FICO® Xpress Optimization

Last update 02 October 2020

37.01

REFERENCE MANUAL

FICO® Xpress Optimizer

Xpress Optimizer

Deliverable Version: A

Last Revised: 02 October 2020

Version 37.01

Contents

CHAPTER 1
Introduction

The FICO Xpress Optimization Suite is a powerful mathematical optimization framework well–suited to a broad range of optimization problems. The core solver of this suite is the FICO Xpress Optimizer, which combines ease of use with speed and flexibility. It can be interfaced via the command line Console Optimizer, via the graphical interface application IVE and through a callable library that is accessible from all the major programming platforms. It combines flexible data access functionality and optimization algorithms, using state–of–the–art methods, which enable the user to handle the increasingly complex problems arising in industry and academia.

The Console Optimizer provides a suite of 'Console Mode' Optimizer functionality. Using the Console Optimizer the user can load problems from widely used problem file formats such as the MPS and LP formats and solve them using any of the algorithms supported by the Optimizer. The results may then be processed and viewed in a variety of ways. The Console Mode provides full access to the Optimizer control variables allowing the user to customize the optimization algorithms to tune the solving performance on the most demanding problems.

The FICO Xpress Optimizer library provides full, high performance access to the internal data structures of the Optimizer and full flexibility to manipulate the problem and customize the optimization process. For example, the Cut Manager framework allows the user to exploit their detailed knowledge of the problem to generate specialized cutting planes during branch and bound that may improve solving performance of Mixed Integer Programs (MIPs).

Of most interest to the library users will be the embedding of the Optimizer functionality within their own applications. The available programming interfaces of the library include interfaces for C/C++, .NET and Java. Note that the interface specifications in the following documentation are given exclusively in terms of the C/C++ language. Short examples of the interface usage using other languages may be found in the FICO Xpress Getting Started manual.

1.1 The FICO Xpress Optimizer

The FICO Xpress Optimizer is a mathematical programming framework designed to provide high performance solving capabilities. Problems can be loaded into the Optimizer via matrix files such as MPS and LP files and/or constructed in memory and loaded using a variety of approaches via the library interface routines. Note that in most cases it is more convenient for the user to construct their problems using FICO Xpress Mosel or FICO Xpress BCL and then solve the problem using the interfaces provided by these packages to the Optimizer.

The solving algorithms provided with the Optimizer include the primal simplex, the dual simplex and the Newton barrier algorithms. For solving Mixed Integer Programs (MIPs) the Optimizer provides a powerful branch and bound framework. The various types of problems the Optimizer can solve are outlined in Chapter 3.

Solution information can be exported to file using a variety of ASCII and binary formats or accessed via memory using the library interface. Advanced solution information, such as solution bases, can be both exported and imported either via files or via memory, using the library interface. Note that bases can be useful for so called 'warm–starting' the solving of Linear Programming (LP) problems.

1.2 Starting the First Time

We recommend that new FICO Xpress Optimizer users first try running the Console Optimizer 'optimizer' executable from the command prompt before using the other interfaces of Optimizer. This is because (i) it is the easiest way to confirm the license status of the installation and (ii) it is an introduction to a powerful tool with many uses during the development cycle of optimization applications. For this reason we focus mainly on discussing the Console Optimizer in this section as an introduction to various basic functions of the Optimizer.

1.2.1 Licensing

To run the Optimizer from any interface it is necessary to have a valid licence file, `xpauth.xpr`. 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.

If the Optimizer fails to verify a valid license then a message can be obtained that describes the reasons for the failure and the Optimizer will be unusable. When using the Console Optimizer the licensing failure message will be displayed on the console. Library users can call the function `XPRSgetlicerrmsg` to get the licensing failure message.

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.

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, e.g., `/opt/xpressmp/bin/xpauth.xpr`.

1.2.2 Starting Console Optimizer

Console Optimizer is an interactive command line interface to the Optimizer. Console Optimizer is started from the command line using the following syntax:

```
C:\> optimizer [problem_name] [@filename]
```

From the command line an initial problem name can be optionally specified together with an optional second argument specifying a text "script" file from which the console input will be read as if it had been typed interactively.

Note that the syntax example above shows the command as if it were input from the Windows Command Prompt (i.e., it is prefixed with the command prompt string `C:\>`). For Windows users Console Optimizer can also be started by typing `optimizer` into the "Run ..." dialog box in the Start menu.

The Console Optimizer provides a quick and convenient interface for operating on a single problem loaded into the Optimizer. Compare this with the more powerful library interface that allows one or more problems to co–exist in a process. The Console Optimizer problem contains the problem data as well as (i) control variables for handling and solving the problem and (ii) attributes of the problem and its solution information.

Useful features of Console Optimizer include a help system, auto–completion of command names and integration of system commands.

Typing "`help`" will list the various options for getting help. Typing "`help commands`" will list available commands. Typing "`help attributes`" and "`help controls`" will list the available attributes and controls, respectively. Typing "`help`" followed by a command name or control/attribute name will list the help for this item. For example, typing "`help lpoptimize`" will get help for the `LPOPTIMIZE` command.

The Console Optimizer auto–completion feature is a useful way of reducing key strokes when issuing commands. To use the auto–completion feature, type the first part of an optimizer command name followed by the Tab key. For example, by typing "`lpopt`" followed by the Tab key Console Optimizer will complete to the `LPOPTIMIZE` command. Note that once you have finished inputting the command name portion of your command line, Console Optimizer can also auto–complete on file names. For example, if you have a matrix file named `hpw15.mps` in the current working directory then by typing "`readprob hpw`" followed by the Tab key the command should auto–complete to the string "`readprob hpw15.mps`". Note that the auto–completion of file names is case–sensitive.

Console Optimizer also features integration with the operating system's shell commands. For example, by typing "`dir`" (or "`ls`" under Unix) you will directly run the operating system's directory listing command. Using the "`cd`" command will change the working directory, which will be indicated in the prompt string:

```
[xpress bin] cd \
[xpress C:\]
```

Finally, note that when the Console Optimizer is first started it will attempt to read in an initialization file named `optimizer.ini` from the current working directory. This is an ASCII file that may contain any lines that can normally be entered at the console prompt, such as commands or control parameter settings. The lines of the `optimizer.ini` file are run with at start up, and can be useful for setting up a customized default Console Optimizer environment for the user (e.g., defining custom controls settings on the Console Optimizer problem).

1.2.3 Scripting Console Optimizer

The Console Optimizer interactive command line hosts a TCL script parser (http://www.tcl.tk). With TCL scripting the user can program flow control into their Console Optimizer scripts. Also TCL scripting provides the user with programmatic access to a powerful suite of functionality in the TCL library. With scripting support the Console Optimizer provides a high level of control and flexibility well beyond that which can be achieved by combining operating system batch files with simple piped script files. Indeed, with scripting support, Console Optimizer is ideal for (i) early application development, (ii) tuning of model formulations and solving performance and (iii) analyzing difficulties and bugs in models.

Note that the TCL parser has been customized and simplified to handle intuitive access to the controls and attributes of the Optimizer. The following example shows how to proceed with write and read access to the `MIPLOG` Optimizer control:

```
[xpress C:\] miplog=3
[xpress C:\] miplog
3
```

The following shows how this would usually be achieved using TCL syntax:

```
[xpress C:\] set miplog 3
3
[xpress C:\] $miplog
3
```

The following set of examples demonstrate how with some simple TCL commands and some basic flow control constructs the user can quickly and easily create powerful programs.

The first example demonstrates a loop through a list of matrix files where a simple regular expression on the matrix file name and a simple condition on the number of rows in the problem decide whether or not the problem is solved using `lpoptimize`. Note the use of:

- the creation of a list of file names using the TCL `glob` command
- the use of the TCL square bracket notation (`[]`) for evaluating commands to their string result value
- the TCL `foreach` loop construct iterating over the list of file names
- dereferencing the string value of a variable using '`$`'
- the use of the TCL `regexp` regular expression command
- the two TCL `if` statements and their condition statements
- the use of the two Optimizer commands `READPROB` and `MINIM`
- the TCL `continue` command used to skip to the next loop iteration

```
set filelist [glob *.mps]
foreach filename $filelist {
  if { [regexp -all {large_problem} $filename] } continue
  readprob $filename
  if { $rows > 200 } continue
  lpoptimize
}
```

The second example demonstrates a loop though some control settings and the tracking of the control setting that gave the best performance. Note the use of:

- the TCL `for` loop construct iterating over the values of variable `i` from `--1` to `3`
- console output formatting with the TCL `puts` command
- setting the values of Optimizer controls `CUTSTRATEGY` and `MAXNODE`
- multiple commands per line separated using a semicolon
- use of the `MIPSTATUS` problem attribute in the TCL `if` statement
- comment lines using the hash character '`#`'

```
set bestnodes 10000000
set p hpw15
for { set i -1 } { $i <= 3 } { incr i } {
  puts "Solving with cutstrategy : $i"
  cutstrategy=$i; maxnode=$bestnodes
  readprob $p
  mipoptimize
  if { $mipstatus == 6 } {
    # Problem was solved within $bestnodes
    set bestnodes $nodes; set beststrat $i
  }
}
puts "Best cutstrategy : $beststrat : $bestnodes"
```

1.2.4 Interrupting Console Optimizer

Console Optimizer users may interrupt the running of the commands (e.g., `lpoptimize`) by typing Ctrl–C. Once interrupted Console Optimizer will return to its command prompt. If an optimization algorithm has been interrupted in this way, any solution process will stop at the first 'safe' place before returning to the prompt. Optimization iterations may be resumed by re–entering the interrupted command. Note that by using this interrupt–resume functionality the user has a convenient way of dynamically changing controls during an optimization run.

When Console Optimizer is being run with script input then Ctrl–C will not return to the command prompt and the Console Optimizer process will simply stop.

Lastly, note that "typing ahead" while the console is writing output to screen can cause Ctrl–C input to fail on some operating systems.

1.3 Manual Layout

So far the manual has given a basic introduction to the FICO Xpress Optimization Suite. The reader should be able to start the Console Optimizer command line tool and have the license verified correctly. They should also be able to enter some common commands used in Console Optimizer (e.g., `READPROB` and `LPOPTIMIZE`) and get help on command usage using the Console Optimizer help functionality.

The remainder of this manual is divided into two parts. These are the first chapters up to but not including Chapter 8 and the remaining chapters from Chapter 8.

The first part of the manual, beginning with Chapter 2, provides a brief overview of common Optimizer usage, introducing the various routines available and setting them in the typical context they are used. This is followed in Chapter 3 by a brief overview of the types of problems that the FICO Xpress Optimizer can be used to solve. Chapter 4 provides a description of the solution methods and some of the more frequently used parameters for controlling these methods along with some ideas of how they may be used to enhance the solution process. Finally, Chapter 5 details some more advanced topics in Optimizer usage.

The second half of the manual is the main reference section. Chapter 8 details all functions in both the Console and Advanced Modes alphabetically. Chapters 9 and 10 then provide a reference for the various controls and attributes, followed by a list of Optimizer error and return codes in Chapter 11. A description of several of the file formats is provided in Appendix A.

CHAPTER 2

Basic Usage

The FICO Xpress Optimization Suite is a powerful and flexible framework catering for the development of a wide range of optimization applications. From the script–based Console Optimizer providing rapid development access to a subset of Optimizer functionality (Console Mode) to the more advanced, high performance access of the full Optimizer functionality through the library interface.

In the previous section we looked at the Console Optimizer interface and introduced some basic functions that all FICO Xpress Optimizer users should be familiar with. In this section we expand on the discussion and include some basic functions of the library interface.

2.1 Initialization

Before the FICO Xpress Optimization Suite can be used from any of the interfaces the Optimizer library must be initialized and the licensing status successfully verified. Details about licensing your installation can be found in FICO Xpress Installation Guide.

When Console Optimizer is started from the command line the initialization and licensing security checks happen immediately and the results are displayed with the banner in the console window. For the library interface users, the initialization and licensing are triggered by a call to the library function `XPRSinit`, which must be made before any of the other Optimizer library routines can be successfully called. If the licensing security checks fail to *check out* a license then library users can obtain a string message explaining the issue using the function `XPRSgetlicerrmsg`.

Note that it is recommended that the users having licensing problems use the Console Optimizer as a means of checking the licensing status while resolving the issues. This is because it is the quickest and easiest way to check and display the licensing status.

Once the Optimizer functionality is no longer required the license and any system resources held by the Optimizer should be released. The Console Optimizer releases these automatically when the user exits the Console Optimizer with the `QUIT` or `STOP` command. For library users the Optimizer can be triggered to release its resources with a call to the routine `XPRSfree`, which will *free* the license checked out in the earlier call to `XPRSinit`.

```
{
  if(XPRSinit(NULL)) printf("Problem with XPRSinit\n");
  XPRSfree();
}
```

In general, library users will call `XPRSinit` once when their application starts and then call `XPRSfree` before it exits. This approach is recommended since calls to `XPRSinit` can have non–negligible (approx. 0.5 sec) overhead when using floating network licensing.

Although it is recommended that the user writes their code such that `XPRSinit` and `XPRSfree` are called only in sequence note that the routine `XPRSinit` may be called repeatedly before a call to

XPRSfree. Each subsequent call to XPRSinit after the first will simply return without performing any tasks. In this case note that the routine XPRSfree must be called the same number of times as the calls to XPRSinit to fully release the resources held by the library. Only on the last of these calls to XPRSfree will the library be released and the license *freed*.

2.2 The Problem Pointer

The Optimizer provides a container or problem pointer to contain an optimization problem and its associated controls, attributes and any other resources the user may attach to help construct and solve the problem. Console Optimizer has one of these problem pointers that it uses to provide the user with loading and solving functionality. This problem pointer is automatically initialized when Console Optimizer is started and release again when it is stopped.

In contrast to Console Optimizer, library interface users can have multiple problem pointers coexisting simultaneously in a process. The library user creates and destroys a problem pointer using the routines XPRScreateprob and XPRSdestroyprob, respectively. In the C library interface, the user passes the problem pointer as the first argument in routines that are used to operate on the problem pointer's data. Note that it is recommended that the library user destroys all problem pointers before calling XPRSfree.

```
{
  XPRSprob prob;
  XPRScreateprob(&prob);
  XPRSdestroyprob(prob);
}
```

2.3 Logging

The Optimizer provides useful text logging messages for indicating progress during the optimization algorithms and for indicating the status of certain important commands such as XPRSreadprob. The messages from the optimization algorithms report information on iterations of the algorithm. The most important use of the logging, however, is to convey error messages reported by the Optimizer. Note that once a system is in production the error messages are typically the only messages of interest to the user.

Conveniently, Console Optimizer automatically writes the logging messages for its problem pointer to the console screen. Although message management for the library users is more complicated than for Console Optimizer users, library users have more flexibility with the handling and routing of messages. The library user can route messages directly to file or they can intercept the messages via callback and marshal the message strings to appropriate destinations depending on the type of message and/or the problem pointer from which the message originates.

To write the messages sent from a problem pointer directly to file the user can call XPRSsetlogfile with specification of an output file name. To get messages sent from a problem pointer to the library user's application the user will define and then register a messaging callback function with a call to the XPRSaddcbmessage routine.

```
{
  XPRSprob prob;
  XPRScreateprob(&prob);
  XPRSsetlogfile(prob, "logfile.log");
  XPRSdestroyprob(prob);
}
```

Note that a high level messaging framework is also available — which handles messages from all problem pointers created by the Optimizer library and messages related to initialization of the library itself — by calling the XPRS_ge_setcbmsghandler function. A convenient use of this callback, particularly when developing and debugging an application, is to trap all messages to file. The following line of code shows how to use the library function XPRSlogfilehandler together with XPRS_ge_setcbmsghandler to write all library message output to the file log.txt.

```
XPRS_ge_setcbmsghandler(XPRSlogfilehandler, "log.txt");
```

Details about the use of callback functions can be found in section 5.8.

2.4 Problem Loading

Once a problem pointer has been created, an optimization problem can be loaded into it. The problem can be loaded either from file or from memory via the suite of problem loading and problem manipulation routines available in the Optimizer library interface. The simplest of these approaches, and the only approach available to Console Optimizer users, is to read a matrix from an MPS or LP file using XPRSreadprob (READPROB).

```
{
  XPRSprob prob;
  XPRScreateprob(&prob);
  XPRSsetlogfile(prob, "logfile.log");
  XPRSreadprob(prob, "hpw15", "");
  XPRSdestroyprob(prob);
}
```

Library users can construct the problem in their own arrays and then load this problem specification using one of the functions XPRSloadlp, XPRSloadqp, XPRSloadqcqp, XPRSloadglobal, XPRSloadqglobal or XPRSloadqcqpglobal. During the problem load routine the Optimizer will use the user's data to construct the internal problem representation in new memory that is associated with the problem pointer. Note, therefore, that the user's arrays can be freed immediately after the call. Once the problem has been loaded, any subsequent call to one of these load routines will overwrite the problem currently represented in the problem pointer.

The names of the problem loading routines indicate the type of problem that can be represented using the routine. The following table outlines the components of an optimization problem as denoted by the codes used in the function names.

Code	Problem Content
lp	Linear Program (LP) (linear constraints and linear objective)
qp	Quadratic Program (LP with quadratic objective)
global	Global Constraints (LP with discrete entities e.g., binary variables)
qc	Quadratic Constraints (LP with quadratic constraints)

Many of the array arguments of the load routines can optionally take NULL pointers if the associated component of the problem is not required to be defined. Note, therefore, that the user need only use the XPRSloadqcqpglobal routine to load any problem that can be loaded by the other routines.

Finally, note that the names of the rows and columns of the problem are not loaded together with the problem specification. These may be loaded afterwards using a call to the function XPRSaddnames.

2.5 Problem Solving

With a problem loaded into a problem pointer the user can run the optimization algorithms on the problem to solve it.

The two main commands to run the optimization algorithms on a problem are `XPRSmipoptimize`(`MIPOPTIMIZE`) and `XPRSlpoptimize`(`LPOPTIMIZE`) depending on whether the problem needs to be solved with or without global entities. The `XPRSlpoptimize` function will solve LPs, QPs and QCQPs or the initial continuous relaxation of a MIP problem, depending on the type of problem loaded in the problem pointer. The `XPRSmipoptimize` function will solve MIPs, MIQPs and MIQCQPs.

For problems with global entities the Optimizer can be told to stop after having solved the initial relaxation by passing the 'l' flag to the `XPRSmipoptimize` function. The remaining MIP search can be run by calling the `XPRSmipoptimize` function without the 'l' flag.

```
{
  XPRSprob prob;
  XPRScreateprob(&prob);
  XPRSsetlogfile(prob, "logfile.log");
  XPRSreadprob(prob, "hpw15", "");
  XPRSmipoptimize(prob, "");
  XPRSdestroyprob(prob);
}
```

2.6 Interrupting the Solve

It is common that users need to interrupt iterations before a solving algorithm is complete. This is particularly common when solving MIP problems since the time to solve these to completion can be large and users are often satisfied with near–optimal solutions. The Optimizer provides for this with structured interrupt criteria using controls and with user–triggered interrupts.

As described previously in section 1.2.4 Console Optimizer can receive a user–triggered interrupt from the keyboard Ctrl–C event. It was also described in this previous section how interrupted commands could be resumed by simply reissuing the command. Similarly, optimization runs started from the library interface and interrupted by either structured or user–triggered interrupts, will return from the call in such a state that the run may be resumed with a follow–on call.

To setup structured interrupts the user will need to set the value of certain controls. Controls are scalar values that are accessed by their name in Console Optimizer and by their id number via the library interface using functions such as `XPRSgetintcontrol` and `XPRSsetintcontrol`. These particular library functions are used for getting and setting the values of integer controls. Similar library functions are used for accessing double precision and string type controls.

Some types of structured interrupts include limits on iterations of the solving algorithms and a limit on the overall time of the optimization run. Limits on the simplex algorithms' iterations are set using the control `LPITERLIMIT`. Iterations of the Newton barrier algorithm are limited using the control `BARITERLIMIT`. A limit on the number of nodes processed in the branch and bound search when solving MIP problems is provided with the `MAXNODE` control. The integer control `MAXTIME` is used to limit the overall run time of the optimization run.

Note that it is important to be careful when using interrupts, to ensure that the optimization run is not being unduly restricted. This is particularly important when using interrupts on MIP optimization runs. Specific controls to use as stopping criteria for the MIP search are discussed in section 4.3.7.

```
{
  XPRSprob prob;
  XPRScreateprob(&prob);
  XPRSsetlogfile(prob, "logfile.log");
  XPRSreadprob(prob, "hpw15", "");
  XPRSsetintcontrol(prob, XPRS_MAXNODE, 20000);
  XPRSmipoptimize(prob, "");
  XPRSdestroyprob(prob);
}
```

Finally note that library users can trigger an interrupt on an optimization run (in a similar way to the Ctrl–C interrupt in Console Optimizer) using a call to the function XPRSinterrupt. It is recommended that the user call this function from a callback during the optimization run. See section 5.8 for details about using callbacks.

2.7 Results Processing

Once the optimization algorithms have completed, either a solution will be available, or else the problem will have been identified as infeasible or unbounded. In the latter case, the user might want to know what caused this particular outcome and take steps to correct it. How to identify the causes of infeasibility and unboundedness are discussed in Chapter 6. In the former case, however, the user typically wants to retrieve the solution information into the required format.

The FICO Xpress Optimizer provides a number of functions for accessing solution information. An ASCII solution file can be obtained by XPRSwriteslxsol (WRITESLXSOL). The .slx format is similar format to the .mps format for MIP models and to the .sol format. Files in .slx format can be read back into the optimizer using the XPRSreadslxsol function. An extended solution file with additional information per column may be obtained as an ASCII file using either of XPRSwritesol (WRITESOL) or XPRSwriteprtsol(WRITEPRTSOL).

Library interface users may additionally access the current LP, QP or QCQP solution information via memory using XPRSgetlpsol. By calling XPRSgetlpsol the user can obtain copies of the double precision values of the decision variables, the slack variables, dual values and reduced costs. Library interface users can obtain the last MIP solution information with the XPRSgetmipsol function.

In addition to the arrays of solution information provided by the Optimizer, summary solution information is also available through *problem attributes*. These are named scalar values that can be accessed by their id number using the library functions XPRSgetintattrib, XPRSgetdblattrib and XPRSgetstrattrib. Examples of attributes include LPOBJVAL and MIPOBJVAL, which return the objective function values for the current LP, QP or QCQP solution and the last MIP solution, respectively. A full list of attributes may be found in Chapter 10.

When the optimization routine returns it is recommended that the user check the status of the run to ensure the results are interpreted correctly. For continuous optimization runs (started with XPRSlpoptimize) the status is available using the LPSTATUS integer problem attribute. For MIP optimization runs (started with XPRSmipoptimize) the status is available using the MIPSTATUS integer problem attribute. See the attribute's reference section for the definition of their values.

```
{
  XPRSprob prob;
  int nCols;
  double *x;
  XPRScreateprob(&prob);
  XPRSsetlogfile(prob, "logfile.log");
  XPRSreadprob(prob, "hpw15", "");
  XPRSgetintattrib(prob, XPRS_COLS, &nCols);
  XPRSsetintcontrol(prob, XPRS_MAXNODE, 20000);
  XPRSmipoptimize(prob, "");
  XPRSgetintattrib(prob, XPRS_MIPSTATUS, &iStatus);
  if(iStatus == XPRS_MIP_SOLUTION || iStatus == XPRS_MIP_OPTIMAL) {
    x = (double *) malloc(sizeof(double) * nCols);
    XPRSgetmipsol(prob, x, NULL);
  }
  XPRSdestroyprob(prob);
}
```

Note that, unlike for LP, QP or QCQP solutions, dual solution information is *not* provided with the call to XPRSgetmipsol and is not automatically generated with the MIP solutions. Only the decision and slack variable values for a MIP solution are obtained when calling XPRSgetmipsol. The reason for this is that MIP problems do not satisfy the theoretical conditions by which dual information is derived (i.e., Karush–Kuhn–Tucker conditions). In particular, this is because the MIP constraint functions are, in general, not continuously differentiable (indeed, the domains of integer variables are not continuous).

Despite this, some useful dual information can be generated if a MIP has continuous variables and we solve the resulting LP problem generated by fixing the non–continuous component of the problem to their solution values. Because this process can be expensive it is left to the user to perform this in a post solving phase where the user will simply call the function `XPRSfixglobals` followed with a call to the optimization routine `XPRSlpoptimize`.

2.8 Function Quick Reference

2.8.1 Administration

`XPRSinit`	Initialize the Optimizer.
`XPRScreateprob`	Create a problem pointer.
`XPRSsetlogfile`	Direct all Optimizer output to a log file.
`XPRSaddcbmessage`	Define a message handler callback function.
`XPRSgetintcontrol`	Get the value of an integer control,
`XPRSsetintcontrol`	Set the value of an integer control.
`XPRSinterrupt`	Set the interrupt status of an optimization run.
`XPRSdestroyprob`	Destroy a problem pointer.
`XPRSfree`	Release resources used by the Optimizer.

2.8.2 Problem Loading

`XPRSreadprob`	Read an MPS or LP format file.
`XPRSloadlp`	Load an LP problem.
`XPRSloadqp`	Load a quadratic objective problem.
`XPRSloadqcqp`	Load a quadratically constrained, quadratic objective problem.
`XPRSloadglobal`	Load a MIP problem.
`XPRSloadqglobal`	Load a quadratic objective MIP problem.
`XPRSloadqcqpglobal`	Load a quadratically constrained, quadratic objective MIP problem.
`XPRSaddnames`	Load names for a range of rows or columns in a problem.

2.8.3 Problem Solving

`XPRSreadbasis`	Read a basis from file.
`XPRSloadbasis`	Load a basis from user arrays.
`XPRSreaddirs`	Read a directives file.
`XPRSlpoptimize`	Solve the problem without global entities.
`XPRSmipoptimize`	Run the problem with global entities.
`XPRSfixglobals`	Fix the discrete variables in the problem to the values of the current MIP solution stored with the problem pointer.
`XPRSgetbasis`	Copy the current basis into user arrays.
`XPRSwritebasis`	Write the current basis to file.

2.8.4 Results Processing

`XPRSwritesol`	Write the current solution to ASCII files.
`XPRSwriteprtsol`	Write the current solution in printable format to file.
`XPRSgetlpsol`	Copy the current LP solution values into user arrays.
`XPRSgetmipsol`	Copy the values of the last MIP solution into user arrays.
`XPRSgetintattrib`	Get the value of an integer problem attribute e.g., by passing the id `MIPSOLS` the user can get the number of MIP solutions found.
`XPRSgetdblattrib`	Get the value of a double problem attribute e.g., by passing the id `MIPOBJVAL` the user can get the objective value of the last MIP solution.
`XPRSgetstrattrib`	Get the value of a string problem attribute.

2.9 Summary

In the previous sections a brief introduction is provided to the most common features of the FICO Xpress Optimizer and its most general usage. The reader should now be familiar with the main routines in the Optimizer library. These routines allow a user to create problem pointers and load problems into these problem pointers. The reader should be familiar with the requirements for setting up message handling with the Optimizer library. Also the reader should know how to run the optimization algorithms on the loaded problems and be familiar with the various ways that results can be accessed.

Examples of using the Optimizer are available from a number of sources, most notably from FICO Xpress Getting Started manual. This provides a straight forward, "hands–on" approach to the FICO Xpress Optimization Suite and it is highly recommended that users read the relevant chapters before considering the reference manuals. Additionally, more advanced, examples may be downloaded from the website.

CHAPTER 3

Problem Types

The FICO Xpress Optimization Suite is a powerful optimization tool for solving Mathematical Programming problems. Users of FICO Xpress formulate real–world problems as Mathematical Programming problems by defining a set of decision variables, a set of constraints on these variables and an objective function of the variables that should be maximized or minimized. Our FICO Xpress users have applications that define and solve important Mathematical Programming problems in academia and industry, including areas such as production scheduling, transportation, supply chain management, telecommunications, finance and personnel planning.

Mathematical Programming problems are usually classified according to the types of decision variables, constraints and objective function in the problem. Perhaps the most popular application of the FICO Xpress Optimizer is for the class of Mixed Integer Programs (MIPs). In this section we will briefly introduce some important types of problems.

3.1 Linear Programs (LPs)

Linear Programming (LP) problems are a very common type of optimization problems. In this type of problem all constraints and the objective function are linear expressions of the decision variables. Each decision variable is restricted to some continuous interval (typically non–negative). Although the methods for solving these types of problems are well known (e.g., the simplex method), only a few efficient implementations of these methods (and additional specialized methods for particular classes of LPs) exists, and these are often crucial for solving the increasingly large instances of LPs arising in industry.

3.2 Mixed Integer Programs (MIPs)

Many problems can be modeled satisfactorily as Linear Programs (LPs), i.e., with variables that are only restricted to having values in continuous intervals. However, a common class of problems requires modeling using discrete variables. These problems are called Mixed Integer Programs (MIPs). MIP problems are often hard to solve and may require large amounts of computation time to obtain even satisfactory, if not optimal, results.

Perhaps the most common use of the FICO Xpress Optimization Suite is for solving MIP problems and it is designed to handle the most challenging of these problems. Besides providing solution support for MIP problems the Optimizer provides support for a variety of popular MIP modeling constructs:

Binary variables – decision variables that have value either `0` or `1`, sometimes called 0/1 variables;

Integer variables – decision variables that have integer values;

Semi–continuous variables – decision variables that either have value `0`, or a continuous value above a specified non–negative limit. Semi–continuous variables are useful for modeling cases where, for example, if a quantity is to be supplied at all then it will be supplied starting from some minimum level (e.g., a power generation unit);

Semi–continuous integer variables – decision variables that either have value `0`, or an integer value above a specified non–negative limit;

Partial integer variables – decision variables that have integer values below a specified limit and continuous values above the limit. Partial integer variables are useful for modeling cases where a supply of some quantity needs to be modeled as discrete for small values but we are indifferent whether it is discrete when the values are large (e.g., because, say, we do not need to distinguish between 10000 items and 10000.25 items);

Special ordered sets of type one (SOS1) — 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. SOS1s are useful for modeling quantities that are taken from a specified discrete set of continuous values (e.g., choosing one of a set of transportation capacities);

Special ordered sets of type two (SOS2) – 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. SOS2s are useful for modeling a piecewise linear quantity (e.g., unit cost as a function of volume supplied);

Indicator constraints – 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. Indicator constraints are useful for modeling cases where supplying some quantity implies that a fixed cost is incurred; otherwise if no quantity is supplied then there is no fixed cost (e.g., starting up a production facility to supply various types of goods and the total volume of goods supplied is bounded above).

Piecewise linear constraints – 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 piecewise 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. Piecewise linear constraints are useful to model e.g. transportation costs that are constant/linear in specific intervals but may jump between the different brackets.

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

All of the above MIP variable types are collectively referred to as `global entities`.

3.3 Quadratic Programs (QPs)

Quadratic Programming (QP) problems are an extension of Linear Programming (LP) problems where the objective function may include a second order polynomial. An example of this is where the user wants to minimize the statistical variance (a quadratic function) of the solution values.

The FICO Xpress Optimizer can be used directly for solving QP problems with support for quadratic objectives in the MPS and LP file formats and library routines for loading QPs and manipulating quadratic objective functions. Note that the Optimizer requires the quadratic function to be `convex` see section 3.4.2 for a description about convexity)

3.4 Quadratically Constrained Quadratic Programs (QCQPs)

Quadratically Constrained Quadratic Programs (QCQPs) are an extension of the Quadratic Programming (QP) problem where the constraints may also include second order polynomials.

A QCQP problem may be written as:

minimize:	$c_1x_1+...+c_nx_n+x^TQ_0x$		
subject to:	$a_{11}x_1+...+a_{1n}x_n+x^TQ_1x$	$\leq$	b_1
	...		
	$a_{m1}x_1+...+a_{mn}x_n+x^TQ_mx$	$\leq$	b_m
	$l_1 \leq x_1 \leq u_1,...,l_n \leq x_n \leq u_n$		

where any of the lower or upper bounds l_i or u_i may be infinite.

The FICO Xpress Optimizer can be used directly for solving QCQP problems with support for quadratic constraints and quadratic objectives in the MPS and LP file formats and library routines for loading QCQPs and manipulating quadratic objective functions and the quadratic component of constraints.

Properties of QCQP problems are discussed in the following few sections.

3.4.1 Algebraic and matrix form

Each second order polynomial can be expressed as x^TQx where Q is an appropriate symmetric matrix: the quadratic expressions are generally either given in the algebraic form

$$a_{11}x_1^2 + 2a_{12}x_1x_2 + 2a_{13}x_1x_3 + ... + a_{22}x_2^2 + 2a_{23}x_2x_3 + ...$$

like in LP files, or in the matrix form x^TQx where

$$Q = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & \\ \vdots & & \ddots & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

like in MPS files. As symmetry is always assumed, $a_{ij} = a_{ji}$ for all index pairs (i, j).

3.4.2 Convexity

A fundamental property for nonlinear optimization problems, thus in QCQP as well, is convexity. A region is called *convex* if for any two points from the region the connecting line segment is also part of the region.

The lack of convexity may give rise to several unfavorable model properties. Lack of convexity in the objective may introduce the phenomenon of locally optimal solutions that are not global ones (a local optimal solution is one for which a neighborhood in the feasible region exists in which that solution is the best). While the lack of convexity in constraints can also give rise to local optima, they may even introduce non–connected feasible regions as shown in Figure 3.1.

In this example, the feasible region is divided into two parts. In region B, the objective function has two alternative locally optimal solutions, while in region A the objective function is not even bounded.

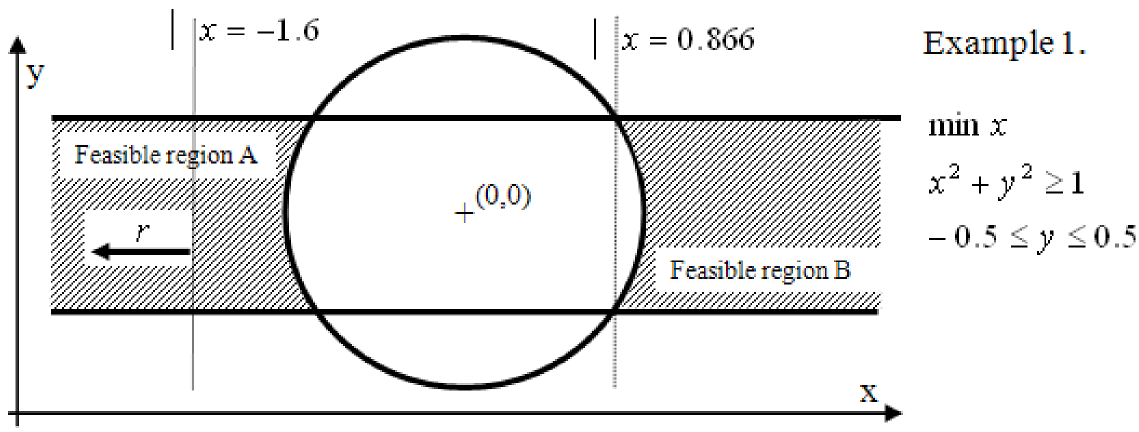


Figure 3.1: Non-connected feasible regions

For convex problems, each locally optimal solution is a global one, making the characterization of the optimal solution efficient.

3.4.3 Characterizing Convexity in Quadratic Constraints

A quadratic constraint of the form

$$a_1x_1 + ... + a_nx_n + x^TQx \leq b$$

defines a convex region if and only if Q is a so–called *positive semi–definite* (PSD) matrix.

A square matrix Q is PSD by definition if for any vector (not restricted to the feasible set of a problem) x it holds that $x^TQx \geq 0$.

It follows that for greater-than or equal constraints

$$a_1x_1 + ... + a_nx_n + x^TQx \geq b$$

the negative of Q must be PSD.

A nontrivial quadratic equality constraint (one for which not every coefficient is zero) always defines a nonconvex region (or in other words, if both Q and its negative is PSD, then Q must be equal to the 0 matrix). Therefore, quadratic equality constraints are not allowed by the Optimizer.

Determining whether a matrix is PSD is not always obvious nor trivial. There are certain constructs, however, that can easily be recognized as being non convex:

1. the product of two variables, say xy, without having both x^2 and y^2 present;
2. having $-x^2$ in any quadratic expression in a less–than or equal constraint, or having x^2 in any greater– than or equal constraint.

3.5 Second Order Cone problems (SOCPs)

Second order cone problems (SOCP) are a special class of quadratically constrained problems, where the quadratic matrix Q is not required to be semi–definite.

The FICO Xpress Optimizer supports (mixed integer) second order cone problems that satisfy the following requirements.

Each quadratic constraint satisfies one of the following two forms:

1. **Second order (or Lorentz) cone**: $x_1^2 + x_1^2 + ... + x_k^2 - t^2 \leq 0$ where $t \geq 0$
2. **Rotated second order (or Lorentz) cone**: $x_1^2 + x_1^2 + ... + x_k^2 - 2t_1t_2 \leq 0$ where $t_1, t_2 \geq 0$

All of the cone coefficients must be exactly one, except for the coefficient of 2 for the t_1t_2 product. Constants or linear terms are not allowed.

Cones cannot be overlapping. That is, a variable x can appear in at most one second–order cone constraint.

Second order cone problems are loaded using the same API functions as for quadratic constraints, and the conic constraints are auto–detected by the optimizer at run time.

CHAPTER 4

Solution Methods

The FICO Xpress Optimization Suite provides three fundamental optimization algorithms for LP or QP problems: the *primal simplex*, the *dual simplex* and the *Newton barrier* algorithm (QCQP and SOCP problems are always solved with the *Newton barrier* algorithm). Using these algorithms the Optimizer implements solving functionality for the various types of continuous problems the user may want to solve.

Typically the user will allow the Optimizer to choose what combination of methods to use for solving their problem. For example, by default, the FICO Xpress Optimizer uses the dual simplex method for solving LP problems and the barrier method for solving QP problems. For the initial continuous relaxation of a MIP, the defaults will be different and depends both on the number of solver threads used, the type of the problem and the MIP technique selected.

For most users the default behavior of the Optimizer will provide satisfactory solution performance and they need not consider any customization. However, if a problem seems to be taking an unusually long time to solve or if the solving performance is critical for the application, the user may consider, as a first attempt, to force the Optimizer to use an algorithm other than the default.

The main points where the user has a choice of what algorithm to use are (i) when the user calls the optimization routine `XPRSlpoptimize` (`LPOPTIMIZE`) and (ii) when the Optimizer solves the node relaxation problems during the branch and bound search. The user may force the use of a particular algorithm by specifying flags to the optimization routine `XPRSlpoptimize` (`LPOPTIMIZE`). If the user specifies flags to `XPRSmipoptimize` (`MIPOPTIMIZE`) to select a particular algorithm then this algorithm will be used for the initial relaxation only. To specify what algorithm to use when solving the node relaxation problems during branch and bound use the special control parameter, `DEFAULTALG`.

As a guide for choosing optimization algorithms other than the default consider the following. As a general rule, the dual simplex is usually much faster than the primal simplex if the problem is neither infeasible nor near–infeasible. If the problem is likely to be infeasible or if the user wishes to get diagnostic information about an infeasible problem then the primal simplex is the best choice. This is because the primal simplex algorithm finds a basic solution that minimizes the sum of infeasibilities and these solutions are typically helpful in identifying causes of infeasibility. The Newton barrier algorithm can perform much better than the simplex algorithms on certain classes of problems. The barrier algorithm will, however, likely be slower than the simplex algorithms if, for a problem coefficient matrix A, A^TA is large and dense.

In the following few sections, performance issues relating to these methods will be discussed in more detail. Performance issues relating to the search for MIP solutions will also be discussed.

4.1 Simplex Method

The simplex method was the first efficient method devised for solving Linear Programs (LPs). This method is still commonly used today and there are efficient implementations of the primal and dual simplex methods available in the Optimizer. We briefly outline some basic simplex theory to give the user a general idea of the simplex algorithm's behavior and to define some terminology that is used in the reference sections.

A region defined by a set of constraints is known in Mathematical Programming as a *feasible region*. When these constraints are linear the feasible region defines the solution space of a Linear Programming (LP) problem. Each value of the objective function of an LP defines a hyperplane or a *level set*. A fundamental result of simplex algorithm theory is that an optimal value of the LP objective function will occur when the level set grazes the boundary of the feasible region. The optimal level set either intersects a single point (or *vertex*) of the feasible region (if such a point exists), in which case the solution is unique, or it intersects a boundary set of the feasible region in which case there is an infinite set of solutions.

In general, vertices occur at points where as many constraints and variable bounds as there are variables in the problem intersect. Simplex methods usually only consider solutions at vertices, or *bases* (known as *basic solutions*) and proceed or iterate from one vertex to another until an optimal solution has been found, or the problem proves to be infeasible or unbounded. The number of iterations required increases with model size, and typically goes up slightly faster than the increase in the number of constraints.

The primal and dual simplex methods differ in which vertices they consider and how they iterate. The dual is the default for LP problems, but may be explicitly invoked using the `d` flag with `XPRSlpoptimize` (`LPOPTIMIZE`).

4.1.1 Output

While the simplex methods iterate, the Optimizer produces iteration logs. Console Optimizer writes these logging messages to the screen. Library users can setup logging management using the various relevant functions in the Optimizer library e.g., `XPRSsetlogfile`, `XPRSaddcbmessage` or `XPRSaddcblplog`. The simplex iteration log is produced for every `LPLOG` simplex iteration. When LPLOG is set to `0`, a log is displayed only when the optimization run terminates. If it is set to a positive value, a summary style log is output; otherwise, a detailed log is output.

4.2 Newton Barrier Method

In contrast to the simplex methods that iterate through boundary points (vertices) of the feasible region, the Newton barrier method iterates through solutions in the interior of the feasible region and will typically find a close approximation of an optimal solution. Consequently, the number of barrier iterations required to complete the method on a problem is determined more so by the required proximity to the optimal solution than the number of decision variables in the problem. Unlike the simplex method, therefore, the barrier often completes in a similar number of iterations regardless of the problem size.

