothing
`itype.`	See the XPRSaddnames documention

Return value

the prob pointer, for possible data pipelines

xprs_addcol

Purpose

Create and add a new empty column to the problem.
This is a variant of `addcols()` that only adds a single column and also allows specifying type and name of the column.

Synopsis

```
xprs_addcol(prob, lb, ub, coltype = NULL, name = NULL, objcoef = NULL)
```

Arguments

`prob` The problem to which the column is to be added.

`lb` The lower bound for the new column.

`ub` The upper bound for the new column.

`coltype` Character specifying the column type. If this is `NULL` then the default Xpress column type is used (continuous). Possible values are:

- `C` indicates a continuous column.
- `B` indicates a binary column.
- `I` indicates an integer column.
- `S` indicates a semicontinuous column.
- `R` indicates a semiinteger column.
- `P` indicates a partial integer column.

`name` The name for the new column.

`objcoef` The objective coefficient for the new column.

Return value

The function always returns `prob`.

xprs_addrow

Purpose

Create and add a new row to a problem.
This is a variant of `addrows()` that only adds a single row and also allows specifying the name of the row.

Synopsis

```
xprs_addrow(prob, colind, rowcoef, rowtype, rhs, name = NULL, rng = NULL)
```

Arguments

`prob`	The problem to which the row is to be added.
`colind`	The variable indices for the variables in the new row.
`rowcoef`	The coefficients for the variables in the new row.
`rowtype`	Character specifying the type of the new row. Possible values are:
	`L` indicates a <= row.
	`G` indicates >= row.
	`E` indicates an = row.
	`R` indicates a range constraint.
	`N` indicates a nonbinding constraint.
`rhs`	The right-hand side of the new row.
`name`	The name of the new row.
`rng`	The range value for the new row.

Return value

The function always returns `prob`.

xprs_getdoubleattributes

Purpose

Get a list of all attributes of type `double`.

Synopsis

```
attributes <- xprs_getdoubleattributes()
```

Return value

A list of all attributes of type `double`

Example

```
prob <- createprob()
attrib_names = names(xprs_getdoubleattributes())
```

xprs_getdoublecontrols

Purpose

Get a list of all controls of type `double`.

Synopsis

```
controls <- xprs_getdoublecontrols()
```

Return value

A list of all controls of type `double`

Example

```
prob <- createprob()
control_names = names(xprs_getdoublecontrols())
```

xprs_getintattributes

Purpose

Get a list of all attributes of type `int`.

Synopsis

```
attributes <- xprs_getintattributes()
```

Return value

A list of all attributes of type `int`

Example

```
prob <- createprob()
attrib_names = names(xprs_getintattributes())
```

xprs_getintcontrols

Purpose

Get a list of all controls of type `int`.

Synopsis

```
controls <- xprs_getintcontrols()
```

Return value

A list of all controls of type `int`

Example

```
prob <- createprob()
control_names = names(xprs_getintcontrols())
```

xprs_getsolution

Purpose

returns the optimal or best known solution of a solution process.
It is an error to invoke xprs_getsolution() before an optimization process was started using xprs_optimize(), mipoptimize(), or lpoptimize().

Synopsis

```
xprs_getsolution(prob)
```

Argument

`prob` the XPRSprob object holding the result of the optimization

Return value

Numeric vector of length COLS containing the optimal or best known solution for 'prob', or NULL if no such solution exists. In this case, a warning is raised.

xprs_getstringattributes

Purpose

Get a list of all attributes of type `string`.

Synopsis

```
attributes <- xprs_getstringattributes()
```

Return value

A list of all attributes of type `string`

Example

```
prob <- createprob()
attrib_names = names(xprs_getstringattributes())
```

xprs_getstringcontrols

Purpose

Get a list of all controls of type `string`.

Synopsis

```
controls <- xprs_getstringcontrols()
```

Return value

A list of all controls of type `string`

Example

```
prob <- createprob()
control_names = names(xprs_getstringcontrols())
```

xprs_hasglobals

Purpose

Does the currently loaded optimization problem have MIP entities?

Synopsis

```
xprs_hasglobals(prob)
```

Further information

This function is deprecated and will be removed from future releases. Please use `xprs_hasmipentities`

xprs_hasmipentities

Purpose

Does the currently loaded optimization problem have MIP entities?
MIP entities comprise non-continuous column types such as integer or binary variables, but also indicator constraints, general constraints, and piecewise linear constructs.