The barrier solver can be invoked on a problem by using the '`b`' flag with `XPRSlpoptimize` (`LPOPTIMIZE`). This is used by default for QP problems, whose quadratic objective functions in general result in optimal solutions that lie on a face of the feasible region, rather than at a vertex.

4.2.1 Crossover

Typically the barrier algorithm terminates when it is within a given tolerance of an optimal solution. Since this solution will not lie exactly on the boundary of the feasible region, the Optimizer can be optionally made to perform a, so–called, 'crossover' phase to obtain an optimal solution on the boundary. The nature of the 'crossover' phase results in a basic optimal solution, which is at a vertex of the feasible region. In the crossover phase the simplex method is used to continue the optimization from the solution found by the barrier algorithm. The `CROSSOVER` control determines whether the Optimizer performs crossover. When set to `1` (the default for LP problems), crossover is performed. If

`CROSSOVER` is set to `0`, no crossover will be attempted and the solution provided will be that determined purely by the barrier method. Note that if a basic optimal solution is required, then the `CROSSOVER` option must be activated before optimization starts.

4.2.2 Output

While the barrier method iterates, the Optimizer produces iteration log messages. Console Optimizer writes these log messages to the screen. Library users can setup logging management using the various relevant functions in the Optimizer library, e.g. `XPRSsetlogfile`, `XPRSaddcbmessage` or `XPRSaddcbbarlog`. Note that the amount of barrier iteration logging is dependent on the value of the `BAROUTPUT` control.

4.3 Branch and Bound

The FICO Xpress Optimizer uses the approach of LP based branch and bound with cutting planes for solving Mixed Integer Programming (MIP) problems. That is, the Optimizer solves the optimization problem (typically an LP problem) resulting from relaxing the discreteness constraints on the variables and then uses branch and bound to search the relaxation space for MIP solutions. It combines this with heuristic methods to quickly find good solutions, and cutting planes to strengthen the LP relaxations.

The Optimizer's MIP solving methods are coordinated internally by sophisticated algorithms so the Optimizer will work well on a wide range of MIP problems with a wide range of solution performance requirements without any user intervention in the solving process. Despite this the user should note that the formulation of a MIP problem is typically not unique and the solving performance can be highly dependent on the formulation of the problem. It is recommended, therefore, that the user undertake careful experimentation with the problem formulation using realistic examples before committing the formulation for use on large production problems. It is also recommended that users have small scale examples available to use during development.

Because of the inherent difficulty in solving MIP problems and the variety of requirements users have on the solution performance on these problems it is not uncommon that users would like to improve over the default performance of the Optimizer. In the following sections we discuss aspects of the branch and bound method for which the user may want to investigate when customizing the Optimizer's MIP search.

4.3.1 Theory

In this section we present a brief overview of branch and bound theory as a guide for the user on where to look to begin customizing the Optimizer's MIP search and also to define the terminology used when describing branch and bound methods.

To simplify the text in the following, we limit the discussion to MIP problems with linear constraints and a linear objective function. Note that it is not difficult to generalize the discussion to problems with quadratic constraints and a quadratic objective function.

The branch and bound method has three main concepts: relaxation, branching and fathoming.

The relaxation concept relates to the way discreteness or integrality constraints are dropped or 'relaxed' in the problem. The initial relaxation problem is a Linear Programming (LP) problem which we solve resulting in one of the following cases:

(a) The LP is infeasible so the MIP problem must also be infeasible;

(b) The LP has a feasible solution, but some of the integrality constraints are not satisfied – the MIP has not yet been solved;

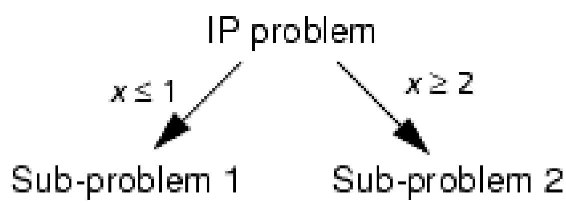
(c) The LP has a feasible solution and all the integrality constraints are satisfied so the MIP has also been solved;

(d) The LP is unbounded.

Case (d) is a special case. It can only occur when solving the initial relaxation problem and in this situation the MIP problem itself is not well posed (see Chapter 6 for details about what to do in this case). For the remaining discussion we assume that the LP is not unbounded.

Outcomes (a) and (c) are said to 'fathom' the particular MIP, since no further work on it is necessary. For case (b) more work is required, since one of the unsatisfied integrality constraints must be selected and the concept of separation applied.

To illustrate the branching concept suppose, for example, that the optimal LP value of an integer variable x is 1.34, a value which violates the integrality constraint. It follows that in any solution to the original problem either $x \leq 1.0$ or $x \geq 2.0$. If the two resulting MIP problems are solved (with the integrality constraints), all integer values of x are considered in the combined solution spaces of the two MIP problems and no solution to one of the MIP problems is a solution to the other. In this way we have branched the problem into two disjoint *sub–problems*.



If both of these sub–problems can be solved and the better of the two is chosen, then the MIP is solved. By recursively applying this same strategy to solve each of the sub–problems and given that in the limiting case the integer variables will have their domains divided into fixed integer values then we can guarantee that we solve the MIP problem.

Branch and bound can be viewed as a *tree–search* algorithm. Each *node* of the tree is a MIP problem. A MIP node is relaxed and the LP relaxation is solved. If the LP relaxation is not fathomed, then the node MIP problem is partitioned into two more sub–problems, or *child* nodes. Each child MIP will have the same constraints as the *parent* node MIP, plus one additional inequality constraint. Each node is therefore either fathomed or has two children or *descendants*.

We now introduce the concept of a *cutoff*, which is an extension of the fathoming concept. To understand the cutoff concept we first make two observations about the behavior of the node MIP problems. Firstly, the optimal MIP objective of a node problem can be no better than the optimal objective of the LP relaxation. Secondly, the optimal objective of a child LP relaxation can be no better than the optimal objective of its parent LP relaxation. Now assume that we are exploring the tree and we are keeping the value of the best MIP objective found so far. Assume also that we keep a 'cutoff value' equal to the best MIP objective found so far. To use the cutoff value we reason that if the optimal LP relaxation objective is no better than the cutoff then any MIP solution of a descendant can be no better than the cutoff and the node can be fathomed (or cutoff) and need not be considered further in the search.

The concept of a cutoff can be extended to apply even when no integer solution has been found in situations where it is known, or may be assumed, from the outset that the optimal solution must be better than some value. If the relaxation is worse than this cutoff, then the node may be fathomed. In this way the user can reduce the number of nodes processed and improve the solution performance. Note that there is the possibility, however, that all MIP solutions, including the optimal one, may be missed if an overly optimistic cutoff value is chosen.

The cutoff concept may also be extended in a different way if the user intends only to find a solution within a certain tolerance of the overall optimal MIP solution. Assume that we have found a MIP solution to our problem and assume that the cutoff is maintained at a value 100 objective units better than the current best MIP solution. Proceeding in this way we are guaranteed to find a MIP solution within 100 units of the overall MIP optimal since we only cutoff nodes with LP relaxation solutions worse than 100 units better than the best MIP solution that we find.

If the MIP problem contains SOS entities then the nodes of the branch and bound tree are determined by branching on the sets. Note that each member of the set has a double precision reference row entry and the sets are ordered by these reference row entries. Branching on the sets is done by choosing a position in the ordering of the set variables and setting all members of the set to 0 either above or below the chosen point. The optimizer used the reference row entries to decide on the branching position and so it is important to choose the reference row entries which reflect the cost of setting the set member to 0. In some cases it maybe better to model the problem with binary variables instead of special ordered sets. This is especially the case if the sets are small.

4.3.2 Variable Selection and Cutting

The branch and bound technique leaves many choices open to the user. In practice, the success of the technique is highly dependent on several key choices.

(a) Deciding which variable to branch on is known as the *variable selection problem* and is often the most critical choice.

(b) *Cutting planes* are used to strengthen the LP relaxation of a sub problem, and can often bring a significant reduction in the number of sub–problems that must be solved

The Optimizer incorporates a default strategy for both choices which has been found to work adequately on most problems. Several controls are provided to tailor the search strategy to a particular problem.

4.3.3 Variable Selection for Branching

Each global entity has a priority for branching, or one set by the user in the directives file. A *low* priority value means that the variable is *more* likely to be selected for branching. The Optimizer uses a priority range of 400–500 by default. To guarantee that a particular global entitity is always branched first, the user should assign a priority value less than 400. Likewise, to guarantee that a global entity is only branched on when it is the only candidate left, a priority value above 500 should be used.

The Optimizer uses a wide variety of information to select among those entities that remain unsatisifed and which belong to the lowest valued priority class. A *pseudo cost* is calculated for each candidate entity, which is typically an estimate of how much the LP relaxation objective value will change (degradationas a result of branching on this particular candidate. Estimates are calculated separately for the up and down branches and combined according to the strategy selected by the `VARSELECTION` control.

The default strategy is based on calculating pseudo costs using the method of *strong branching*. With strong brancing, the LP relaxations of the two potential sub problems that would result from branching on a candidate global entity, are solved partially. Dual simplex is applied for a limited number of iterations and the change in objective value is recorded as a pseudo cost. This can be very expensive to apply to every candidate for every node of the branch and bound search, which is why the Optimizer by default will reuse pseudo costs collected from one node, on subsequent nodes of the search.

Selecting a global entity for branching is a multi–stage process, which combines estimates that are cheap to compute, with the more expensive strong branching based pseudo costs. The basic selection process is given by the following outline, together with the controls that affect each step:

1. Pre–filter the set of candidate entities using very cheap estimates.
   `SBSELECT`: determine the filter size.

2. Calculate simple estimates based on local node information and rank the selected candidates.
   SBESTIMATE: local ranking function.
3. Calculate strong–branching pseudo costs for candidates lacking such information.
   SBBEST: number of variables to strong branch on.
   SBITERLIMIT: LP iteration limit for strong branching.
4. Select the best candidate using a combination of pseudo costs and the local ranking functions.

The overall amount of effort put into this process can be adjusted using the SBEFFORT control.

4.3.4 Cutting Planes

Cutting planes are valid constraints used for tightening the LP relaxation of a MIP problem, without affecting the MIP solution space. They can be very effective at reducing the amount of sub problems that the branch and bound search has to solve. The Optimizer will automatically create many different well–known classes of cutting planes, such as *mixed integer Gomory cuts*, *lift–and–project cuts*, *mixed integer rounding* (MIR) *cuts*, *clique cuts*, *implied bound cuts*, *flow–path cuts*, *zero–half cuts*, etc. These classes of cuts are grouped together into two groups that can be controlled separately. The following table lists the main controls and the related cut classes that are affected by those control:

COVERCUTS	Mixed integer rounding cuts
TREECOVERCUTS	Lifted cover cuts
	Clique cuts
	Implied bound cuts
	Flow–path cuts
	Zero–half cuts
GOMCUTS	Mixed integer Gomory cuts
TREEGOMCUTS	Lift–and–project cuts

The controls COVERCUTS and GOMCUTS sets an upper limit on the number of rounds of cuts to create for the root problem, for their respective groups. Correspondingly, TREECOVERCUTS and TREEGOMCUTS sets an upper limit on the number of rounds of cuts for any sub problem in the tree.

An important aspect of cutting is the choice of how many cuts to add to a sub problem. The more cuts that are added, the harder it becomes to solve the LP relaxation of the node problem. The tradeoff is therefore between the additional effort in solving the LP relaxation versus the strengthening of the sub problem. The CUTSTRATEGY control sets the general level of how many cuts to add, expressed as a value from `0` (no cutting at all) to `3` (high level of cuts).

Another important aspect of cutting is how often cuts should be created and added to a sub problem. The Optimizer will automatically decide on a frequency that attempts to balance the effort of creating cuts versus the benefits they provide. It is possible to override this and set a fixed strategy using the CUTFREQ control. When set to a value *k*, cutting will be applied to every *k*'th level of the branch and bound tree. Note that setting CUTFREQ = `0` will disable cutting on sub problems completely, leaving only cutting on the root problem.

4.3.5 Node Selection

The Optimizer applies a search scheme involving best–bound first search combined with dives. Sub problems that have not been fathomed or which have not been branched further into new sub problems are referred to as *active nodes* of the branch and bound search tree. Such activate nodes are maintained by the Optimizer in a pool.

The search process involves selecting a sub problem (or node) from this active nodes pool and commencing a dive. When the Optimizer branches on a global entity and creates the two sub problems, it has a choice of which of the two sub problems to work on next. This choice is determined by the BRANCHCHOICE control. The dive is a recursive search, where it selects a child problem, branches on it to create two new child problems, and repeats with one of the new child problems, until it ends with a sub problem that should not be branched further. At this point it will go back to the active nodes pool and pick a new sub problem to perform a dive on. This is called a *backtrack* and the choice of node is determined by the BACKTRACK control. The default backtrack strategy will select the active node with the best bound.

4.3.6 Adjusting the Cutoff Value

The parameter MIPADDCUTOFF determines the cutoff value set by the Optimizer when it has identified a new MIP solution. The new cutoff value is set as the objective function value of the MIP solution plus the value of MIPADDCUTOFF. If MIPADDCUTOFF has not been set by the user, the value used by the Optimizer will be calculated after the initial LP optimization step as:

$$\max(\mathtt{MIPADDCUTOFF}, 0.01 \cdot \mathtt{MIPRELCUTOFF} \cdot \mathit{LP\_value})$$

using the initial values for MIPADDCUTOFF and MIPRELCUTOFF, and where *LP_value* is the optimal objective value of the initial LP relaxation.

4.3.7 Stopping Criteria

Often when solving a MIP problem it is sufficient to stop with a good solution instead of waiting for a potentially long solve process to find an optimal solution. The Optimizer provides several stopping criteria related to the solutions found, through the MIPRELSTOP and MIPABSSTOP parameters. If MIPABSSTOP is set for a minimization problem, the Optimizer will stop when it finds a MIP solution with an objective value equal to or less than MIPABSSTOP. The MIPRELSTOP parameter can be used to stop the solve process when the found solution is sufficiently close to optimality, as measure relative to the best available bound. The optimizer will stop due to MIPRELSTOP when the following is satisfied:

$$|\mathtt{MIPOBJVAL} - \mathtt{BESTBOUND}| \leq \mathtt{MIPRELSTOP} \cdot \max(|\mathtt{BESTBOUND}|, |\mathtt{MIPOBJVAL}|)$$

It is also possible to set limits on the solve process, such as number of nodes (MAXNODE), time limit (MAXTIME) or on the number of solutions found (MAXMIPSOL). If the solve process is interrupted due to any of these limits, the problem will be left in its unfinished state. It is possible to resume the solve from an unfinished state by calling XPRSmipoptimize (MIPOPTIMIZE) again.

To return an unfinished problem to its starting state, where it can be modified again, the user should use the function XPRSpostsolve (POSTSOLVE). This function can be used to restore a problem from an interrupted global search even if the problem is not in a presolved state.

4.3.8 Integer Preprocessing

If MIPPRESOLVE has been set to a nonzero value before solving a MIP problem, integer preprocessing will be performed at each node of the branch and bound tree search (including the root node). This incorporates reduced cost tightening of bounds and tightening of implied variable bounds after branching. If a variable is fixed at a node, it remains fixed at all its child nodes, but it is not deleted from the matrix (unlike the variables fixed by presolve).

MIPPRESOLVE is a bitmap whose values are acted on as follows:

Bit	Value	Action
0	1	Reduced cost fixing;
1	2	Integer implication tightening.
2	4	*Unused*
3	8	Tightening of implied continuous variables.
4	16	Fixing of variables based on dual (i.e. optimality) implications.

So a value of 1+2=3 for `MIPPRESOLVE` causes reduced cost fixing and tightening of implied bounds on integer variables.

4.4 QCQP and SOCP Methods

Continuous QCQP and SOCP problems are always solved by the Xpress Newton–barrier solver. For QCQP, SOCP and QP problems, there is no solution purification method applied after the barrier (such as the crossover for linear problems). This means that solutions tend to contain more active variables than basic solutions, and fewer variables will be at or close to one of their bounds.

When solving a linearly constrained quadratic program (QP) from scratch, the Newton barrier method is usually the algorithm of choice. In general, the quadratic simplex methods are better if a solution with a low number of active variables is required, or when a good starting basis is available (e.g., when reoptimizing).

4.4.1 Convexity Checking

The Optimizer requires that the quadratic coefficient matrix in each constraint or in the objective function is either positive semi–definite or negative semi–definite, depending on the sense of for constraints or the direction of optimization for the objective. The only exception is when a quadratic constraint describes a second order cone. Quadratic constraints and a quadratic objective is therefore automatically checked for convexity. Note that this convexity checker will reject any problem where this requirement is violated by more than a small tolerance.

Each constraint is checked individually for convexity. In certain cases it is possible that the problem itself is convex, but the representation of it is not. A simple example would be

minimize:	x		
subject to:	$x^2 - y^2 + 2xy$	$\leq$	1
	$y=0$		

The optimizer will deny solving this problem if the automatic convexity check is on, although the problem is clearly convex. The reason is that convexity of QCQPs is checked before any presolve takes place. To understand why, consider the following example:

minimize:	y		
subject to:	$y - x^2$	$\leq$	1
	$y=2$		

This problem is clearly feasible, and an optimal solution is $(x, y) = (1, 2)$. However, when presolving the problem, it will be found infeasible, since assuming that the quadratic part of the first constraint is convex the constraint cannot be satisfied (remember that if a constraint is convex, then removing the quadratic part is always a relaxation). Thus since presolve makes use of the assumption that the problem is convex, convexity must be checked before presolve.

Note that for quadratic programming (QP) and mixed integer quadratic programs (MIQP) where the quadratic expressions appear only in the objective, the convexity check takes place after presolve, making it possible to accept matrices that are not PSD, but define a convex function over the feasible region (note that this is only a chance).

It is possible to turn the automatic convexity check off. By doing so, one may save time when solving problems that are known to be convex, or one might even want to experiment trying to solve non–convex problems. For a non–convex problem, any of the following might happen:

1. the algorithm converges to a local optimum which it declares optimal (and which might or might not be the actual optimum);
2. the algorithm doesn't converge and stops after reaching the iteration limit;
3. the algorithm cannot make sufficient improvement and stops;
4. the algorithm stops because it cannot solve a subproblem (in this case it will declare the matrix non semidefinite);
5. presolve declares a feasible problem infeasible;
6. presolve eliminates variables that otherwise play an important role, thus significantly change the model;
7. different solutions (even feasibility/infeasibility) are generated to the same problem, only by slightly changing its formulation.

There is no guarantee on which of the cases above will occur, and as mentioned before, the behavior/outcome might be formulation dependent. One should take extreme care when interpreting the solution information returned for a non–convex problem.

4.4.2 Quadratically Constrained and Second Order Cone Problems

Quadratically constrained and second order cone problems are solved by the barrier algorithm.

Mixed integer quadratically constrained (MIQCQP) and mixed integer second order problems (MISOCP) are solved using traditional branch and bound using the barrier to solve the node problems, or by means of outer approximation, as defined by control `MIQCPALG`.

It is sometimes beneficial to solve the root node of an MIQCQP or MISOCP by the barrier, even if outer approximation is used later; controlled by the `QCROOTALG` control. The number of cut rounds on the root for outer approximation is defined by `QCCUTS`.

CHAPTER 5

Advanced Usage

5.1 Problem Names

Problems loaded in the Optimizer have a name. The name is either taken from the file name if the problem is read into the optimizer or it is specified as a string in a function call when a problem is loaded into the Optimizer using the library interface. Once loaded the name of the problem can be queried and modified. For example, the library provides the function `XPRSsetprobname` for changing the name of a problem.

When reading a problem from a matrix file the user can optionally specify a file extension. The search order used for matrix files in the case where the file extension is not specified is described in the reference for the function `XPRSreadprob`. In this case, the problem name becomes the file name, including the full path, but without the file extension.

Note that matrix files can be read directly from a gzip compressed file. Recognized names of matrix files stored with gzip compression have an extension that is one of the usual matrix file format extensions followed by the `.gz` extension. For example, `hpw15.mps.gz`.

The problem name is used as a default base name for the various file system interactions that the Optimizer may make when handling a problem. For example, when commanded to read a basis file for a problem and the basis file name is not supplied with the read basis command the Optimizer will try to open a file with the problem name appended with the `.bss` extension.

It is useful to note that the problem name can include file system path information. For example, `c:/matrices/hpw15`. Note the use of forward slashes in the Windows path string. It is recommended that Windows users use forward slashes as path delimiters in all file name specifications for the Optimizer since (i) this will work in all situations and (ii) it avoids any problems with the back slash being interpreted as the escape character.

5.2 Manipulating the Matrix

In general, the basic usage of the FICO Xpress Optimizer described in the previous chapters will be sufficient for most users' requirements. Using the Optimizer in this way simply means load the problem, solve the problem, get the results and finish.

In some cases, however, it is required that the problem is first solved, then modified, and solved again. We may want to do this, for example, if a problem was found to be infeasible. In this case, to find a feasible subset of constraints we iteratively remove some constraints and re–solve the problem. Another example is when a user wants to 'generate' columns using the optimal duals of a 'restricted' LP problem. In this case we will first need to load a problem and then we will need to add columns to this problem after it has been solved.

For library users, FICO Xpress provides a suite of functions providing read and modify access to the matrix.

5.2.1 Reading the Matrix

The Optimizer provides a suite of routines for read access to the optimization problem including access to the objective coefficients, constraint right hand sides, decision variable bounds and the matrix coefficients.

It is important to note that the information returned by these functions will depend on whether or not the problem has been run through an optimization algorithm or if the problem is currently being solved using an optimization algorithm, in which case the user will be calling the access routines from a callback (see section 5.8 for details about callbacks). Note that the dependency on when the access routine is called is mainly due to the way "presolve" methods are applied to modify the problem. How the presolve methods affect what the user accesses through the read routines is discussed in section 5.3.

The user can access the names of the problem's constraints, or 'rows', as well as the decision variables, or 'columns', using the `XPRSgetnames` routine.

The linear coefficients of the problem constraints can be read using `XPRSgetrows`. Note that for the cases where the user requires access to the linear matrix coefficients in the column–wise sense the Optimizer includes the `XPRSgetcols` function. The type of the constraint, the right hand side and the right hand side range are accessed using the functions `XPRSgetrowtype`, `XPRSgetrhs` and `XPRSgetrhsrange`, respectively.

The coefficients of the objective function can be accessed using the `XPRSgetobj` routine, for the linear coefficients, and `XPRSgetqobj` for the quadratic objective function coefficients. The type of a column (or decision variable) and its upper and lower bounds can be accessed using the routines `XPRSgetcoltype`, `XPRSgetub` and `XPRSgetlb`, respectively.

The quadratic coefficients in constraints can be accessed either in matrix form, using the `XPRSgetqrowqmatrix` routine, or as a list of quadratic coefficients with the `XPRSgetqrowqmatrixtriplets`.

Note that the reference section in Chapter 8 of this manual provides details on the usage of these functions.

5.2.2 Modifying the Matrix

The Optimizer provides a set of routines for manipulating the problem data. These include a set of routines for adding and deleting problem constraints ('rows') and decision variables ('columns'). A set of routines is also provided for changing individual coefficients of the problem and for changing the types of decision variables in the problem.

Rows and columns can be added to a problem together with their linear coefficients using `XPRSaddrows` and `XPRSaddcols`, respectively. Rows and columns can be deleted using `XPRSdelrows` and `XPRSdelcols`, respectively.

The Optimizer provides a suite of routines for modifying the data for existing rows and columns. The linear matrix coefficients can be modified using `XPRSchgcoef` (or use `XPRSchgmcoef` if a batch of coefficients are to be changed). Row and column types can be changed using the routines `XPRSchgrowtype` and `XPRSchgcoltype`, respectively. Right hand sides and their ranges may be changed with `XPRSchgrhs` and `XPRSchgrhsrange`. The linear objective function coefficients may be changed with `XPRSchgobj` while the quadratic objective function coefficients are changed using `XPRSchgqobj` (or use `XPRSchgmqobj` if a batch of coefficients are to be changed). Likewise, quadratic coefficients in constraints are changed with `XPRSchgqrowcoeff`.

Examples of the usage of all the above functions and their syntax may be found in the reference section of this manual in Chapter 8.

Finally, it is important to note that it is not possible to modify a matrix when it has been 'presolved' and has not been subsequently 'postsolved'. The following section 5.3 discusses some important points concerning reading and modifying a problem that is "presolved".

5.3 Working with Presolve

The Optimizer provides a number of algorithms for simplifying a problem prior to the optimization process. This elaborate collection of procedures, known as *presolve*, can often greatly improve the Optimizer's performance by modifying the problem matrix, making it easier to solve. The presolve algorithms identify and remove redundant rows and columns, reducing the size of the matrix, for which reason most users will find it a helpful tool in reducing solution times. However, presolve is included as an option and can be disabled if not required by setting the `PRESOLVE` control to `0`. Usually this is set to `1` and presolve is called by default.

For some users the presolve routines can result in confusion since a problem viewed in its presolved form will look very different to the original model. Under standard use of the Optimizer this may cause no difficulty. On a few occasions, however, if errors occur or if a user tries to access additional properties of the matrix for certain types of problem, the presolved values may be returned instead. In this section we provide a few notes on how such confusion may be best avoided. If you are unsure if the matrix is in a presolved state or not, check the `PRESOLVESTATE` attribute

It is important to note that when solving a problem with presolve on, the Optimizer will take a copy of the matrix and modify the copy. The original matrix is therefore preserved, but will be inaccessible to the user while the presolved problem exists. Following optimization, the whole matrix is automatically *postsolved* to recover a solution to the original problem and restoring the original matrix. Consequently, either before or after, but not during, a completed optimization run, the full matrix may be viewed and altered as described above, being in its original form.

A problem might be left in a presolved state if the solve was interrupted, for example due to the Ctrl–C key combination, or if a time limit (set by `MAXTIME`) was reached. In such a case, the matrix can always be returned to its original state by calling `XPRSpostsolve` (`POSTSOLVE`). If the matrix is already in the original state then `XPRSpostsolve` (`POSTSOLVE`) will return without doing anything.

While a problem is in a presolved state it is not possible to make any modifications to it, such as adding rows or columns. The problem must first be returned to its original state by calling `XPRSpostsolve` before it can be changed.

5.3.1 (Mixed) Integer Programming Problems

If a model contains global entities, integer presolve methods such as bound tightening and coefficient tightening are applied to tighten the LP relaxation. As a simple example of this might be if the matrix has a binary variable x and one of the constraints of the matrix is $x \leq 0.2$. It follows that x can be fixed at zero since it can never take the value 1. If presolve uses the global entities to alter the matrix in this way, then the LP relaxation is said to have been *tightened*. For Console users, notice of this is sent to the screen; for library users it may be sent to a callback function, or printed to the log file if one has been set up. In such circumstances, the optimal objective function value of the LP relaxation for a presolved matrix may be different from that for the unpresolved matrix.

The strict LP solution to a model with global entities can be obtained by calling the `XPRSlpoptimize` (`LPOPTIMIZE`) command. This removes the global constraints from the variables, preventing the LP relaxation from being tightened and solves the resulting matrix. In the example above, x would not be fixed at 0, but allowed to range between 0 and 0.2.

When `XPRSmipoptimize` (`GLOBAL`) finds an integer solution, it is postsolved and saved in memory. The solution can be read with the `XPRSgetmipsol` function. A permanent copy can be saved to a solution file by calling `XPRSwritebinsol` (`WRITEBINSOL`), or `XPRSwriteslxsol` (`WRITESLXSOL`) for a simpler text file. This can be retrieved later by calling `XPRSreadbinsol` (`READBINSOL`) or `XPRSreadslxsol` (`READSLXSOL`), respectively.

After calling `XPRSmipoptimize` (MIPOPTIMIZE), the matrix will be postsolved whenever the MIP search has completed. If the MIP search hasn't completed the matrix can be postsolved by calling the `XPRSpostsolve` (`POSTSOLVE`) function.

5.4 Working with LP Folding

In addition to presolve procedures, the Optimizer provides an algorithm called LP folding that can further simplify LP problems. The LP folding is applicable to LP problems that can be partitioned into *equitable partitions*, and it works by aggregating matrix columns of equitable partitions and then reducing the problem size.

Solutions for the folded problem are also valid for the original problem. While it is straightforward to transfer a solution from the folded problem to the original problem, it is non–trivial to do so for the basis. When an LP problem is solved to optimality and a basis is needed, the LP unfolding will use the crossover algorithm to provide one. When the folded LP problem is unbounded or infeasible, or when the solving process is stopped due to time or iteration limit, the basis will not be available. Please note that LP folding tends to provide solutions with a larger support (number of variables that are not at any of their bounds).

LP folding is applied automatically when appropriate. It can be enabled or disabled by setting the `LPFOLDING` control.

5.5 Working with Heuristics

The Optimizer contains several primal heuristics that help to find feasible solutions during a global search. These heuristics fall broadly into one of three classes:

1. Simple rounding heuristics
   These take the continuous relaxation solution to a node and, through simple roundings of the solution values for global entities, try to construct a feasible MIP solution. These are typically run on every node.
2. Diving heuristics
   These start from the continuous relaxation solution to a node and combines rounding and fixing of global entities with occasional reoptimization of the continuous relaxation to construct a better quality MIP solution. They are run frequently on both the root node and during the branch and bound tree search.
3. Local search heuristics
   The local search heuristics are generally the most expensive heuristics and involve solving one or more smaller MIPs whose feasible regions describe a neighborhood around a candidate MIP solution. These heuristics are run at the end of the root solve and typically on every 500–1000 nodes during the tree search.

Some simple heuristics and a few fast diving heuristics, which do not require a starting solution, will be tried before the initial continuous relaxation of a MIP is solved. On very simple problems, it is possible that an optimal MIP solution will be found at this point, which can lead to the initial relaxation being cut off. These heuristics can be enabled or disabled using the `HEURBEFORELP` control.

There are a few controls that affect all of the heuristics:

`HEURSTRATEGY`	Determines the level of heuristics to use. A value of 3 will allow all heuristics to be run and a value of 1 will only allow the faster rounding and diving heuristics to be run. Setting `HEURSTRATEGY` to 0 will disable all heuristics.

`HEURTHREADS`	The number of additional heuristic threads to start in parallel with cutting on the root node. If set to zero, heuristics will be run interleaved with cutting.

The simple rounding heuristics do not have any controls associated with them. The diving heuristics have the following controls:

`HEURFREQ`	The frequency at which to run a diving heuristic during the branch and bound tree search. If `HEURFREQ=k`, a diving heuristic will be applied when at least `k` nodes of the tree search have been solved since the last run. Set this control to zero to disable diving heuristics during the tree search. With a default setting of `-1`, the Optimizer will automatically select a frequency that depends on how expensive it is to run and how many integer variables need to be rounded. Typically, this results in a diving heuristic being run for every 10–50 nodes.
`HEURDIVESTRATEGY`	Can be used to select one specific out of 10 predefined diving strategies, otherwise the Optimizer will automatically select which appears to work best. Set this control to zero to disable the diving heuristic.
`HEURDIVERANDOMIZE`	How much randomization to introduce into the diving heuristics.
`HEURDIVESPEEDUP`	The amount of effort to put into the individual dives. This essentially determines how often the continuous relaxation is reoptimized during a dive.

The local search heuristics have the following controls:

`HEURSEARCHFREQ`	The frequency at which to run the local search heuristics during the branch and bound tree search. If `HEURSEARCHFREQ=k`, the local search heuristics will be run when at least `k` nodes of the tree search have been solved since the last run.
`HEURSEARCHEFFORT`	Determines the complexity of the local search MIP problems solved and, if `HEURSEARCHFREQ=-1`, also how often they are applied.
`HEURSEARCHROOTSELECT`	Selects which local search heuristics are allowed to be run on the root node. Each bit of this integer control represents an individual heuristic.
`HEURSEARCHTREESELECT`	Selects which local search heuristics are allowed to be run during the branch and bound tree search.

5.6 Analyzing and Handling Numerical Issues

There are many optimization applications which give rise to numerically challenging models. You might notice that the Optimizer takes unexpectedly long for simplex reoptimization, that minimal changes in the models lead to an unexpectedly large change in the optimal solution value or that the optimal solution shows a certain amount of violation in the postsolved state. The Optimizer provides various tools to analyze whether a model is numerically challenging and to handle numerical issues when they occur.

5.6.1 Analyzing Models for Numerical Issues

There are two main reasons which can make models numerically challenging: Firstly, using coefficients that span many orders of magnitude, e.g., using numbers as large as 100 million mixed with numbers as small as 1 over 100 million. Those span 16 orders of magnitude. A double–precision floating point number, however, can only represent 15 precise digits. Thus, round–off errors are inevitable. Secondly, if a model contains structures that amplify the effect of numeric error propagation, e.g., when the result of subtracting two almost identical values is scaled up and then used for further computations.

To both ends, Xpress provides features to analyze models for numerical stability. Addressing the first issue, Xpress provides the user with information on the coefficient ranges in both the original problem and the problem that is solved after presolving and scaling has been applied. In the log, the minimum and maximum absolute values of the matrix coefficients, the right–hand side/bounds and the objective are printed. The relevant part for the numerical behavior of the solution process are the coefficient ranges in the solved model. The difference between the exponents of the min and max values tells you how many orders of magnitude are covered. As a rule of thumb, those should not be more than nine (and not more than six in an individual row or column of the original matrix). For MIP solves, Xpress will notify the user after the solution of the root LP when the coefficient ranges and other stability measures indicate that the solve might become numerically cumbersome. In such a case, it will print a warning "High attention level predicted from matrix features" to the log.

The second issue, error propagation, is a bit trickier to trace. The most important source to consider for this is the multiplication of a vector with the constraint matrix, which gets stored in a factorized fashion. Hence, it makes sense to consider the condition number of the basis inverse matrix. Computing this can be expensive and is hence not done by default. You can activate it by setting the `MIPKAPPAFREQ` control to one. When setting this control, you will get a final statistic report that summarizes the condition numbers collected during search. Besides the percentage of stable, suspicious, unstable, and ill–posed basis inverse matrices, the Optimizer will report a quantity called the attention level after the solve. The attention level takes values between zero and one. It is equal to zero if all basis inverse matrices are stable, and one if all basis inverse matrices are ill–posed. The higher the attention level, the more likely are numerical errors. As a rule of thumb, matrices with an attention level larger than 0.1 should be investigated further. The attention level is available as an attribute: `ATTENTIONLEVEL`.

After having solved the root LP relaxation of a MIP solve, the Optimizer applies a Machine Learning model to predict the attention level of the current MIP solve. If the prediction is larger than 0.1, it will print a message to the log: "High attention level predicted from matrix features". The predicted attention level is available as an attribute: `PREDICTEDATTLEVEL`. Finally, if the Optimizer undergoes numerical failures during the optimization process, it will report these at the end of the solve. If you see dual, primal or barrier failures, or single inverts being reported, it might be worthwhile to try some of the methods described in the following sections.

5.6.2 Scaling

Scaling is a widely used preconditioning technique that aims at reducing the condition number of the constraint matrix, at reducing error propagation, and at reducing the number of LP iterations required to solve the problem. In Xpress, both columns and rows are scaled, however, only by powers of 2 to avoid round–off errors. By default, Xpress applies a machine learning algorithm to choose a scaling variant that is predicted to give the most stable performance. Although this prediction is correct in most of the cases, one can try the opposite setting, i.e., setting `SCALING` to 163 when autoscaling selected Curtis–Reid scaling and setting scaling to 16 when autoscaling selected standard scaling. Furthermore, disabling special handling of big-M rows and conduction scaling before presolving, represented by bits 6 and 9 of the SCALING control, is useful for some problems.

5.6.3 Solution Refinement

The Optimizer offers two methods of refining solutions, both are independent and complement each other. The first is called LP Refinement and aims at providing LP solutions of a higher precision, i.e., with more significant bits. It consists of two parts. Standard LP Refinement iteratively attempts to increase the accuracy of the solution until either both `FEASTOLTARGET` and `OPTIMALITYTOLTARGET` are satisfied, or accuracy cannot further be increased, or some effort limit is exhausted. It is applied by default both to LP solutions and to MIP solutions. Iterative refinement has the same goal, but uses more expensive, but also more promising measures of doing so, e.g., quad precision computing. If the postsolved LP solution is slightly infeasible, setting bits 5 and 6 of the `REFINEOPS` control aims at reducing those infeasibilities.

The second refinement scheme is called MIP Refinement and aims at providing MIP solutions which are truly integral and will not lead to infeasibilities when fixing integer variables in the original space. Note that both Iterative Refinement and MIP Refinement can lead to a slowdown of the solution process which is more considerable the more numerically challenging the matrix is.

5.6.4 Other Ways to Handle Numerical Issues

In addition to the methods named above, the Optimizer gives the user the possibility to change the numerical tolerances, such as `FEASTOL` and `MATRIXTOL`, but caution is advised here. Finally, if the numerical issues mainly come from the behavior of the simplex algorithm, setting `DUALSTRATEGY` to values 7 or 32 might help, or even using only barrier for solving LPs during a MIP solve, achieved by changing `DEFAULTALG` to 4.

In any case, it is best practice to reconsider the model. If you have very small and/or very large values in there — are those really necessary? Or could they be adapted to some significantly more stable value while still representing the same logic? Can you determine places where large values might cancel each other out and the residual is used for further computations? Have you tried using indicator instead of big-M formulations?

5.7 Common Causes of Confusion

It should be noted that most of the library routines described above and in chapter 8, which modify the matrix will not work on a presolved matrix. The only exception is inside a callback for a MIP solve, where cuts may be added or variable bounds tightened (using `XPRSchgbounds`). Any of these functions expect references to the presolved problem. If one tries to retrieve rows, columns, bounds or the number of these, such information will come from the presolved matrix and not the original. A few functions exist which are specifically designed to work with presolved and scaled matrices, although care should be exercised in using them. Examples of these include the commands `XPRSgetpresolvesol`, `XPRSgetpresolvebasis`, `XPRSgetscaledinfeas`, `XPRSloadpresolvebasis` and `XPRSloadpresolvedirs`.

5.8 Using the Callbacks

Console users are constantly provided with information on the standard output device by the Optimizer as it searches for a solution to the current problem. The same output is also available to library users if a log file has been set up using `XPRSsetlogfile`. However, whilst Console users can respond to this information as it is produced and allow it to influence their session, the same is not immediately true for library users, since their program must be written and compiled before the session is initiated. For such users, a more interactive alternative to the above forms of output is provided by the use of *callback functions*.

The library *callbacks* are a collection of functions which allow user–defined routines to be specified to the Optimizer. In this way, users may define their own routines which should be called at various stages during the optimization process, prompting the Optimizer to return to the user's program before continuing with the solution algorithm. Perhaps the three most general of the callback functions are those associated with the search for an LP solution. However, the vast majority of situations in which such routines might be called are associated with the global search, and will be addressed below.

5.8.1 Output Callbacks

Instead of catching the standard output from the Optimizer and saving it to a log file, the callback `XPRSaddcbmessage` allows the user to define a routine which should be called every time a text line is output by the Optimizer. Since this returns the status of each message output, the user's routine could test for error or warning messages and take appropriate action accordingly.

5.8.2 LP Callbacks

The functions `XPRSaddcblplog` and `XPRSaddcbbarlog` allow the user to respond after each iteration of either the simplex or barrier algorithms, respectively. The controls `LPLOG` and `BAROUTPUT` may additionally be set to reduce the frequency at which these routines should be called.

5.8.3 Global Search Callbacks

When a problem with global entities is to be optimized, a large number of sub problems, called *nodes*, must typically be solved as part of the global tree search. At various points in this process user–defined routines can be called, depending on the callback that is used to specify the routine to the Optimizer.

In a global tree search, the Optimizer starts by selecting an active node amongst all candidates (known as a *full backtrack*) and then proceed with solving it, which can lead to new descendent nodes being created. If there is a descendent node, the optimizer will by default select one of these next to solve and repeat this iterative descend while new descendent nodes are being created. This *dive* stops when it reaches a node that is found to be infeasible or cutoff, at which point the Optimizer will perform a *full backtrack* again and repeat the process with a new active node.

A routine may be called whenever a node is selected by the optimizer during a *full backtrack*, using `XPRSaddcbchgnode`. This will also allow a user to directly select the active node for the optimizer. Whenever a new node is created, a routine set by `XPRSaddcbnewnode` will be called, which can be used to record the identifier of the new node, e.g. for use with `XPRSaddcbchgnode`.

When the Optimizer solves a new node, it will first call any routine set by `XPRSaddcbprenode`, which can be used to e.g. tighten bounds on columns (with `XPRSchgbounds`) as part of a user node presolve. Afterwards, the LP relaxation of the node problem is solved to obtain a feasible solution and a best bound for the node. This might be followed by one or more rounds of cuts. If the node problem is found to be infeasible or cutoff during this process, a routine set by `XPRSaddcbinfnode` will be called. Otherwise, a routine set by `XPRSaddcboptnode` will be called to let the user know that the optimizer now has an optimal solution to the LP relaxation of the node problem. In this routine, the user is allowed to add cuts (see section 5.9) and tighten bounds to tighten the node problem, or apply branching objects (see `XPRS_bo_create`) to separate on the current node problem. If the user modifies the problem inside this *optnode* callback routine, the optimizer will automatically resolve the node LP and, if the LP is still feasible, call the *optnode* routine again.

If the LP relaxation solution to the node problem also satisfies all global entities and the user has not added any branching objects, i.e., if it is a MIP solution, the Optimizer will call a routine set by `XPRSaddcbpreintsol` *before* saving the new solution, and call a routine set by `XPRSaddcbintsol` *after* saving the solution. These two routines will also be called whenever a new MIP solution is found using one of the Optimizer heuristics.

Otherwise, if the node LP solution does not satisfy the global entities (or any user branching objects), the Optimizer will proceed with branching. After the optimizer has selected the candidate entity for branching, a routine set by `XPRSaddcbchgbranch` will be called, which also allows a user to change the selected candidate. If, during the candidate evaluation the optimizer discovers that e.g. bounds can be tightened, it will tighten the node problem and go back to resolving the node LP, followed by the callback routines explained above.

When the Optimizer finds a better MIP solution, it is possible that some of the nodes in the active nodes pool are cut off due to having an LP solution bound that is worse than the new cutoff value. For such nodes, a routine set by `XPRSaddcbnodecutoff` will be called and the node will be dropped from the active nodes pool.

The final global callback, `XPRSaddcbgloballog`, is more similar to the LP search callbacks, allowing a user's routine to be called whenever a line of the global log is printed. The frequency with which this occurs is set by the control `MIPLOG`.

5.9 Working with the Cut Manager

5.9.1 Cuts and the Cut Pool

Solving the LP relaxations during a global search is often made more efficient by supplying additional rows (constraints) to the matrix which reduce the size of the feasible region, whilst ensuring that it still contains an optimal integer solution. Such additional rows are called *cutting planes*, or *cuts*.

By default, cuts are automatically added to the matrix by the Optimizer during a global search to speed up the solution process. However, for advanced users, the Optimizer library provides greater freedom, allowing the possibility of choosing which cuts are to be added at particular nodes, or removing cuts entirely. The cutting planes themselves are held in a *cut pool*, which may be manipulated using library functions.

Cuts may be added directly to the matrix at a particular node, or may be stored in the cut pool first before subsequently being loaded into the matrix. It often makes little difference which of these two approaches is adopted, although as a general rule if cuts are cheap to generate, it may be preferable to add the cuts directly to the matrix and delete any redundant cuts after each sub–problem (node) has been optimized. Any cuts added to the matrix at a node and not deleted at that node will automatically be added to the cut pool. If you wish to save all the cuts that are generated, it is better to add the cuts to the cut pool first. Cuts can then be loaded into the matrix from the cut pool. This approach has the advantage that the cut pool routines can be used to identify duplicate cuts and save only the stronger cuts.

To help track the cuts that have been added to the matrix at different nodes, the cuts can be classified according to a user–defined *cut type*. The cut type can either be a number such as the node number or it can be a bit map. In the latter case each bit of the cut type may be used to indicate a property of the cut. For example, cuts could be classified as local cuts applicable at the current node and its descendants, or as global cuts applicable at all nodes. If the first bit of the cut type is set this could indicate a local cut and if the second bit is set this could indicate a global cut. Other bits of the cut type could then be used to signify other properties of the cuts. The advantage of using bit maps is that all cuts with a particular property can easily be selected, for example all local cuts.

5.9.2 Cut Management Routines

Cuts may be added directly into the matrix at the current node using `XPRSaddcuts`. Any cuts added to the matrix at a node will be automatically added to the cut pool and hence restored at descendant nodes unless specifically deleted at that node, using `XPRSdelcuts`. Cuts may be deleted from a parent node which have been automatically restored, as well as those added to the current node using `XPRSaddcuts`, or loaded from the cut pool using `XPRSloadcuts`.

It is recommended to delete only those cuts with basic slacks. Otherwise, the basis will no longer be valid and it may take many iterations to recover an optimal basis. If the second argument to `XPRSdelcuts` is set to `1`, this will ensure that cuts with non–basic slacks will not be deleted, even if the other controls specify that they should be. It is highly recommended that this is always set to `1`.

Cuts may be saved directly to the cut pool using the function `XPRSstorecuts`. Since cuts added to the cut pool are *not* automatically added to the matrix at the current node, any such cut must be explicitly loaded into the matrix using `XPRSloadcuts` before it can become active. If the third argument of `XPRSstorecuts` is set to `1`, the cut pool will be checked for duplicate cuts with a cut type identical to the cuts being added. If a duplicate cut is found, the new cut will only be added if its right hand side value makes the cut stronger. If the cut in the cut pool is weaker than the added cut, it will be removed unless it has already been applied to active nodes of the tree. If, instead, this argument is set to `2`, the same test is carried out on all cuts, ignoring the cut type. The routine `XPRSdelcpcuts` allows the user to remove cuts from the cut pool, unless they have already been applied to active nodes in the branch and bound tree.

A list of cuts in the cut pool may be obtained using the command `XPRSgetcpcuts`, whilst `XPRSgetcpcutlist` returns a list of their indices. A list of those cuts which are active at the current node is returned using `XPRSgetcutlist`.

5.9.3 User Cut Manager Routines

Users may also write their own cut manager routines to be called during the branch and bound search. Cuts can be added or removed on any node of the branch and bound search using a callback function set by the routine `XPRSaddcboptnode` (see section 5.8.3).

Further details of these functions may be found in chapter 8 within the functional reference which follows.

5.10 Solving Problems Using Multiple Threads

It is possible to use multiple processors when solving any type of problem with the Optimizer. On the more common processor types, such as those from Intel or AMD, the Optimizer will detect how many logical processors are available in the system and attempt to solve the problem in parallel using as many threads as possible. The number detected can be read through the `CORESDETECTED` integer attribute. It is also possible to adjust the number of threads to use by setting the integer parameter `THREADS`.

By default a problem will be solved deterministically, in the sense that the same solution path will be followed each time the problem is solved when given the same number of threads. For an LP this means that the number of iterations and the optimal, feasible solution returned will always be the same.

When solving a MIP deterministically, each node of the branch–and–bound tree will always be solved the same. Each node of the branch–and–bound tree can be identified by a unique number, available through the attribute `CURRENTNODE`. The tree will always have the same parent/child relationship in terms of these identifiers. A deterministic MIP solve will always find integer solutions on the same nodes and the attributes and solutions on a node will always be returned the same from one run to another. Since nodes will be solved in parallel the order in which nodes are solved can vary. There is an overhead in synchronizing the threads to make the parallel runs deterministic and it can be faster to run in non–deterministic mode. This can be done by setting the `DETERMINISTIC` control to 0.

For an LP problem (or the initial continuous relaxation of a MIP), there are several choices of parallelism. Both the barrier algorithm and the dual simplex algorithm support multiple threads. The number of threads to use can be set with `BARTHREADS` or `DUALTHREADS`, respectively. It is also possible to run some or all of primal simplex, dual simplex and the Barrier algorithm side–by–side in separate threads, known as a *concurrent* LP solve. This can be useful when none of the methods is the obvious choice. In this mode, the Optimizer will stop with the first algorithm to solve the problem. The number of threads for the concurrent LP solve can be set using `CONCURRENTTHREADS`. The algorithms to use for the concurrent solve can be specified by concatenating the required "d", "p", "n" and "b" flags when calling `XPRSlpoptimize` (`LPOPTIMIZE`) or `XPRSmipoptimize` (`MIPOPTIMIZE`); please refer to section 5.10.1 for more details.

When solving a MIP problem, the Optimizer will try to run the branch and bound tree search in parallel. Use the `MIPTHREADS` control to set the number of threads specifically for the tree search.

The operation of the optimizer for MIPs is fairly similar in serial and parallel mode. The MIP callbacks can still be used in parallel and callbacks are called when each MIP worker problem is created and destroyed. The mipthread callback (declared with `XPRSaddcbmipthread`) is called whenever a MIP worker problem is created and the callback declared with `XPRSaddcbdestroymt` is called whenever the worker problem is destroyed. Each worker problem has a unique ID which can be obtained from the `MIPTHREADID` integer attribute. When an executing thread solves a branch–and–bound node, it will also do so on a worker problem assigned to it. Note that a given worker problem can be assigned to different threads during its lifetime and the threads might differ from one run to another.

When the MIP callbacks are called they are MUTEX protected to allow non threadsafe user callbacks. If a significant amount of time is spent in the callbacks then it is worth turning off the automatic MUTEX protection by setting the `MUTEXCALLBACKS` control to 0. It this is done then the user must ensure that their callbacks are threadsafe.

On some problems it is also possible to obtain a speedup by using multiple threads for the MIP solve process between the initial LP relaxation solve and the branch and bound search. The default behavior here is for the Optimizer to use a single thread to create its rounds of cuts and to run its heuristic methods to obtain MIP solutions. Extra threads can be started, dedicated to running the heuristics only, by setting the `HEURTHREADS` control. By setting `HEURTHREADS` to a non–zero value, the heuristics will be run in separate threads, in parallel with cutting.

When a MIP solve is terminated early, due to e.g. a time or node limit, it is possible to select between two different termination behaviors. This has implications for the determinism of callbacks called near termination and how quickly the Optimizer stops. In the default behavior, when termination is detected, all work is immediately stopped and any partial node solves are discarded. It is therefore possible that some callbacks will have been called for nodes that are discarded at termination. Note that this termination method does not affect the final state the problem is left in after termination and that any integer solution for which the *preintsol* and *intsol* callbacks are called will never be dropped. By setting the control `MIPTERMINATIONMETHOD` to 1, the termination behavior will be changed such that partial work is never discarded. Instead, all worker threads will be allowed to complete their current work before the solve stops. This termination behavior might cause a longer delay between termination is detected and the Optimizer stops, but it will ensure that work is never dropped for any callbacks that have already been called.

5.10.1 The concurrent solver

The concurrent solve is activated either by passing multiple algorithm flags to `XPRSlpoptimize` (e.g. "pb" for running primal and the barrier) or by setting `CONCURRENTTHREADS` to a positive number. The order in which threads are allocated to the algorithms is not affected by the order of the flags provided.

If algorithm flags are specified, then concurrent will run the specified algorithms, if the setting of `CONCURRENTTHREADS` allows for a sufficient number of threads. When no flags are specified, the automatic order of selecting algorithms starts with dual, followed by barrier and then primal. The network solver is only used if specified by flags.

`CONCURRENTTHREADS` represents the total target number of threads that can be used by concurrent. The optimizer will then first start dual then barrier (if `CONCURRENTTHREADS` >1) followed by primal (if `CONCURRENTTHREADS` >2). Any remaining threads will be allocated to parallel barrier.

If manual algorithm selection has been made using algorithm flags, then `CONCURRENTTHREADS` will limit the number of algorithms started (if smaller than the number of algorithms provided), in which case the number of algorithms started will be the first `CONCURRENTTHREADS` in the dual → barrier →

primal $\rightarrow$ network order.

Once an algorithm is started, the direct thread controls `BARTHREADS` and `DUALTHREADS` are respected. Note that due to the latter controls the total number of theads may exceed `CONCURRENTTHREADS`.

In case a single algorithm is started and relevant controls are on automatic, the value of the `THREADS` control is used.

If multiple algorithms have been started and `CONCURRENTTHREADS` is on automatic, then `THREADS` will be used as the overall number of threads used in the concurrent (unless overwritten by the relevant algorithm specific control on a per–algorithm basis).

5.11 Solving Large Models (the 64 bit Functions)

The size of the models that can be loaded into the optimizer using the standard optimizer functions is limited by the largest number that can be held in a 32–bit integer. This means that it is not possible to load any problem with more than 2147483648 matrix elements with the standard optimizer functions. On 64–bit machines, it is possible to use the optimizer 64–bit functions to load problems with a larger number of elements (these functions have 64 appended to the standard optimizer function names). For example, it is possible to load a problem with a large number of elements with the `XPRSloadlp64` function. The only difference between `XPRSloadlp64` and `XPRSloadlp` is that the `mstart` array containing the starting points of the elements in each column that is passed to `XPRSloadlp64` is a pointer to an array of 64–bit integers. Typically, the declaration and allocation of space for the 64–bit `mstart` array would be as follows:

```
XPRSint64 *mstart = malloc( ncol * sizeof( *mstart) );
```

The starting points of the elements in `mstart` can then exceed the largest 32–bit integer.

Wherever there is a need to pass a large array to an optimizer subroutine there is a corresponding 64–bit function. Once a large model has been loaded into the optimizer then all the standard optimizer functions (such as `XPRSlpoptimize`) can be used.

Note that although the 64–bit functions allow very large models to be loaded, there is no guarantee that such large problems can be solved in a reasonable time. Also if the machine doesn't have sufficient memory, the matrix will be constantly swapped in and out of memory which can slow the solution process considerably.

5.12 Using the Tuner

For a given optimization problem, setting suitable control parameters frequently results in improved performance such as solution time reduction. The Xpress Optimizer built–in tuner can help a user to identify such set of control settings that allows the Xpress Optimizer or Xpress SLP to solve problems faster than by using defaults.

5.12.1 Basic Usage

With a loaded problem, the tuner can be started by simply calling `TUNE` from the console, or `XPRStune` from a user application. The tuner will then search for better control settings from a list of controls (called the tuner method). To achieve this, the tuner will solve the problem with its default baseline control settings and then solve the problem multiple times with each individual control and certain combinations of these controls.

As the tuner works by solving a problem mutiple times, it is important and recommended to set time limits. Setting `MAXTIME` will limit the effort spent on each individual solve and setting `TUNERMAXTIME` will limit the overall effort of the tuner.

The tuner works with LP and MIP problems. It automatically determines the problem type by examining the characteristics of the current problem. It is possible to tune a MIP problem as an LP or vice versa by passing the flag `l` or `g` to `XPRStune` or `TUNE`.

The tuner can also work with SLP and MISLP problems when Xpress Nonlinear is available. Note that for SLP or MISLP problems, the time limit is set with `XSLP_MAXTIME`.

5.12.2 The Tuner Method

A tuner method consists of a list of controls for the tuner to try with. It is possible to run the tuner with different pre–defined lists of controls, so–called factory tuner methods, or with a user–defined list of controls. When using the tuner, it will automatically choose a default factory tuner method according to the problem type. A non–default factory tuner method can be selected by setting the `TUNERMETHOD` control. There are several choices available for factory tuner methods, among them:

- A simple MIP method, which only features a few controls and can be used in situations where tuning with the default method would take too long, e.g., because the instance to be tuned takes a long time for each individual solve
- A comprehensive MIP method, which features a larger list of controls (and control settings) and can be used when individual instance solves are relatively fast or the default method could not reveal a satisfying improvement
- A root–focus method, which only considers controls that affect the root node processing of the MIP solve. It can either be used when root and tree behavior should be tuned independently, in a two stage process, or when it is evident that improvements have to come from root node processing. When tuning with a root–focus, it might make sense to choose minimizing the primal dual integral as a tuner target.
- A tree–focus method, which only considers controls that affect the tree search behavior of the MIP solve. It can either be used when root and tree behavior should be tuned independently, in a two stage process, or when it is evident that improvements have to come from the tree search, e.g., because the dual bound needs better branching.
- A method for tuning primal heuristics, which should be used when finding a better MIP solutions is the only focus and improving the best bound can be neglected. In this case, it might make sense to choose improvement of the primal bound also as a tuner target.

Please also refer to the documentation of the `TUNERMETHOD` control.

A tuner method can be written out using `XPRStunerwritemethod`. This function will create a file in XTM format, that is effectively a list of Xpress Optimizer controls, each with a set of possible settings to try in tuning. When writing out one of the factory methods, it is recommended to first select the tuner method by setting `TUNERMETHOD`, or to load a targeting problem, so that the tuner can write out suitable tuner methods for the respective problem types.

Users can provide their own method to the tuner by setting up an XTM file (or editing one that has been written out). This can be read into the tuner with `XPRStunerreadmethod`.

An alternate way to load a user–defined tuner method is to set the `TUNERMETHODFILE` control to the file name. This will only work when no tuner method has been loaded by explicitly calling `XPRStunerreadmethod`. If a user–defined method is successfully loaded, the tuner will use it and not load any factory tuner method.

Please refer to Appendix A.9 for the format of tuner method file.

5.12.3 The Tuner Output

While the tuner examines various control settings, it prints a progress report to the console. At the same time, it writes out the result and individual logs to the file system.

On the console, the tuner will print a one–line summary for each finished run. When a new better control setting is identified, it will be highlighted with an asterisk (*) at the begining of its log line, and followed by details of the control setting and its log file name. The console progress logging can be switched off by disabling the `OUTPUTLOG` control. Please refer to Appendix A.13 for a more detailed description of the tuner logging.

In the background, the tuner will output the result and individual logs to the file system. By default, all the output files will be stored in the directory `tuneroutput/probname/`. The root folder path can be changed by setting the `TUNEROUTPUTPATH` control. This is the central folder in which all subfolders for the results and logs of different problems will be stored. The subfolders themselves are automatically named using the current problem name. They can be manually given a different name by setting the `TUNERSESSIONNAME` control. The subfolder contains one result file in XML format, and many log files, one for each evaluated control setting. The XML result file consists of the control settings, solution results and pointers to the log files of all finished tuner runs.

The file output can be turned off completely by disabling the `TUNEROUTPUT` control.

5.12.4 The Tuner Target

A tuner target defines how to compare two finished runs with different control settings.

A common usage of the tuner is to pursue a solution time reduction, where two runs will be compared by their solution time, the faster one is considered the better. However, when both of the runs time out, it will be more meaningful to compare other attributes of the two runs, for example the final gap or the best integer solution for MIP problems.

The tuner will choose a default tuner target according to problem types. For instance, comparing the time firstly and then the gap is the default tuner target for MIP problems. A user can select a different target by setting the `TUNERTARGET` control. Please refer to the documentation of `TUNERTARGET` for a list of supported tuner targets.

5.12.5 Restarting the Tuner

When tuning the same problem again, the tuner will attempt to pick up results from previous tuner runs so that it can avoid testing with the same control settings again. For this, it checks whether an XML result file is available in the directory `tuneroutput/probname/`, see Section 5.12.3. Reusing of the history results even works when a user changes the baseline settings or uses a different tuner method. In this case, the tuner will only pick up history results which match the new control combinations. By default, when a new control setting is evaluated, the result will be appended to the existing result file from the previous tuner session.

This feature of reusing and appending to previous results can be switched off by setting the `TUNERHISTORY` control. This control has the default value `2`, which allows both, reusing and appending. Setting it to `1` will switch off reusing of the results, while still allowing to append new result to the XML result file. Setting it to `0` will switch off appending as well; consequently, the old result file will be overwritten. Note that all log files from previous tuner session will always be kept even if they run with identical settings. This is realized by having a time stamp and a unique number in the file name. Log files can only be removed manually.

5.12.6 Tuner with Multiple Threads

The tuner can work in parallel, i.e., it can run several evaluations of different control settings simultaneously. When setting TUNERTHREADS larger than 1, the tuner will start in parallel mode with the given number of threads. Setting the tuner threads won't affect the number of threads used by each individual run. However, it is natural that, when solving different control settings in parallel, each of the runs may slow down.

When using the parallel tuner, it is worth considering to set the THREADS control as well; ideally such that the product of THREADS and TUNERTHREADS is at most the number of system threads.

5.12.7 Tuner with Problem Permutations

For a certain problem, there may exist several "lucky" controls, that show a better performance by coincidence and not due to structural reasons. Such lucky controls will typically not work with other problems of the same type, or when a user modifies the problem slightly or updates Xpress. They can be thought of as false positives of tuning.

To address this issue, the tuner can exploit a phenomenon known as performance variability and solve the problem with multiple random permutations. When setting TUNERPERMUTE to a positive number, for each control setting, the tuner will solve the original problem and the corresponding number of permuted problems and finally aggregate their results as one. Generally, tuner results with permutations are expected to be more stable.

5.12.8 Tuning a Set of Problems

The tuner can tune a set of problems to search for an overall best control setting for all the problems in the set. Tuning a problem set can be started from the optimizer console with the command

`tune probset problem.set`,

where the problem.set is a plain text file which contains a list of problem files in MPS or LP format.

The tuner starts by checking all the problems defined in the problem set file. It will read in each problem to find out its type (one of LP, MIP, SLP and MISLP) and optimization direction. When there are mixed problem types, the tuner will quit with a warning message. The tuner can work with mixed optimization directions and it will treat the whole problem set as a minimization problem. For a given problem set, it is possible to force the tuner to tune the problem set as LP or MIP problems with the command

`tune lpset problem.set` or `tune mipset problem.set`

respectively.

For a problem set, the tuner works by solving each individual problem in the set for each specific combination of control settings separately. When all the problems in the set are solved for a specific control setting, the tuner combines the individual problem results into a consolidated one and reports it on the console. During the solve, for each problem in the set, the tuner will output its result and log files to a path defined by TUNEROUTPUTPATH/PROBLEMNAME. For the main problem set, the tuner will write the consolidated results to the main output path, together with a concatenated copy of all the individual problem logs.

When tuning a problem set again, the tuner can pick up the result of existing runs for the main problem set and for each separate problem in the set as well. If the full problem set can be recovered from the existing tuning records, the tuner will omit solving them as usual. Otherwise, the tuner will go through all the problems in the set. For each problem in the set, the tuner will also check whether it is possible to pick up an existing result with the specific control setting and omit solving for existing ones when possible.

5.12.9 Advanced Topics

Besides explicitly calling TUNE or XPRStune, the tuner can also be started by enabling the TUNERMODE control. When enabling this control (setting to 1), all the optimization such as XPRSmipoptimize or XPRSlpoptimize will be carried out as a tuned optimization. The Optimizer will first use the tuner to find the best setting and then apply the best setting to solve the problem. On the other hand, a user can disable this control (setting to 0) to always disable the tuner, such that a call to XPRStune will have no effect. This TUNERMODE has a default value of -1, which won't affect the behaviour of any of the above mentioned functions.

When using the tuner from a user application with callbacks, the callbacks will also be passed on to each individual runs. A user needs to keep in mind that these callbacks may be called mutiple times from the tuner, as the tuner will solve the problem mutiples times. Moreover, when using the parallel tuner, it is the user's responsibility to ensure that callbacks are thread–safe.

Though the tuner is built–in with the Xpress Optimizer, it can tune nonlinear problems when Xpress Nonlinear is available. Currently, parallel tuning and permutations will be disabled in this case.

5.13 Remote Solving with Xpress Insight Compute Interface

The Xpress Optimizer libraries can be configured to outsource optimization computation to a remote Insight server that supports the Compute Interface. Software applications which depend on the Optimizer libraries for optimization computation therefore inherit the ability to transparently send jobs to Insight. This includes the Xpress applications (Optimizer Console, Mosel, Workbench).

When a solve is started, the Optimizer library directs any operations that can be solved remotely to the remote server. Some features such as callbacks, multi-start, and the solution enumerator have restrictions applied which are documented here.

To integrate Xpress with an Insight Compute Server you must provide some configuration. Please see Chapter 3 of the Xpress Insight Compute Interface guide here:
(https://www.fico.com/fico-xpress-optimization/docs/latest/insight5/compute/)

The single solve operations XPRSlpoptimize, XPRSmipoptimize, XSLPnlpoptimize, XSLPminim, XSLPmaxim, XPRSiis are supported. Calls to XPRSrepairinfeas and XPRStune which generate multiple problem solves are also supported and each solve will be outsourced to the remote Insight server. The number of parallel solves in the tuner is driven by the TUNERTHREADS control.

The Xpress Insight execution service and the local client application must be using the same major version of Xpress. Remote solves by Insight are supported by Xpress v8.10 and higher. Note: If you get a solve path difference, update the version of Xpress to match the version on the server with that on the client machine and check hardware controls, in particular threading controls. Solves will use the default execution service unless you specify one using the COMPUTEEXECSERVICE control or the configuration file as described in Appendix A.14.3.

Compute solves do not support the continuation of solves once they are interrupted, nor the multistart nonlinear algorithm.

A remote solve can be terminated by calling XPRSinterrupt. When called from the supported callbacks - with the exception of the message callback - this will stop the optimizer the same way as for a local solve. Otherwise, calling XPRSinterrupt outside of the supported callbacks will terminate the solve at the earliest opportunity, and no results will be generated.

The remote solve is resilient to a temporary loss of connection between client and server. Xpress will try to reconnect for a period of time and a message will appear in the run log if this is successful, or the solve will terminate with an error if it is not. If the connection between client and server is established when the connection between server and the executing worker is lost then the solve will be restarted to maintain determinism, and a computerestart callback will be fired to notify the calling application. Any work done by the disconnected remote worker, including any integer solution callbacks already fired, will be repeated.

Support for the following features are disabled when solving remotely and calling the related API methods will cause a runtime error:

- multiple solution pools,
- solution enumeration,
- callbacks not listed as supported above.

5.13.1 Authentication

Please refer to the Insight Compute Interface guide Chapter 2 and 3 for details of connecting Xpress to a remote Insight server.
(https://www.fico.com/fico-xpress-optimization/docs/latest/insight5/compute/)

5.13.2 Callbacks

Callbacks are supported. When submitting a job to a remote machine, these callbacks are restricted to the message, barlog, globallog, lplog and cutlog, gapnotify, and intsol callbacks. Attempting to set any other callbacks will cause a runtime error. Controls can be changed in the usual way in all the supported callbacks with the exception of the message callback. Note: when solving remotely, the value of control XSLP_AUTOSAVE is always ignored.

Within the supported callbacks, calls can be made to functions that retrieve attributes and setting control values. Within the intsol callback, calls to XPRSgetmipsol and XPRSgetmipsolvalue are permitted. Calling any other API function will cause a runtime error, including any XSLP and BCL API calls. Note: While calling XPRSgetlpsol is also permitted in the intsol callback, it will return the same solution as XPRSgetmipsol, both being the solution associated to the intsol callback. This is different to non–remote solves where getmipsol would return the overall best solution instead. It is therefore safe to keep using getlpsol in applications utilizing compute solves.

Any job that features callbacks which return hardware related attributes will use values from the remote server. For example, XPRS_CORESDETECTED will reflect the hardware on which the problem is being solved, not the hardware of the local client.

5.13.3 Licensing

When an Xpress application or an application embedding the Optimizer library is started with remote solving configured, the local license check is omitted and no local license is required to execute the application.

When a solve is started, the Optimizer instance will direct any operations that can be solved remotely to the remote server. This will also be the case if additional Optimizer instances are initiated as separate threads of the same process.

5.13.4 Advanced Configuration

There are some advanced settings that can be set using the Remote Solving Configuration file; this is described in section A.14.

Chapter 6

Infeasibility, Unboundedness and Instability

All users will, generally, encounter occasions in which an instance of the model they are developing is solved and found to be *infeasible* or *unbounded*. An infeasible problem is a problem that has no solution while an unbounded problem is one where the constraints do not restrict the objective function and the objective goes to infinity. Both situations often arise due to errors or shortcomings in the formulation or in the data defining the problem. When such a result is found it is typically not clear what it is about the formulation or the data that has caused the problem.

Problem instability arises when the coefficient values of the problem are such that the optimization algorithms find it difficult to converge to a solution. This is typically because of large ratios between the largest and smallest coefficients in the rows or columns and the handling of the range of numerical values in the algorithm is causing floating point accuracy issues. Problem instability generally manifests in either long run times or spurious infeasibilities.

It is often difficult to deal with these issues since it is often difficult to diagnose the cause of the problems. In this chapter we discuss the various approaches and tools provided by the Optimizer for handling these issues.

6.1 Infeasibility

A problem is said to be *infeasible* if no solution exists which satisfies all the constraints. The FICO Xpress Optimizer provides functionality for diagnosing the cause of infeasibility in the user's problem.

Before we discuss the infeasibility diagnostics of the Optimizer we will define some types of infeasibility in terms of the type of problem it relates to and how the infeasibility is detected by the Optimizer.

We will consider two basic types of infeasibility. The first we will call continuous infeasibility and the second discrete or integer infeasibility. Continuous infeasibility is where a non–MIP problem is infeasible. In this case the feasible region defined by the intersecting constraints is empty. Discrete or integer infeasibility is where a MIP problem has a feasible relaxation (a relaxation of a MIP is the problem we get when we drop the discreteness requirement on the variables) but the feasible region of the relaxation contains no solution that satisfies the discreteness requirement.

Either type of infeasibility may be detected at the presolve phase of an optimization run. Presolve is the analysis and processing of the problem before the problem is run through the optimization algorithm. If continuous infeasibility is not detected in presolve then the optimization algorithm will detect the infeasibility. If integer infeasibility is not detected in presolve, a branch and bound search will be necessary to detect the infeasibility. These scenarios are discussed in the following sections.

6.1.1 Diagnosis in Presolve

The presolve processing, if activated (see section 5.3), provides a variety of checks for infeasibility. When presolve detects infeasibility, it is possible to "trace" back the implications that determined an inconsistency and identify a particular cause. This diagnosis is carried out whenever the control parameter `TRACE` is set to `1` before the optimization routine `XPRSlpoptimize` (`LPOPTIMIZE`) is called. In such a situation, the cause of the infeasibility is then reported as part of the output from the optimization routine.

6.1.2 Diagnosis using Primal Simplex

The trace presolve functionality is typically useful when the infeasibility is simple, such that the sequence of bound implications that explains the infeasibility is short. If, however, this sequence is long or there are a number of sequences on different sets of variables, it might be useful to try forcing presolve to continue processing and then solve the problem using the primal simplex to get the, so called, 'phase 1' solution. To force presolve to continue even when an infeasibility is discovered the user can set the control `PRESOLVE` to `--1`. The 'phase 1' solution is useful because the sum of infeasibilities is minimized in the solution and the resulting set of violated constraints and violated variable bounds provides a clear picture of what aspect of the model is causing the infeasibility.

6.1.3 Irreducible Infeasible Sets

A general technique to analyze infeasibility is to find a small subset of the matrix that is itself infeasible. The Optimizer does this by finding *irreducible infeasible sets* (IISs). An IIS is a minimal set of constraints and variable bounds which is infeasible, but becomes feasible if any constraint or bound in it is removed.

A model may have several infeasibilities. Repairing a single IIS may not make the model feasible, for which reason the Optimizer can attempt to find an IIS for each of the infeasibilities in a model. The IISs found by the optimizer are independent in the sense that each constraint and variable bound may only be present in at most one IIS. In some problems there are overlapping IISs. The number of all IISs present in a problem may be exponential, and no attempt is made to enumerate all. If the infeasibility can be represented by several different IISs the Optimizer will attempt to find the IIS with the smallest number of constraints in order to make the infeasibility easier to diagnose (the Optimizer tries to minimize the number of constraints involved, even if it means that the IIS will contain more bounds).

Using the library functions IISs can be generated iteratively using the `XPRSiisfirst` and `XPRSiisnext` functions. All (a maximal set of independent) IISs can also be obtained with the `XPRSiisall` function. Note that if the problem is modified during the iterative search for IISs, the process has to be started from scratch. After a set of IISs is identified, the information contained by any one of the IISs (size, constraint and bound lists, duals, etc.) may be retrieved with the function `XPRSgetiisdata`. A summary on the generated IISs is provided by function `XPRSiisstatus`, while it is possible to save the IIS data or the IIS subproblem directly into a file in MPS or LP format using `XPRSiiswrite`. The information about the IISs is available while the problem remains unchanged. The information about an IIS may be obtained at any time after it has been generated. Function `XPRSiisclear` clears the information already stored about IISs.

On the console, all the IIS functions are available by passing different flags to the `IIS` console command. A single IIS may be found with the command `IIS`. If further IISs are required (e.g., if trying to find the smallest one) the `IIS --n` command may be used to generate subsequent IISs, or the `IIS --a` to generate all independent IISs, until no further independent IIS exists. These functions display the constraints and bounds that are identified to be in an IIS as they are found. If further information is required, the `IIS --p` *num* command may be used to retrieve all the data for a given IIS, or the `IISw` and `IISe` functions to create an LP/MPS or CSV containing the IIS subproblem or the additional information about the IIS in a file.

Once an IIS has been found it is useful to know if dropping a single constraint or bound in the IIS will completely remove the infeasibility represented by the IIS, thus an attempt is made to identify a subset of the IIS called a `sub--IIS isolation`. A `sub--IIS isolation` is a special constraint or bound in an IIS. Removing an IIS isolation constraint or bound will remove all infeasibilities in the IIS without increasing the infeasibilities outside the IIS, that is, in any other independent IISs.

The `IIS isolations` thus indicate the likely cause of each independent infeasibility and give an indication of which constraint or bound to drop or modify. This procedure is computationally expensive, and is carried out separately by function `XPRSiisisolations` (`IIS--i`) for an already identified IIS. It is not always possible to find `IIS isolations`.

After an optimal but infeasible first phase primal simplex, it is possible to identify a subproblem containing all the infeasibilities (corresponding to the given basis) to reduce the IIS work–problem dramatically. Rows with zero duals (thus with slack variables having zero reduced cost) and columns that have zero reduced costs may be excluded from the search for IISs. Moreover, for rows and columns with nonzero costs, the sign of the cost is used to relax equality rows either to less then or greater than equal rows, and to drop either possible upper or lower bounds on variables. This process is referred to as sensitivity filter for IISs.

The identification of an IIS, especially if the isolations search is also performed, may take a very long time. For this reason, using the sensitivity filter for IISs, it is possible to find only an approximation of the IISs, which typically contains all the IISs (and may contain several rows and bounds that are not part of any IIS). This approximation is a sub–problem identified at the beginning of the search for IISs, and is referred to as the initial infeasible sub–problem. Its size is typically crucial to the running time of the IIS procedure. This sub–problem is accessible by setting the input parameters of `XPRSiisfirst` or by calling (`IIS --f`) on the console. Note that the IIS approximation and the IISs generated so far are always available.

The `XPRSgetiisdata` function also returns dual multipliers. These multipliers are associated with Farkas' lemma of linear optimization. Farkas' lemma in its simplest form states that if $Ax = b, x \geq 0$ has no solution, then there exists a y for which $y^T A \geq 0$ and $y^T b < 0$. In other words, if the constraints and bounds are contradictory, then an inequality of form $d^T x < 0$ may be derived, where d is a constant vector of nonnegative values. The vector y, i.e., the multipliers with which the constraints and bounds have to be combined to get the contradiction is called dual multipliers. For each IIS identified, these multipliers are also provided. For an IIS all the dual multipliers should be nonzero.

6.1.4 The Infeasibility Repair Utility

In some cases, identifying the cause of infeasibility, even if the search is based on IISs may prove very demanding and time consuming. In such cases, a solution that violates the constraints and bounds minimally can greatly assist modeling. This functionality is provided by the `XPRSrepairweightedinfeas` function.

Based on preferences provided by the user, the Optimizer relaxes the constraints and bounds in the problem by introducing penalized deviation variables associated with selected rows and columns. Then a weighted sum of these variables (sometimes referred to as infeasibility breakers) is minimized, resulting in a solution that violates the constraints and bounds minimally regarding the provided preferences. The preference associated with a constraint or bound reflects the modeler's will to relax the corresponding right–hand–side or bound. The higher the preference, the more willing the modeler is to relax (the penalty value associated is the reciprocal of the preference). A zero preference reflects that the constraint or bound cannot be relaxed. It is the responsibility of the modeler to provide preferences that yield a feasible relaxed problem. Note, that if all preferences are nonzero, the relaxed problem is always feasible (with the exception of problems containing binary or semi–continuous variables, since because of their special associated modeling properties, such variables are not relaxed).

Note, that this utility does not repair the infeasibility of the original model, but based on the preferences provided by the user, it introduces extra freedom into it to make it feasible, and minimizes the utilization of the added freedom.

The magnitude of the preferences does not affect the quality of the resulting solution, and only the ratios of the individual preferences determine the resulting solution. If a single penalty value is used for each constraint and bound group (less than and greater than or equal constraints, as well as lower and upper bounds are treated separately) the `XPRSrepairinfeas` (`REPAIRINFEAS`) function may be used, which provides a simplified interface to `XPRSrepairweightedinfeas`.

Using the new variables introduced, it is possible to warm start the primal simplex algorithm with a basic solution. However, based on the value of the control `KEEPBASIS`, the function may modify the actual basis to produce a warm start basis for the solution process. An infeasible, but first phase optimal primal solution typically speeds up the solution of the relaxed problem.

Once the optimal solution to the relaxed problem is identified (and is automatically projected back to the original problem space), it may be used by the modeler to modify the problem in order to become feasible. However, it may be of interest to know what the optimal objective value will be if the original problem is relaxed according to the solution found be the infeasibility repair function.

In order to provide such information, the infeasibility repair tool may carry out a second phase, in which the weighted violation of the constraints and bounds are restricted to be no greater than the optimum of the first phase in the infeasibility repair function, and the original objective function is minimized or maximized.

It is possible to slightly relax the restriction on the weighted violation of the constraints and bounds in the second phase by setting the value of the parameter `delta` in `XPRSrepairweightedinfeas`, or using the `--delta` option with the Console Optimizer command `REPAIRINFEAS`. If the minimal weighted violation in the first phase is `p`, a nonzero `delta` would relax the restriction on the weighted violations to be less or equal than `(1+delta)p`. While such a relaxation allows considering the effect of the original objective function in more detail, on some problems the trade–off between increasing `delta` to improve the objective can be very large, and the modeler is advised to carefully analyze the effect of the extra violations of the constraints and bounds to the underlying model.

Note, that it is possible that an infeasible problem becomes unbounded in the second phase of the infeasibility repair function. In such cases, the cause of the problem being unbounded is likely to be independent from the cause of its infeasibility.

When not all constraints and bounds are relaxed it is possible for the relaxed problem to remain infeasible. In such cases it is possible to run the IIS tool on the relaxed problem, which can be used to identify why it is still infeasible.

It is also possible to limit the amount of relaxation allowed on a per constraint side or bound by using `XPRSrepairweightedinfeasbounds`.

It can sometimes be desired to achieve an even distribution of relaxation values. This can be achieved by using quadratic penalties on the added relaxation variables, and is indicated to the optimizer by specifying a negative preference value for the constraint or bound on which a quadratic penalty should be added.

6.1.5 Integer Infeasibility

In rare cases a MIP problem can be found to be infeasible although its LP relaxation was found to be feasible. In such circumstances the feasible region for the LP relaxation, while nontrivial, contains no solutions which satisfy the various integrality constraints. These are perhaps the worst kind of infeasibilities as it can be hard to determine the cause. In such cases it is recommended that the user try to introduce some flexibility into the problem by adding slack variables to all of the constraints each with some moderate penalty cost. With the solution to this problem the user should be able to identify, from the non–zero slack variables, where the problem is being overly restricted and with this decide how to modify the formulation and/or the data to avoid the problem.

6.2 Unboundedness

A problem is said to be *unbounded* if the objective function may be improved indefinitely without violating the constraints and bounds. This can happen if a problem is being solved with the wrong optimization sense, e.g., a maximization problem is being minimized. However, when a problem is unbounded and the problem is being solved with the correct optimization sense then this indicates a problem in the formulation of the model or the data. Typically, the problem is caused by missing constraints or the wrong signs on the coefficients. Note that unboundedness is often diagnosed by presolve.

6.3 Instability

6.3.1 Scaling

When developing a model and the definition of its input data users often produce problems that contain constraints and/or columns with large ratios in the absolute values of the largest and smallest coefficients. For example:

maximize:	$10^6x + 7y$	=	z
subject to:	$10^6x + 0.1y$	$\leq$	100
	$10^7x + 8y$	$\leq$	500
	$10^{12}x + 10^6y$	$\leq$	$50*10^6$

Here the objective coefficients, constraint coefficients, and right–hand side values range between 0.1 and 10^{12}. We say that the model is *badly scaled*.

During the optimization process, the Optimizer must perform many calculations involving subtraction and division of quantities derived from the constraints and the objective function. When these calculations are carried out with values differing greatly in magnitude, the finite precision of computer arithmetic and the fixed tolerances employed by FICO Xpress result in a build up of rounding errors to a point where the Optimizer can no longer reliably find the optimal solution.

To minimize undesirable effects, when formulating your problem try to choose units (or equivalently scale your problem) so that objective coefficients and matrix elements do not range by more than 10^6, and the right–hand side and non–infinite bound values do not exceed 10^6. One common problem is the use of large finite bound values to represent infinite bounds (i.e., no bounds) — if you have to enter explicit infinite bounds, make sure you use values greater than 10^{20} which will be interpreted as infinity by the Optimizer. Avoid having large objective values that have a small relative difference — this makes it hard for the dual simplex algorithm to solve the problem. Similarly, avoid having large right–hand side or bound values that are close together, but not identical.

In the above example, both the coefficient for x and the last constraint might be better scaled. Issues arising from the first may be overcome by *column scaling*, effectively a change of coordinates, with the replacement of 10^6x by some new variable. Those from the second may be overcome by *row scaling*. If we set $x = 10^6x'$ and scale the last row by 10^{-6}, the example becomes the much better scaled problem:

maximize:	$x' + 7y$	=	z
subject to:	$x' + 0.1y$	$\leq$	100
	$10x' + 8y$	$\leq$	500
	$x' + y$	$\leq$	50

FICO Xpress also incorporates a number of automatic scaling options to improve the scaling of the matrix. However, the general techniques described below cannot replace attention to the choice of units specific to your problem. The best option is to scale your problem following the advice above, and use the automatic scaling provided by the Optimizer.

The form of scaling provided by the Optimizer depends on the setting of the bits of the control parameter SCALING. To get a particular form of scaling, set SCALING to the sum of the values corresponding to the scaling required. For instance, to get row scaling, column scaling and then row scaling again, set SCALING to 1+2+4=7. The scaling processing is applied after presolve and before the optimization algorithm. The most important of the defined bits are given in the following table. For a full list, refer to SCALING in Chapter 9

Bit	Value	Type of Scaling
0	1	Row scaling.
1	2	Column scaling.
2	4	Row scaling again.
3	8	Maximin.
4	16	Curtis–Reid.
5	32	0– scale by geometric mean; 1– scale by maximum element (not applicable if maximin or Curtis–Reid is specified).
7	128	Objective function scaling.
8	256	Exclude the quadratic part of constraint when calculating scaling factors.
9	512	Scale the problem before presolve is applied.

If scaling is not required, SCALING should be set to 0.

If the user wants to get quick results when attempting to solve a badly scaled problem it may be useful to try running customized scaling on a problem before calling the optimization algorithm. To run the scaling process on a problem the user can call the routine XPRSscale(SCALE). The SCALING control determines how the scaling will be applied.