Synopsis

```
xprs_hasmipentities(prob)
```

Argument

`prob` XPRSprob pointer with an already loaded problem

Return value

TRUE if 'prob' has any form of MIP entities, FALSE otherwise

xprs_loadproblemdata

Purpose

Loads a new optimization problem into the optimizer

Synopsis

```
xprs_loadproblemdata(prob = NULL, problemdata)
```

Arguments

`prob`	an XPRSprob object, or NULL to create a new one
`problemdata`	a named list that describes the input data. The 'problemdata' understands all input arguments of the function XPRSloadmiqcqp. See the details below.

Return value

the XPRSprob object with the given optimization model loaded; this is either the `prob` passed to the call, or a newly created XPRSprob object.

Further information

The data of a new optimization model is passed as a list object. Each named property declares parts of the input data for the FICO Xpress Optimizer. The linear part of the constraints, usually denoted by a coefficient matrix A, can be specified in two alternative ways through 'problemdata'. Either as a dense or sparse matrix object, or in the classical column-major representation of the C-API of Xpress. The C-style input should specify the following attributes in 'problemdata'. Note that all index vectors must be 0-based, which is different from the usual R-convention! All names of the C-style input come from `XPRSloadmiqcqp()`.

`ncols`	Number of structural columns in the matrix. This is optional and usually inferred from 'lb', 'ub', and 'objcoef' problemdata attributes
`nrows`	Number of rows in the matrix (not including the objective row). This is optional and usually inferred from 'rowtype', 'rhs', and 'rng' problemdata attributes
`start`	0-based(!) integer array containing the offsets in the rowind and rowcoef arrays of the start of the elements for each column. This array is of length ncols or, if collen is NULL, length ncols+1. If collen is NULL the extra entry of start, start[ncols+1], contains the position in the rowind and rowcoef arrays at which an extra column would start, if it were present. In C, this value is also the length of the rowind and rowcoef arrays
`collen`	Integer array of length ncols containing the number of nonzero elements in each column. May be NULL if all elements are contiguous and start[ncols+1] contains the offset where the elements for column ncols+1 would start. This array is not required if the non-zero coefficients in the rowind and rowcoef arrays are continuous, and the start array has ncols+1 entries as described above. It may be NULL if not required
`rowind`	Integer array containing the 0-based(!) 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[ncols]+collen[ncols] or, if collen is NULL, start[ncols+1].
`rowcoef`	Double array containing the nonzero element values; length as for rowind

Alternatively, A can be specified directly as matrix input.

`A`	a dense or sparse matrix (column major or sparse triplet format) to input the linear constraint coefficients.

It is an error to specify A *and* any of the C-style input. The (in-)equalities are further described by a right-hand-side vector (usually denoted as b), the type of each constraint, and an optional rng vector.

`rhs`	Double array of length nrows containing the right hand side coefficients of the rows. The right hand side value for a rng row gives the upper bound on the row

`rowtype` String array of length 'nrows' containing the row types:

- `"L"` indicates a '<=' constraint (use this one for quadratic constraints as well)
- `"E"` indicates an '=' constraint
- `"G"` indicates a '>=' constraint
- `"R"` indicates a rng constraint
- `"N"` indicates a nonbinding constraint

`rng` Double array of length nrows containing the rng values for rng rows. Values for all other rows will be ignored. May be NULL if there are no ranged constraints. The lower bound on a rng row is the right hand side value minus the rng value. The sign of the rng value is ignored - the absolute value is used in all cases

Column properties are lower bounds, upper bounds, and objective coefficients.

`objcoef` Double array of length 'ncols' containing the objective function coefficients

`lb` Double array of length 'ncols' containing the lower bounds on the columns. Use XPRS_MINUSINFINITY to represent a lower bound of minus infinity

`ub` Double array of length 'ncols' containing the upper bounds on the columns. Use XPRS_PLUSINFINITY to represent an upper bound of plus infinity

Column types can be specified for all columns. By default, all columns are continuous. However, columns can also have various discrete types. This happens either by indexing those columns that should be non-continuous in C-style, or by providing the "columntypes" (and optionally, a "columnlimits")-vector, thereby avoiding any 0 vs. 1-based index confusion.