If the user is applying customized scaling to their problem and they are subsequently modifying the problem, it is important to note that the addition of new elements in the matrix can cause the problem to become badly scaled again. This can be avoided by reapplying their scaling strategy after completing their modifications to the matrix.

Finally, note that the scaling operations are determined by the matrix elements only. The objective coefficients, right hand side values and bound values do not influence the scaling. Only continuous variables (i.e., their bounds and coefficients) and constraints (i.e., their right–hand sides and coefficients) are scaled. Discrete entities such as integer variables are not scaled so the user should choose carefully the scaling of these variables.

6.3.2 Accuracy

The accuracy of the computed variable values and objective function value is affected in general by the various tolerances used in the Optimizer. Of particular relevance to MIP problems are the accuracy and cut off controls. The MIPRELCUTOFF control has a non–zero default value, which will prevent solutions very close but better than a known solution being found. This control can of course be set to zero if required.

When the LP solver stops at an optimal solution, the scaled constraints will be violated by no more than FEASTOL and the variables will be within FEASTOL of their scaled bounds. However once the constraints and variables have been unscaled the constraint and variable bound violation can increase to more than FEASTOL. If this happens then it indicates the problem is badly scaled. Reducing FEASTOL can help however this can cause the LP solve to be unstable and reduce solution performance.

However, for all problems it is probably ambitious to expect a level of accuracy in the objective of more than 1 in 1,000,000. Bear in mind that the default feasibility and optimality tolerances are 10^{-6}. It is often not practially possible to compute the solution values and reduced costs from a basis, to an accuracy better than 10^{-8} anyway, particularly for large models. It depends on the condition number of the basis matrix and the size of the right—hand side and cost coefficients. Under reasonable assumptions, an upper bound for the computed variable value accuracy is $4xKx\|RHS\|/10^{16}$, where $\|RHS\|$ denotes the L–infinity norm of the right–hand side and K is the basis condition number. The basis condition number can be found using the `XPRSbasiscondition` (`BASISCONDITION`) function.

You should also bear in mind that the matrix is scaled, which would normally have the effect of increasing the apparent feasibility tolerance.

CHAPTER 7

Goal Programming

7.1 Overview

Note that the Goal Programming functionality of the Optimizer will be dropped in a future release. This functionality will be replaced by an example program, available with this release (see `goal_example.c` in the `examples/optimizer/c` folder of the installation), that provides the same functionality as the original library function `XPRSgoal` (`GOAL`) but is implemented using the Optimizer library interface.

Goal programming is an extension of linear programming in which targets are specified for a set of constraints. In goal programming there are two basic models: the *pre–emptive* (lexicographic) model and the *Archimedean* model. In the pre–emptive model, goals are ordered according to priorities. The goals at a certain priority level are considered to be infinitely more important than the goals at the next level. With the Archimedean model, weights or penalties for not achieving targets must be specified and one attempts to minimize the weighted sum of goal under–achievement.

In the Optimizer, goals can be constructed either from constraints or from objective functions (N rows). If constraints are used to construct the goals, then the goals are to minimize the violation of the constraints. The goals are met when the constraints are satisfied. In the pre–emptive case we try to meet as many goals as possible, taking them in priority order. In the Archimedean case, we minimize a weighted sum of penalties for not meeting each of the goals. If the goals are constructed from N rows, then, in the pre–emptive case, a target for each N row is calculated from the optimal value for the N row. This may be done by specifying either a percentage or absolute deviation that may be allowed from the optimal value for the N rows. In the Archimedean case, the problem becomes a multi–objective linear programming problem in which a weighted sum of the objective functions is to be minimized.

In this section four examples will be provided of the four different types of goal programming available. Goal programming itself is performed using the `XPRSgoal` (`GOAL`) command, whose syntax is described in full in the reference section of this manual.

7.2 Pre–emptive Goal Programming Using Constraints

For this case, goals are ranked from most important to least important. Initially we try to satisfy the most important goal. Then amongst all the solutions that satisfy the first goal, we try to come as close as possible to satisfying the second goal. We continue in this fashion until the only way we can come closer to satisfying a goal is to increase the deviation from a higher priority goal.

An example of this is as follows:

goal 1 (G1):	7x + 3y	$\geq$	40
goal 2 (G2):	10x + 5y	=	60
goal 3 (G3):	5x + 4y	$\leq$	35
LIMIT:	100x + 60y	$\leq$	600

Initially we try to meet the first goal (G1), which can be done with `x=5.0` and `y=1.6`, but this solution does not satisfy goal 2 (G2) or goal 3 (G3). If we try to meet goal 2 while still meeting goal 1, the solution `x=6.0` and `y=0.0` will satisfy. However, this does not satisfy goal 3, so we repeat the process. On this occasion no solution exists which satisfies all three.

7.3 Archimedean Goal Programming Using Constraints

We must now minimize a weighted sum of violations of the constraints. Suppose that we have the following problem, this time with penalties attached:

				Penalties
goal 1 (G1):	7x + 3y	$\geq$	40	8
goal 2 (G2):	10x + 5y	=	60	3
goal 3 (G3):	5x + 4y	$\leq$	35	1
LIMIT:	100x + 60y	$\leq$	600	

Then the solution will be the solution of the following problem:

minimize:	$8d_1 + 3d_2 + 3d_3 + 1d_4$		
subject to:	$7x + 3y + d_1$	$\geq$	40
	$10x + 5y + d_2 - d_3$	=	60
	$5x + 4y + d_4$	$\geq$	35
	$100x + 60y$	$\leq$	600
	$d_1 \geq 0, d_2 \geq 0, d_3 \geq 0, d_4 \geq 0$		

In this case a penalty of `8` units is incurred for each unit that `7x + 3y` is less than `40` and so on. the final solution will minimize the weighted sum of the penalties. Penalties are also referred to as *weights*. This solution will be x=6, y=0, d_1=d_2=d_3=0 and d_4=5, which means that the first and second most important constraints can be met, while for the third constraint the right hand side must be reduced by `5` units in order to be met.

Note that if the problem is infeasible after all the goal constraints have been relaxed, then no solution will be found.

7.4 Pre–emptive Goal Programming Using Objective Functions

Suppose that we have a set of objective functions and knowing which are the most important. As in the pre–emptive case with constraints, goals are ranked from most to least important. Initially we find the optimal value of the first goal. Once we have found this value we turn this objective function into a constraint such that its value does not differ from its optimal value by more than a certain amount. This can be a *fixed* amount (or *absolute* deviation) or a percentage of (or relative deviation from) the optimal value found before. Now we optimize the next goal (the second most important objective function) and so on.

For example, suppose we have the following problem:

				Sense	D/P	Deviation
goal 1 (OBJ1):	5x + 2y	–	20	max	P	10
goal 2 (OBJ2):	–3x + 15y	–	48	min	D	4
goal 3 (OBJ3):	1.5x + 21y	–	3.8	max	P	20
LIMIT:	42x + 13y	≤	100			

For each goal the sense of the optimization (max or min) and the percentage (P) or absolute (D) deviation must be specified. For `OBJ1` and `OBJ3` a percentage deviation of 10% and 20%, respectively, have been specified, whilst for `OBJ2` an absolute deviation of 4 units has been specified.

We start by maximizing the first objective function, finding that the optimal value is `--4.615385`. As a 10% deviation has been specified, we change this objective function into the following constraint:

5x + 2y – 20	≥	–4.615385 – 0.14615385

Now that we know that for any solution the value for the former objective function must be within 10% of the best possible value, we minimize the next most important objective function (`OBJ2`) and find the optimal value to be `51.133603`. Goal 2 (`OBJ2`) may then be changed into a constraint such that:

–3x + 15y – 48	≤	51.133603 + 4

and in this way we ensure that for any solution, the value of this objective function will not be greater than the best possible minimum value plus `4` units.

Finally we have to maximize `OBJ3`. An optimal value of `141.943995` will be obtained. Since a 20% allowable deviation has been specified, this objective function may be changed into the following constraint:

1.5x + 21y – 3.8	≥	141.943995 – 0.2141.943995

The solution of this problem is `x=0.238062` and `y=6.923186`.

7.5 Archimedean Goal Programming Using Objective Functions

In this, the final case, we optimize a weighted sum of objective functions. In other words we solve a multi–objective problem. For consider the following:

				Weights	Sense
goal 1 (OBJ1):	5x + 2y	–	20	100	max
goal 2 (OBJ2):	–3x + 15y	–	48	1	min
goal 3 (OBJ3):	1.5x + 21y	–	3.8	0.01	max
LIMIT:	42x + 13y	≤	100		

In this case we have three different objective functions that will be combined into a single objective function by weighting them by the values given in the *weights* column. The solution of this model is one that minimizes:

1(–3x + 15y – 48) – 100(5x + 2y – 20) – 0.01(1.5x + 21y – 3.8)

The resulting values that each of the objective functions will have are as follows:

OBJ1:	$5x + 2y - 20$	=	-4.615389
OBJ2:	$-3x + 15y - 48$	=	67.384613
OBJ3:	$1.5x + 21y - 3.8$	=	157.738464

The solution is `x=0.0` and `y=7.692308`.

CHAPTER 8

Console and Library Functions

A large number of routines are available for both Console and Library users of the FICO Xpress Optimizer, ranging from simple routines for the input and solution of problems from matrix files to sophisticated callback functions and greater control over the solution process. Of these, the core functionality is available to both sets of users and comprises the 'Console Mode'. Library users additionally have access to a set of more 'advanced' functions, which extend the functionality provided by the Console Mode, providing more control over their program's interaction with the Optimizer and catering for more complicated problem development.

8.1 Console Mode Functions

With both the Console and Advanced Mode functions described side-by-side in this chapter, library users can use this as a quick reference for the full capabilities of the Optimizer library. For users of Console Optimizer, only the following functions will be of relevance:

Command	Description	Page
CHECKCONVEXITY	Convexity checker.	p. 146
EXIT	Terminate the Console Optimizer.	p. 176
HELP	Quick reference help for the Optimizer console.	p. 263
IIS	Console IIS command.	p. 264
PRINTRANGE	Writes the ranging information to screen.	p. 315
PRINTSOL	Write the current solution to screen.	p. 316
QUIT	Terminate the Console Optimizer.	p. 317
STOP	Terminate the Console Optimizer.	p. 376
TUNE	Console Tuner command.	p. 382
SETARCHCONSISTENCY	Sets whether to force the same execution path on various CPU architecture extensions, in particular (pre-)AVX and AVX2.	p. 81
ALTER	Alters or changes matrix elements, right hand sides and constraint senses in the current problem.	p. 138
BASISCONDITION	This function is deprecated, and will be removed in future releases. Please use the XPRSbasisstability function instead. Calculates the condition number of the current basis after solving the LP relaxation.	p. 139
BASISSTABILITY	Calculates various measures for the stability of the current basis, including the basis condition number.	p. 140
CHGOBJSENSE	Changes the problem's objective function sense to minimize or maximize.	p. 154
DUMPCONTROLS	Displays the list of controls and their current value for those controls that have been set to a non default value.	p. 175
FIXGLOBALS	Fixes all the global entities to the values of the last found MIP solution. This is useful for finding the reduced costs for the continuous variables after the global variables have been fixed to their optimal values.	p. 179

For a list of functions by task, refer to 2.8.

8.2 Layout for Function Descriptions

All functions mentioned in this chapter are described under the following set of headings:

Function Name

The description of each routine starts on a new page. The library name for a function is on the left and the Console Optimizer command name, if one exists, is on the right.

Purpose

A short description of the routine and its purpose begins the information section.

Synopsis

A synopsis of the syntax for usage of the routine is provided. "Optional" arguments and flags may be specified as `NULL` if not required. Where this possibility exists, it will be described alongside the argument, or in the Further Information at the end of the routine's description. If the function is available in the Console, the library syntax is described first, followed by the Console Optimizer syntax.

Arguments

A list of arguments to the routine with a description of possible values for them follows.

Error Values

Optimizer return codes are described in Chapter 11. For library users, however, a return code of `32` indicates that additional error information may be obtained, specific to the function which caused the error. Such is available by calling

```
XPRSgetintattrib(prob,XPRS_ERRORCODE,&errorcode);
```

Likely error values returned by this for each function are listed in the Error Values section. A description of the error may be obtained using the `XPRSgetlasterror` function. If no attention need be drawn to particular error values, this section will be omitted.

Associated Controls

Controls which affect a given routine are listed next, separated into lists by type. The control name given here should have `XPRS_` prefixed by library users, in a similar way to the `XPRSgetintattrib` example in the Error Values section above. Console Xpress users should use the controls without this prefix, as described in FICO Xpress Getting Started manual. These controls must be set before the routine is called if they are to have any effect.

Examples

One or two examples are provided which explain certain aspects of the routine's use.

Further Information

Additional information not contained elsewhere in the routine's description is provided at the end.

Related Topics

Finally a list of related routines and topics is provided for comparison and reference.

XPRS_bo_addbounds

Purpose

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

Synopsis

```
int XPRS_CC XPRS_bo_addbounds(XPRSbranchobject obranch, int ibranch, int
      nbounds, const char cbndtype[], const int mbndcol[], const double
      dbndval[]);
```

Arguments

`obranch`	The user branching object to modify.
`ibranch`	The number of the branch to add the new bounds for. This branch must already have been created using `XPRS_bo_addbranches`. Branches are indexed starting from zero.
`nbounds`	Number of new bounds to add.
`cbndtype`	Character array of length `nbounds` indicating the type of bounds to add: `L` Lower bound. `U` Upper bound.
`mbndcol`	Integer array of length `nbounds` containing the column indices for the new bounds.
`dbndval`	Double array of length `nbounds` giving the bound values.

Example

See `XPRS_bo_create` for an example using `XPRS_bo_addbounds`.

Related topics

`XPRS_bo_create`.

XPRS_bo_addbranches

Purpose

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

Synopsis

```
int XPRS_CC XPRS_bo_addbranches(XPRSbranchobject obranch, int nbranches);
```

Arguments

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

Example

See `XPRS_bo_create` for an example using `XPRS_bo_addbranches`.

Related topics

`XPRS_bo_create`.

XPRS_bo_addcuts

Purpose

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

Synopsis

```
int XPRS_CC XPRS_bo_addcuts(XPRSbranchobject obranch, int ibranch, int
      ncuts, const XPRScut mcutind[]);
```

Arguments

`obranch`	The user branching object to modify.
`ibranch`	The number of the branch to add the cuts for. This branch must already have been created using `XPRS_bo_addbranches`. Branches are indexed starting from zero.
`ncuts`	Number of cuts to add.
`mcutind`	Array of length `ncuts` containing the pointers to user cuts that should be added to the branch.

Related topics

`XPRS_bo_create`, `XPRS_bo_addrows`.

XPRS_bo_addrows

Purpose

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

Synopsis

```
int XPRS_CC XPRS_bo_addrows(XPRSbranchobject obranch, int ibranch, int
      nrows, int nelems, const char crtype[], const double drrhs[], const
      int mrbeg[], const int mcol[], const double dval[]);
```

Arguments

`obranch`	The user branching object to modify.
`ibranch`	The number of the branch to add the new constraints for. This branch must already have been created using `XPRS_bo_addbranches`. Branches are indexed starting from zero.
`nrows`	Number of new constraints to add.
`nelems`	Number of non-zero coefficients in all new constraints.
`crtype`	Character array of length `nrows` indicating the type of constraints to add: `L` Less than type. `G` Greater than type. `E` Equality type.
`drrhs`	Double array of length `nrows` containing the right hand side values.
`mrbeg`	Integer array of length `nrows` containing the offsets of the `mcol` and `dval` arrays of the start of the non zero coefficients in the new constraints.
`mcol`	Integer array of length `nelems` containing the column indices for the non zero coefficients.
`dval`	Double array of length `nelems` containing the non zero coefficient values.

Example

The following function will create a branching object that branches on constraints $x_1 + x_2 \geq 1$ or $x_1 + x_2 \leq 0$:

```
XPRSbranchobject CreateConstraintBranch(XPRSprob xp_mip, int icol)
{
  char   cRowType;
  double dRowRHS;
  int    mRowBeg;
  int    mElemCol[2];
  double dElemVal[2];

  XPRSbranchobject bo = NULL;
  int isoriginal = 1;

  /* Create the new object with two empty branches. */
  XPRS_bo_create(&bo, xp_mip, isoriginal);
  XPRS_bo_addbranches(bo, 2);

  /* Add the constraint x1 + x2 >= 1. */
  cRowType = 'G';
  dRowRHS  = 1.0;
  mRowBeg  = 0;
  mElemCol[0] = 0; mElemCol[1] = 1;
  dElemVal[0] = 1.0; dElemVal[1] = 1.0;
  XPRS_bo_addrows
    (bo, 0, 1, 2, &cRowType, &dRowRHS, &mRowBeg, mElemCol, dElemVal);
```

```
    /* Add the constraint x1 + x2 <= 0. */
    cRowType = 'L';
    dRowRHS  = 0.0;
    XPRS_bo_addrows
      (bo, 1, 1, 2, &cRowType, &dRowRHS, &mRowBeg, mElemCol, dElemVal);

    /* Set a low priority value so our branch object is picked up */
    /* before the default branch candidates. */
    XPRS_bo_setpriority(bo, 100);

    return bo;
  }
```

Related topics

XPRS_bo_create.

XPRS_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 `XPRSaddcboptnode` or `XPRSaddcbchgbranchobject`.

Synopsis

```
int XPRS_CC XPRS_bo_create(XPRSbranchobject* p_object, XPRSprob prob, int
        isoriginal);
```

Arguments

`p_object`	Pointer to where the new object should be returned.
`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.

Further information

1. In addition to the standard global entities supported by the Optimizer, the Optimizer also allows the user to define their own global entities for branching, using branching objects.
2. A branching object of type `XPRSbranchobject` should provide a linear description of how to branch on the current node for a user's global entities. Any number of branches is allowed and each branch description can contain any combination of columns bounds and new constraints.
3. Branching objects must always contain at least one branch and all branches of the object must contain at least one bound or constraint.
4. When the Optimizer branches the current node on a user's branching object, a new child node will be created for each branch defined in the object. The child nodes will inherit the bounds and constraint of the current node, plus any new bounds or constraints defined for that branch in the object.
5. Inside the callback function set by `XPRSaddcboptnode`, a user can define any number of branching objects and pass them to the Optimizer. These objects are added to the set of infeasible global entities for the current node and the Optimizer will select a best candidate from this extended set using all of its normal evaluation methods.
6. The callback function set by `XPRSaddcbchgbranchobject` can be used to override the Optimizer's selected branching candidate with the user's own object. This can for example be used to modify how to branch on the global entity selected by the Optimizer.
7. The following functions are available to set up a new user branching object:

`XPRS_bo_create`	Creates a new, empty branching object with no branches.
`XPRS_bo_addbranches`	Adds new, empty branches to the object. Branches must be created before column bounds or rows can be added to a branch.
`XPRS_bo_addbounds`	Adds new column bounds to a given branch of the object.
`XPRS_bo_addrows`	Adds new constraints to a given branch of the object.
`XPRS_bo_setpriority`	Sets the priority value for the object. These are equivalent to the priority values for regular global entities that can be set through directives (see also Appendix A.6).
`XPRS_bo_setpreferredbranch`	Specifies which of the child nodes corresponding to the branches of the object should be explored first.
`XPRS_bo_store`	Adds the created object to the candidate list for branching.

Example

The following function will create a branching object equivalent to a standard binary branch on a column:

```
XPRSbranchobject CreateBinaryBranchObject(XPRSprob xp_mip, int icol)
{
  char   cBndType;
  double dBndValue;
  int isoriginal = 1;

  XPRSbranchobject bo = NULL;

  /* Create the new object with two empty branches. */
  XPRS_bo_create(&bo, xp_mip, isoriginal);
  XPRS_bo_addbranches(bo, 2);

  /* Add bounds to branch the column to either zero or one. */
  cBndType = 'U';
  dBndValue = 0.0;
  XPRS_bo_addbounds(bo, 0, 1, &cBndType, &icol, &dBndValue);
  cBndType = 'L';
  dBndValue = 1.0;
  XPRS_bo_addbounds(bo, 1, 1, &cBndType, &icol, &dBndValue);

  /* Set a low priority value so our branch object is picked up */
  /* before the default branch candidates. */
  XPRS_bo_setpriority(bo, 100);

  return bo;
}
```

Related topics

XPRSaddcboptnode, XPRSaddcbchgbranchobject.

XPRS_bo_destroy

Purpose

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

Synopsis

```
int XPRS_CC XPRS_bo_destroy(XPRSbranchobject obranch);
```

Argument

`obranch`	The user branching object to free.

Related topics

`XPRS_bo_create`, `XPRS_bo_store`.

XPRS_bo_getbounds

Purpose

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

Synopsis

```
int XPRS_CC XPRS_bo_getbounds(XPRSbranchobject obranch, int ibranch, int*
      p_nbounds, int nbounds_size, char cbndtype[], int mbndcol[], double
      dbndval[]);
```

Arguments

`obranch`	The branching object to inspect.
`ibranch`	The number of the branch to get the bounds for.
`p_nbounds`	Location where the number of bounds for the given branch should be returned.
`nbounds_size`	Maximum number of bounds to return.
`cbndtype`	Character array of length `nbounds_size` where the types of bounds will be returned: `L` Lower bound. `U` Upper bound. Allowed to be NULL.
`mbndcol`	Integer array of length `nbounds_size` where the column indices will be returned. Allowed to be NULL.
`dbndval`	Double array of length `nbounds_size` where the bound values will be returned. Allowed to be NULL.

Related topics

`XPRS_bo_create`, `XPRS_bo_addbounds`.

XPRS_bo_getbranches

Purpose

Returns the number of branches of a branching object.

Synopsis

```
int XPRS_CC XPRS_bo_getbranches(XPRSbranchobject obranch, int*
        p_nbranches);
```

Arguments

`obranch`	The user branching object to inspect.
`p_nbranches`	Memory where the number of branches should be returned.

Related topics

`XPRS_bo_create`, `XPRS_bo_addbranches`.

XPRS_bo_getid

Purpose

Returns the unique identifier assigned to a branching object.

Synopsis

```
int XPRS_CC XPRS_bo_getid(XPRSbranchobject obranch, int* p_id);
```

Arguments

`obranch`	A branching object.
`p_id`	Pointer to an integer where the identifier should be returned.

Further information

1. Branching objects associated with existing column entities (binaries, integers, semi–continuous and partial integers), are given an identifier from 1 to `MIPENTS`.
2. Branching objects associated with existing Special Ordered Sets, are given an identifier from `MIPENTS`+1 to `MIPENTS`+`SETS`.
3. User created branching objects will always have a negative identifier.

Related topics

`XPRS_bo_create`.

XPRS_bo_getlasterror

Purpose

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

Synopsis

```
int XPRS_CC XPRS_bo_getlasterror(XPRSbranchobject obranch, int* iMsgCode,
      char* _msg, int _iStringBufferBytes, int* _iBytesInInternalString);
```

Arguments

`obranch`	The branch object.
`iMsgCode`	Location where the error code will be returned. Can be NULL if not required.
`_msg`	A character buffer of size `iStringBufferBytes` in which the last error message relating to the given branching object will be returned.
`iStringBufferBytes`	The size of the character buffer `_msg`.
`_iBytesInInternalString`	The size of the required character buffer to fully return the error string.

Example

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

```
XPRSbranchobject obranch;
...
char* cbuf;
int cbuflen;
if (XPRS_bo_setpreferredbranch(obranch,3)) {
  XPRS_bo_getlasterror(obranch,NULL,NULL,0,&cbuflen);
  cbuf = malloc(cbuflen);
  XPRS_bo_getlasterror(obranch,NULL, cbuf, cbuflen, NULL);
  printf("ERROR when setting preferred branch: %s\n", cbuf);
}
```

Related topics

`XPRS_ge_setcbmsghandler`.

XPRS_bo_getrows

Purpose

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

Synopsis

```
int XPRS_CC XPRS_bo_getrows(XPRSbranchobject obranch, int ibranch, int*
       p_nrows, int nrows_size, int* p_nelems, int nelems_size, char
       crtype[], double drrhs[], int mrbeg[], int mcol[], double dval[]);
```

Arguments

`obranch`	The user branching object to inspect.
`ibranch`	The number of the branch to get the constraints from.
`p_nrows`	Memory location where the number of rows should be returned.
`nrows_size`	Maximum number of rows to return.
`p_nelems`	Memory location where the number of non zero coefficients in the constraints should be returned.
`nelems_size`	Maximum number of non zero coefficients to return.
`crtype`	Character array of length `nrows_size` where the types of the rows will be returned: `L` Less than type. `G` Greater than type. `E` Equality type.
`drrhs`	Double array of length `nrows_size` where the right hand side values will be returned.
`mrbeg`	Integer array of length `nrows_size` which will be filled with the offsets of the `mcol` and `dval` arrays of the start of the non zero coefficients in the returned constraints.
`mcol`	Integer array of length `nelems_size` which will be filled with the column indices for the non zero coefficients.
`dval`	Double array of length `nelems_size` which will be filled with the non zero coefficient values.

Related topics

`XPRS_bo_create`, `XPRS_bo_addrows`.

XPRS_bo_setpreferredbranch

Purpose

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

Synopsis

```
int XPRS_CC XPRS_bo_setpreferredbranch(XPRSbranchobject obranch, int
      ibranch);
```

Arguments

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

Related topics

`XPRS_bo_create`.

XPRS_bo_setpriority

Purpose

Sets the priority value of a user branching object.

Synopsis

```
int XPRS_CC XPRS_bo_setpriority(XPRSbranchobject obranch, int ipriority);
```

Arguments

`obranch`	The user branching object.
`ipriority`	The new priority value to assign to the branching object, which must be a number from 0 to 1000. User branching objects are created with a default priority value of 500.

Further information

1. A candidate branching object with lowest priority number will always be selected for branching before an object with a higher number.
2. Priority values must be an integer from 0 to 1000. User branching objects and global entities are by default assigned a priority value of 500. Special branching objects, such as those arising from structural branches or split disjunctions are assigned a priority value of 400.

Related topics

`XPRS_bo_create`, Section A.6.

XPRS_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 function set by XPRSaddcboptnode.

Synopsis

```
int XPRS_CC XPRS_bo_store(XPRSbranchobject obranch, int* p_status);
```

Arguments

obranch	The new user branching object to store. After this call the obranch object is no longer valid and should not be referred to again.
p_status	The returned status from checking the provided branching object: 0 The object was accepted successfully. 1 Failed to presolve the object due to dual reductions in presolve. 2 Failed to presolve the object due to duplicate column reductions in presolve. 3 The object contains an empty branch. The object was not added to the candidate list if a non zero status is returned.

Further information

1. To ensure that a user branching object expressed in terms of the original matrix columns can be applied to the presolved problem, it might be necessary to turn off certain presolve operations.
2. If any of the original matrix columns referred to in the object are unbounded, dual reductions might prevent the corresponding bound or constraint from being presolved. To avoid this, dual reductions should be turned off in presolve, by clearing bit 3 of the integer control PRESOLVEOPS.
3. If one or more of the original matrix columns of the object are duplicates in the original matrix, but not in the branching object, it might not be possible to presolve the object due to duplicate column eliminations in presolve. To avoid this, duplicate column eliminations should be turned off in presolve, by clearing bit 5 of PRESOLVEOPS.
4. As an alternative to turning off the above mentioned presolve features, it is possible to protect individual columns of a the problem from being modified by presolve. Use the XPRSloadsecurevecs function to mark any columns that might be branched on using branching objects.

Related topics

XPRS_bo_create, XPRS_bo_validate.

XPRS_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

```
int XPRS_CC XPRS_bo_validate(XPRSbranchobject obranch, int* p_status);
```

Arguments

`obranch`	A branching object.
`p_status`	The returned status from checking the provided branching object: `0` The object is acceptable. `1` Failed to presolve the object due to dual reductions in presolve. `2` Failed to presolve the object due to duplicate column reductions in presolve. `3` The object contains an empty branch.

Related topics

`XPRS_bo_create`.

XPRS_ge_addcbmsghandler

Purpose

Declares an output callback function in the global environment, called every time a line of message text is output by any object in the library. This callback function will be called in addition to any output callbacks already added by `XPRS_ge_addcbmsghandler`.

Synopsis

```
int XPRS_CC XPRS_ge_addcbmsghandler(int (XPRS_CC *f_msghandler)(XPRSobject
        vXPRSObject, void * vUserContext, void * vSystemThreadId, const char
        * sMsg, int iMsgType, int iMsgNumber), void *object, int priority);
```

Arguments

`f_msghandler` The callback function which takes six arguments, `vXPRSObject`, `vUserContext`, `vSystemThreadId`, `sMsg`, `iMsgType` and `iMsgNumber`. Use a NULL value to cancel a callback function.

`vXPRSObject` The object sending the message. Use `XPRSgetobjecttypename` to get the name of the object type.

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

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

`sMsg` A null terminated character array (string) containing the message, which may simply be a new line. When the callback is called for the first time `sMsg` will be a NULL pointer.

`iMsgType` Indicates the type of output message:
- `1` information messages;
- `2` (not used);
- `3` warning messages;
- `4` error messages.

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

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

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

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

Further information

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

```
XPRS_ge_addcbmsghandler(XPRSlogfilehandler, "log.txt", 0);
```

Related topics

`XPRS_ge_removecbmsghandler`, `XPRSgetobjecttypename`.

XPRS_ge_getcbmsghandler

Purpose

This function is deprecated and may be removed in future releases.

Get the output callback function for the global environment, as set by `XPRS_ge_setcbmsghander`.

Synopsis

```
int XPRS_CC XPRS_ge_getcbmsghandler(int (XPRS_CC
      **r_f_msghandler)(XPRSobject vXPRSObject, void * vUserContext, void *
      vSystemThreadId, const char * sMsg, int iMsgType, int iMsgNumber),
      void **object);
```

Arguments

`r_f_msghandler`	Pointer to the memory where the callback function will be returned.
`object`	Pointer to the memory where the callback function context value will be returned.

Related topics

`XPRS_ge_setcbmsghandler`.

XPRS_ge_getlasterror

Purpose

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

Synopsis

```
int XPRS_CC XPRS_ge_getlasterror(int* iMsgCode, char* _msg, int
      _iStringBufferBytes, int* _iBytesInInternalString);
```

Arguments

`iMsgCode` Memory location in which the error code will be returned. Can be NULL if not required.

`_msg` A character buffer of size `iStringBufferBytes` in which the last error message relating to the global environment will be returned.

`iStringBufferBytes` The size of the character buffer `_msg`.

`_iBytesInInternalString` Memory location in which the minimum required size of the buffer to hold the full error string will be returned. Can be NULL if not required.

Example

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

```
char* cbuf;
int cbuflen;
if (XPRS_ge_setcbmsghandler(myfunc,NULL)!=0) {
  XPRS_ge_getlasterror(NULL,NULL,0,&cbuflen);
  cbuf = malloc(cbuflen);
  XPRS_ge_getlasterror(NULL, cbuf, cbuflen, NULL);
  printf("ERROR from Xpress global environment: %s\n", cbuf);
}
```

Related topics

`XPRS_ge_setcbmsghandler`.

XPRS_ge_removecbmsghandler

Purpose

Removes a message callback function previously added by `XPRS_ge_addcbmsghandler`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRS_ge_removecbmsghandler(int (XPRS_CC *f_msghandler)
      (XPRSobject vXPRSObject, void * vUserContext, void * vSystemThreadId,
      const char * sMsg, int iMsgType, int iMsgNumber), void * object);
```

Arguments

`f_msghandler`	The callback function to remove. If NULL then all message callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all message callbacks with the function pointer `f_msghandler` will be removed.

Related topics

`XPRS_ge_addcbmsghandler`

XPRS_ge_setcbmsghandler

Purpose

This function is deprecated and may be removed in future releases. Please use `XPRS_ge_addcbmsghandler` instead.

Declares an output callback function, called every time a line of message text is output by any object in the library.

Synopsis

```
int XPRS_CC XPRS_ge_setcbmsghandler(int (XPRS_CC *f_msghandler)
        (XPRSobject vXPRSObject, void * vUserContext, void * vSystemThreadId,
        const char * sMsg, int iMsgType, int iMsgNumber), void * p);
```

Arguments

`f_msghandler` The callback function which takes six arguments, `vXPRSObject`, `vUserContext`, `vSystemThreadId`, `sMsg`, `iMsgType` and `iMsgNumber`. Use a NULL value to cancel a callback function.

`vXPRSObject` The object sending the message. Use `XPRSgetobjecttypename` to get the name of the object type.

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

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

`sMsg` A null terminated character array (string) containing the message, which may simply be a new line. When the callback is called for the first time `sMsg` will be a NULL pointer.

`iMsgType` Indicates the type of output message:

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

A negative value means the callback is being called for the first time.

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

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

Further information

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

```
XPRS_ge_setcbmsghandler(XPRSlogfilehandler, "log.txt");
```

Related topics

`XPRSgetobjecttypename`.

XPRS_ge_setarchconsistency SETARCHCONSISTENCY

Purpose

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

Synopsis

```
int XPRS_CC XPRS_ge_setarchconsistency(int ifArchConsistent);
SETARCHCONSISTENCY ifArchConsistent
```

Argument

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

Further information

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

XPRS_nml_addnames

Purpose

The `XPRS_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 `XPRS_nml_addnames` to add names to a name list, or modify existing names on a namelist.

Synopsis

```
int XPRS_CC XPRS_nml_addnames(XPRSnamelist nml, const char buf[], int
        firstIndex, int lastIndex);
```

Arguments

`nml`	The name list to which you want to add names. Must be an object previously returned by `XPRS_nml_create`, as `XPRSnamelist` objects returned by other functions are immutable and cannot be changed.
`names`	Character buffer containing the null-terminated string names.
`first`	The index of the first name to add/replace. Name indices in a namelist always start from 0.
`last`	The index of the last name to add/replace.

Example

```
char mynames[0] = "fred\0jim\0sheila"
...
XPRS_nml_addnames(nml,mynames,0,2);
```

Related topics

`XPRS_nml_create`, `XPRS_nml_removenames`, `XPRS_nml_copynames`, `XPRSaddnames`.

XPRS_nml_copynames

Purpose

The `XPRS_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. `XPRS_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, `XPRS_nml_copynames` will be most often used to copy such names to a namelist where they can be modified, for some later use.

Synopsis

```
int XPRS_CC XPRS_nml_copynames(XPRSnamelist dst, XPRSnamelist src);
```

Arguments

`dst`	The namelist object to copy names to. Any names already in this name list will be removed. Must be an object previously returned by `XPRS_nml_create`.
`src`	The namelist object from which all the names should be copied.

Example

```
XPRSprob prob;
XPRSnamelist rnames, rnames_on_prob;
...
/* Create a namelist */
XPRS_nml_create(&rnames);
/* Get a namelist through which we can access the row names */
XPRSgetnamelistobject(prob,1,&rnames_on_prob);
/* Now copy these names from the immutable 'XPRSprob' namelist
   to another one */
XPRS_nml_copynames(rnames,rnames_on_prob);
/* The names in the list can now be modified then put to some
   other use */
```

Related topics

`XPRS_nml_create`, `XPRS_nml_addnames`, `XPRSgetnamelistobject`.

XPRS_nml_create

Purpose

The XPRS_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. XPRS_nml_create will create a new namelist to which the user can add, remove and otherwise modify names.

Synopsis

```
int XPRS_CC XPRS_nml_create(XPRSnamelist* p_nml);
```

Argument

p_nml	Pointer to location where the new namelist will be returned.

Example

```
XPRSnamelist mylist;
XPRS_nml_create(&mylist);
```

Related topics

XPRSgetnamelistobject, XPRS_nml_destroy.

XPRS_nml_destroy

Purpose

Destroys a namelist and frees any memory associated with it. Note you need only destroy namelists created by `XPRS_nml_destroy` - those returned by `XPRSgetnamelistobject` are automatically destroyed when you destroy the problem object.

Synopsis

```
int XPRS_CC XPRS_nml_destroy(XPRSnamelist nml);
```

Argument

`nml`	The namelist to be destroyed.

Example

```
XPRSnamelist mylist;
XPRS_nml_create(&mylist);
...
XPRS_nml_destroy(&mylist);
```

Related topics

`XPRS_nml_create`, `XPRSgetnamelistobject`, `XPRSdestroyprob`.

XPRS_nml_findname

Purpose

The `XPRS_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.
`XPRS_nml_findname` returns the index of the given name in the given name list.

Synopsis

```
int XPRS_CC XPRS_nml_findname(XPRSnamelist nml, const char* name, int*
      r_index);
```

Arguments

`nml`	The namelist in which to look for the name.
`name`	Null-terminated string containing the name for which to search.
`r_index`	Pointer to variable in which the index of the name is returned, or in which is returned -1 if the name is not found in the namelist.

Example

```
XPRSnamelist mylist;
int idx;
...
XPRS_nml_findname(mylist, "profit_after_work", &idx);
if (idx==-1)
  printf("'profit_after_work' was not found in the namelist");
else
  printf("'profit_after_work' was found at position %d", idx);
```

Related topics

`XPRS_nml_addnames`, `XPRS_nml_getnames`.

XPRS_nml_getlasterror

Purpose

Returns the last error encountered during a call to a namelist object.

Synopsis

```
int XPRS_CC XPRS_nml_getlasterror(XPRSnamelist nml, int* iMsgCode, char*
      _msg, int _iStringBufferBytes, int* _iBytesInInternalString);
```

Arguments

`nml` The namelist object.

`iMsgCode` Variable in which the error code will be returned. Can be NULL if not required.

`_msg` A character buffer of size `iStringBufferBytes` in which the last error message relating to this namelist will be returned.

`_iStringBufferBytes` The size of the character buffer `_msg`.

`_iBytesInInternalString` Memory location in which the minimum required size of the buffer to hold the full error string will be returned. Can be NULL if not required.

Example

```
XPRSnamelist nml;
char* cbuf;
int cbuflen;
...
if (XPRS_nml_removenames(nml,2,35)) {
  XPRS_nml_getlasterror(nml, NULL, NULL, 0, &cbuflen);
  cbuf = malloc(cbuflen);
  XPRS_nml_getlasterror(nml, NULL, cbuf, cbuflen, NULL);
  printf("ERROR removing names: %s\n", cbuf);
}
```

Related topics

None.

XPRS_nml_getmaxnamelen

Purpose

The `XPRS_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. `XPRS_nml_getmaxnamelen` returns the length of the longest name in the namelist.

Synopsis

```
int XPRS_CC XPRS_nml_getmaxnamelen(XPRSnamelist nml, int* namlen);
```

Arguments

`nml`	The namelist object.
`namelen`	Pointer to a variable into which shall be written the length of the longest name.

Related topics

None.

XPRS_nml_getnamecount

Purpose

The `XPRS_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. `XPRS_nlm_getnamecount` returns the number of names in the namelist.

Synopsis

```
int XPRS_CC XPRS_nml_getnamecount(XPRSnamelist nml, int* count);
```

Arguments

`nml`	The namelist object.
`count`	Pointer to a variable into which shall be written the number of names.

Example

```
XPRSnamelist mylist;
int count;
...
XPRS_nml_getnamecount(mylist,&count);
printf("There are %d names", count);
```

Related topics

None.

XPRS_nml_getnames

Purpose

The `XPRS_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 `XPRS_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

```
int XPRS_CC XPRS_nml_getnames(XPRSnamelist nml, int padlen, char buf[], int
      buflen, int* r_buflen_reqd, int firstIndex, int lastIndex);
```

Arguments

`nml`	The namelist object.
`padlen`	The minimum length of each name. If > 0 then names shorter than `padlen` will be concatenated with whitespace to make them this length.
`buf`	Buffer of length `buflen` into which the names shall be returned.
`buflen`	The maximum number of bytes that may be written to the character buffer `buf`.
`r_buflen_reqd`	A pointer to a variable into which will be written the number of bytes required to contain the names. May be NULL if not required.
`firstIndex`	The index of the first name in the namelist to return. Note name list indexes always start from 0.
`lastIndex`	The index of the last name in the namelist to return.

Example

```
XPRSnamelist mylist;
char* cbuf;
int o, i, cbuflen;
...
/* Find out how much space we'll require for these names */
XPRS_nml_getnames(mylist, 0, NULL, 0, &cbuflen, 0, 5 );
/* Allocate a buffer large enough to hold the names */
cbuf = malloc(cbuflen);
/* Retrieve the names */
XPRS_nml_getnames(mylist, 0, cbuf, cbuflen, NULL, 0, 5);
/* Display the names */
o=0;
for (i=0;i<6;i++) {
  printf("Name #%d = %s\n", i, cbuf+o);
  o += strlen(cbuf)+1;
}
```

Related topics

None.

XPRS_nml_removenames

Purpose

The XPRS_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.
XPRS_nml_removenames will remove the specified names from the name list. Any subsequent names will be moved down to replace the removed names.

Synopsis

```
int XPRS_CC XPRS_nml_removenames(XPRSnamelist nml, int firstIndex, int
      lastIndex);
```

Arguments

nml	The name list from which you want to remove names. Must be an object previously returned by XPRS_nml_create, as XPRSnamelist objects returned by other functions are immutable and cannot be changed.
firstIndex	The index of the first name to remove. Note that indices for names in a name list always start from 0.
lastIndex	The index of the last name to remove.

Example

```
XPRS_nml_removenames(mylist, 3, 5);
```

Related topics

XPRS_nml_addnames.

XPRSaddcbbariteration

Purpose

Declares a barrier iteration callback function, called after each iteration during the interior point algorithm, with the ability to access the current barrier solution/slack/duals or reduced cost values, and to ask barrier to stop. This callback function will be called in addition to any callbacks already added by XPRSaddcbbariteration.

Synopsis

```
int XPRS_CC XPRSaddcbbariteration (XPRSprob prob, void (XPRS_CC
       *f_bariteration)( XPRSprob my_prob, void *my_object, int
       *barrier_action), void *object, int priority);
```

Arguments

`prob`	The current problem.
`f_bariteration`	The callback function itself. This takes three arguments, `my_prob`, `my_object`, and `barrier_action` serving as an integer return value. This function is called at every barrier iteration.
`my_prob`	The problem passed to the callback function, `f_bariteration`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `XPRSaddcbbariteration`.
`barrier_action`	Defines a return value controlling barrier: `<0` continue with the next iteration; `=0` let barrier decide (use default stopping criteria); `1` barrier stops with status not defined; `2` barrier stops with optimal status; `3` barrier stops with dual infeasible status; `4` barrier stops wih primal infeasible status;
`object`	A user-defined object to be passed to the callback function, `f_bariteration`.
`priority`	An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

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

```
// Barrier iteration callback
void XPRS_CC BarrierIterCallback(XPRSprob my_prob,
                  void *my_object, int *barrier_action) {
  int current_iteration;
  double PrimalObj, DualObj, Gap, PrimalInf, DualInf,
    ComplementaryGap;

  my_object_s *my = (my_object_s *) my_object;

  XPRSgetintattrib(my_prob, XPRS_BARITER, &current_iteration);

  // try to get all the solution values
  XPRSgetlpsol(my_prob, my->x, my->slacks, my->y, my->dj);

  XPRSgetdblattrib(my_prob, XPRS_BARPRIMALOBJ, &PrimalObj);
  XPRSgetdblattrib(my_prob, XPRS_BARDUALOBJ, &DualObj);
  Gap = DualObj - PrimalObj;
  XPRSgetdblattrib(my_prob, XPRS_BARPRIMALINF, &PrimalInf);
```

```
    XPRSgetdblattrib(my_prob, XPRS_BARDUALINF, &DualInf);
    XPRSgetdblattrib(my_prob, XPRS_BARCGAP, &ComplementaryGap);

    // decide if stop or continue
    *barrier_action = BARRIER_CHECKSTOPPING;
    if (current_iteration >= 50
        || Gap <= 0.1*max(fabs(PrimalObj),fabs(DualObj))) {
      *barrier_action = BARRIER_OPTIMAL;
    }
  }

  // To set callback:
  XPRSaddcbbariteration(xprob, BarrierIterCallback, (void *) &my, 0);
```

Further information

1. Only the following functions are expected to be called from the callback: `XPRSgetlpsol` and the attribute/control value retrieving and setting routines.
2. General barrier iteration values are available by using `XPRSgetdblattrib` to retrieve:
   - `BARPRIMALOBJ` - current primal objective
   - `BARDUALOBJ` - current dual objective
   - `BARPRIMALINF` - current primal infeasibility
   - `BARDUALINF` - current dual infeasibility
   - `BARCGAP` - current complementary gap
3. Please note that these values refer to the scaled and presolved problem used by barrier, and may differ from the ones calculated from the postsolved solution returned by `XPRSgetlpsol`.

Related topics

`XPRSremovecbbariteration`.

XPRSaddcbbarlog

Purpose

Declares a barrier log callback function, called at each iteration during the interior point algorithm. This callback function will be called in addition to any barrier log callbacks already added by XPRSaddcbbarlog.

Synopsis

```
int XPRS_CC XPRSaddcbbarlog (XPRSprob prob, int (XPRS_CC
      *f_barlog)(XPRSprob my_prob, void *my_object), void *object, int
      priority);
```

Arguments

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

Example

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

```
XPRSaddcbbarlog(prob,barLog,NULL,0);
XPRSlpoptimize(prob,"b");
```

The callback function might resemble:

```
int XPRS_CC barLog(XPRSprob prob, void *object)
{
  printf("Next barrier iteration\n");
}
```

Further information

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

Related topics

`XPRSremovecbbarlog`, `XPRSaddcbgloballog`, `XPRSaddcblplog`, `XPRSaddcbmessage`.

XPRSaddcbcomputerestart

Purpose

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

Synopsis

```
int XPRS_CC XPRSaddcbcomputerestart (XPRSprob prob, void (XPRS_CC
      *f_computerestart)(XPRSprob my_prob, void *my_object), void *object,
      int priority);
```

Arguments

`prob`	The current problem.
`f_presolve`	The callback function itself. This takes two arguments, `my_prob` and `my_object`, and has an integer return value. This function is called when a solve had to be restarted in compute mode.
`my_prob`	The problem passed to the callback function, `f_computerestart`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `XPRSaddcbcomputerestart`.
`object`	A user-defined object to be passed to the callback function, `f_computerestart`.
`priority`	An integer that determines the order in which multiple computerestart callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Related topics

`XPRSremovecbcomputerestart`

XPRSaddcbpresolve

Purpose

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

Synopsis

```
int XPRS_CC XPRSaddcbpresolve (XPRSprob prob, void (XPRS_CC
      *f_presolve)(XPRSprob my_prob, void *my_object), void *object, int
      priority);
```

Arguments

`prob`	The current problem.
`f_presolve`	The callback function itself. This takes two arguments, `my_prob` and `my_object`, and has an integer return value. This function is called after presolve is complete.
`my_prob`	The problem passed to the callback function, `f_barlog`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `XPRSaddcbpresolve`.
`object`	A user-defined object to be passed to the callback function, `f_presolve`.
`priority`	An integer that determines the order in which multiple presolve callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Related topics

XPRSremovecbpresolve

XPRSaddcbchecktime

Purpose

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

Synopsis

```
int XPRS_CC XPRSaddcbchecktime(XPRSprob prob, int (XPRS_CC
      *f_checktime)(XPRSprob my_prob, void* my_object), void* object, int
      priority);
```

Arguments

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

Further information

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

Related topics

`XPRSremovecbchecktime` `MAXTIME` `CHECKSONMAXTIME` `MAXCHECKSONMAXTIME`

XPRSaddcbchgbranch

Purpose

This function is deprecated and may be removed in future releases. Please use `XPRSaddcbchgbranchobject` instead.

Declares a branching variable callback function, called every time a new branching variable is set or selected during the branch and bound search. This callback function will be called in addition to any change branch callbacks already added by XPRSaddcbchgbranch.

Synopsis

```
int XPRS_CC XPRSaddcbchgbranch(XPRSprob prob, void (XPRS_CC
       *f_chgbranch)(XPRSprob my_prob, void *my_object, int *entity, int
       *up, double *estdeg), void *object, int priority);
```

Arguments

`prob`	The current problem.
`f_chgbranch`	The callback function, which takes five arguments, `my_prob`, `my_object`, `entity`, `up` and `estdeg`, and has no return value. This function is called every time a new branching variable or set is selected.
`my_prob`	The problem passed to the callback function, `f_chgbranch`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `XPRSaddcbchgbranch`.
`entity`	A pointer to the variable or set on which to branch. Ordinary global variables are identified by their column index, i.e. `0`, `1`,...(`COLS- 1`) and by their set index, i.e. `0`, `1`,...,(`SETS- 1`).
`up`	If `entity` is a variable, this is `1` if the upward branch is to be made first, or `0` otherwise. If `entity` is a set, this is `3` if the upward branch is to be made first, or `2` otherwise.
`estdeg`	This value is obsolete. It will be set to zero and any returned value is ignored.
`object`	A user-defined object to be passed to the callback function, `f_chgbranch`.
`priority`	An integer that determines the order in which multiple branching variable callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Further information

The arguments initially contain the default values of the branching variable, branching variable, branching direction and estimated degradation. If they are changed then the Optimizer will use the new values, if they are not changed then the default values will be used.

Related topics

`XPRSremovecbchgbranch`, `XPRSaddcbchgnode`, `XPRSaddcboptnode`, `XPRSaddcbinfnode`, `XPRSaddcbintsol`, `XPRSaddcbnodecutoff`, `XPRSaddcbprenode`.

XPRSaddcbchgbranchobject

Purpose

Declares a callback function that will be called after the selection of a global entity to branch on. This callback allows the user to inspect and replace the Optimizer's choice of how to branch the current node. This callback will also be called in the case when there are no candidates to branch on, that is, when all global entities are already satisfied. This callback function will be called in addition to any callbacks already added by XPRSaddcbchgbranchobject.

Synopsis

```
int XPRS_CC XPRSaddcbchgbranchobject(XPRSprob prob, void (XPRS_CC
      *f_chgbranchobject)(XPRSprob my_prob, void* my_object,
      XPRSbranchobject obranch, XPRSbranchobject* p_newobject), void*
      object, int priority);
```

Arguments

`prob`	The current problem.
`f_chgbranchobject`	The callback function, which takes four arguments: `my_prob`, `my_object`, `obranch` and `p_newobject`. This function is called every time the Optimizer has selected a candidate entity for branching.
`my_prob`	The problem passed to the callback function, `f_chgbranchobject`.
`my_object`	The user defined object passed as `object` when setting up the callback with `XPRSaddcbchgbranchobject`.
`obranch`	The candidate branching object selected by the Optimizer. Will be `NULL` if no candidates exist.
`p_newobject`	Optional new branching object to replace the Optimizer's selection. If `obranch` or `NULL` is passed back, no change will be applied.
`object`	A user-defined object to be passed to the callback function, `f_chgbranchobject`.
`priority`	An integer that determines the order in which multiple callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Further information

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

Related topics

`XPRSremovecbchgbranchobject`, `XPRS_bo_create`.

XPRSaddcbchgnode

Purpose

This function is deprecated and may be removed in future releases.

Declares a node selection callback function. This is called every time the code backtracks to select a new node during the MIP search. This callback function will be called in addition to any callbacks already added by XPRSaddcbchgnode.

Synopsis

```
int XPRS_CC XPRSaddcbchgnode(XPRSprob prob, void (XPRS_CC
      *f_chgnode)(XPRSprob my_prob, void *my_object, int *nodnum), void
      *object, int priority);
```

Arguments

`prob`	The current problem.
`f_chgnode`	The callback function which takes three arguments, `my_prob`, `my_object` and `nodnum`, and has no return value. This function is called every time a new node is selected.
`my_prob`	The problem passed to the callback function, `f_chgnode`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `XPRSaddcbchgnode`.
`nodnum`	A pointer to the number of the node, `nodnum`, selected by the Optimizer. By changing the value pointed to by this argument, the selected node may be changed with this function.
`object`	A user-defined object to be passed to the callback function, `f_chgnode`.
`priority`	An integer that determines the order in which multiple node selection callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Related controls

Integer

`NODESELECTION` Node selection control.

Example

The following prints out the node number every time a new node is selected during the global search:

```
XPRSsetintcontrol(prob,XPRS_MIPLOG,3);
XPRSsetintcontrol(prob,XPRS_NODESELECTION,2);
XPRSaddcbchgnode(prob,nodeSelection,NULL,0);
XPRSmipoptimize(prob,"");
```

The callback function may resemble:

```
XPRS_CC void nodeSelection(XPRSprob prob, void *object,
                           int *Node)
{
  printf("Node number %d\n", *Node);
}
```

See the example `depthfirst.c` in the `examples/optimizer/c` folder for an example of using a node selection callback.

Related topics

`XPRSremovecbchgnode`, `XPRSaddcboptnode`, `XPRSaddcbinfnode`, `XPRSaddcbintsol`, `XPRSaddcbnodecutoff`, `XPRSaddcbchgbranch`, `XPRSaddcbprenode`.

XPRSaddcbcutlog

Purpose

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

Synopsis

```
int XPRS_CC XPRSaddcbcutlog(XPRSprob prob, int (XPRS_CC *f_cutlog)(XPRSprob
      my_prob, void *my_object), void *object, int priority);
```

Arguments

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

Further information

Return a non-zero value from `f_cutlog` to stop cutting on the current node.

Related topics

`XPRSremovecbcutlog`, `XPRSaddcbcutmgr`.

XPRSaddcbcutmgr

Purpose

This function is deprecated and may be removed in future releases. Please use `XPRSaddcboptnode` *instead.*

Declares a user-defined cut manager routine, called at each node of the branch and bound search. This callback function will be called in addition to any callbacks already added by XPRSaddcbcutmgr.

Synopsis

```
int XPRS_CC XPRSaddcbcutmgr(XPRSprob prob, int (XPRS_CC *f_cutmgr)(XPRSprob
      my_prob, void *my_object), void *object, int priority);
```

Arguments

`prob`	The current problem
`f_cutmgr`	The callback function which takes two arguments, `my_prob` and `my_object`, and has an integer return value. This function is called at each node in the Branch and Bound search.
`my_prob`	The problem passed to the callback function, `f_cutmgr`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `XPRSaddcbcutmgr`.
`object`	A user-defined object to be passed to the callback function, `f_cutmgr`.
`priority`	An integer that determines the order in which multiple global log callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Further information

1. When returning from the user function `f_cutlog`, the Optimizer will automatically reoptimize the LP relaxation of the node problem. If a non-zero value is returned from `f_cutlog`, the function will be called again afterwards, unless the LP relaxation has become infeasible or was cut off due to the objective function value. Return `0` from `f_cutlog` to prevent the function from being called again for the same branch and bound node.

2. `f_cutlog` is called for a branch-and-bound node problem after the Optimizer has already applied any internal cuts and heuristics, but before determining if the node problem should be branched or if the node LP relaxation solution is MIP feasible.

3. The Optimizer ensures that cuts added to a node are automatically restored at descendant nodes. To do this, all cuts are stored in a cut pool and the Optimizer keeps track of which cuts from the cut pool must be restored at each node.

Related topics

`XPRSremovecbcutmgr`, `XPRSaddcbcutlog`, `CALLBACKCOUNT_CUTMGR`.

XPRSaddcbdestroymt

Purpose

Declares a destroy MIP thread callback function, called every time a MIP thread is destroyed by the parallel MIP code. This callback function will be called in addition to any callbacks already added by XPRSaddcbdestroymt.

Synopsis

```
int XPRS_CC XPRSaddcbdestroymt(XPRSprob prob, void (XPRS_CC
      *f_destroymt)(XPRSprob my_prob, void *my_object), void *object, int
      priority);
```

Arguments

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

Related controls

Integer

`MIPTHREADS`	Number of MIP threads to create.

Further information

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

Related topics

`XPRSremovecbdestroymt`,`XPRSaddcbmipthread`.

XPRSaddcbestimate

Purpose

This function is deprecated and may be removed in future releases. Please use branching objects instead.

Declares an estimate callback function. If defined, it will be called at each node of the branch and bound tree to determine the estimated degradation from branching the user's global entities. This callback function will be called in addition to any callbacks already added by XPRSaddcbestimate.

Synopsis

```
int XPRS_CC XPRSaddcbestimate(XPRSprob prob, int (XPRS_CC
        *f_estimate)(XPRSprob my_prob, void *my_object, int *iglsel, int
        *iprio, double *degbest, double *degworst, double *curval, int
        *ifupx, int *nglinf, double *degsum, int *nbr), void *object, int
        priority);
```

Arguments

`prob`	The current problem.
`f_estimate`	The callback function which takes eleven arguments, `my_prob`, `my_object`, `iglsel`, `iprio`, `degbest`, `degworst`, `curval`, `ifupx`, `nglinf`, `degsum` and `nbr`, and has an integer return value. This function is called at each node of the branch and bound search.
`my_prob`	The problem passed to the callback function, `f_estimate`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `XPRSaddcbestimate`.
`iglsel`	Selected user global entity. Must be non-negative or -1 to indicate that there is no user global entity candidate for branching. If set to -1, all other arguments, except for `nglinf` and `degsum` are ignored. This argument is initialized to -1.
`iprio`	Priority of selected user global entity. This argument is initialized to a value larger (i.e., lower priority) than the default priority for global entities (see Section 4.3.3 in Section 4.3).
`degbest`	Estimated degradation from branching on selected user entity in preferred direction.
`degworst`	Estimated degradation from branching on selected user entity in worst direction.
`curval`	Current value of user global entities.
`ifupx`	Preferred branch on user global entity (`0`,...,`nbr−1`).
`nglinf`	Number of infeasible user global entities.
`degsum`	Sum of estimated degradations of satisfying all user entities.
`nbr`	Number of branches. The user separate routine (set up with `XPRSaddcbsepnode`) will be called `nbr` times in order to create the actual branches.
`object`	A user-defined object to be passed to the callback function, `f_estimate`.
`priority`	An integer that determines the order in which multiple estimate callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Further information

Consider using the more flexible *branching objects*, as described for the `XPRS_bo_create` function.

Related topics

`XPRSremovecbestimate`, `XPRSsetbranchcuts`, `XPRSaddcbsepnode`, `XPRS_bo_create`.

XPRSaddcbgapnotify

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

```
int XPRS_CC XPRSaddcbgapnotify(XPRSprob prob, void (XPRS_CC
      *f_gapnotify)(XPRSprob my_prob, void* my_object, double*
      newRelGapNotifyTarget, double* newAbsGapNotifyTarget, double*
      newAbsGapNotifyObjTarget, double* newAbsGapNotifyBoundTarget), void*
      object, int priority);
```

Arguments

`prob` The current problem.

`f_gapnotify` The callback function.

`my_prob` The current problem.

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

`newRelGapNotifyTarget` 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.

`newAbsGapNotifyTarget` 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.

`newAbsGapNotifyObjTarget` 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.

`newAbsGapNotifyBoundTarget` The value the MIPABSGAPNOTIFYBOUND control will be set to after this callback. May be modified within the callback in order to set a new notification target.

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

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

Example

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

```
void XPRS_CC gapnotify(XPRSprob prob, void* object,
  double* newRelGapNotifyTarget, double* newAbsGapNotifyTarget,
  double* newAbsGapNotifyObjTarget, double* newAbsGapNotifyBoundTarget)
{
  double obj, bound, relgap;
  XPRSgetdblattrib(prob, XPRS_MIPOBJVAL, &obj);
  XPRSgetdblattrib(prob, XPRS_BESTBOUND, &bound);
  if (obj != 0.0 || bound != 0.0)
    relgap = fabs((obj - bound)/ max(fabs(obj), fabs(bound)));
  else
    relgap = 0.0;
  if (relgap<=0.10) {
    printf("Gap reached 10%");
    *newRelGapNotifyTarget = 0.1;
  }
  if (relgap<=0.01) {
    printf("Gap reached 1%");
```

```
      *newRelGapNotifyTarget = -1; /* Don't call gapnotify again */
    }
  }

  XPRSsetdblcontrol( prob, XPRS_MIPRELGAPNOTIFY, 0.10 );
  XPRSaddcbgapnotify( prob, gapnotify, NULL, 0 );
  XPRSmipoptimize(prob, "");
```

Further information

The target values that caused the callback to be triggered will automatically be reset to prevent the same callback from being fired again.

Related topics

`MIPRELGAPNOTIFY`, `MIPABSGAPNOTIFY`, `MIPABSGAPNOTIFYOBJ`, `MIPABSGAPNOTIFYBOUND`, `XPRSremovecbgapnotify`.

XPRSaddcbgloballog

Purpose

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

Synopsis

```
int XPRS_CC XPRSaddcbgloballog(XPRSprob prob, int (XPRS_CC
      *f_globallog)(XPRSprob my_prob, void *my_object), void *object, int
      priority);
```

Arguments

`prob`	The current problem.
`f_globallog`	The callback function which takes two arguments, `my_prob` and `my_object`, and has an integer return value. This function is called whenever the global log is printed as determined by the `MIPLOG` control.
`my_prob`	The problem passed to the callback function, `f_globallog`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `XPRSaddcbgloballog`.
`object`	A user-defined object to be passed to the callback function, `f_globallog`.
`priority`	An integer that determines the order in which multiple global log callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Related controls

Integer

`MIPLOG`	Global print flag.

Example

The following example prints at each node of the global search the node number and its depth:

```
XPRSsetintcontrol(prob, XPRS_MIPLOG, 3);
XPRSaddcbgloballog(prob, globalLog, NULL, 0);
XPRSmipoptimize(prob,"");
```

The callback function may resemble:

```
int XPRS_CC globalLog(XPRSprob prob, void *object)
{
  int node, nodedepth;

  XPRSgetintattrib(prob, XPRS_NODEDEPTH, &nodedepth);
  XPRSgetintattrib(prob, XPRS_CURRENTNODE, &node);
  printf("Node %d with depth %d has just been processed\n",
         node, nodedepth);

  return 0;
}
```

See the example `depthfirst.c` in the `examples/optimizer/c` folder.

Further information

If the callback function returns a nonzero value, the global search will be interrupted.

Related topics

`XPRSremovecbgloballog`, `XPRSaddcbbarlog`, `XPRSaddcblplog`, `XPRSaddcbmessage`.

XPRSaddcbinfnode

Purpose

Declares a user infeasible node callback function, called after the current node has been found to be infeasible during the Branch and Bound search. This callback function will be called in addition to any callbacks already added by XPRSaddcbinfnode.

Synopsis

```
int XPRS_CC XPRSaddcbinfnode(XPRSprob prob, void (XPRS_CC
      *f_infnode)(XPRSprob my_prob, void *my_object), void *object, int
      priority);
```

Arguments

`prob`	The current problem
`f_infnode`	The callback function which takes two arguments, `my_prob` and `my_object`, and has no return value. This function is called after the current node has been found to be infeasible.
`my_prob`	The problem passed to the callback function, `f_infnode`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `XPRSaddcbinfnode`.
`object`	A user-defined object to be passed to the callback function, `f_infnode`.
`priority`	An integer that determines the order in which multiple user infeasible node callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following notifies the user whenever an infeasible node is found during the global search:

```
XPRSaddcbinfnode(prob,nodeInfeasible,NULL,0);
XPRSmipoptimize(prob,"");
```

The callback function may resemble:

```
void XPRS_CC nodeInfeasible(XPRSprob prob, void *object)
{
  int node;
  XPRSgetintattrib(prob, XPRS_CURRENTNODE, &node);
  printf("Node %d infeasible\n", node);
}
```

See the example `depthfirst.c` in the `examples/optimizer/c` folder.

Related topics

`XPRSremovecbinfnode`, `XPRSaddcboptnode`, `XPRSaddcbintsol`, `XPRSaddcbnodecutoff`.

XPRSaddcbintsol

Purpose

Declares a user integer solution callback function, called every time an integer solution is found by heuristics or during the Branch and Bound search. This callback function will be called in addition to any callbacks already added by XPRSaddcbintsol.

Synopsis

```
int XPRS_CC XPRSaddcbintsol(XPRSprob prob, void (XPRS_CC
      *f_intsol)(XPRSprob my_prob, void *my_object), void *object, int
      priority);
```

Arguments

`prob`	The current problem.
`f_intsol`	The callback function which takes two arguments, `my_prob` and `my_object`, and has no return value. This function is called if the current node is found to have an integer feasible solution, i.e. every time an integer feasible solution is found.
`my_prob`	The problem passed to the callback function, `f_intsol`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `XPRSaddcbintsol`.
`object`	A user-defined object to be passed to the callback function, `f_intsol`.
`priority`	An integer that determines the order in which multiple integer solution callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following example prints integer solutions as they are discovered in the global search:

```
XPRSaddcbintsol(prob,printsol,NULL,0);
XPRSmipoptimize(prob,"");
```

The callback function might resemble:

```
void XPRS_CC printsol(XPRSprob my_prob, void *object)
{
  int i, cols;
  double objval, *x;

  XPRSgetintattrib(my_prob, XPRS_ORIGINALCOLS, &cols);
  XPRSgetdblattrib(my_prob, XPRS_LPOBJVAL, &objval);
  x = malloc(cols * sizeof(double));
  if (!x) return;
  XPRSgetlpsol(my_prob, x, NULL, NULL, NULL);

  printf("\nInteger solution found: %f\n", objval);
  for(i=0;i<cols;i++) printf(" x[%d] = %d\n", i, x[i]);
  free(x);
}
```

Further information

1. This callback is useful if the user wants to retrieve the integer solution when it is found.
2. To retrieve the integer solution, use either `XPRSgetlpsol` or `XPRSgetpresolvesol`. `XPRSgetmipsol` always returns the last integer solution found and, if called from the `intsol` callback, it will not necessarily return the solution that caused the invocation of the callback (for example, it is possible that when solving with multiple MP threads, another thread finds a new integer solution before the user calls `XPRSgetmipsol`).
3. This callback is called after a new integer solution was found by the Optimizer. Use a callback set by `XPRSaddcbpreintsol` in order to be notified before a new integer solution is accepted by the Optimizer, which allows for the new solution to be rejected.

Related topics

`XPRSremovecbintsol`, `XPRSaddcbpreintsol`.

XPRSaddcblplog

Purpose

Declares a simplex log callback function which is called after every `LPLOG` iterations of the simplex algorithm. This callback function will be called in addition to any callbacks already added by XPRSaddcblplog.

Synopsis

```
int XPRS_CC XPRSaddcblplog(XPRSprob prob, int (XPRS_CC *f_lplog)(XPRSprob
      my_prob, void* my_object), void* object, int priority);
```

Arguments

`prob`	The current problem.
`f_lplog`	The callback function which takes two arguments, `my_prob` and `my_object`, and has an integer return value. This function is called every `LPLOG` simplex iterations including iteration `0` and the final iteration.
`my_prob`	The problem passed to the callback function, `f_lplog`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `XPRSaddcblplog`.
`object`	A user-defined object to be passed to the callback function, `f_lplog`.
`priority`	An integer that determines the order in which multiple lplog callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Related controls

Integer

`LPLOG`	Frequency and type of simplex algorithm log.

Example

The following code sets a callback function, `lpLog`, to be called every `10` iterations of the optimization:

```
XPRSsetintcontrol(prob,XPRS_LPLOG,10);
XPRSaddcblplog(prob,lpLog,NULL,0);
XPRSreadprob(prob,"problem","");
XPRSmipoptimize(prob,"");
```

The callback function may resemble:

```
int XPRS_CC lpLog(XPRSprob my_prob, void *object)
{
  int iter; double obj;

  XPRSgetintattrib(my_prob, XPRS_SIMPLEXITER, &iter);
  XPRSgetdblattrib(my_prob, XPRS_LPOBJVAL, &obj);
  printf("At iteration %d objval is %g\n", iter, obj);
  return 0;
}
```

Further information

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

Related topics

`XPRSremovecblplog`, `XPRSaddcbbarlog`, `XPRSaddcbgloballog`, `XPRSaddcbmessage`.

XPRSaddcbmessage

Purpose

Declares an output callback function, called every time a text line relating to the given XPRSprob is output by the Optimizer. This callback function will be called in addition to any callbacks already added by XPRSaddcbmessage.

Synopsis

```
int XPRS_CC XPRSaddcbmessage(XPRSprob prob, void (XPRS_CC
        *f_message)(XPRSprob my_prob, void *my_object, const char *msg, int
        len, int msgtype), void *object, int priority);
```

Arguments

`prob`	The current problem.
`f_message`	The callback function which takes five arguments, `my_prob`, `my_object`, `msg`, `len` and `msgtype`, and has no return value. Use a `NULL` value to cancel a callback function.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed as `object` when setting up the callback with `XPRSaddcbmessage`.
`msg`	A null terminated character array (string) containing the message, which may simply be a new line.
`len`	The length of the message string, excluding the null terminator.
`msgtype`	Indicates the type of output message: `1` information messages; `2` (not used); `3` warning messages; `4` error messages. A negative value indicates that the Optimizer is about to finish and the buffers should be flushed at this time if the output is being redirected to a file.
`object`	A user-defined object to be passed to the callback function.
`priority`	An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Related controls

Integer

`OUTPUTLOG` All messages are disabled if set to zero.

Example

The following example simply sends all output to the screen (`stdout`):

```
XPRSaddcbmessage(prob,Message,NULL,0);
```

The callback function might resemble:

```
void XPRS_CC Message(XPRSprob my_prob, void* object,
                     const char *msg, int len, int msgtype)
{
  switch(msgtype)
  {
    case 4:  /* error */
    case 3:  /* warning */
    case 2:  /* not used */
    case 1:  /* information */
             printf("%s\n", msg);
```

```
                  break;
         default: /* exiting - buffers need flushing */
                  fflush(stdout);
                  break;
       }
     }
```

Further information

1. Screen output is automatically created by the Optimizer Console only. To produce output when using the Optimizer library, it is necessary to define this callback function and use it to print the messages to the screen (`stdout`).

2. This function offers one method of handling the messages which describe any warnings and errors that may occur during execution. Other methods are to check the return values of functions and then get the error code using the `ERRORCODE` attribute, obtain the last error message directly using `XPRSgetlasterror`, or send messages direct to a log file using `XPRSsetlogfile`.

3. Visual Basic users must use the alternative function `XPRSaddcbmessageVB` to define the callback; this is required because of the different way VB handles strings.

Related topics

`XPRSremovecbmessage`, `XPRSaddcbbarlog`, `XPRSaddcbgloballog`, `XPRSaddcblplog`, `XPRSsetlogfile`.

XPRSaddcbmipthread

Purpose

Declares a MIP thread callback function, called every time a MIP worker problem is created by the parallel MIP code. This callback function will be called in addition to any callbacks already added by XPRSaddcbmipthread.

Synopsis

```
int XPRS_CC XPRSaddcbmipthread(XPRSprob prob, void (XPRS_CC
       *f_mipthread)(XPRSprob my_prob, void *my_object, XPRSprob
       thread_prob), void *object, int priority);
```

Arguments

`prob`	The current problem.
`f_mipthread`	The callback function which takes three arguments, `my_prob`, `my_object` and `thread_prob`, and has no return value.
`my_prob`	The problem passed to the callback function.
`my_object`	The user-defined object passed to the callback function.
`thread_prob`	The problem pointer for the MIP thread
`object`	A user-defined object to be passed to the callback function.
`priority`	An integer that determines the order in which multiple callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Related controls

Integer

`MIPTHREADS` Number of MIP threads to create.

Example

The following example clears the message callback for each of the MIP threads:

```
XPRSaddcbmipthread(prob,mipthread,NULL, 0);

void XPRS_CC mipthread(XPRSprob my_prob, void* my_object,
                       XPRSprob mipthread)
{
  /* clear the message callback*/
  XPRSremovecbmessage(my_prob, mipthread, NULL);

}
```

Further information

This function will be called when a new MIP worker problem is created. Each worker problem receives a unique identifier that can be obtained through the `MIPTHREADID` attribute. Worker problems can be matched with different system threads at different points of a solve, so the system thread that is responsible for executing the callback is not necessarily the same thread used for all subsequent callbacks for the same worker problem. On the other hand, worker problems are always assigned to a single thread at a time and the same nodes are always solved on the same worker problem in repeated runs of a deterministic MIP solve. A worker problem therefore acts as a virtual thread through the node solves.

Related topics

`XPRSremovecbmipthread`,`XPRSaddcbdestroymt`, `MIPTHREADS`, `MAXMIPTASKS`.

XPRSaddcbnewnode

Purpose

Declares a callback function that will be called every time a new node is created during the branch and bound search. This callback function will be called in addition to any callbacks already added by XPRSaddcbnewnode.

Synopsis

```
int XPRS_CC XPRSaddcbnewnode(XPRSprob prob, void (XPRS_CC
      *f_newnode)(XPRSprob my_prob, void* my_object, int parentnode, int
      newnode, int branch), void* object, int priority);
```

Arguments

`prob`	The current problem.
`f_newnode`	The callback function, which takes five arguments: `myprob`, `my_object`, `parentnode`, `newnode` and `branch`. This function is called every time a new node is created through branching.
`my_prob`	The problem passed to the callback function, `f_newnode`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `XPRSaddcbnewnode`.
`parentnode`	Unique identifier for the parent of the new node.
`newnode`	Unique identifier assigned to the new node.
`branch`	The sequence number of the new node amongst the child nodes of `parentnode`. For regular branches on a global entity this will be either `0` or `1`.
`object`	A user-defined object to be passed to the callback function.
`priority`	An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Further information

1. For regular branches on a global entity, `branch` will be either zero or one, depending on whether the new node corresponds to branching the global entity up or down.
2. When branching on an `XPRSbranchobject`, `branch` refers to the given branch index of the object.
3. For new nodes created using the `XPRSaddcbestimate`/`XPRSaddcbsepnode` callback functions, branch is identical to the `ifup` argument of the `XPRSaddcbsepnode` callback function.

Related topics

`XPRSremovecbnewnode`, `XPRSaddcbchgnode`.

XPRSaddcbnodecutoff

Purpose

Declares a user node cutoff callback function, called every time a node is cut off as a result of an improved integer solution being found during the branch and bound search. This callback function will be called in addition to any callbacks already added by XPRSaddcbnodecutoff.

Synopsis

```
int XPRS_CC XPRSaddcbnodecutoff(XPRSprob prob, void (XPRS_CC
       *f_nodecutoff)(XPRSprob my_prob, void *my_object, int node), void
       *object, int priority);
```

Arguments

`prob`	The current problem.
`f_nodecutoff`	The callback function, which takes three arguments, `my_prob`, `my_object` and `node`, and has no return value. This function is called every time a node is cut off as the result of an improved integer solution being found.
`my_prob`	The problem passed to the callback function, `f_nodecutoff`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `XPRSaddcbnodecutoff`.
`node`	The number of the node that is cut off.
`object`	A user-defined object to be passed to the callback function, `f_nodecutoff`.
`priority`	An integer that determines the order in which multiple node-optimal callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following notifies the user whenever a node is cutoff during the global search:

```
XPRSaddcbnodecutoff(prob,Cutoff,NULL,0);
XPRSmipoptimize(prob,"");
```

The callback function might resemble:

```
void XPRS_CC Cutoff(XPRSprob prob, void *object, int node)
{
  printf("Node %d cutoff\n", node);
}
```

See the example `depthfirst.c` in the `examples/optimizer/c` folder.

Further information

This function allows the user to keep track of the eligible nodes. Note that the LP solution will not be available from this callback.

Related topics

`XPRSremovecbnodecutoff`, `XPRSaddcboptnode`, `XPRSaddcbinfnode`, `XPRSaddcbintsol`.

XPRSaddcboptnode

Purpose

Declares an optimal node callback function, called during the branch and bound search, after the LP relaxation has been solved for the current node, and after any internal cuts and heuristics have been applied, but before the Optimizer checks if the current node should be branched. This callback function will be called in addition to any callbacks already added by XPRSaddcboptnode.

Synopsis

```
int XPRS_CC XPRSaddcboptnode(XPRSprob prob, void (XPRS_CC
      *f_optnode)(XPRSprob my_prob, void *my_object, int *feas), void
      *object, int priority);
```

Arguments

`prob`	The current problem.
`f_optnode`	The callback function which takes three arguments, `my_prob`, `my_object` and `feas`, and has no return value.
`my_prob`	The problem passed to the callback function, `f_optnode`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `XPRSaddcboptnode`.
`feas`	The feasibility status. If set to a nonzero value by the user, the current node will be declared infeasible.
`object`	A user-defined object to be passed to the callback function, `f_optnode`.
`priority`	An integer that determines the order in which multiple node-optimal callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following prints the optimal objective value of the node LP relaxations:

```
XPRSaddcboptnode(prob,nodeOptimal,NULL,0);
XPRSmipoptimize(prob,"");
```

The callback function might resemble:

```
void XPRS_CC nodeOptimal(XPRSprob prob, void *object, int *feas)
{
  int node;
  double objval;

  XPRSgetintattrib(prob, XPRS_CURRENTNODE, &node);
  printf("NodeOptimal: node number %d\n", node);
  XPRSgetdblattrib(prob, XPRS_LPOBJVAL, &objval);
  printf("\tObjective function value = %f\n", objval);
}
```

See the example `depthfirst.c` in the `examples/optimizer/c` folder.

Related topics

`XPRSremovecboptnode`, `XPRSaddcbinfnode`, `XPRSaddcbintsol`, `XPRSaddcbnodecutoff`, `CALLBACKCOUNT_OPTNODE`.

XPRSaddcbpreintsol

Purpose

Declares a user integer solution callback function, called when an integer solution is found by heuristics or during the branch and bound search, but before it is accepted by the Optimizer. This callback function will be called in addition to any integer solution callbacks already added by XPRSaddcbpreintsol.

Synopsis

```
int XPRS_CC XPRSaddcbpreintsol(XPRSprob prob, void (XPRS_CC
      *f_preintsol)(XPRSprob my_prob, void *my_object, int soltype, int
      *ifreject, double *cutoff), void *object, int priority);
```

Arguments

`prob` The current problem.

`f_preintsol` The callback function which takes five arguments, `my_prob`, `my_object`, `soltype`, `ifreject` and `cutoff`, and has no return value. This function is called when an integer solution is found, but before the solution is accepted by the Optimizer, allowing the user to reject the solution.

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

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

`soltype` The type of MIP solution that has been found: Set to 1 if the solution was found using a heuristic. Otherwise, it will be the global feasible solution to the current node of the global search.

- `0` The continuous relaxation solution to the current node of the global search, which has been found to be global feasible.
- `1` A MIP solution found by a heuristic.
- `2` A MIP solution provided by the user.
- `3` A solution resulting from refinement of primal or dual violations of a previous MIP solution.

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

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

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

Related controls

Integer

`MIPABSCUTOFF` Branch and Bound: If the user knows that they are interested only in values of the objective function which are better than some value, this can be assigned to MIPABSCUTOFF. This allows the Optimizer to ignore solving any nodes which may yield worse objective values, saving solution time. When a MIP solution is found a new cut off value is calculated and the value can be obtained from the CURRMIPCUTOFF attribute. The value of CURRMIPCUTOFF is calculated using the MIPRELCUTOFF and MIPADDCUTOFF controls.

Further information

1. If a solution is rejected, the Optimizer will drop the found solution without updating any attributes, including the cutoff value. To change the cutoff value when rejecting a solution, the control `MIPABSCUTOFF` should be set instead.
2. When a node solution (`soltype = 0`) is rejected, the node itself will be dropped without further branching.
3. To retrieve the integer solution, use either `XPRSgetlpsol` or `XPRSgetpresolvesol`. `XPRSgetmipsol` will not return the newly found solution because it has not been saved at this point.

Related topics

`XPRSremovecbpreintsol`, `XPRSaddcbintsol`.

XPRSaddcbprenode

Purpose

Declares a preprocess node callback function, called before the LP relaxation of a node has been optimized, so the solution at the node will not be available. This callback function will be called in addition to any callbacks already added by XPRSaddcbprenode.

Synopsis

```
int XPRS_CC XPRSaddcbprenode(XPRSprob prob, void (XPRS_CC
      *f_prenode)(XPRSprob my_prob, void *my_object, int *nodinfeas), void
      *object, int priority);
```

Arguments

`prob`	The current problem.
`f_prenode`	The callback function, which takes three arguments, `my_prob`, `my_object` and `nodinfeas`, and has no return value. This function is called before a node is reoptimized and the node may be made infeasible by setting `*nodinfeas` to `1`.
`my_prob`	The problem passed to the callback function, `f_prenode`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `XPRSaddcbprenode`.
`nodinfeas`	The feasibility status. If set to a nonzero value by the user, the current node will be declared infeasible by the Optimizer.
`object`	A user-defined object to be passed to the callback function, `f_prenode`.
`priority`	An integer that determines the order in which multiple preprocess node callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

The following example notifies the user before each node is processed:

```
XPRSaddcbprenode(prob, preNode, NULL, 0);
XPRSmipoptimize(prob,"");
```

The callback function might resemble:

```
void XPRS_CC preNode(XPRSprob prob, void* object, int *nodinfeas)
{
 *nodinfeas = 0; /* set to 1 if node is infeasible */
}
```

Related topics

`XPRSremovecbprenode`, `XPRSaddcbchgnode`, `XPRSaddcbinfnode`, `XPRSaddcbintsol`, `XPRSaddcbnodecutoff`, `XPRSaddcboptnode`.

XPRSaddcbsepnode

Purpose

This function is deprecated and may be removed in future releases. Please use branching objects instead.

Declares a separate callback function to specify how to branch on a node in the branch and bound tree using a global object. A node can be branched by applying either cuts or bounds to each node. These are stored in the cut pool. This callback function will be called in addition to any callbacks already added by XPRSaddcbsepnode.

Synopsis

```
int XPRS_CC XPRSaddcbsepnode(XPRSprob prob, int (XPRS_CC
      *f_sepnode)(XPRSprob my_prob, void *my_object, int ibr, int iglsel,
      int ifup, double curval), void *object, int priority);
```

Arguments

`prob`	The current problem.
`f_sepnode`	The callback function, which takes six arguments, `my_prob`, `my_object`, `ibr`, `iglsel`, `ifup` and `curval`, and has an integer return value.
`my_prob`	The problem passed to the callback function, `f_sepnode`.
`my_object`	The user-defined object passed as `object` when setting up the callback with `XPRSaddcbsepnode`.
`ibr`	The branch number.
`iglsel`	The global entity number.
`ifup`	The direction of branch on the global entity (same as `ibr`).
`curval`	Current value of the global entity.
`object`	A user-defined object to be passed to the callback function, `f_sepnode` .
`priority`	An integer that determines the order in which callbacks of this type will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to 0 if not required.

Example

This example solves a MIP, using a separation callback function to branch on fractional integer variables. It assumes the presence of an estimation callback function (not shown), defined by `XPRSaddcbestimate`, to identify a fractional integer variable.