`nentities` Number of binary, integer, semi-continuous, semi-continuous integer and partial integer entities. This is optional, as it can be inferred from 'gqtype', and 'entind'

`coltype` Character array of length nentities containing the entity types

- `"B"` binary variables
- `"I"` integer variables
- `"P"` partial integer variables
- `"S"` semi-continuous variables
- `"R"` semi-continuous integer variables

`entind` Integer array of length nentities containing the column indices of the MIP entities

`limit` Double array of length nentities 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 NULL if not required

Or full column type specification:

`columntypes` A single character array of length 'ncols'. Besides the column types for 'coltype', specify all continuous columns by "C"

`columnlimits` A single numeric array of length 'ncols' to specify limits for partial integer variables and lower bounds for semi-continuous and semi-continuous integer variables. This is not needed if no such types are present

It is an error if problemdata mixes columntypes in both representations, C-style and full column types style. A quadratic objective matrix can be specified as single (sparse or dense) matrix input 'Qobj', or in C-style notation. As matrix 'Qobj', in which case no index confusion between 0- and 1-based indexing can occur

`Qobj` a dense or sparse matrix (column major or sparse triplet format) to input the quadratic objective terms

Or C-style input:

`nobjqcoef` Number of quadratic terms of the objective

`objqcol1` Integer array of size nobjqcoef containing the 0-based(!) column index of the first variable in each quadratic objective term

`objqcol2` Integer array of size nobjqcoef containing the 0-based(!) column index of the second variable in each quadratic objective term

`objqcoef` Double vector of size nobjqcoef containing the quadratic objective coefficients

The same for quadratic terms in constraints. Specify them in C-style notation, or as a single list 'Qrowlist' of length 'nrows' of matrix objects, sparse or dense, with NULL elements for the rows that do not have quadratic terms. The matrix-style input argument:

`Qrowlist` list of dense or sparse numeric matrices (column major or sparse triplet format) of length 'nrows' that define the quadratic terms in each constraint.

Alternatively, use the classic C-style input variant:

`nqrows` Number of rows containing quadratic matrices

`qrowind` Integer vector of size qmn, 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` Integer vector of size qmn, containing the number of nonzeros in each quadratic constraint matrix

`rowqcol1` Integer vector of size nqcelem, where nqcelem equals the sum of the elements in nrowqcoefs (i.e. the total number of quadratic matrix elements in all the constraints). It contains the first column indices of the quadratic matrices. Indices for the first matrix are listed from 0 to qcnquads[0]-1, for the second matrix from qcnquads[0] to qcnquads[0]+ qcnquads[1]-1, etc

`rowqcol2` Integer array of size nqcelem, containing the second index for the quadratic constraint matrices

`rowqcoef` Double array of size nqcelem, containing the coefficients for the quadratic constraint matrices

Special Ordered Sets (SOS) restrict the number of columns with a nonzero solution value. SOS constraints must be input in sparse row representation. Each row represents one SOS constraint whose member columns are given as indices together with a reference row value. It is also necessary to specify the type of each set using the 'settype' argument.

`settype` Character array of length nsets containing the set types:

- `1` SOS1 type sets
- `2` SOS2 type sets

`nsets` Number of SOS1 and SOS2 sets. This is readily inferred from settype and need not be specified.

`setstart` Integer 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

`setind` Integer array of length setstart[nsets]-1 containing the columns in each set

`refval` Double array of length setstart[nsets]-1 containing the reference row entries for each member of the sets

`probname` A string of up to MAXPROBNAMELENGTH characters containing a name for the problem

`colname` (optional) a vector specifying a name for each column of the problem

`rowname` (optional) a vector specifying a name for each row of the problem

Example

```
# We load and solve the following Linear Program with 2 variables and 2 constraints.
#      min   x_1 +    x_2
#          5 x_1 +    x_2  >= 7
#            x_1 +  4 x_2  >= 9
#                 x_1,x_2  >= 0
library(xpress)
# create a list object to hold all input
problemdata <- list()
# objective coefficients
problemdata$objcoef <- c(1,1)
# row coefficients
problemdata$A <- matrix(c(5,1,1,4), nrow=2, byrow=TRUE)
# right-hand side
problemdata$rhs <- c(7,9)
# row sense
problemdata$rowtype <- c("G", "G")
# lower bounds (defaulting to 0 if unspecified)
problemdata$lb <- c(0,0)
# upper bounds(defaulting to Inf if unspecified)
problemdata$ub <- c(Inf,Inf)
# names for writing to MPS/LP files
problemdata$colname <- c("x_1", "x_2")
# Problem Name displayed when the solver solves the problem.
problemdata$probname <- "FirstExample"
# load everything into a new XPRSprob 'p'. You may also use the equivalent
#
# p <- createprob()
# xprs_loadproblemdata(p, problemdata=problemdata)
#
# for convenience and the use inside pipes,
# xprs_loadproblemdata returns the prob pointer.
#
p <- xprs_loadproblemdata(problemdata=problemdata)
```

xprs_newcol

Purpose

Create and add a new empty column to a problem.
This function is similar to `xprs_addcol` but instead of returning the `prob` object, it returns the index of the newly created column.