```
XPRSaddcbsepnode(prob,nodeSep,NULL,0);
XPRSmipoptimize(prob,"");
```

where the function `nodeSep` may be defined as follows:

```
int nodeSep(XPRSprob my_prob, void *my_object, int ibr,
             int iglsel, int ifup, double curval)
 {
   XPRScut index;
   double dbd;

   if( ifup )
   {
     dbd = floor(xval);
     XPRSstorebounds(my_prob, 1, &iglsel, "U", &dbd, &index);
   }
   else
   {
     dbd = ceil(xval);
```

```
      XPRSstorebounds(my_prob, 1, &iglsel, "L", &dbd, &index);
    }
    XPRSsetbranchbounds(prob, index);
    return 0;
  }
```

Further information

1. The return value of the `f_sepnode` callback function is currently ignored.
2. Consider using the more flexible *branching objects*, as described for the `XPRS_bo_create` function.
3. The user separate routine is called `nbr` times where `nbr` is returned by the estimate callback function, `XPRSaddcbestimate`. This allows multi-way branching to be performed.
4. The bounds and/or cuts to be applied at a node must be specified in the user separate routine by calling `XPRSsetbranchbounds` and/or `XPRSsetbranchcuts`.

Related topics

`XPRSremovecbsepnode`, `XPRSsetbranchbounds`, `XPRSsetbranchcuts`, `XPRSaddcbestimate`, `XPRSstorebounds`, `XPRSstorecuts`.

XPRSaddcbusersolnotify

Purpose

Declares a callback function to be called each time a solution added by `XPRSaddmipsol` has been processed. This callback function will be called in addition to any callbacks already added by `XPRSaddcbusersolnotify`.

Synopsis

```
int XPRS_CC XPRSaddcbusersolnotify(XPRSprob prob, void (XPRS_CC
      *f_usersolnotify)(XPRSprob my_prob, void* my_object, const char*
      solname, int status), void* object, int priority);
```

Arguments

`prob`	The current problem.
`f_usersolnotify`	The callback function which takes four arguments, `my_prob`, `my_object`, `solname` and `status` and has no return value.
`my_prob`	The problem passed to the callback function, `f_usersolnotify`.
`my_object`	The user-defined object passed as object when setting up the callback with `XPRSaddcbusersolnotify`.
`solname`	The string name assigned to the solution when it was loaded into the Optimizer using `XPRSaddmipsol`.
`status`	One of the following status values: 0 An error occurred while processing the solution. 1 Solution is feasible. 2 Solution is feasible after reoptimizing with fixed globals. 3 A local search heuristic was applied and a feasible solution discovered. 4 A local search heuristic was applied but a feasible solution was not found. 5 Solution is infeasible and a local search could not be applied. 6 Solution is partial and a local search could not be applied. 7 Failed to reoptimize the problem with globals fixed to the provided solution. Likely because a time or iteration limit was reached. 8 Solution is dropped. This can happen if the MIP problem is changed or solved to completion before the solution could be processed.
`object`	A user-defined object to be passed to the callback function, `f_usersolnotify`.
`priority`	An integer that determines the order in which multiple callbacks will be invoked. The callback added with a higher priority will be called before a callback with a lower priority. Set to `0` if not required.

Further information

If presolve is turned on, any solution added with `XPRSaddmipsol` will first be presolved before it can be checked. The value returned in `status` refers to the presolved solution, which might have had values adjusted due to bound changes, fixing of variables, etc.

Related topics

`XPRSremovecbusersolnotify`, `XPRSaddmipsol`.

XPRSaddcols, XPRSaddcols64

Purpose

Allows columns to be added to the matrix after passing it to the Optimizer using the input routines.

Synopsis

```
int XPRS_CC XPRSaddcols(XPRSprob prob, int newcol, int newnz, const double
      objx[], const int mstart[], const int mrwind[], const double
      dmatval[], const double bdl[], const double bdu[]);

int XPRS_CC XPRSaddcols64(XPRSprob prob, int newcol, XPRSint64 newnz, const
      double objx[], const XPRSint64 mstart[], const int mrwind[], const
      double dmatval[], const double bdl[], const double bdu[]);
```

Arguments

`prob`	The current problem.
`newcol`	Number of new columns.
`newnz`	Number of new nonzeros in the added columns.
`objx`	Double array of length `newcol` containing the objective function coefficients of the new columns.
`mstart`	Integer array of length `newcol` containing the offsets in the `mrwind` and `dmatval` arrays of the start of the elements for each column.
`mrwind`	Integer array of length `newnz` containing the row indices for the elements in each column.
`dmatval`	Double array of length `newnz` containing the element values.
`bdl`	Double array of length `newcol` containing the lower bounds on the added columns.
`bdu`	Double array of length `newcol` containing the upper bounds on the added columns.

Related controls

Integer

`EXTRACOLS`	Number of extra columns to be allowed for.
`EXTRAELEMS`	Number of extra matrix elements to be allowed for.
`EXTRAMIPENTS`	Number of extra global entities to be allowed for.

Double

`MATRIXTOL`	Tolerance on matrix elements.

Example

In this example, we consider the two problems:

(a)	maximize:	$2x + y$			(b)	maximize:	$2x + y + 3z$		
	subject to:	$x + 4y$	$\leq$	24		subject to:	$x + 4y + 2z$	$\leq$	24
		y	$\leq$	5			$y + z$	$\leq$	5
		$3x + y$	$\leq$	20			$3x + y$	$\leq$	20
		$x + y$	$\leq$	9			$x + y + 3z$	$\leq$	9
							z	$\leq$	12

Using `XPRSaddcols`, the following transforms (a) into (b) and then names the new variable using `XPRSaddnames`:

```
obj[0] = 3;
mstart[] = {0};
mrwind[] = {0, 1, 3};
```

```
matval[] = {2.0, 1.0, 3.0};
bdl[0] = XPRS_MINUSINFINITY; bdu[0] = 12.0;
...
XPRSaddcols(prob,1,3,obj,mstart,mrwind,matval,bdl,bdu);
XPRSaddnames(prob,2,"z",2,2);
```

Further information

1. The double constants `XPRS_PLUSINFINITY` and `XPRS_MINUSINFINITY` defined in the library header file can be used to represent plus and minus infinity respectively in the bound arrays.
2. If the columns are added to a MIP problem then they will be continuous variables. Use `XPRSchgcoltype` to impose integrality conditions on such new columns.

Related topics

`XPRSaddnames`, `XPRSaddrows`, `XPRSdelcols`, `XPRSchgcoltype`.

XPRSaddcuts, XPRSaddcuts64

Purpose

Adds cuts directly to the matrix at the current node. Any cuts added to the matrix at the current node and not deleted at the current node will be automatically added to the cut pool. The cuts added to the cut pool will be automatically restored at descendant nodes.

Synopsis

```
int XPRS_CC XPRSaddcuts(XPRSprob prob, int ncuts, const int mtype[], const
      char qrtype[], const double drhs[], const int mstart[], const int
      mcols[], const double dmatval[]);

int XPRS_CC XPRSaddcuts64(XPRSprob prob, int ncuts, const int mtype[],
      const char qrtype[], const double drhs[], const XPRSint64 mstart[],
      const int mcols[], const double dmatval[]);
```

Arguments

`prob`	The current problem.
`ncuts`	Number of cuts to add.
`mtype`	Integer array of length `ncuts` containing the user assigned cut types. The cut types can be any integer chosen by the user, and are used to identify the cuts in other cut manager routines using user supplied parameters. The cut type can be interpreted as an integer or a bitmap - see `XPRSdelcuts`.
`qrtype`	Character array of length `ncuts` containing the row types: `L` indicates a $\leq$ row; `G` indicates a $\geq$ row; `E` indicates an = row.
`drhs`	Double array of length `ncuts` containing the right hand side elements for the cuts.
`mstart`	Integer array containing offset into the `mcols` and `dmatval` arrays indicating the start of each cut. This array is of length `ncuts+1` with the last element, `mstart[ncuts]`, being where cut `ncuts+1` would start.
`mcols`	Integer array of length `mstart[ncuts]` containing the column indices in the cuts.
`dmatval`	Double array of length `mstart[ncuts]` containing the matrix values for the cuts.

Further information

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

2. The cuts added to the matrix are always added at the end of the matrix and the number of rows is always set to the original number of cuts added. If `ncuts` have been added, then the rows `0`,...,`ROWS-ncuts-1` are the original rows, whilst the rows `ROWS-ncuts`,...,`ROWS-1` are the added cuts. The number of cuts can be found by consulting the `CUTS` problem attribute.

3. This function should be called only from within callback functions set by either `XPRSaddcboptnode` or `XPRSaddcbcutmgr`.

Related topics

`XPRSaddrows`, `XPRSdelcpcuts`, `XPRSdelcuts`, `XPRSgetcpcutlist`, `XPRSgetcutlist`, `XPRSloadcuts`, `XPRSstorecuts`, Section 5.9.

XPRSaddgencons, XPRSaddgencons64

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

```
int XPRS_CC XPRSaddgencons(XPRSprob prob, int ngencons, int ncols, int
      nvals, const int type[], const int resultant[], const int colstart[],
      const int col[], const int valstart[], const double val[]);

int XPRS_CC XPRSaddgencons64(XPRSprob prob, int ngencons, XPRSint64 ncols,
      XPRSint64 nvals, const int type[], const int resultant[], const
      XPRSint64 colstart[], const int col[], const XPRSint64 valstart[],
      const double val[]);
```

Arguments

`prob`	The current problem.
`ngencons`	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.
`type`	Integer array of length `ngencons` containing the types of the general constraints: `XPRS_GENCONS_MAX (0)` indicates a `maximum` constraint; `XPRS_GENCONS_MIN (1)` indicates a `minimum` constraint; `XPRS_GENCONS_AND (2)` indicates an `and` constraint. `XPRS_GENCONS_OR (3)` indicates an `or` constraint; `XPRS_GENCONS_ABS (4)` indicates an `absolute value` constraint.
`resultant`	Integer array of length `ngencons` containing the indices of the output variables of the general constraints.
`colstart`	Integer array of length `ngencons` containing the start index of each general constraint in the `col` array.
`col`	Integer array of length `ncols` containing the input variables in all general constraints.
`valstart`	Integer array of length `ngencons` containing the start index of each general constraint in the `val` array (may be `NULL` if `nvals = 0`).
`val`	Double array of length `nvals` containing the constant values in all general constraints (may be `NULL` if `nvals = 0`).

Example

This adds two new general constraints `x2 = max(x0, x1, 5)` and `x3 = |x1|`:

```
int type[] = {XPRS_GENCONS_MAX, XPRS_GENCONS_ABS};
int resultant[] = {2, 3};
int colstart[] = {0, 2};
int col[] = {0, 1, 1};
int valstart[] = {0, 1};
double val[] = {5.0};

...
XPRSaddgencons(prob, 2, 3, 1, type, resultant, colstart, col, valstart, val);
```

```
XPRSmipoptimize(prob,"");
```

Further information

General constraints must be set up before solving the problem. They are converted to additional binary variables, indicator and linear constraints with the exact formulation and number of added entities depending on the performed presolving.

Note that using non-binary variables in `and/or` constraints or adding constant values to them or `absolute value` constraints will give an error at solve time.

Related controls

Integer

`GENCONSDUALREDUCTIONS` Controls whether dual reductions may be applied to reduce the number of added variables and constraints.

Related topics

`XPRSgetgencons`, `XPRSdelgencons`.

XPRSaddmipsol

Purpose

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

Synopsis

```
int XPRS_CC XPRSaddmipsol(XPRSprob prob, int ilength, const double
       mipsolval[], const int mipsolcol[], const char* solname);
```

Arguments

`prob`	The current problem.
`ilength`	Number of columns for which a value is provided.
`mipsolval`	Double array of length `ilength` containing solution values.
`mipsolcol`	Optional integer array of length `ilength` containing the column indices for the solution values provided in `mipsolval`. Should be `NULL` when `ilength` is equal to `COLS`, in which case it is assumed that `mipsolval` provides a complete solution vector.
`solname`	An optional name to associate with the solution. Can be NULL.

Further information

1. The function returns immediately after passing the solution to the Optimizer. The solution is placed in a pool until the Optimizer is able to analyze the solution during a MIP solve.

2. If the provided solution is found to be infeasible, a limited local search heuristic will be run in an attempt to find a close feasible integer solution.

3. If a partial solution is provided, global columns will be fixed to any provided values and a limited local search will be run in an attempt to find integer feasible values for the remaining unspecified columns. Values provided for continuous column in partial solutions are currently ignored.

4. The `XPRSaddcbusersolnotify` callback function can be used to discover the outcome of a loaded solution. The optional name provided as `solname` will be returned in the callback function.

5. If one or more solutions are loaded during the `XPRSaddcboptnode` callback, the Optimizer will process all loaded solutions and fire the callback again. This will be repeated as long as new solutions are loaded during the callback. You can check the `CALLBACKCOUNT_OPTNODE` attribute if you only want to run once.

Related controls

Integer

`CALLBACKCOUNT_OPTNODE`	Counts the number of times the `XPRSaddcboptnode` callback has been called, in particular after rerunning due to XPRSaddmipsol.
`USERSOLHEURISTIC`	Controls the local search heuristic for an infeasible or partial solution.

Related topics

`XPRSaddcbusersolnotify`, `XPRSaddcboptnode`.

XPRSaddnames

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 `XPRSaddnames`.

Synopsis

```
int XPRS_CC XPRSaddnames(XPRSprob prob, int type, const char cnames[], int
      first, int last);
```

Arguments

`prob`	The current problem.	
`type`	`1`	for row names;
	`2`	for column names.
	`3`	for set names.
	`4`	for piecewise linear constraint names.
	`5`	for general constraint names.
`cnames`	Character buffer containing the null-terminated string names.	
`first`	Start of the range of rows, columns, sets, piecewise linear or general constraints.	
`last`	End of the range of rows, columns, sets, piecewise linear of general constraints.	

Example

Add variable names (`a` and `b`), objective function (`profit`) and constraint names (`first` and `second`) to a problem:

```
char rnames[] = "profit\0first\0second"
char cnames[] = "a\0b";
...
XPRSaddnames(prob,1,rnames,0,nrow-1);
XPRSaddnames(prob,2,cnames,0,ncol-1);
```

Related topics

`XPRSaddcols`, `XPRSaddrows`, `XPRSgetnames`.

XPRSaddpwlcons, XPRSaddpwlcons64

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

```
int XPRS_CC XPRSaddpwlcons(XPRSprob prob, int npwls, int npoints, const int
      col[], const int resultant[], const int start[], const double xval[],
      const double yval[]);

int XPRS_CC XPRSaddpwlcons64(XPRSprob prob, int npwls, XPRSint64 npoints,
      const int col[], const int resultant[], const XPRSint64 start[],
      const double xval[], const double yval[]);
```

Arguments

`prob`	The current problem.
`npwls`	The number of piecewise linear constraints to add.
`npoints`	The total number of breakpoints of all piecewise linear constraints that should be added.
`col`	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.

Example

The following example adds a new piecewise linear constraint `y = f(x)`, where

`f(x) = -x`	if `x < 0`
`f(x) = 1`	if `0 <= x <= 2`
`f(x) = 2x-3`	if `x > 2`

This function can be defined using the breakpoints `(x = -1, y = 1), (0,0), (0,1), (2,1), (3,3)` (note that the first breakpoint could also be replaced, e.g., by `(x = -2, y = 2)`, similarly for the last):

```
int col[] = {0};
int resultant[] = {1};
int start[] = {0};
double xval[] = {-1,0,0,2,3};
double yval[] = {1,0,1,1,3};
...
XPRSaddpwlcons(prob,1,5,col,resultant,start,xval,yval);
XPRSmipoptimize(prob,"");
```

Further information

Piecewise linear constraints must be set up before solving the problem. They are converted to additional linear constraints, continuous variables and SOS2 constraints, with the exact formulation and number of added entities depending on the convexity of the piecewise linear function and some presolving steps that are applied.

Related controls

Integer

`PWLDUALREDUCTIONS` Controls whether dual reductions may be applied to reduce the number of added variables and constraints.

Related topics

`XPRSgetpwlcons`, `XPRSdelpwlcons`.

XPRSaddqmatrix, XPRSaddqmatrix64

Purpose

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

Synopsis

```
int XPRS_CC XPRSaddqmatrix(XPRSprob prob, int irow, int nqtr, const int
      mqc1[], const int mqc2[], const double dqe[]);

int XPRS_CC XPRSaddqmatrix64(XPRSprob prob, int irow, XPRSint64 nqtr, const
      int mqc1[], const int mqc2[], const double dqe[]);
```

Arguments

prob	The current problem.
irow	Index of the row where the quadratic matrix is to be added.
nqtr	Number of triplets used to define the quadratic matrix. This may be less than the number of coefficients in the quadratic matrix, since off diagonals and their transposed pairs are defined by one triplet.
mqc1	First index in the triplets.
mqc2	Second index in the triplets.
dqe	Coefficients in the triplets.

Further information

1. The triplets should be filled to define the upper-triangular part of the quadratic expression. This means that to add $[x^2 + 6\ xy]$ the dqe array shall contain the coefficients 1 and 3, respectively.
2. The matrix defined by mqc1, mqc2 and dqe should be positive semi-definite for <= and negative semi-definite for >= rows.
3. The row must not be an equality or a ranged row.

Related topics

XPRSloadqcqp, XPRSgetqrowcoeff, XPRSchgqrowcoeff, XPRSgetqrowqmatrix, XPRSgetqrowqmatrixtriplets, XPRSgetqrows, XPRSchgqobj, XPRSchgmqobj, XPRSgetqobj.

XPRSaddrows, XPRSaddrows64

Purpose

Allows rows to be added to the matrix after passing it to the Optimizer using the input routines.

Synopsis

```
int XPRS_CC XPRSaddrows(XPRSprob prob, int newrow, int newnz, const char
      qrtype[], const double rhs[], const double range[], const int
      mstart[], const int mclind[], const double dmatval[]);

int XPRS_CC XPRSaddrows64(XPRSprob prob, int newrow, XPRSint64 newnz, const
      char qrtype[], const double rhs[], const double range[], const
      XPRSint64 mstart[], const int mclind[], const double dmatval[]);
```

Arguments

`prob`	The current problem.
`newrow`	Number of new rows.
`newnz`	Number of new nonzeros in the added rows.
`qrtype`	Character array of length newrow containing the row types: `L` indicates a $\leq$ row; `G` indicates $\geq$ row; `E` indicates an = row. `R` indicates a range constraint; `N` indicates a nonbinding constraint.
`rhs`	Double array of length `newrow` containing the right hand side elements.
`range`	Double array of length `newrow` containing the row range elements. This may be `NULL` if there are no ranged constraints. The values in the `range` array will only be read for `R` type rows. The entries for other type rows will be ignored.
`mstart`	Integer array of length `newrow` containing the offsets in the `mclind` and `dmatval` arrays of the start of the elements for each row.
`mclind`	Integer array of length `newnz` containing the (contiguous) column indices for the elements in each row.
`dmatval`	Double array of length `newnz` containing the (contiguous) element values.

Related controls

Integer

EXTRAELEMS	Number of extra matrix elements to be allowed for.
EXTRAROWS	Number of extra rows to be allowed for.

Double

MATRIXTOL	Tolerance on matrix elements.

Example

Suppose the current problem is:

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

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

```
qrtype[0] = 'L';
```

```
rhs[0] = 25.0;
mstart[] = {0};
mclind[] = {0, 1, 2};
dmatval[] = {8.0, 9.0, 10.0};
...
XPRSaddrows(prob,1,3,qrtype,rhs,NULL,mstart,mclind, dmatval);
XPRSaddnames(prob,1,"NewRow",4,4);
```

Further information

Range rows are automatically converted to type `L`, with an upper bound in the slack. This must be taken into consideration, when retrieving row type, right–hand side values or range information for rows.

Related topics

`XPRSaddcols`, `XPRSaddcuts`, `XPRSaddnames`, `XPRSdelrows`.

XPRSaddsets, XPRSaddsets64

Purpose

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

Synopsis

```
int XPRS_CC XPRSaddsets(XPRSprob prob, int newsets, int newnz, const char
      qrtype[], const int msstart[], const int mclind[], const double
      dref[]);

int XPRS_CC XPRSaddsets64(XPRSprob prob, int newsets, XPRSint64 newnz,
      const char qrtype[], const XPRSint64 msstart[], const int mclind[],
      const double dref[]);
```

Arguments

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

Related topics

`XPRSdelsets`.

XPRSaddsetnames

Purpose

When a model with global 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

```
int XPRS_CC XPRSaddsetnames(XPRSprob prob, const char names[], int first,
        int 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.

Example

Add set names (`set1` and `set2`) to a problem:

```
char snames[] = "set1\0set2"
...
XPRSaddsetnames(prob,snames,0,1);
```

Related topics

`XPRSaddnames`, `XPRSloadglobal`, `XPRSloadqglobal`.

XPRSalter ALTER

Purpose

Alters or changes matrix elements, right hand sides and constraint senses in the current problem.

Synopsis

```
int XPRS_CC XPRSalter(XPRSprob prob, const char *filename);
ALTER [filename]
```

Arguments

`prob`	The current problem.
`filename`	A string of up to `MAXPROBNAMELENGTH` characters specifying the file to be read. If omitted, the default *problem_name* is used with a `.alt` extension.

Related controls

Integer

`EXTRAELEMS`	Number of extra matrix elements to be allowed for.

Double

`MATRIXTOL`	Tolerance on matrix elements.

Example 1 (Library)

Since the following call does not specify a filename, the file *problem_name*`.alt` is read in, from which commands are taken to alter the current matrix.

```
XPRSalter(prob,"");
```

Example 2 (Console)

The following example reads in the file `fred.alt`, from which instructions are taken to alter the current matrix:

```
ALTER fred
```

Further information

1. The file `filename.alt` is read. It is an ASCII file containing matrix revision statements in the format described in Section A.8. The `MODIFY` format of the MPS `REVISE` data is also supported.
2. It is not possible to alter a problem that is in a presolved state. Call `XPRSpostsolve` to bring the problem back to its original state.
3. If the problem was read from an .lp file, the name to use for the right-hand side is the one given by the attribute `RHSNAME` which by default is set to `RHS00001`.

Related topics

Section A.8.

XPRSbasiscondition BASISCONDITION

Purpose

This function is deprecated, and will be removed in future releases. Please use the `XPRSbasisstability` function instead. Calculates the condition number of the current basis after solving the LP relaxation.

Synopsis

```
int XPRS_CC XPRSbasiscondition(XPRSprob prob, double *condnum, double
      *scondnum);
BASISCONDITION
```

Arguments

`prob`	The current problem.
`condnum`	The returned condition number of the current basis.
`scondnum`	The returned condition number of the current basis for the scaled problem.

Example 1 (Library)

Get the condition number after optimizing a problem.

```
XPRSlpoptimize(prob," ");
XPRSbasiscondition(prob,&condnum,&scondnum);
printf("Condition no's are %g %g\n",condnum,scondnum);
```

Example 2 (Console)

Print the condition number after optimizing a problem.

```
READPROB problem.mps
LPOPTIMIZE
BASISCONDITION
```

Further information

1. The condition number of an invertible matrix is the norm of the matrix multiplied with the norm of its inverse. This number is an indication of how accurate the solution can be calculated and how sensitive it is to small changes in the data. The larger the condition number is, the less accurate the solution is likely to become.

2. When using the `BASISCONDITION` command in the Console Optimizer, the condition number is shown both for the scaled problem and in parenthesis for the original problem.

XPRSbasisstability BASISSTABILITY

Purpose

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

Synopsis

```
int XPRS_CC XPRSbasisstability(XPRSprob prob, int type, int norm, int
        ifscaled, double *dval);
BASISSTABILITY [-flags]
```

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.
`ifscaled`	If the stability values are to be calculated in the scaled, or the unscaled matrix.	
`dval`	Pointer to a double, where the calculated value is to be returned.	
`flags`	`x`	Stability measure for the solution and right–hand side values relative to the current basis.
	`d`	Stability measure for the duals and the basic part of the objective relative to the current basis.
	`c`	Condition number of the basis (default).
	`i`	Use the infinity norm (default).
	`o`	Use the one norm.
	`e`	Use the Euclidian norm for vectors, and the Frobenius norm for matrices.
	`u`	Calculate values in the unscaled matrix.

Further information

1. The Console Optimizer command `BASISSTABILITY` uses `0` as the default value for type and norm, and calculates the values in the scaled matrix.
2. The condition number (`type` = `0`) of an invertible matrix is the norm of the matrix multiplied with the norm of its inverse. This number is an indication of how accurate the solution can be calculated and how sensitive it is to small changes in the data. The larger the condition number is, the less accurate the solution is likely to become.
3. The stability measures (`type` = `1...4`) are using the original matrix and the basis to recalculate the various vectors related to the solution and the duals. The returned stability measure is the norm of the difference of the recalculated vector to the original one.

XPRSbtran

Purpose

Post-multiplies a (row) vector provided by the user by the inverse of the current basis.

Synopsis

```
int XPRS_CC XPRSbtran(XPRSprob prob, double vec[]);
```

Arguments

`prob`	The current problem.
`vec`	Double array of length `ROWS` containing the values by which the basis inverse is to be multiplied. The transformed values will also be returned in this array.

Related controls

Double

`ETATOL`	Tolerance on eta elements.

Example

Get the (unscaled) tableau row `z` of constraint number `irow`, assuming that all arrays have been dimensioned.

```
/* Minimum size of arrays:
 * y: nrow + ncol;
 * mstart: 2;
 * mrowind, dmatval: nrow.
 */

/* set up the unit vector y to pick out row irow */
for(i = 0; i < nrow; i++) y[i] = 0.0;
y[irow] = 1.0;

rc = XPRSbtran(prob,y);        /* y = e*B^{-1} */

/* Form z = y * A */
for(j = 0; j < ncol, j++) {
  rc = XPRSgetcols(prob, mstart, mrowind, dmatval,
                   nrow, &nelt, j, j);
  for(d = 0.0, ielt = 0, ielt < nelt; ielt++)
    d += y[mrowind[ielt]] * dmatval[ielt];
  y[nrow + j] = d;
}
```

Further information

If the matrix is in a presolved state, `XPRSbtran` works with the basis for the presolved problem.

Related topics

`XPRSftran`.

XPRScalcobjective

Purpose

Calculates the objective value of a given solution.

Synopsis

```
int XPRS_CC XPRScalcobjective(XPRSprob prob, const double solution[],
        double* objective);
```

Arguments

`prob`	The current problem.
`solution`	Double array of length COLS that holds the solution.
`objective`	Pointer to a double in which the calculated objective value is returned.

Further information

The calculations are always carried out in the original problem, even if the problem is currently presolved.

Related topics

`XPRScalcslacks`, `XPRScalcsolinfo`, `XPRScalcreducedcosts`.

XPRScalcreducedcosts

Purpose

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

Synopsis

```
int XPRS_CC XPRScalcreducedcosts(XPRSprob prob, const double duals[], const
      double solution[], double calculateddjs[]);
```

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. This is necessary for quadratic problems.
`calculateddjs`	Double array of length COLS in which the calculated reduced costs are returned.

Further information

1. The calculations are always carried out in the original problem, even if the problem is currently presolved.
2. If using the function during a solve (e.g. from a callback), use ORIGINALCOLS and ORIGINALROWS to retrieve the non-presolved dimensions of the problem.

Related topics

`XPRScalcslacks`, `XPRScalcsolinfo`, `XPRScalcobjective`.

XPRScalcslacks

Purpose

Calculates the row slack values for a given solution.

Synopsis

```
int XPRS_CC XPRScalcslacks(XPRSprob prob, const double solution[], double
        calculatedslacks[]);
```

Arguments

`prob`	The current problem.
`solution`	Double array of length COLS that holds the solution to calculate the slacks for.
`calculatedslacks`	Double array of length ROWS in which the calculated row slacks are returned.

Further information

1. The calculations are always carried out in the original problem, even if the problem is currently presolved.
2. If using the function during a solve (e.g. from a callback), use ORIGINALCOLS and ORIGINALROWS to retrieve the non-presolved dimensions of the problem.

Related topics

`XPRScalcreducedcosts`, `XPRScalcsolinfo`, `XPRScalcobjective`.

XPRScalcsolinfo

Purpose

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

Synopsis

```
int XPRS_CC XPRScalcsolinfo(XPRSprob prob, const double solution[], const
      double dual[], int Property, double* Value);
```

Arguments

`prob`	The current problem.	
`solution`	Double array of length COLS that holds the solution.	
`dual`	Double array of length ROWS that holds the dual solution.	
`Property`	`XPRS_SOLINFO_ABSPRIMALINFEAS`	the calculated maximum absolute primal infeasibility is returned.
	`XPRS_SOLINFO_RELPRIMALINFEAS`	the calculated maximum relative primal infeasibility is returned.
	`XPRS_SOLINFO_ABSDUALINFEAS`	the calculated maximum absolute dual infeasibility is returned.
	`XPRS_SOLINFO_RELDUALINFEAS`	the calculated maximum relative dual infeasibility is returned.
	`XPRS_SOLINFO_MAXMIPFRACTIONAL`	the calculated maximum absolute MIP fractionality or SOS infeasibility.
	`XPRS_SOLINFO_ABSMIPINFEAS`	the calculated maximum absolute MIP infeasibility (including delayed rows, indicators, general and piecewise linear constraints) is returned.
	`XPRS_SOLINFO_RELMIPINFEAS`	the calculated maximum relative MIP infeasibility (including delayed rows, indicators, general and piecewise linear constraints) is returned.
`Value`	Pointer to a double where the calculated value is returned.	

Further information

The calculations are always carried out in the original problem, even if the problem is currently presolved.

Related topics

`XPRScalcslacks`, `XPRScalcobjective`, `XPRScalcreducedcosts`.

CHECKCONVEXITY

Purpose

Checks if the loaded problem is convex. Applies to quadratic, mixed integer quadratic and quadratically constrained problems. Checking convexity takes some time, thus for problems that are known to be convex it might be reasonable to switch the checking off. Returns an error if the problem is not convex.

Synopsis

```
CHECKCONVEXITY
```

Further information

This console function checks the positive semi-definiteness of all quadratic matrices in the problem. Note, that when optimizing a problem, for quadratic programming and mixed integer quadratic problems, the checking of the objective function is performed after presolve, thus it is possible that an otherwise indefinite quadratic matrix will be found positive semi-definite (the indefinite part might have been fixed and dropped by presolve).

Related topics

XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE), IFCHECKCONVEXITY, EIGENVALUETOL.

XPRSchgbounds

Purpose

Used to change the bounds on columns in the matrix.

Synopsis

```
int XPRS_CC XPRSchgbounds(XPRSprob prob, int nbnds, const int mindex[],
      const char qbtype[], const double bnd[]);
```

Arguments

`prob`	The current problem.
`nbnds`	Number of bounds to change.
`mindex`	Integer array of size `nbnds` containing the indices of the columns on which the bounds will change.
`qbtype`	Character array of length `nbnds` indicating the type of bound to change: `U` indicates change the upper bound; `L` indicates change the lower bound; `B` indicates change both bounds, i.e. fix the column.
`bnd`	Double array of length `nbnds` giving the new bound values.

Example

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

```
mindex[0] = 0;
qbtype[0] = 'U';
bnd[0] = 0.5;
XPRSchgbounds(prob,1,mindex,qbtype,bnd);
```

Further information

1. A column index may appear twice in the `mindex` array so it is possible to change both the upper and lower bounds on a variable in one go.
2. `XPRSchgbounds` may be applied to the problem in a presolved state, in which case it expects references to the presolved problem.
3. The double constants `XPRS_PLUSINFINITY` and `XPRS_MINUSINFINITY` defined in the library header file can be used to represent plus and minus infinity respectively in the bound (`bnd`) array.
4. If the upper bound on a binary variable is changed to be greater than 1 or the lower bound is changed to be less than 0 then the variable will become an integer variable.

Related topics

`XPRSgetlb`, `XPRSgetub`, `XPRSstorebounds`.

XPRSchgcoef

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

```
int XPRS_CC XPRSchgcoef(XPRSprob prob, int irow, int icol, double dval);
```

Arguments

`prob`	The current problem.
`irow`	Row index for the coefficient.
`icol`	Column index for the coefficient.
`dval`	New value for the coefficient. If `dval` is zero, any existing coefficient will be deleted.

Related controls

Double

`MATRIXTOL`	Tolerance on matrix elements.

Example

In the following, the element in row `2`, column `1` of the matrix is changed to `0.33`:

```
XPRSchgcoef(prob,2,1,0.33);
```

Further information

`XPRSchgmcoef` is more efficient than multiple calls to `XPRSchgcoef` and should be used in its place in such circumstances.

Related topics

`XPRSaddcols`, `XPRSaddrows`, `XPRSchgmcoef`, `XPRSchgmqobj`, `XPRSchgobj`, `XPRSchgqobj`, `XPRSchgrhs`, `XPRSgetcols`, `XPRSgetrows`.

XPRSchgcoltype

Purpose

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

Synopsis

```
int XPRS_CC XPRSchgcoltype(XPRSprob prob, int nels, const int mindex[],
        const char qctype[]);
```

Arguments

`prob`	The current problem.
`nels`	Number of columns to change.
`mindex`	Integer array of length `nels` containing the indices of the columns.
`qctype`	Character array of length `nels` giving the new column types: `C` indicates a continuous column; `B` indicates a binary column; `I` indicates an integer column. `S` indicates a semi–continuous column. The semi–continuous lower bound will be set to `1.0`. `R` indicates a semi–integer column. The semi–integer lower bound will be set to `1.0`. `P` indicates a partial integer column. The partial integer bound will be set to `1.0`.

Example

The following changes columns 3 and 5 of the matrix to be integer and binary respectively:

```
mindex[0] = 3; mindex[1] = 5;
qctype[0] = 'I'; qctype[1] = 'B';
XPRSchgcoltype(prob,2,mindex,qctype);
```

Further information

1. The column types can only be changed before the global search is started.
2. Calling `XPRSchgcoltype` to change any variable into a binary variable causes the bounds previously defined for the variable to be deleted and replaced by bounds of 0 and 1.
3. Calling `XPRSchgcoltype` to change a continuous variable into an integer variable cause its lower bound to be rounded up to the nearest integer value and its upper bound to be rounded down to the nearest integer value.

Related topics

`XPRSaddcols`, `XPRSchgrowtype`, `XPRSdelcols`, `XPRSgetcoltype`.

XPRSchgglblimit

Purpose

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

Synopsis

```
int XPRS_CC XPRSchgglblimit(XPRSprob prob, int ncols, const int mindex[],
       const double dlimit[]);
```

Arguments

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

Further information

1. The new limits are not allowed to be negative.
2. Partial integer limits can be at most `2ˆ28`.

Related topics

`XPRSchgcoltype`, `XPRSgetglobal`.

XPRSchgmcoef, XPRSchgmcoef64

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

```
int XPRS_CC XPRSchgmcoef(XPRSprob prob, int nels, const int mrow[], const
      int mcol[], const double dval[]);

int XPRS_CC XPRSchgmcoef64(XPRSprob prob, XPRSint64 nels, const int mrow[],
      const int mcol[], const double dval[]);
```

Arguments

`prob`	The current problem.
`nels`	Number of new coefficients.
`mrow`	Integer array of length `nels` containing the row indices of the coefficients to be changed.
`mcol`	Integer array of length `nels` containing the column indices of the coefficients to be changed.
`dval`	Double array of length `nels` containing the new coefficient values. If an element of `dval` is zero, the coefficient will be deleted.

Related controls

Double

`MATRIXTOL`	Tolerance on matrix elements.

Example

```
mrow[0] = 0; mrow[1] = 3;
mcol[0] = 1; mcol[1] = 5;
dval[0] = 2.0; dval[1] = 0.0;
XPRSchgmcoef(prob,2,mrow,mcol,dval);
```

This changes two elements to values `2.0` and `0.0`.

Further information

`XPRSchgmcoef` is more efficient than repeated calls to `XPRSchgcoef` and should be used in its place if many coefficients are to be changed.

Related topics

`XPRSchgcoef`, `XPRSchgmqobj`, `XPRSchgobj`, `XPRSchgqobj`, `XPRSchgrhs`, `XPRSgetcols`, `XPRSgetrhs`.

XPRSchgmqobj, XPRSchgmqobj64

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

```
int XPRS_CC XPRSchgmqobj(XPRSprob prob, int nels, const int mqcol1[], const
      int mqcol2[], const double dval[]);

int XPRS_CC XPRSchgmqobj64(XPRSprob prob, XPRSint64 nels, const int
      mqcol1[], const int mqcol2[], const double dval[]);
```

Arguments

`prob`	The current problem.
`nels`	The number of coefficients to change.
`mqcol1`	Integer array of size `ncol` containing the column index of the first variable in each quadratic term.
`mqcol2`	Integer array of size `ncol` containing the column index of the second variable in each quadratic term.
`dval`	New values for the coefficients. If an entry in `dval` is `0`, the corresponding entry will be deleted. These are the coefficients of the quadratic Hessian matrix.

Example

The following code results in an objective function with terms: $[6x_1^2 + 3x_1x_2 + 3x_2x_1]/2$

```
mqcol1[0] = 0; mqcol2[0] = 0; dval[0] = 6.0;
mqcol1[1] = 1; mqcol2[1] = 0; dval[1] = 3.0;
XPRSchgmqobj(prob,2,mqcol1,mqcol2,dval);
```

Further information

1. The columns in the arrays `mqcol1` and `mqcol2` must already exist in the matrix. If the columns do not exist, they must be added with `XPRSaddcols`.
2. `XPRSchgmqobj` is more efficient than repeated calls to `XPRSchgqobj` and should be used in its place when several coefficients are to be changed.

Related topics

`XPRSchgcoef`, `XPRSchgmcoef`, `XPRSchgobj`, `XPRSchgqobj`, `XPRSgetqobj`.

XPRSchgobj

Purpose

Used to change the objective function coefficients.

Synopsis

```
int XPRS_CC XPRSchgobj(XPRSprob prob, int nels, const int mindex[], const
      double obj[]);
```

Arguments

`prob`	The current problem.
`nels`	Number of objective function coefficient elements to change.
`mindex`	Integer array of length `nels` containing the indices of the columns on which the range elements will change. An index of `-1` indicates that the fixed part of the objective function on the right hand side should change.
`obj`	Double array of length `nels` giving the new objective function coefficient.

Example

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

```
mindex[0] = 0; mindex[1] = 2; mindex[2] = 5;
obj[0] = 25.0; obj[1] = 5.3; obj[2] = 0.0;
XPRSchgobj(prob,3,mindex,obj);
```

Further information

The value of the fixed part of the objective function can be obtained using the `OBJRHS` problem attribute.

Related topics

`XPRSchgcoef`, `XPRSchgmcoef`, `XPRSchgmqobj`, `XPRSchgqobj`, `XPRSgetobj`.

XPRSchgobjsense CHGOBJSENSE

Purpose

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

Synopsis

```
int XPRS_CC XPRSchgobjsense(XPRSprob prob, int objsense);
CHGOBJSENSE [ min | max ]
```

Arguments

`prob`	The current problem.
`objsense`	`XPRS_OBJ_MINIMIZE` to change into a minimization, or `XPRS_OBJ_MAXIMIZE` to change into maximization problem.

Related topics

`XPRSlpoptimize`, `XPRSmipoptimize`.

XPRSchgqobj

Purpose

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

Synopsis

```
int XPRS_CC XPRSchgqobj(XPRSprob prob, int icol, int jcol, double dval);
```

Arguments

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

Example

The following code adds the terms $[15x_1^2 + 7x_1x_2]/2$ to the objective function:

```
XPRSchgqobj(prob, 0, 0, 15);
XPRSchgqobj(prob, 0, 1, 3.5);
```

Further information

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

Related topics

`XPRSchgcoef`, `XPRSchgmcoef`, `XPRSchgmqobj`, `XPRSchgobj`, `XPRSgetqobj`.

XPRSchgqrowcoeff

Purpose

Changes a single quadratic coefficient in a row.

Synopsis

```
int XPRS_CC XPRSchgqrowcoeff(XPRSprob prob, int irow, int icol, int jcol,
        double dval);
```

Arguments

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

Further information

1. This function may be used to add new nonzero coefficients, or even to define the whole quadratic expression with it. Doing that, however, is significantly less efficient than adding the whole expression with `XPRSaddqmatrix`.
2. The row must not be an equality or a ranged row.

Related topics

`XPRSloadqcqp`, `XPRSgetqrowcoeff`, `XPRSaddqmatrix`, `XPRSchgqrowcoeff`, `XPRSgetqrowqmatrix`, `XPRSgetqrowqmatrixtriplets`, `XPRSgetqrows`, `XPRSchgqobj`, `XPRSchgmqobj`, `XPRSgetqobj`.

XPRSchgrhs

Purpose

Used to change right–hand side values of the problem.

Synopsis

```
int XPRS_CC XPRSchgrhs(XPRSprob prob, int nels, const int mindex[], const
      double rhs[]);
```

Arguments

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

Example

Here we change the three right hand sides in rows `2`, `6`, and `8` to new values:

```
mindex[0] = 2; mindex[1] = 8; mindex[2] = 6;
rhs[0] = 5.0; rhs[1] = 3.8; rhs[2] = 5.7;
XPRSchgrhs(prob,3,mindex,rhs);
```

Related topics

`XPRSchgcoef`, `XPRSchgmcoef`, `XPRSchgrhsrange`, `XPRSgetrhs`, `XPRSgetrhsrange`.

XPRSchgrhsrange

Purpose

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

Synopsis

```
int XPRS_CC XPRSchgrhsrange(XPRSprob prob, int nels, const int mindex[],
      const double rng[]);
```

Arguments

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

Example

Here, the constraint $x + y \leq 10$ (with row index 5) in the problem is changed to $8 \leq x + y \leq 10$:

```
mindex[0] = 5; rng[0] = 2.0;
XPRSchgrhsrange(prob,1,mindex,rng);
```

Further information

If the range specified on the row is r, what happens depends on the row type and value of r. It is possible to convert non-range rows using this routine.

Value of r	Row type	Effect
$r \geq 0$	$= b, \leq b$	$b - r \leq \sum a_j x_j \leq b$
$r \geq 0$	$\geq b$	$b \leq \sum a_j x_j \leq b + r$
$r < 0$	$= b, \leq b$	$b \leq \sum a_j x_j \leq b - r$
$r < 0$	$\geq b$	$b + r \leq \sum a_j x_j \leq b$

Related topics

XPRSchgcoef, XPRSchgmcoef, XPRSchgrhs, XPRSgetrhsrange.

XPRSchgrowtype

Purpose

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

Synopsis

```
int XPRS_CC XPRSchgrowtype(XPRSprob prob, int nels, const int mindex[],
        const char qrtype[]);
```

Arguments

`prob`	The current problem.
`nels`	Number of rows to change.
`mindex`	Integer array of length `nels` containing the indices of the rows.
`qrtype`	Character array of length `nels` giving the new row types: `L` indicates a $\leq$ row; `E` indicates an = row; `G` indicates a $\geq$ row; `R` indicates a range row; `N` indicates a free row.

Example

Here row `4` is changed to an equality row:

```
mindex[0] = 4; qrtype[0] = 'E';
XPRSchgrowtype(prob,1,mindex,qrtype);
```

Further information

A row can be changed to a range type row by first changing the row to an `R` or `L` type row and then changing the range on the row using `XPRSchgrhsrange`.

Related topics

`XPRSaddrows`, `XPRSchgcoltype`, `XPRSchgrhs`, `XPRSchgrhsrange`, `XPRSdelrows`, `XPRSgetrowrange`, `XPRSgetrowtype`.

XPRScopycallbacks

Purpose

Copies callback functions defined for one problem to another.

Synopsis

```
int XPRS_CC XPRScopycallbacks(XPRSprob dest, XPRSprob src);
```

Arguments

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

Example

The following sets up a message callback function `callback` for problem `prob1` and then copies this to the problem `prob2`.

```
XPRScreateprob(&prob1);
XPRSaddcbmessage(prob1,callback,NULL,0);
XPRScreateprob(&prob2);
XPRScopycallbacks(prob2,prob1);
```

Related topics

`XPRScopycontrols`, `XPRScopyprob`.

XPRScopycontrols

Purpose

Copies controls defined for one problem to another.

Synopsis

```
int XPRS_CC XPRScopycontrols(XPRSprob dest, XPRSprob src);
```

Arguments

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

Example

The following turns off Presolve for problem `prob1` and then copies this and other control values to the problem `prob2` :

```
XPRScreateprob(&prob1);
XPRSsetintcontrol(prob1,XPRS_PRESOLVE,0);
XPRScreateprob(&prob2);
XPRScopycontrols(prob2,prob1);
```

Related topics

`XPRScopycallbacks`, `XPRScopyprob`.

XPRScopyprob

Purpose

Copies information defined for one problem to another.

Synopsis

```
int XPRS_CC XPRScopyprob(XPRSprob dest, XPRSprob src, const char
        *probname);
```

Arguments

`dest`	The new problem pointer to which information is copied.
`src`	The old problem pointer from which information is copied.
`probname`	A string of up to 1024 characters (including `NULL` terminator) containing the name for the problem copy. This must be unique when file writing is to be expected, and particularly for global problems.

Example

The following copies the problem, its controls and it callbacks from `prob1` to `prob2`:

```
XPRSprob prob1, prob2;
...
XPRScreateprob(&prob2);
XPRScopyprob(prob2,prob1,"MyProb");
XPRScopycontrols(prob2,prob1);
XPRScopycallbacks(prob2,prob1);
```

Further information

`XPRScopyprob` copies only the problem and does not copy the callbacks or controls associated to a problem. These must be copied separately using `XPRScopycallbacks` and `XPRScopycontrols`, respectively.

Related topics

`XPRScopycallbacks`, `XPRScopycontrols`, `XPRScreateprob`.

XPRScreateprob

Purpose

Sets up a new problem within the Optimizer.

Synopsis

```
int XPRS_CC XPRScreateprob(XPRSprob *prob);
```

Argument

`prob` Pointer to a variable holding the new problem.

Example

The following creates a problem which will contain `myprob`:

```
XPRSprob prob;
XPRSinit(NULL);
XPRScreateprob(&prob);
XPRSreadprob(prob,"myprob","");
```

Further information

1. `XPRScreateprob` must be called after `XPRSinit` and before using the other Optimizer routines.
2. Any number of problems may be created in this way, depending on your license details. All problems should be removed using `XPRSdestroyprob` once you have finished working with them.
3. If `XPRScreateprob` cannot complete successfully, a nonzero value is returned and `*prob` is set to NULL (as a consequence, it is not possible to retrieve further error information using e.g. `XPRSgetlasterror`).

Related topics

`XPRSdestroyprob`, `XPRScopyprob`, `XPRSinit`.

XPRScrossoverlpsol

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

```
int XPRS_CC XPRScrossoverlpsol(XPRSprob prob, int *status);
```

Arguments

`prob`	The current problem.
`status`	Pointer to an `int` where the status will be returned. The status is one of: `0` The crossover is successful. `1` The crossover is not performed because the problem has no solution.

Related controls

Integer

`ALGAFTERCROSSOVER`	Specifies which algorithm to use for cleaning up the solution.
`PREPROTECTDUAL`	Whether or not to protect the given dual solution during presolve.

Example

This example loads a problem, loads a solution for the problem and then uses `XPRScrossoverlpsol` to find a basic optimal solution.

```
XPRSreadprob(prob, "problem", "");
XPRSloadlpsol(prob, x, NULL, dual, NULL, &status);
XPRScrossoverlpsol(prob, &status);
```

A solution can also be loaded from an ASCII solution file using `XPRSreadslxsol`.

Further information

1. The crossover contains two phases: a crossover phase for finding a basic solution and a clean-up phase for finding a basic optimal solution. Setting `ALGAFTERCROSSOVER` to `0` will allow the crossover to skip the clean-up phase.

2. The given solution is expected to be feasible or nearly feasible, otherwise the crossover may take a long time to find a basic feasible solution. More importantly, the given solution is expected to have a small duality gap. A small duality gap indicates that the given solution is close to the optimal solution. If the given solution is far away from the optimal solution, the clean-up phase may need many simplex iterations to move to a basic optimal solution.

Related topics

`XPRSloadlpsol`, `XPRSreadslxsol`, Section 4.2.1.

XPRSdelcols

Purpose

Delete columns from a matrix.

Synopsis

```
int XPRS_CC XPRSdelcols(XPRSprob prob, int ncols, const int mindex[]);
```

Arguments

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

Example

In this example, column `3` is deleted from the matrix:

```
mindex[0] = 3;
XPRSdelcols(prob,1,mindex);
```

Further information

1. After columns have been deleted from a problem, the numbers of the remaining columns are moved down so that the columns are always numbered from `0` to `COLS-1` where `COLS` is the problem attribute containing the number of non-deleted columns in the matrix.
2. If the problem has already been optimized, or an advanced basis has been loaded, and you delete a basis column the current basis will no longer be valid - the basis is "lost".
   If you go on to re-optimize the problem, a warning message is displayed (140) and the Optimizer automatically generates a corrected basis.
   You can avoid losing the basis by only deleting non-basic columns (see `XPRSgetbasis`), taking a basic column out of the basis first if necessary (see `XPRSgetpivots` and `XPRSpivot`).

Related topics

`XPRSaddcols`, `XPRSdelrows`.

XPRSdelcpcuts

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 XPRSdelcuts, 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 XPRSdelcpcuts.

Synopsis

```
int XPRS_CC XPRSdelcpcuts(XPRSprob prob, int itype, int interp, int ncuts,
        const XPRScut mcutind[]);
```

Arguments

prob	The current problem.
itype	User defined cut type to match against.
interp	Way in which the cut itype is interpreted: −1 match all cut types; 1 treat cut types as numbers; 2 treat cut types as bit maps - delete if any bit matches any bit set in itype; 3 treat cut types as bit maps - delete if all bits match those set in itype.
ncuts	The number of cuts to delete. A value of −1 indicates delete all cuts.
mcutind	Array containing pointers to the cuts which are to be deleted. This array may be NULL if ncuts is −1, otherwise it has length ncuts.

Related topics

XPRSaddcuts, XPRSdelcuts, XPRSloadcuts, Section 5.9.

XPRSdelcuts

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 XPRSaddcuts or XPRSloadcuts. 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

```
int XPRS_CC XPRSdelcuts(XPRSprob prob, int ibasis, int itype, int interp,
        double delta, int num, const XPRScut mcutind[]);
```

Arguments

prob	The current problem.
ibasis	Ensures the basis will be valid if set to 1. If set to 0, cuts with non-basic slacks may be deleted.
itype	User defined type of the cut to be deleted.
interp	Way in which the cut itype is interpreted: −1 match all cut types; 1 treat cut types as numbers; 2 treat cut types as bit maps - delete if any bit matches any bit set in itype; 3 treat cut types as bit maps - delete if all bits match those set in itype.
delta	Only delete cuts with an absolute slack value greater than delta. To delete all the cuts, this argument should be set to XPRS_MINUSINFINITY.
num	Number of cuts to drop if a list of cuts is provided. A value of −1 indicates all cuts.
mcutind	Array containing pointers to the cuts which are to be deleted. This array may be NULL if num is set to −1 otherwise it has length num.

Further information

1. It is usually best to drop only those cuts with basic slacks, otherwise the basis will no longer be valid and it may take many iterations to recover an optimal basis. If the ibasis parameter is set to 1, this will ensure that cuts with non-basic slacks will not be deleted even if the other parameters specify that these cuts should be deleted. It is highly recommended that the ibasis parameter is always set to 1.

2. The cuts to be deleted can also be specified by the size of the slack variable for the cut. Only those cuts with a slack value greater than the delta parameter will be deleted.

3. A list of indices of the cuts to be deleted can also be provided. The list of active cuts at a node can be obtained with the XPRSgetcutlist command.

4. This function should be called only from within callback functions set by either XPRSaddcboptnode or XPRSaddcbcutmgr.

Related topics

XPRSaddcuts, XPRSdelcpcuts, XPRSgetcutlist, XPRSloadcuts, Section 5.9.

XPRSdelgencons

Purpose

Delete general constraints from a problem.

Synopsis

```
int XPRS_CC XPRSdelgencons(XPRSprob prob, int ngencons, const int
        mindex[]);
```

Arguments

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

Example

In this example, general constraints `0` and `2` are deleted from the problem:

```
mindex[0] = 0; mindex[1] = 2;
XPRSdelgencons(prob,2,mindex);
```

Further information

After general constraints have been deleted from a problem, the indices of the remaining constraints are reduced down so that the general constraints are always numbered from `0` to `GENCONS-1` where `GENCONS` is the problem attribute containing the number of non-deleted general constraints in the problem.

Related topics

`XPRSaddgencons`, `XPRSgetgencons`.

XPRSdelindicators

Purpose

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

Synopsis

```
int XPRS_CC XPRSdelindicators(XPRSprob prob, int first, int last);
```

Arguments

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

Example

In this example, if any of the first two rows of the matrix is an indicator constraint, they are turned into normal rows:

```
XPRSdelindicators(prob,0,1);
```

Further information

This function has no effect on rows that are not indicator constraints.

Related topics

`XPRSgetindicators`, `XPRSsetindicators`.

XPRSdelpwlcons

Purpose

Delete piecewise linear constraints from a problem.

Synopsis

```
int XPRS_CC XPRSdelpwlcons(XPRSprob prob, int npwls, const int mindex[]);
```

Arguments

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

Example

In this example, piecewise linear constraints `0` and `2` are deleted from the problem:

```
mindex[0] = 0; mindex[1] = 2;
XPRSdelpwlcons(prob,2,mindex);
```

Further information

After piecewise linear constraints have been deleted from a problem, the indices of the remaining constraints are reduced so that the piecewise linear constraints are always numbered from `0` to `PWLCONS-1` where `PWLCONS` is the problem attribute containing the number of non-deleted piecewise linear constraints in the problem.

Related topics

`XPRSaddpwlcons`, `XPRSgetpwlcons`.

XPRSdelqmatrix

Purpose

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

Synopsis

```
int XPRS_CC XPRSdelqmatrix(XPRSprob prob, int row);
```

Arguments

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

Further information

If a row index of `-1` is used, the function deletes the quadratic coefficients from the objective function.

Related topics

`XPRSaddrows`, `XPRSdelcols`, `XPRSdelrows`.

XPRSdelrows

Purpose

Delete rows from a matrix.

Synopsis

```
int XPRS_CC XPRSdelrows(XPRSprob prob, int nrows, const int mindex[]);
```

Arguments

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

Example

In this example, rows `0` and `10` are deleted from the matrix:

```
mindex[0] = 0; mindex[1] = 10;
XPRSdelrows(prob,2,mindex);
```

Further information

1. After rows have been deleted from a problem, the numbers of the remaining rows are moved down so that the rows are always numbered from `0` to `ROWS-1` where `ROWS` is the problem attribute containing the number of non-deleted rows in the matrix.
2. If the problem has already been optimized, or an advanced basis has been loaded, and you delete a row for which the slack column is non-basic, the current basis will no longer be valid - the basis is "lost".

   If you go on to re-optimize the problem, a warning message is displayed (140) and the Optimizer automatically generates a corrected basis.
   You can avoid losing the basis by only deleting basic rows (see `XPRSgetbasis`), bringing a non-basic row into the basis first if necessary (see `XPRSgetpivots` and `XPRSpivot`).

Related topics

`XPRSaddrows`, `XPRSdelcols`, `XPRSgetbasis`, `XPRSgetpivots`, `XPRSpivot`.

XPRSdelsets

Purpose

Delete sets from a problem.

Synopsis

```
int XPRS_CC XPRSdelsets(XPRSprob prob, int nsets, const int mindex[]);
```

Arguments

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

Example

In this example, sets `0` and `2` are deleted from the problem:

```
mindex[0] = 0; mindex[1] = 2;
XPRSdelsets(prob,2,mindex);
```

Further information

After sets have been deleted from a problem, the numbers of the remaining sets are moved down so that the sets are always numbered from `0` to `SETS-1` where `SETS` is the problem attribute containing the number of non-deleted sets in the problem.

Related topics

`XPRSaddsets`.

XPRSdestroyprob

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

Synopsis
```
int XPRS_CC XPRSdestroyprob(XPRSprob prob);
```

Argument
`prob`	The problem to be destroyed.

Example
The following creates, loads and solves a problem called `myprob`, before subsequently freeing any resources allocated to it:

```
XPRScreateprob(&prob);
XPRSreadprob(prob,"myprob","");
XPRSlpoptimize(prob,"");
XPRSdestroyprob(prob);
```

Further information
After work is finished, all problems must be destroyed. If a `NULL` problem pointer is passed to `XPRSdestroyprob`, no error will result.

Related topics
`XPRScreateprob`, `XPRSfree`, `XPRSinit`.

XPRSdumpcontrols DUMPCONTROLS

Purpose

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

Synopsis

```
int XPRS_CC XPRSdumpcontrols(XPRSprob prob);
DUMPCONTROLS
```

Related topics

SETDEFAULTS, SETDEFAULTCONTROL

EXIT

Purpose

Terminates the Console Optimizer, returning a zero exit code to the operating system. Alias of QUIT.

Synopsis

```
EXIT
```

Example

The command is called simply as:

```
EXIT
```

Further information

1. Fatal error conditions return nonzero exit values which may be of use to the host operating system. These are described in Chapter 11.
2. If you wish to return an exit code reflecting the final solution status, then use the `STOP` command instead.

Related topics

`STOP`, `QUIT`, `XPRSsave` (`SAVE`).

XPRSestimaterowdualranges

Purpose

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

Synopsis

```
int XPRS_CC XPRSestimaterowdualranges(XPRSprob prob, const int nRows, const
      int rowIndices[], const int iterationLimit, double minDualActivity[],
      double maxDualActivity[]);
```

Arguments

`prob`	The current problem.
`nRows`	The number of rows to analyze.
`rowIndices`	Row indices to analyze.
`iterationLimit`	Effort limit expressed as simplex iterations per row.
`minDualActivity`	Estimated lower bounds on the possible dual ranges.
`maxDualActivity`	Estimated upper bounds on the possible dual ranges.

Further information

This function may provide better results for individual row dual ranges when called for a larger number of rows.

Related topics

`XPRSlpoptimize`, `XPRSstrongbranch`

XPRSfeaturequery

Purpose

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

Synopsis

```
int XPRS_CC XPRSfeaturequery(const char *feature, int *featurestatus);
```

Arguments

`feature`	The feature string to be checked in the license.
`featurestatus`	Return status of the check, a value of 1 indicates the feature is available.

XPRSfixglobals FIXGLOBALS

Purpose

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

Synopsis

```
int XPRS_CC XPRSfixglobals(XPRSprob prob, int options);
FIXGLOBALS [-flags]
```

Arguments

`prob` The current problem.

`options` Options how to fix the globals.

Values

Bit	Meaning
`0`	If all global 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.

`flags` Flags to pass to FIXGLOBALS:

`r` round all global entities to the nearest feasible value in the solution before being fixed;

`t` keep piecewise linear and general constraints and only fix their non-convex decisions;

Example 1 (Library)

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

```
XPRSreadprob(prob,"myprob","");
XPRSmipoptimize(prob," ");
XPRSfixglobals(prob, 1);
XPRSlpoptimize(prob," ");
XPRSwriteprtsol(prob);
```

Example 2 (Console)

A similar set of commands at the console would be as follows:

```
READPROB
MIPOPTIMIZE
FIXGLOBALS -r
LPOPTIMIZE
PRINTSOL
```

Further information

1. Because of tolerances, it is possible for e.g. a binary variable to be slightly fractional in the MIP solution, where it might have the value `0.999999` instead of being at exactly `1.0`. With `ifround = 0`, such a binary will be fixed at `0.999999`, but with `ifround = 1`, it will be fixed at `1.0`.

2. This command is useful for inspecting the reduced costs of the continuous variables in a matrix after the global entities have been fixed. Sensitivity analysis can also be performed on the continuous variables in a MIP problem using `XPRSrhssa` or `XPRSobjsa` after calling `XPRSfixglobals` (`FIXGLOBALS`).

Related topics

XPRSmipoptimize (MIPOPTIMIZE).

XPRSfree

Purpose

Frees any allocated memory and closes all open files.

Synopsis

```
int XPRS_CC XPRSfree(void);
```

Example

The following frees resources allocated to the problem `prob` and then tidies up before exiting:

```
XPRSdestroyprob(prob);
XPRSfree();
return 0;
```

Further information

After a call to `XPRSfree` no library functions may be used without first calling `XPRSinit` again.

Related topics

`XPRSdestroyprob`, `XPRSinit`.

XPRSftran

Purpose

Pre-multiplies a (column) vector provided by the user by the inverse of the current matrix.

Synopsis

```
int XPRS_CC XPRSftran(XPRSprob prob, double vec[]);
```

Arguments

`prob`	The current problem.
`vec`	Double array of length ROWS containing the values which are to be multiplied by the basis inverse. The transformed values appear in the array.

Related controls

Double

ETATOL	Tolerance on eta elements.

Example

To get the (unscaled) tableau column of structural variable number `jcol`, assuming that all arrays have been dimensioned, do the following:

```
/* Min size of arrays: mstart: 2; mrowind, dmatval & y: nrow. */
/* Get column as loaded originally, in sparse format */
rc = XPRSgetcols(prob, mstart, mrowind, dmatval, nrow, &nelt,
                 jcol, jcol);

/* Unpack into the zeroed array */
for(i = 0; i < nrow; i++)
y[i] = 0.0;
for(ielt = 0; ielt < nelt; ielt++)
y[mrowind[ielt]] = dmatval[ielt];

rc = XPRSftran(prob,y);
```

Get the (unscaled) tableau column of the slack variable for row number `irow`, assuming that all arrays have been dimensioned.

```
/* Min size of arrays: y: nrow */
/* Set up the original slack column in full format */
for(i = 0; i < nrow; i++)
y[i] = 0.0;
y[irow] = 1.0;

rc = XPRSftran(prob,y);
```

Further information

If the matrix is in a presolved state, the function will work with the basis for the presolved problem.

Related topics

XPRSbtran.

XPRSgetattribinfo

Purpose

Accesses the id number and the type information of an attribute given its name. An attribute name may be for example `XPRS_ROWS`. Names are case-insensitive and may or may not have the `XPRS_` prefix. The id number is the constant used to identify the attribute for calls to functions such as `XPRSgetintattrib`. 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 `XPRS_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.

XPRS_TYPE_NOTDEFINED	The name was not recognized.
XPRS_TYPE_INT	32 bit integer.
XPRS_TYPE_INT64	64 bit integer.
XPRS_TYPE_DOUBLE	Double precision floating point.
XPRS_TYPE_STRING	String.

Synopsis

```
int XPRS_CC XPRSgetattribinfo(XPRSprob prob, const char* sCaName, int*
      iHeaderId, int* iTypeinfo);
```

Arguments

`prob`	The current problem.
`sCaName`	The name of the attribute to be queried. Names are case-insensitive and may or may not have the `XPRS_` prefix. A full list of all attributes may be found in Chapter 9, or from the list in the `xprs.h` header file.
`iHeaderId`	Pointer to an integer where the id number will be returned.
`iTypeInfo`	Pointer to an integer where the type id will be returned.

Example

The following code example obtains the id number and the type id of the control or attribute with name given by `sCaName`. Note that the name happens to be a control name in this example:

```
const char *sCaName = "presolve";
int iHeaderId, iTypeInfo;
...
if(XPRSgetattribinfo(prob, sCaName, &iHeaderId,
                     &iTypeInfo) || iHeaderId==0) {
  if(XPRSgetcontrolinfo(prob, sCaName, &iHeaderId,
                        &iTypeInfo) || iHeaderId==0) {
    printf("Unrecognized name: %s\n", sCaName);
  }
}
```

Related topics

`XPRSgetcontrolinfo`.

XPRSgetbanner

Purpose
Returns the banner and copyright message.

Synopsis
```
int XPRS_CC XPRSgetbanner(char *banner);
```

Argument
`banner`	A buffer of at least `XPRS_MAXBUFFERLENGTH` characters in which the null terminated banner string will be returned.

Example
The following calls `XPRSgetbanner` to return banner information at the start of the program:

```
char banner[XPRS_MAXBUFFERLENGTH];
...
if(XPRSinit(NULL))
{
  /* The error message when XPRSinit fails is written to the banner. */
  XPRSgetbanner(banner);
  printf("%s\n", banner);
  return 1;
}
XPRSgetbanner(banner);
printf("%s\n", banner);
```

Further information
This function can most usefully be employed to return extra information if a problem occurs with `XPRSinit`.

Related topics
`XPRSinit`.

XPRSgetbasis

Purpose

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

Synopsis

```
int XPRS_CC XPRSgetbasis(XPRSprob prob, int rstatus[], int cstatus[]);
```

Arguments

`prob` The current problem.

`rstatus` Integer array of length `ROWS` to the basis status of the slack, surplus or artificial variable associated with each row. The status will be one of:
- `0` slack, surplus or artificial is non-basic at lower bound;
- `1` slack, surplus or artificial is basic;
- `2` slack or surplus is non-basic at upper bound.
- `3` slack or surplus is super-basic.

May be `NULL` if not required.

`cstatus` Integer array of length `COLS` to hold the basis status of the columns in the constraint matrix. The status will be one of:
- `0` variable is non-basic at lower bound, or superbasic at zero if the variable has no lower bound;
- `1` variable is basic;
- `2` variable is non-basic at upper bound;
- `3` variable is super-basic.

May be `NULL` if not required.

Example

The following example minimizes a problem before saving the basis for later:

```
int rows, cols, *rstatus, *cstatus;
...
XPRSgetintattrib(prob,XPRS_ROWS,&rows);
XPRSgetintattrib(prob,XPRS_COLS,&cols);
rstatus = (int *) malloc(sizeof(int)*rows);
cstatus = (int *) malloc(sizeof(int)*cols);
XPRSlpoptimize(prob,"");
XPRSgetbasis(prob,rstatus,cstatus);
```

Related topics

`XPRSgetpresolvebasis`, `XPRSloadbasis`, `XPRSloadpresolvebasis`.

XPRSgetbasisval

Purpose

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

Synopsis

```
int XPRS_CC XPRSgetbasisval(XPRSprob prob, int row, int column, int
      *rstatus, int *cstatus);
```

Arguments

`prob`	The current problem.
`row`	Row index to get the row basis status for.
`column`	Column index to get the column basis status for.
`rstatus`	Integer pointer where the value of the row basis status will be returned. May be `NULL` if not required.
`cstatus`	Integer pointer where the value of the column basis status will be returned. May be `NULL` if not required.

Related topics

`XPRSgetbasis`, `XPRSgetpresolvebasis`, `XPRSloadbasis`, `XPRSloadpresolvebasis`.

XPRSgetcheckedmode

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 `XPRSsetcheckedmode`.

Synopsis

```
int XPRS_CC XPRSgetcheckedmode(int* r_checked_mode);
```

Argument

`r_checked_mode` Variable that is set to 0 if checking and validation of Optimizer function calls is disabled for the current process, non-zero otherwise.

Related topics

`XPRSsetcheckedmode`.

XPRSgetcoef

Purpose

Returns a single coefficient in the constraint matrix.

Synopsis

```
int XPRS_CC XPRSgetcoef(XPRSprob prob, int irow, int icol, double *dval);
```

Arguments

`prob`	The current problem.
`irow`	Row of the constraint matrix.
`icol`	Column of the constraint matrix.
`dval`	Pointer to a double where the coefficient will be returned.

Further information

It is quite inefficient to get several coefficients with the `XPRSgetcoef` function. It is better to use `XPRSgetcols` or `XPRSgetrows`.

Related topics

`XPRSgetcols`, `XPRSgetrows`.

XPRSgetcolrange

Purpose

Returns the column ranges computed by XPRSrange.

Synopsis

```
int XPRS_CC XPRSgetcolrange(XPRSprob prob, double upact[], double loact[],
      double uup[], double udn[], double ucost[], double lcost[]);
```

Arguments

prob	The current problem.
upact	Double array of length COLS for upper column activities.
loact	Double array of length COLS for lower column activities.
uup	Double array of length COLS for upper column unit costs.
udn	Double array of length COLS for lower column unit costs.
ucost	Double array of length COLS for upper costs.
lcost	Double array of length COLS for lower costs.

Example

Here the column ranges are retrieved into arrays as in the synopsis:

```
int cols;
double *upact, *loact, *uup, *udn, *ucost, *lcost;
...
XPRSgetintattrib(prob,XPRS_COLS,&cols);
upact = malloc(cols*(sizeof(double)));
loact = malloc(cols*(sizeof(double)));
uup   = malloc(cols*(sizeof(double)));
udn   = malloc(cols*(sizeof(double)));
ucost = malloc(cols*(sizeof(double)));
lcost = malloc(cols*(sizeof(double)));
XPRSrange(prob);
XPRSgetcolrange(prob,upact,loact,uup,udn,ucost,lcost);
```

Further information

The activities and unit costs are obtained from the range file (*problem_name*.rng). The meaning of the upper and lower column activities and upper and lower unit costs in the ASCII range files is described in Appendix A.

Related topics

XPRSgetrowrange, XPRSrange.

XPRSgetcols, XPRSgetcols64

Purpose

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

Synopsis

```
int XPRS_CC XPRSgetcols(XPRSprob prob, int mstart[], int mrwind[], double
      dmatval[], int size, int *nels, int first, int last);

int XPRS_CC XPRSgetcols64(XPRSprob prob, XPRSint64 mstart[], int mrwind[],
      double dmatval[], XPRSint64 size, XPRSint64 *nels, int first, int
      last);
```

Arguments

`prob`	The current problem.
`mstart`	Integer array which will be filled with the indices indicating the starting offsets in the `mrwind` and `dmatval` arrays for each requested column. It must be of length at least `last-first+2`. Column `i` starts at position `mstart[i]` in the `mrwind` and `dmatval` arrays, and has `mstart[i+1]-mstart[i]` elements in it. May be `NULL` if not required.
`mrwind`	Integer array of length `size` which will be filled with the row indices of the nonzero coefficents for each column. May be `NULL` if not required.
`dmatval`	Double array of length `size` which will be filled with the nonzero coefficient values. May be `NULL` if not required.
`size`	The size of the `mrwind` and `dmatval` arrays. This is the maximum number of nonzero coefficients that the Optimizer is allowed to return.
`nels`	Pointer to an integer where the number of nonzero coefficients in the selected columns will be returned. If `nels` exceeds `size`, only the `size` first nonzero coefficients will be returned.
`first`	First column in the range.
`last`	Last column in the range.

Example

The following examples retrieves the number of nonzero coefficients in all columns of the problem:

```
int nels, cols, first = 0, last;
...
XPRSgetintattrib(prob,XPRS_COLS,&cols);
last = cols-1;
XPRSgetcols(prob,NULL,NULL,NULL,0,&nels,first,last);
```

Further information

It is possible to obtain just the number of elements in the range of columns by replacing `mstart`, `mrwind` and `dmatval` by `NULL`, as in the example. In this case, `size` must be set to `0` to indicate that the length of arrays passed is zero. This is demonstrated in the example above.

Related topics

`XPRSgetrows`.

XPRSgetcoltype

Purpose

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

Synopsis

```
int XPRS_CC XPRSgetcoltype(XPRSprob prob, char coltype[], int first, int
      last);
```

Arguments

`prob`	The current problem.
`coltype`	Character array of length `last-first+1` where the column types will be returned: `C` indicates a continuous variable; `I` indicates an integer variable; `B` indicates a binary variable; `S` indicates a semi-continuous variable; `R` indicates a semi-continuous integer variable; `P` indicates a partial integer variable.
`first`	First column in the range.
`last`	Last column in the range.

Example

This example finds the types for all columns in the matrix and prints them to the console:

```
int cols, i;
char *types;
...
XPRSgetintattrib(prob,XPRS_COLS,&cols);
types = (char *)malloc(sizeof(char)*cols);
XPRSgetcoltype(prob,types,0,cols-1);

for(i=0;i<cols;i++) printf("%c\n",types[i]);
```

Related topics

`XPRSchgcoltype`, `XPRSgetrowtype`.

XPRSgetcontrolinfo

Purpose

Accesses the id number and the type information of a control given its name. A control name may be for example `XPRS_PRESOLVE`. Names are case-insensitive and may or may not have the `XPRS_` prefix. The id number is the constant used to identify the control for calls to functions such as `XPRSgetintcontrol`.

The function will return an id number of `0` and a type value of `XPRS_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.

The type information returned will be one of the below integer constants defined in the `xprs.h` header file.

XPRS_TYPE_NOTDEFINED	The name was not recognized.
XPRS_TYPE_INT	32 bit integer.
XPRS_TYPE_INT64	64 bit integer.
XPRS_TYPE_DOUBLE	Double precision floating point.
XPRS_TYPE_STRING	String.

Synopsis

```
int XPRS_CC XPRSgetcontrolinfo(XPRSprob prob, const char* sCaName, int*
      iHeaderId, int* iTypeinfo);
```

Arguments

`prob`	The current problem.
`sCaName`	The name of the control to be queried. Names are case-insensitive and may or may not have the `XPRS_` prefix. A full list of all controls may be found in 9, or from the list in the `xprs.h` header file.
`iHeaderId`	Pointer to an integer where the id number will be returned.
`iTypeInfo`	Pointer to an integer where the type information will be returned.

Example

The following code example obtains the id number and the type information of the control or attribute with name given by `sCaName`. Note that the name happens to be a control name in this example:

```
const char *sCaName = "presolve";
int iHeaderId, iTypeInfo;
...
if(XPRSgetattribinfo(prob, sCaName, &iHeaderId,
                     &iTypeInfo) || iHeaderId==0) {
  if(XPRSgetcontrolinfo(prob, sCaName, &iHeaderId,
                        &iTypeInfo) || iHeaderId==0) {
    printf("Unrecognized name: %s\n", sCaName);
  }
}
```

Related topics

`XPRSgetattribinfo`.

XPRSgetcpcutlist

Purpose

Returns a list of cut indices from the cut pool.

Synopsis

```
int XPRS_CC XPRSgetcpcutlist(XPRSprob prob, int itype, int interp, double
      delta, int *ncuts, int size, XPRScut mcutind[], double dviol[]);
```

Arguments

`prob`	The current problem.
`itype`	The user defined type of the cuts to be returned.
`interp`	Way in which the cut type is interpreted: `-1` get all cuts; `1` treat cut types as numbers; `2` treat cut types as bit maps - get cut if any bit matches any bit set in `itype`; `3` treat cut types as bit maps - get cut if all bits match those set in `itype`.
`delta`	Only those cuts with a signed violation greater than delta will be returned.
`ncuts`	Pointer to the integer where the number of cuts of type `itype` in the cut pool will be returned.
`size`	Maximum number of cuts to be returned.
`mcutind`	Array of length `size` where the pointers to the cuts will be returned.
`dviol`	Double array of length `size` where the values of the signed violations of the cuts will be returned.

Further information

1. The violated cuts can be obtained by setting the `delta` parameter to the size of the (signed) violation required. If unviolated cuts are required as well, `delta` may be set to `XPRS_MINUSINFINITY` which is defined in the library header file.

2. If the number of active cuts is greater than `size`, only `size` cuts will be returned and `ncuts` will be set to the number of active cuts. If `ncuts` is less than `size`, then only `ncuts` positions will be filled in `mcutind`.

3. In case of a cut of type '`L`', the violation equals the negative of the slack associated with the row of the cut. In case of a cut of type '`G`', the violation equals the slack associated with the row of the cut. For cuts of type '`E`', the violation equals the absolute value of the slack.

4. Please note that the violations returned are absolute violations, while feasibility is checked by the Optimizer in the scaled problem.

Related topics

`XPRSdelcpcuts`, `XPRSgetcpcuts`, `XPRSgetcutlist`, `XPRSloadcuts`, `XPRSgetcutmap`, `XPRSgetcutslack`, Section 5.9.

XPRSgetcpcuts, XPRSgetcpcuts64

Purpose

Returns cuts from the cut pool. A list of cut pointers in the array `mindex` must be passed to the routine. The columns and elements of the cut will be returned in the regions pointed to by the `mcols` and `dmatval` parameters. The columns and elements will be stored contiguously and the starting point of each cut will be returned in the region pointed to by the `mstart` parameter.

Synopsis

```
int XPRS_CC XPRSgetcpcuts(XPRSprob prob, const XPRScut mindex[], int ncuts,
      int size, int mtype[], char qrtype[], int mstart[], int mcols[],
      double dmatval[], double drhs[]);

int XPRS_CC XPRSgetcpcuts64(XPRSprob prob, const XPRScut mindex[], int
      ncuts, XPRSint64 size, int mtype[], char qrtype[], XPRSint64
      mstart[], int mcols[], double dmatval[], double drhs[]);
```

Arguments

`prob`	The current problem.
`mindex`	Array of length `ncuts` containing the pointers to the cuts.
`ncuts`	Number of cuts to be returned.
`size`	Maximum number of column indices of the cuts to be returned.
`mtype`	Integer array of length at least `ncuts` where the cut types will be returned. May be NULL if not required.
`qrtype`	Character array of length at least `ncuts` where the sense of the cuts (`L`, `G`, or `E`) will be returned. May be NULL if not required.
`mstart`	Integer array of length at least `ncuts+1` containing the offsets into the `mcols` and `dmatval` arrays. The last element indicates where cut `ncuts+1` would start. May be NULL if not required.
`mcols`	Integer array of length `size` where the column indices of the cuts will be returned. May be NULL if not required.
`dmatval`	Double array of length `size` where the matrix values will be returned. May be NULL if not required.
`drhs`	Double array of length at least `ncuts` where the right hand side elements for the cuts will be returned. May be NULL if not required.

Example

The following example gets the first two cuts:

```
int mtype[2], mstart[3];
int *mcols;
int mindex[] = { 0, 1 };
double drhs[2];
double *dmatval;
char * qrtype;
...
XPRSgetcpcuts(prob,mindex,2,0,NULL,NULL,mstart,NULL,NULL,NULL);
mcols = (int*) malloc(mstart[2]*sizeof(int));
dmatval = (double*) malloc(mstart[2]*sizeof(double));
XPRSgetcpcuts(prob,mindex,2,0,mtype,qrtype,mstart,mcols,dmatval,drhs);
```

Further information

It is possible to obtain just the number of nonzeros in the range of queried cuts by calling the functions with all output arays except for mstart equaling `NULL` and checking the value of `mstart[ncuts]`. In this case, `size` must be set to `0` to indicate that the length of arrays passed is `0`.

Related topics

`XPRSgetcpcutlist`, `XPRSgetcutlist`, 5.9.

XPRSgetcutlist

Purpose

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

Synopsis

```
int XPRS_CC XPRSgetcutlist(XPRSprob prob, int itype, int interp, int
       *ncuts, int size, XPRScut mcutind[]);
```

Arguments

`prob`	The current problem.
`itype`	User defined type of the cuts to be returned. A value of `-1` indicates return all active cuts.
`interp`	Way in which the cut type is interpreted: `-1` get all cuts; `1` treat cut types as numbers; `2` treat cut types as bit maps - get cut if any bit matches any bit set in `itype`; `3` treat cut types as bit maps - get cut if all bits match those set in `itype`.
`ncuts`	Pointer to the integer where the number of active cuts of type `itype` will be returned.
`size`	Maximum number of cuts to be retrieved.
`mcutind`	Array of length `size` where the pointers to the cuts will be returned.

Further information

If the number of active cuts is greater than `size`, then `size` cuts will be returned and `ncuts` will be set to the number of active cuts. If `ncuts` is less than `size`, then only `ncuts` positions will be filled in `mcutind`.

Related topics

`XPRSgetcpcutlist`, `XPRSgetcpcuts`, Section 5.9.

XPRSgetcutmap

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

```
int XPRS_CC XPRSgetcutmap(XPRSprob prob, int ncuts, const XPRScut cuts[],
        int cutmap[]);
```

Arguments

`prob`	The current problem.
`ncuts`	Number of cuts in the cuts array.
`cuts`	Pointer array to the cuts for which the row index is requested.
`cutmap`	Integer array of length `ncuts`, where the row indices are returned.

Further information

For cuts currently not loaded into the problem, a row index of `-1` is returned.

Related topics

`XPRSgetcpcutlist`, `XPRSdelcpcuts`, `XPRSgetcutlist`, `XPRSloadcuts`, `XPRSgetcutslack`, `XPRSgetcpcuts`, Section 5.9.

XPRSgetcutslack

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

```
int XPRS_CC XPRSgetcutslack(XPRSprob prob, XPRScut cut, double* dslack);
```

Arguments

`prob`	The current problem.
`cuts`	Pointer of the cut for which the slack is to be calculated.
`dslack`	Double pointer where the value of the slack is returned.

Related topics

`XPRSgetcpcutlist`, `XPRSdelcpcuts`, `XPRSgetcutlist`, `XPRSloadcuts`, `XPRSgetcutmap`, `XPRSgetcpcuts`, Section 5.9.

XPRSgetdaysleft

Purpose

Returns the number of days left until the license expires.

Synopsis

```
int XPRS_CC XPRSgetdaysleft(int *days);
```

Argument

`days`	Pointer to an integer where the number of days is to be returned. For a permanent license, the return value will be `XPRS_MAXINT`

Example

The following calls `XPRSgetdaysleft` to print information about the license:

```
int days;
...
XPRSinit(NULL);
if(XPRSgetdaysleft(&days) != 0) {
  printf("An error occurred\n");
} else if (days==XPRS_MAXINT) {
  printf("License will never expire\n");
} else {
  printf("License expires in %d days\n", days);
}
```

Related topics

`XPRSgetbanner`.

XPRSgetdblattrib

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

Synopsis
```
int XPRS_CC XPRSgetdblattrib(XPRSprob prob, int ipar, double *dval);
```

Arguments
`prob`	The current problem.
`ipar`	Problem attribute whose value is to be returned. A full list of all available problem attributes may be found in Chapter 10, or from the list in the `xprs.h` header file.
`dval`	Pointer to a double where the value of the problem attribute will be returned.

Example
The following obtains the optimal value of the objective function and displays it to the console:

```
double lpobjval;
...
XPRSlpoptimize(prob,"");
XPRSgetdblattrib(prob,XPRS_LPOBJVAL,&lpobjval);
printf("The maximum profit is %f\n",lpobjval);
```

Related topics
`XPRSgetintattrib`, `XPRSgetstrattrib`.

XPRSgetdblcontrol

Purpose

Retrieves the value of a given double control parameter.

Synopsis

```
int XPRS_CC XPRSgetdblcontrol(XPRSprob prob, int ipar, double *dgval);
```

Arguments

`prob`	The current problem.
`ipar`	Control parameter whose value is to be returned. A full list of all controls may be found in Chapter 9, or from the list in the `xprs.h` header file.
`dgval`	Pointer to the location where the control value will be returned.

Example

The following returns the integer feasibility tolerance:

```
XPRSgetdblcontrol(prob,XPRS_MIPTOL,&miptol);
```

Related topics

`XPRSsetdblcontrol`, `XPRSgetintcontrol`, `XPRSgetstrcontrol`.

XPRSgetdirs

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, `XPRSgetdirs` will get the directives for the presolved problem.

Synopsis

```
int XPRS_CC XPRSgetdirs(XPRSprob prob, int *ndir, int mcols[], int mpri[],
        char qbr[], double dupc[], double ddpc[]);
```

Arguments

`prob`	The current problem.
`ndir`	Pointer to an integer where the number of directives will be returned.
`mcols`	Integer array of length `ndir` containing the column numbers (`0`, `1`, `2`,...) or negative values corresponding to special ordered sets (the first set numbered `-1`, the second numbered `-2`,...). May be NULL if not required.
`mpri`	Integer array of length `ndir` containing the priorities for the columns and sets, where columns/sets with smallest priority will be branched on first. May be NULL if not required.
`qbr`	Character array of length `ndir` specifying the branching direction for each column or set: `U` the entity is to be forced up; `D` the entity is to be forced down; `N` not specified.
`dupc`	Double array of length `ndir` containing the up pseudo costs for the columns and sets. May be NULL if not required.
`ddpc`	Double array of length `ndir` containing the down pseudo costs for the columns and sets. May be NULL if not required.

Further information

The value `ndir` denotes the number of directives, at most `MIPENTS`, obtainable with `XPRSgetintattrib(prob,XPRS_MIPENTS,& mipents);`.

Related topics

`XPRSloaddirs`, `XPRSloadpresolvedirs`.

XPRSgetdualray

Purpose

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

Synopsis

```
int XPRS_CC XPRSgetdualray(XPRSprob prob, double dray[], int *hasRay);
```

Arguments

`prob`	The current problem.
`dray`	Double array of length `ROWS` to hold the ray. May be NULL if not required.
`hasRay`	This variable will be set to 1 if the Optimizer is able to return a dual ray, 0 otherwise.