Synopsis

```
xprs_newcol(prob, lb, ub, coltype = NULL, name = NULL, objcoef = NULL)
```

Arguments

`prob` The problem to which the column is to be added.

`lb` The lower bound for the new column.

`ub` The upper bound for the new column.

`coltype` Character specifying the column type. If this is `NULL` then the default Xpress column type is used (continuous). Possible values are:

`C`	indicates a continuous column.
`B`	indicates a binary column.
`I`	indicates an integer column.
`S`	indicates a semicontinuous column.
`R`	indicates a semiinteger column.
`P`	indicates a partial integer column.

`name` The name for the new column.

`objcoef` The objective coefficient for the new column.

Return value

The index of the newly created column.

Further information

This is a variant of `addcols()` that only adds a single column and also allows specifying type and name of the column.

xprs_newrow

Purpose

Create and add a new row to a problem.
This function is similar to `xprs_addrow` but instead of returning the `prob` object, it returns the index of the newly created row.

Synopsis

```
xprs_newrow(prob, colind, rowcoef, rowtype, rhs, name = NULL, rng = NULL)
```

Arguments

`prob`	The problem to which the row is to be added.
`colind`	The variable indices for the variables in the new row.
`rowcoef`	The coefficients for the variables in the new row.
`rowtype`	Character specifying the type of the new row. Possible values are:
	`L` indicates a <= row.
	`G` indicates >= row.
	`E` indicates an = row.
	`R` indicates a range constraint.
	`N` indicates a nonbinding constraint.
`rhs`	The right-hand side of the new row.
`name`	The name of the new row.
`rng`	The range value for the new row.

Return value

The index of the newly created row.

Further information

This is a variant of `addrows()` that only adds a single row and also allows specifying the name of the row.

xprs_optimize

Purpose

Invoke a solution process with the FICO Xpress Optimizer
This function will invoke the suitable optimization routine for an XPRSprob object which holds an already loaded optimization problem. The actual solving behavior is similar to that of `optimize`. However, this function allows to specify the solution method of the initial LP relaxation via flags and returns the prob object itself, making it suitable for usage with pipes.

Synopsis

```
xprs_optimize(prob, sflags = "")
```

Arguments

`prob`	an XPRSprob object, into which a problem has been loaded already.
`sflags`	Flags to pass to the optimization method, which specifies how to solve the initial continuous problem where the MIP entities are relaxed.

Return value

the XPRSprob object

Example

```
problemdata <- list()
problemdata$A <- matrix(c(1,0,0,1),nrow=2)
problemdata$rhs <- c(1,2)
problemdata$objcoef <- c(1,2)
problemdata$lb <- c(0,0)
problemdata$ub <- c(Inf,Inf)
problemdata$rowtype <- c("G", "E")
prob <- xprs_loadproblemdata(problemdata=problemdata)
xprs_optimize(prob)
```

Appendix A

Contacting FICO

FICO provides clients with support and services for all our products.

FICO Customer Support

FICO Customer Support offers technical support and services ranging from self-help tools to direct assistance with a FICO technical support engineer. Support is available to all clients who have an active maintenance contract.

The FICO Customer Self-Service Portal (support.fico.com) is a secure web portal that allows users to open, review, and update their support cases; manage their organization's portal users; find solutions to common problems in the FICO Knowledge Base; and view the availability of their cloud applications 24 hours a day, 7 days a week.

You can find support contact information and a link to the FICO Customer Self-Service Portal (online support) on the Product Support home page (www.fico.com/en/product-support).

Please include *'Xpress'* in the subject line of your support queries.

Documentation

FICO continually looks for new ways to improve and enhance the value of the products and services we provide.

If you have comments or suggestions regarding how we can improve this documentation, let us know by sending your suggestions to techpubs@fico.com. Please include your contact information (name, company, email address, and optionally, your phone number) so we may reach you if we have questions.

FICO Learning

FICO Learning is the principal provider of product training for our clients and partners. FICO Learning offers instructor-led classroom courses, web-based training, seminars, and training tools for both new user enablement and ongoing performance support.

For additional information, visit the FICO Learning home page at www.fico.com/en/product-training or email producteducation@fico.com.

Sales and maintenance

If you need information on other Xpress Optimization products, or you need to discuss maintenance contracts or other sales-related items, contact FICO by:

- Phone: +1 (408) 535-1500 or +44 207 940 8718
- Web: www.fico.com/optimization and use the available contact forms

About FICO

FICO (NYSE:FICO) is a leading analytics software company, helping businesses in 90+ countries make better decisions that drive higher levels of growth, profitability, and customer satisfaction. Learn more at www.fico.com or contact us at www.fico.com/en/contact-us.

Index