Example

The following code tries to retrieve a dual ray:

```
int rows;
double *dualRay;
int hasRay;
...
XPRSgetintattrib(prob, XPRS_ROWS, &rows);
dualRay = malloc(rows*sizeof(double));
XPRSgetdualray(prob, dualRay, &hasRay);
if(!hasRay) printf("Could not retrieve a dual ray\n");
```

Further information

1. It is possible to retrieve a dual ray only when, after solving an LP problem, the final status (`LPSTATUS`) is `XPRS_LP_INFEAS`.
2. Dual rays are not post-solved. If the problem is in a presolved state, the dual ray that is returned will be for the presolved problem. If the problem was solved with presolve on and has been restored to the original state (the default behavior), this function will not be able to return a ray. To ensure that a dual ray can be obtained, it is recommended to solve a problem with presolve turned off (`PRESOLVE` = 0).

Related topics

`XPRSgetprimalray`.

XPRSgetgencons, XPRSgetgencons64

Purpose

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

Synopsis

```
int XPRS_CC XPRSgetgencons(XPRSprob prob, int type[], int resultant[], int
      colstart[], int col[], int colsize, int *ncols, int valstart[],
      double val[], int valsize, int *nvals, int first, int last);

int XPRS_CC XPRSgetgencons64(XPRSprob prob, int type[], int resultant[],
      XPRSint64 colstart[], int col[], XPRSint64 colsize, XPRSint64 *ncols,
      XPRSint64 valstart[], double val[], XPRSint64 valsize, XPRSint64
      *nvals, int first, int last);
```

Arguments

`prob`	The current problem.
`type`	`NULL` if not required or an integer array of length at least `last-first+1` which will be filled with the types of the general constraints: `XPRS_GENCONS_MAX (0)` indicates a `maximum` constraint; `XPRS_GENCONS_MIN (1)` indicates a `minimum` constraint; `XPRS_GENCONS_AND (2)` indicates an `and` constraint. `XPRS_GENCONS_OR (3)` indicates an `or` constraint; `XPRS_GENCONS_ABS (4)` indicates an `absolute value` constraint.
`resultant`	Integer array which will be filled with the indices of the output variables `y`. It must be of length at least `last-first+1`. May be `NULL` if not required.
`colstart`	Integer array of length at least `last-first+2` which will be filled with the start index of each general constraint in the `col` array. May be `NULL` if not required.
`col`	Integer array which will be filled with the indices of the input variables `xi`. May be `NULL` if not required.
`colsize`	Maximum number of input columns to be retrieved.
`ncols`	Pointer to return the number of input columns in the `col` array. If the number of input columns is greater than `colsize`, then only `colsize` elements will be returned. May be `NULL` if not required.
`valstart`	Integer array of length at least `last-first+2` which will be filled with the start index of each general constraint in the `val` array. May be `NULL` if not required.
`val`	Integer array which will be filled with the constant values `ci`. May be `NULL` if not required.
`valsize`	Maximum number of constant values to be retrieved.
`nvals`	Pointer to return the number of constant values in the `val` array. If the number of constant values is greater than `valsize`, then only `valsize` elements will be returned.May be `NULL` if not required.
`first`	First general constraint in the range.
`last`	Last general constraint in the range.

Example

The following example retrieves all general constraints:

```
int ngencons;
int *type;
int *resultant;
int *colstart;
int *col;
```

```
int colsize;
int ncols;
int *valstart;
int *val;
int valsize;
int nvals;
...
XPRSgetdblattrib(prob, XPRS_GENCONS, &ngencons);
XPRSgetgencons(prob, NULL, NULL, NULL, NULL, 0, &colsize, NULL, NULL, 0, &vals
type = (int*) malloc(ngencons*sizeof(int));
resultant = (int*) malloc(ngencons*sizeof(int));
colstart = (int*) malloc((ngencons+1)*sizeof(int));
col = (int*) malloc(colsize*sizeof(int));
valstart = (int*) malloc((ngencons+1)*sizeof(int));
val = (double*) malloc(valsize*sizeof(double));
XPRSgetgencons(prob, type, resultant, colstart, col, colsize, &ncols, valstart
...
```

Further information

It is possible to obtain just the number of input columns and/or constant values in the range of general constraints by calling this function with `colsize` and `valsize` set to `0`, in which case the required size for the arrays will be returned in `ncols` and `nvals` (one of them may be `NULL` if only the other is required).

Related topics

`XPRSaddgencons`, `XPRSdelgencons`.

XPRSgetglobal, XPRSgetglobal64

Purpose

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

Synopsis

```
int XPRS_CC XPRSgetglobal(XPRSprob prob, int *nglents, int *sets, char
      qgtype[], int mgcols[], double dlim[], char qstype[], int msstart[],
      int mscols[], double dref[]);

int XPRS_CC XPRSgetglobal64(XPRSprob prob, int *nglents, int *sets, char
      qgtype[], int mgcols[], double dlim[], char qstype[], XPRSint64
      msstart[], int mscols[], double dref[]);
```

Arguments

`prob`	The current problem.
`nglents`	Pointer to the integer where the number of binary, integer, semi-continuous, semi-continuous integer and partial integer entities will be returned. This is equal to the problem attribute `MIPENTS`.
`sets`	Pointer to the integer where the number of SOS1 and SOS2 sets will be returned. It can be retrieved from the problem attribute `SETS`.
`qgtype`	Character array of length `nglents` where the entity types will be returned. The types will be one of: `B` binary variables; `I` integer variables; `P` partial integer variables; `S` semi-continuous variables; `R` semi-continuous integer variables.
`mgcols`	Integer array of length `nglents` where the column indices of the global entities will be returned.
`dlim`	Double array of length `nglents` where the limits for the partial integer variables and lower bounds for the semi-continuous and semi-continuous integer variables will be returned (any entries in the positions corresponding to binary and integer variables will be meaningless).
`qstype`	Character array of length `sets` where the set types will be returned. The set types will be one of: `1` SOS1 type sets; `2` SOS2 type sets.
`msstart`	Integer array where the offsets into the `mscols` and `dref` arrays indicating the start of the sets will be returned. This array must be of length `sets+1`, the final element will contain the offset where set `sets+1` would start and equals the length of the `mscols` and `dref` arrays, `SETMEMBERS`.
`mscols`	Integer array of length `SETMEMBERS` where the columns in each set will be returned.
`dref`	Double array of length `SETMEMBERS` where the reference row entries for each member of the sets will be returned.

Example

The following obtains the global variables and their types in the arrays `mgcols` and `qrtype`:

```
int nglents, nsets, *mgcols;
char *qgtype;
...
XPRSgetglobal(prob,&nglents,&nsets,NULL,NULL,NULL,NULL,
```

```
                 NULL,NULL,NULL);
mgcols = malloc(nglents*sizeof(int));
qgtype = malloc(nglents*sizeof(char));
XPRSgetglobal(prob,&nglents,&nsets,qgtype,ngcols,NULL,
                 NULL,NULL,NULL,NULL);
```

Further information

Any of the arguments except `prob`, `nglents` and `sets` may be `NULL` if not required.

Related topics

`XPRSloadglobal`, `XPRSloadqglobal`.

XPRSgetiisdata

Purpose

Returns information for an Irreducible Infeasible Set: size, variables (row and column vectors) and conflicting sides of the variables, duals and reduced costs.

Synopsis

```
int XPRS_CC XPRSgetiisdata(XPRSprob prob, int num, int *rownumber, int
      *colnumber, int miisrow[], int miiscol[], char constrainttype[], char
      colbndtype[], double duals[], double rdcs[], char isolationrows[],
      char isolationcols[]);
```

Arguments

`prob` The current problem.

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

`rownumber` Pointer to an integer where the number of rows in the IIS will be returned.

`colnumber` Pointer to an integer where the number of bounds in the IIS will be returned.

`miisrow` Indices of rows in the IIS. Can be NULL if not required.

`miiscol` Indices of bounds (columns) in the IIS. Can be NULL if not required.

`constrainttype` Sense of rows in the IIS:
- `L` for less or equal row;
- `G` for greater or equal row.
- `E` for an equality row (for a non LP IIS);
- `1` for a SOS1 row;
- `2` for a SOS2 row;
- `I` for an indicator row.

Can be NULL if not required.

`colbndtype` Sense of bound in the IIS:
- `U` for upper bound;
- `L` for lower bound.
- `F` for fixed columns (for a non LP IIS);
- `B` for a binary column;
- `I` for an integer column;
- `P` for a partial integer columns;
- `S` for a semi-continuous column;
- `R` for a semi-continuous integer column.

Can be NULL if not required.

`duals` The dual multipliers associated with the rows. Can be NULL if not required.

`rdcs` The dual multipliers (reduced costs) associated with the bounds. Can be NULL if not required.

`isolationrows` The isolation status of the rows:
- `-1` if isolation information is not available for row (run iis isolations);
- `0` if row is not in isolation;
- `1` if row is in isolation.

Can be NULL if not required.

`isolationcols` The isolation status of the bounds:
- `-1` if isolation information is not available for column (run iis isolations);
- `0` if column is not in isolation;
- `1` if column is in isolation. Can be NULL if not required.

Example

This example first retrieves the size of IIS 1, then gets the detailed information for the IIS.

```
XPRSgetiisdata(myprob, 1, &nrow, &ncol, NULL, NULL, NULL, NULL,
```

```
                NULL,NULL,NULL,NULL);

rows = malloc(nrow*sizeof(int));
cols = malloc(ncol*sizeof(int));
constrainttype = malloc(nrow);
colbndtype = malloc(ncol);
duals = malloc(nrow*sizeof(double));
rdcs = malloc(ncol*sizeof(double));
isolationrows = malloc(nrow);
isolationcols = malloc(ncol);
XPRSgetiisdata(myprob, 1, &nrow, &ncol, rows, cols, constrainttype,
               colbndtype, duals, rdcs, isolationrows, isolationcols);
```

Further information

1. Calling `IIS` from the console automatically prints most of the above IIS information to the screen. Extra information can be printed with the `IIS -p` command.
2. IISs are numbered from 1 to `NUMIIS`. Index number 0 refers to the IIS approximation.
3. If `miisrow` and `miiscol` both are NULL, only the `rownumber` and `colnumber` are returned.
4. The arrays may be NULL if not required. However, arrays `constrainttype`, `duals` and `isolationrows` are only returned if `miisrow` is not NULL. Similarly, arrays `colbndtype`, `rdcs` and `isolationcols` are only returned if `miiscol` is not NULL.
5. All the non NULL arrays should be of length `rownumber` or `colnumber`, respectively.
6. For the initial IIS approximation (`num` = 0) the number of rows and columns with a nonzero Lagrange multiplier (dual/reduced cost respectively) are returned. Please note that, in such cases, it might be necessary to call `XPRSiisstatus` to retrieve the necessary size of the return arrays.
7. If there are Special Ordered Sets in the IIS, their number is included in the `miisrow` array.
8. For non LP IISs, some column indices may appear more than once in the `miiscol` array, for example an integrality and a bound restriction for the same column.
9. Duals, reduced cost and isolation information is not available for nonlinear IIS problems, and for those the arrays are filled with zero values in case they are provided.

Related topics

`XPRSiisall`, `XPRSiisclear`, `XPRSiisfirst`, `XPRSiisisolations`, `XPRSiisnext`, `XPRSiisstatus`, `XPRSiiswrite`, `IIS`, Section A.7.

XPRSgetindex

Purpose

Returns the index for a specified row or column name.

Synopsis

```
int XPRS_CC XPRSgetindex(XPRSprob prob, int type, const char *name, int
        *seq);
```

Arguments

`prob`	The current problem.	
`type`	`1`	if a row index is required;
	`2`	if a column index is required.
`name`	Null terminated string.	
`seq`	Pointer of the integer where the row or column index number will be returned. A value of `-1` will be returned if the row or column does not exist.	

Example

The following example loads `problem` and checks to see if "`n 0203`" is the name of a row or column:

```
int seqr, seqc;
...
XPRSreadprob(prob,"problem","");

XPRSgetindex(prob,1,"n   0203", &seqr);
XPRSgetindex(prob,2,"n   0203", &seqc);
if(seqr==-1 && seqc ==-1) printf("n   0203 not there\n");
if(seqr!= -1) printf("n   0203 is row %d\n",seqr);
if(seqc!= -1) printf"n   0203 is column %d\n",seqc);
```

Related topics

XPRSaddnames.

XPRSgetindicators

Purpose

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

Synopsis

```
int XPRS_CC XPRSgetindicators(XPRSprob prob, int inds[], int comps[], int
      first, int last);
```

Arguments

`prob`	The current problem.
`inds`	Integer array of length `last-first+1` where the column indices of the indicator variables are to be placed.
`comps`	Integer array of length `last-first+1` where the indicator complement flags will be returned: `0` not an indicator constraint (in this case the corresponding entry in the `inds` array is ignored); `1` for indicator constraints with condition "`bin = 1`"; `-1` for indicator constraints with condition "`bin = 0`";
`first`	First row in the range.
`last`	Last row in the range (inclusive).

Example

The following example retrieves information about all indicator constraints in the matrix and prints a list of their indices.

```
int i, rows;
double *inds, *comps;
...
XPRSgetintattrib(prob,XPRS_ROWS,&rows);
inds = malloc(rows*(sizeof(int)));
comps = malloc(rows*(sizeof(int)));
XPRSgetindicators(prob,inds,comps,0,rows-1);

printf("Indicator rows:");
for(i=0; i<rows; i++) if(comps[i]!=0) printf(" %d", i);
printf("\n");
```

Related topics

XPRSsetindicators, XPRSdelindicators.

XPRSgetinfeas

Purpose

Returns a list of infeasible primal and dual variables.

Synopsis

```
int XPRS_CC XPRSgetinfeas(XPRSprob prob, int *npv, int *nps, int *nds, int
      *ndv, int mx[], int mslack[], int mdual[], int mdj[]);
```

Arguments

`prob`	The current problem.
`npv`	Pointer to an integer where the number of primal infeasible variables is returned.
`nps`	Pointer to an integer where the number of primal infeasible rows is returned.
`nds`	Pointer to an integer where the number of dual infeasible rows is returned.
`ndv`	Pointer to an integer where the number of dual infeasible variables is returned.
`mx`	Integer array of length `npv` where the primal infeasible variables will be returned. May be `NULL` if not required.
`mslack`	Integer array of length `nps` where the primal infeasible rows will be returned. May be `NULL` if not required.
`mdual`	Integer array of length `nds` where the dual infeasible rows will be returned. May be `NULL` if not required.
`mdj`	Integer array of length `ndv` where the dual infeasible variables will be returned. May be `NULL` if not required.

Error values

`91`	A current problem is not available.
`422`	A solution is not available.

Related controls

Double

`FEASTOL`	Tolerance on RHS.
`OPTIMALITYTOL`	Reduced cost tolerance.

Example

In this example, `XPRSgetinfeas` is first called with nulled integer arrays to get the number of infeasible entries. Then space is allocated for the arrays and the function is again called to fill them in:

```
int npv, nps, nds, ndv, *mx, *mslack, *mdual, *mdj;
...
XPRSgetinfeas(prob, &npv, &nps, &nds, &ndv,
              NULL, NULL, NULL, NULL);
mx = malloc(npv * sizeof(*mx));
mslack = malloc(nps * sizeof(*mslack));
mdual = malloc(nds * sizeof(*mdual));
mdj = malloc(ndv * sizeof(*mdj));
XPRSgetinfeas(prob, &npv, &nps, &nds, &ndv,
              mx, mslack, mdual, mdj);
```

Further information

1. To find the infeasibilities in a previously saved solution, the solution must first be loaded into memory with the `XPRSreadbinsol` (`READBINSOL`) function.

2. If any of the last four arguments are set to `NULL`, the corresponding number of infeasibilities is still returned.

Related topics

`XPRSgetscaledinfeas`, `XPRSgetiisdata`, `XPRSiisall`, `XPRSiisclear`, `XPRSiisfirst`, `XPRSiisisolations`, `XPRSiisnext`, `XPRSiisstatus`, `XPRSiiswrite`, `IIS`.

XPRSgetintattrib, XPRSgetintattrib64

Purpose

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

Synopsis

```
int XPRS_CC XPRSgetintattrib(XPRSprob prob, int ipar, int *ival);

int XPRS_CC XPRSgetintattrib64(XPRSprob prob, int ipar, XPRSint64 *ival);
```

Arguments

`prob`	The current problem.
`ipar`	Problem attribute whose value is to be returned. A full list of all problem attributes may be found in Chapter 10, or from the list in the `xprs.h` header file.
`ival`	Pointer to an integer where the value of the problem attribute will be returned.

Example

The following obtains the number of columns in the matrix and allocates space to obtain lower bounds for each column:

```
int cols;
double *lb;
...
XPRSgetintattrib(prob,XPRS_COLS,&cols);
lb = (double *) malloc(sizeof(double)*cols);
XPRSgetlb(prob,lb,0,cols-1);
```

Related topics

`XPRSgetdblattrib`, `XPRSgetstrattrib`.

XPRSgetintcontrol, XPRSgetintcontrol64

Purpose

Enables users to recover the values of various integer control parameters

Synopsis

```
int XPRS_CC XPRSgetintcontrol(XPRSprob prob, int ipar, int *igval);

int XPRS_CC XPRSgetintcontrol64(XPRSprob prob, int ipar, XPRSint64 *igval);
```

Arguments

`prob`	The current problem.
`ipar`	Control parameter whose value is to be returned. A full list of all controls may be found in Chapter 9, or from the list in the `xprs.h` header file.
`igval`	Pointer to an integer where the value of the control will be returned.

Example

The following obtains the value of `DEFAULTALG` and outputs it to screen:

```
int defaultalg;
...
XPRSlpoptimize(prob,"");
XPRSgetintcontrol(prob,XPRS_DEFAULTALG,&defaultalg);
printf("DEFAULTALG is %d\n",defaultalg);
```

Further information

Some control parameters, such as `SCALING`, are bitmaps. Each bit controls a different behavior. If set, bit `0` has value `1`, bit `1` has value `2`, bit `2` has value `4`, and so on.

Related topics

`XPRSsetintcontrol`, `XPRSgetdblcontrol`, `XPRSgetstrcontrol`.

XPRSgetlastbarsol

Purpose

Used to obtain the last barrier solution values following optimization that used the barrier solver.

Synopsis

```
int XPRS_CC XPRSgetastbarsol(XPRSprob prob, double x[], double slack[],
       double dual[], double dj[], int *barsolstatus);
```

Arguments

`prob`	The current problem.
`x`	Double array of length `COLS` where the values of the primal variables will be returned. May be `NULL` if not required.
`slack`	Double array of length `ROWS` where the values of the slack variables will be returned. May be `NULL` if not required.
`dual`	Double array of length `ROWS` where the values of the dual variables ($c_B^T B^{-1}$) will be returned. May be `NULL` if not required.
`dj`	Double array of length `COLS` where the reduced cost for each variable ($c^T - c_B^T B^{-1} A$) will be returned. May be `NULL` if not required.
`barsolstatus`	Status of the last barrier solve. Value matches that of XPRS_LPSTATUS should the solve have been stopped immediately after the barrier.

Further information

1. If the barrier solver has not been used, `barsolstatus` will return `XPRS_LP_UNSOLVED`.
2. The barrier solution or the solution candidate is always available if the status is not `XPRS_LP_UNSOLVED`.
3. The last barrier solution is available until the next solve, and is not invalidated by otherwise working with the problem.

Related topics

`XPRSgetlpsol`

XPRSgetlasterror

Purpose

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

Synopsis

```
int XPRS_CC XPRSgetlasterror(XPRSprob prob, char *errmsg);
```

Arguments

`prob`	The current problem.
`errmsg`	A 512 character buffer where the last error message will be returned.

Example

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

```
void error(XPRSprob myprob, char *function)
{
  char errmsg[512];
  XPRSgetlasterror(myprob,errmsg);
  printf("Function %s did not execute correctly: %s\n",
         function, errmsg);
  XPRSdestroyprob(myprob);
  XPRSfree();
  exit(1);
}
```

where the main function might contain lines such as:

```
XPRSprob prob;
...
if(XPRSreadprob(prob,"myprob",""))
  error(prob,"XPRSreadprob");
```

Related topics

`ERRORCODE`, `XPRSaddcbmessage`, `XPRSsetlogfile`, Chapter 11.

XPRSgetlb

Purpose

Returns the lower bounds for the columns in a given range.

Synopsis

```
int XPRS_CC XPRSgetlb(XPRSprob prob, double lb[], int first, int last);
```

Arguments

`prob`	The current problem.
`lb`	Double array of length `last-first+1` where the lower bounds are to be placed.
`first`	First column in the range.
`last`	Last column in the range.

Example

The following example retrieves the lower bounds for the columns of the current problem:

```
int cols;
double *lb;
...
XPRSgetintattrib(prob,XPRS_COLS,&cols);
lb = (double *) malloc(sizeof(double)*cols);
XPRSgetlb(prob,lb,0,cols-1);
```

Further information

Values greater than or equal to `XPRS_PLUSINFINITY` should be interpreted as infinite; values less than or equal to `XPRS_MINUSINFINITY` should be interpreted as infinite and negative.

Related topics

`XPRSchgbounds`, `XPRSgetub`.

XPRSgetlicerrmsg

Purpose

Retrieves an error message describing the last licensing error, if any occurred.

Synopsis

```
int XPRS_CC XPRSgetlicerrmsg(char *buffer, int length);
```

Arguments

`buffer`	Buffer long enough to hold the error message (including a null terminator).
`length`	Length of the buffer.

Example

The following calls `XPRSgetlicerrmsg` to find out why `XPRSinit` failed:

```
char message[512];
...
if(XPRSinit(NULL))
{
  XPRSgetlicerrmsg(message,512);
  printf("%s\n", message);
}
```

Further information

1. The error message includes an error code, which in case the user wishes to use it is also returned by the function. If there was no licensing error the function returns 0.
2. It's recommended that you pass a buffer of at least 2048 bytes as licensing errors can be qute long. If the error message is too large to fit in the buffer, the first `length-1` characters will be returned.

Related topics

`XPRSinit`.

XPRSgetlpsol

Purpose

Used to obtain the LP solution values following optimization.

Synopsis

```
int XPRS_CC XPRSgetlpsol(XPRSprob prob, double x[], double slack[], double
      dual[], double dj[]);
```

Arguments

prob	The current problem.
x	Double array of length COLS where the values of the primal variables will be returned. May be NULL if not required.
slack	Double array of length ROWS where the values of the slack variables will be returned. May be NULL if not required.
dual	Double array of length ROWS where the values of the dual variables ($c_B^T B^{-1}$) will be returned. May be NULL if not required.
dj	Double array of length COLS where the reduced cost for each variable ($c^T - c_B^T B^{-1} A$) will be returned. May be NULL if not required.

Example

The following sequence of commands will get the LP solution (x) at the top node of a MIP and the optimal MIP solution (y):

```
int cols;
double *x, *y;
...
XPRSmipoptimize(prob,"l");
XPRSgetintattrib(prob,XPRS_ORIGINALCOLS,&cols);
x = malloc(cols*sizeof(double));
XPRSgetlpsol(prob,x,NULL,NULL,NULL);
XPRSmipoptimize(prob,"");
y = malloc(cols*sizeof(double));
XPRSgetmipsol(prob,y,NULL);
```

Further information

1. If called during a global callback the solution of the current node will be returned.
2. When an integer solution is found during a global search, it is always set up as a solution to the current node; therefore the integer solution is available as the current node solution and can be retrieved with XPRSgetlpsol and XPRSgetpresolvesol.
3. If the matrix is modified after calling XPRSlpoptimize, then the solution will no longer be available.
4. If the problem has been presolved, then XPRSgetlpsol returns the solution to the original problem. The only way to obtain the presolved solution is to call the related function, XPRSgetpresolvesol.

Related topics

XPRSgetlpsolval, XPRSgetpresolvesol, XPRSgetmipsol, XPRSwriteprtsol, XPRSwritesol.

XPRSgetlpsolval

Purpose

Used to obtain a single LP solution value following optimization.

Synopsis

```
int XPRS_CC XPRSgetlpsolval(XPRSprob prob, int col, int row, double *x,
        double *slack, double *dual, double *dj);
```

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.
`x`	Double pointer where the value of the primal variable will be returned. May be `NULL` if not required.
`slack`	Double pointer where the value of the slack variable will be returned. May be `NULL` if not required.
`dual`	Double pointer where the value of the dual variable ($c_B^T B^{-1}$) will be returned. May be `NULL` if not required.
`dj`	Double pointer where the reduced cost for the variable ($c^T - c_B^T B^{-1} A$) will be returned. May be `NULL` if not required.

Further information

This function is currently not supported if the problem is in a presolved state.

Related topics

`XPRSgetlpsol`, `XPRSgetpresolvesol`, `XPRSgetmipsol`, `XPRSwriteprtsol`, `XPRSwritesol`.

XPRSgetmessagestatus

Purpose

Retrieves the current suppression status of a message.

Synopsis

```
int XPRS_CC XPRSgetmessagestatus(XPRSprob prob, int errcode, int *status);
```

Arguments

`prob`	The problem to check for the suppression status of the message error code. Use `NULL` to check for the global suppression status of the message errcode.
`errcode`	The id number of the message. Refer to Chapter 11 for a list of possible message numbers.
`status`	Non-zero if the message is not suppressed; `0` otherwise.

Further information

If a message is suppressed globally then the message will always have `status` return zero from `XPRSgetmessagestatus` when prob is non-NULL.

Related topics

`XPRSsetmessagestatus`.

XPRSgetmipsol

Purpose

Used to obtain the solution values of the last MIP solution that was found.

Synopsis

```
int XPRS_CC XPRSgetmipsol(XPRSprob prob, double x[], double slack[]);
```

Arguments

`prob`	The current problem.
`x`	Double array of length COLS where the values of the primal variables will be returned. May be `NULL` if not required.
`slack`	Double array of length ROWS where the values of the slack variables will be returned. May be `NULL` if not required.

Example

The following sequence of commands will get the solution (`x`) of the last MIP solution for a problem:

```
int cols;
double *x;
...
XPRSmipoptimize(prob,"");
XPRSgetintattrib(prob,XPRS_ORIGINALCOLS,&cols);
x = malloc(cols*sizeof(double));
XPRSgetmipsol(prob,x,NULL);
```

Further information

1. **Warning:** If allocating space for the MIP solution the row and column sizes must be obtained for the original problem and not for the presolve problem. They can be obtained before optimizing or after calling `XPRSpostsolve` for the case where the global search has not completed.
2. During a global `intsol` or `preintsol` callback, in order to retrieve the corresponding integer solution, use either `XPRSgetlpsol` or `XPRSgetpresolvesol`, not `XPRSgetmipsol` (see the documentation of these callbacks for an explanation).

Related topics

`XPRSgetmipsolval`, `XPRSgetpresolvesol`, `XPRSwriteprtsol`, `XPRSwritesol`.

XPRSgetmipsolval

Purpose

Used to obtain a single solution value of the last MIP solution that was found.

Synopsis

```
int XPRS_CC XPRSgetmipsolval(XPRSprob prob, int col, int row, double *x,
       double *slack);
```

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.
`x`	Double pointer where the value of the primal variable will be returned. May be `NULL` if not required.
`slack`	Double pointer where the value of the slack variable will be returned. May be `NULL` if not required.

Related topics

`XPRSgetmipsol`, `XPRSgetpresolvesol`, `XPRSwriteprtsol`, `XPRSwritesol`.

XPRSgetmqobj, XPRSgetmqobj64

Purpose

Returns the nonzeros in the quadratic objective coefficients matrix for the columns in a given range. To achieve maximum efficiency, `XPRSgetmqobj` returns the lower triangular part of this matrix only.

Synopsis

```
int XPRS_CC XPRSgetmqobj (XPRSprob prob, int mstart[], int mclind[], double
       dobjval[], int size, int *nels, int first, int last);

int XPRS_CC XPRSgetmqobj64 (XPRSprob prob, XPRSint64 mstart[], int
       mclind[], double dobjval[], XPRSint64 size, XPRSint64 *nels, int
       first, int last);
```

Arguments

`prob`	The current problem.
`mstart`	Integer array which will be filled with indices indicating the starting offsets in the `mclind` and `dobjval` arrays for each requested column. It must be length of at least `last-first+2`. Column `i` starts at position `mstart[i]` in the `mrwind` and `dmatval` arrays, and has `mstart[i+1]-mstart[i]` elements in it. May be NULL if `size` is `0`.
`mclind`	Integer array of length `size` which will be filled with the column indices of the nonzero elements in the lower triangular part of `Q`. May be `NULL` if `size` is `0`.
`dobjval`	Double array of length `size` which will be filled with the nonzero element values. May be `NULL` if `size` is `0`.
`size`	The maximum number of elements to be returned (size of the arrays).
`nels`	Pointer to an integer where the number of nonzero quadratic objective coefficients will be returned. If the number of nonzero coefficients is greater than `size`, then only `size` elements will be returned. If `nels` is smaller than `size`, then only `nels` will be returned.
`first`	First column in the range.
`last`	Last column in the range.

Further information

The objective function is of the form $c^T x + 0.5 x^T Q x$ where `Q` is positive semi-definite for minimization problems and negative semi-definite for maximization problems. If this is not the case the optimization algorithms may converge to a local optimum or may not converge at all. Note that only the upper or lower triangular part of the `Q` matrix is returned.

Related topics

`XPRSchgmqobj`, `XPRSchgqobj`, `XPRSgetqobj`.

XPRSgetnamelist

Purpose

Returns the names for the rows, columns, sets, piecewise linear of general constraints 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

```
int XPRS_CC XPRSgetnamelist(XPRSprob prob, int type, char names[], int
        names_len, int * names_len_reqd, int first, int last);
```

Arguments

`prob`	The current problem.
`type`	`1` if row names are required; `2` if column names are required. `3` if set names are required. `4` if piecewise linear constraint names are required. `5` if general constraint names are required.
`names`	A buffer into which the names will be returned as a sequence of null-terminated strings. The buffer should be of length `names_len` bytes. May be NULL if `names_len` is `0`.
`names_len`	The maximum number of bytes that may be written to the buffer `names`.
`names_len_reqd`	A pointer to a variable into which will be written the number of bytes required to contain the names in the specified range. May be NULL if not 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.

Example

The following example retrieves and outputs the row and column names for the current problem.

```
int i, o, cols, rows, cnames_len, rnames_len;
char *cnames, *rnames;
...
/* Get problem size */
XPRSgetintattrib(prob,XPRS_COLS,&cols);
XPRSgetintattrib(prob,XPRS_ROWS,&rows);
/* Request number of bytes required to retrieve the names */
XPRSgetnamelist(prob,1,NULL,0,&rnames_len,0,rows-1);
XPRSgetnamelist(prob,2,NULL,0,&cnames_len,0,cols-1);

/* Now allocate buffers big enough then fetch the names */
cnames = (char *) malloc(sizeof(char)*cnames_len);
rnames = (char *) malloc(sizeof(char)*rnames_len);
XPRSgetnamelist(prob,1,rnames,rnames_len,NULL,0,rows-1);
XPRSgetnamelist(prob,2,cnames,cnames_len,NULL,0,cols-1);

/* Output row names */
o=0;
for (i=0;i<rows;i++) {
  printf("Row #%d: %s\n", i, rnames+o);
  o += strlen(rnames+o)+1;
}
/* Output column names */
o=0;
for (i=0;i<cols;i++) {
  printf("Col #%d: %s\n", i, cnames+o);
```

```
          o += strlen(cnames+o)+1;
        }
```

Related topics

XPRSaddnames.

XPRSgetnamelistobject

Purpose

Returns the `XPRSnamelist` object for the rows, columns or sets of a problem. The names stored in this object can be queried using the `XPRS_nml_` functions.

Synopsis

```
int XPRS_CC XPRSgetnamelistobject(XPRSprob prob, int itype, XPRSnamelist
        *r_nml);
```

Arguments

`prob`	The current problem.	
`itype`	1	if the row name list is required;
	2	if the column name list is required;
	3	if the set name list is required.
	4	if piecewise linear constraint name list is required.
	5	if general constraint name list is required.
`r_nml`	Pointer to a variable holding the name list contained by the problem.	

Further information

The `XPRSnamelist` object is a map of names to and from indices.

Related topics

None.

XPRSgetnames

Purpose

Returns the names for the rows, columns, sets, piecewise linear or general constraints in a given range. The names will be returned in a character buffer, each name being separated by a null character.

Synopsis

```
int XPRS_CC XPRSgetnames(XPRSprob prob, int type, char names[], int first,
        int last);
```

Arguments

`prob`	The current problem.
`type`	`1` if row names are required; `2` if column names are required. `3` if set names are required. `4` if piecewise linear constraint names are required. `5` if general constraint names are required.
`names`	Buffer long enough to hold the names. Since each name is `8*NAMELENGTH` characters long (plus a null terminator), the array, `names`, would be required to be at least as long as (`first-last+1`)*(`8*NAMELENGTH+1`) characters. The names of the row/column/set `first+i` will be written into the `names` buffer starting at position `i*8*NAMELENGTH+i`.
`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.

Example

The following example retrieves the row and column names of the current problem:

```
int cols, rows, nl;
...
XPRSgetintattrib(prob,XPRS_COLS,&cols);
XPRSgetintattrib(prob,XPRS_ROWS,&rows);
XPRSgetintattrib(prob,XPRS_NAMELENGTH,&nl);

cnames = (char *) malloc(sizeof(char)*(8*nl+1)*cols);
rnames = (char *) malloc(sizeof(char)*(8*nl+1)*rows);
XPRSgetnames(prob,1,rnames,0,rows-1);
XPRSgetnames(prob,2,cnames,0,cols-1);
```

To display `names[i]`, use

```
int namelength;
...

XPRSgetintattrib(prob,XPRS_NAMELENGTH,&namelength);
printf("%s",names + i*(8*namelength+1));
```

Related topics

`XPRSaddnames`, `XPRSgetnamelist`.

XPRSgetobj

Purpose

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

Synopsis

```
int XPRS_CC XPRSgetobj(XPRSprob prob, double obj[], int first, int last);
```

Arguments

`prob`	The current problem.
`obj`	Double array of length `last-first+1` where the objective function coefficients are to be placed.
`first`	First column in the range.
`last`	Last column in the range.

Example

The following example retrieves the objective function coefficients of the current problem:

```
int cols;
double *obj;
...
XPRSgetintattrib(prob,XPRS_COLS,&cols);
obj = (double *) malloc(sizeof(double)*cols);
XPRSgetobj(prob, obj, 0, cols-1);
```

Related topics

`XPRSchgobj`.

XPRSgetobjecttypename

Purpose

Function to access the type name of an object referenced using the generic Optimizer object pointer `XPRSobject`.

Synopsis

```
int XPRS_CC XPRSgetobjecttypename(XPRSobject object, const char
        **sObjectName);
```

Arguments

`object`	The object for which the type name will be retrieved.
`sObjectName`	Pointer to a char pointer returning a reference to the null terminated string containing the object's type name. For example, if the object is of type `XPRSprob` then the returned pointer points to the string "`XPRSprob`".

Further information

This function is intended to be used typically from within the message callback function registered with the `XPRS_ge_addcbmsghandler` function. In such cases the user will need to identify the type of object sending the message since the message callback is passed only a generic pointer to the Optimizer object (`XPRSobject`) sending the message.

Related topics

`XPRS_ge_addcbmsghandler`.

XPRSgetpivotorder

Purpose

Returns the pivot order of the basic variables.

Synopsis

```
int XPRS_CC XPRSgetpivotorder(XPRSprob prob, int mpiv[]);
```

Arguments

prob	The current problem.
mpiv	Integer array of length ROWS where the pivot order will be returned.

Example

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

```
XPRSgetintattrib(prob,XPRS_ROWS,&rows);
pPivot = malloc(rows*(sizeof(int)));
XPRSgetpivotorder(prob,pPivot);
```

Further information

Row indices are in the range 0 to ROWS-1, whilst columns are in the range ROWS+SPAREROWS to ROWS+SPAREROWS+COLS-1.

Related topics

XPRSgetpivots, XPRSpivot.

XPRSgetpivots

Purpose

Returns a list of potential leaving variables if a specified variable enters the basis.

Synopsis

```
int XPRS_CC XPRSgetpivots(XPRSprob prob, int in, int outlist[], double x[],
        double *dobj, int *npiv, int maxpiv);
```

Arguments

prob	The current problem.
in	Index of the specified row or column to enter basis.
outlist	Integer array of length at least maxpiv to hold list of potential leaving variables. May be NULL if not required.
x	Double array of length ROWS+SPAREROWS+COLS to hold the values of all the variables that would result if in entered the basis. May be NULL if not required.
dobj	Pointer to a double where the objective function value that would result if in entered the basis will be returned.
npiv	Pointer to an integer where the actual number of potential leaving variables will be returned.
maxpiv	Maximum number of potential leaving variables to return.

Error value

425	Indicates in is invalid (out of range or already basic).

Example

The following retrieves a list of up to 5 potential leaving variables if variable 6 enters the basis:

```
int npiv, outlist[5];
double dobj;
...
XPRSgetpivots(prob,6,outlist,NULL,&dobj,&npiv,5);
```

Further information

1. If the variable in enters the basis and the problem is degenerate then several basic variables are candidates for leaving the basis, and the number of potential candidates is returned in npiv. A list of at most maxpiv of these candidates is returned in outlist which must be at least maxpiv long. If variable in were to be pivoted in, then because the problem is degenerate, the resulting values of the objective function and all the variables do not depend on which of the candidates from outlist is chosen to leave the basis. The value of the objective is returned in dobj and the values of the variables into x.

2. Row indices are in the range 0 to ROWS-1, whilst columns are in the range ROWS+SPAREROWS to ROWS+SPAREROWS+COLS-1.

Related topics

XPRSgetpivotorder, XPRSpivot.

XPRSgetpresolvebasis

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

```
int XPRS_CC XPRSgetpresolvebasis(XPRSprob prob, int rstatus[], int
       cstatus[]);
```

Arguments

`prob` The current problem.

`rstatus` Integer array of length `ROWS` to the basis status of the stack, surplus or artificial variable associated with each row. The status will be one of:

- `0` slack, surplus or artificial is non-basic at lower bound;
- `1` slack, surplus or artificial is basic;
- `2` slack or surplus is non-basic at upper bound.

May be `NULL` if not required.

`cstatus` Integer array of length `COLS` to hold the basis status of the columns in the constraint matrix. The status will be one of:

- `0` variable is non-basic at lower bound, or superbasic at zero if the variable has no lower bound;
- `1` variable is basic;
- `2` variable is at upper bound;
- `3` variable is super-basic.

May be `NULL` if not required.

Example

The following obtains and outputs basis information on a presolved problem prior to the global search:

```
XPRSprob prob;
int i, cols, *cstatus;
...
XPRSreadprob(prob,"myglobalprob","");
XPRSmipoptimize(prob,"l");
XPRSgetintattrib(prob,XPRS_COLS,&cols);
cstatus = malloc(cols*sizeof(int));
XPRSgetpresolvebasis(prob,NULL,cstatus);
for(i=0;i<cols;i++)
 printf("Column %d: %d\n", i, cstatus[i]);
XPRSmipoptimize(prob);
```

Related topics

`XPRSgetbasis`, `XPRSloadbasis`, `XPRSloadpresolvebasis`.

XPRSgetpresolvemap

Purpose

Returns the mapping of the row and column numbers from the presolve problem back to the original problem.

Synopsis

```
int XPRS_CC XPRSgetpresolvemap(XPRSprob prob, int rowmap[], int colmap[]);
```

Arguments

`prob`	The current problem.
`rowmap`	Integer array of length `ROWS` where the row maps will be returned.
`colmap`	Integer array of length `COLS` where the column maps will be returned.

Example

The following reads in a (Mixed) Integer Programming problem and gets the mapping for the rows and columns back to the original problem following optimization of the linear relaxation. The elimination operations of the presolve are turned off so that a one-to-one mapping between the presolve problem and the original problem.

```
XPRSreadprob(prob,"MyProb","");
XPRSsetintcontrol(prob,XPRS_PRESOLVEOPS,255);
XPRSmipoptimize(prob,"l");
XPRSgetintattrib(prob,XPRS_COLS,&cols);
colmap = malloc(cols*sizeof(int));
XPRSgetintattrib(prob,XPRS_ROWS,&rows);
rowmap = malloc(rows*sizeof(int));
XPRSgetpresolvemap(prob,rowmap,colmap);
```

Further information

The presolved problem can contain rows or columns that do not map to anything in the original problem. An example of this are cuts created during the MIP solve and temporarily added to the presolved problem. It is also possible that the presolver will introduce new rows or columns. For any row or column that does not have a mapping to a row or column in the original problem, the corresponding entry in the returned `rowmap` and `colmap` arrays will be `-1`.

Related topics

5.3.

XPRSgetpresolvesol

Purpose

Returns the solution for the presolved problem from memory.

Synopsis

```
int XPRS_CC XPRSgetpresolvesol(XPRSprob prob, double x[], double slack[],
      double dual[], double dj[]);
```

Arguments

`prob`	The current problem.
`x`	Double array of length `COLS` where the values of the primal variables will be returned. May be `NULL` if not required.
`slack`	Double array of length `ROWS` where the values of the slack variables will be returned. May be `NULL` if not required.
`dual`	Double array of length `ROWS` where the values of the dual variables will be returned. May be `NULL` if not required.
`dj`	Double array of length `COLS` where the reduced cost for each variable will be returned. May be `NULL` if not required.

Example

The following reads in a (Mixed) Integer Programming problem and displays the solution to the presolved problem following optimization of the linear relaxation:

```
XPRSreadprob(prob,"MyProb","");
XPRSmipoptimize(prob,"l");
XPRSgetintattrib(prob,XPRS_COLS,&cols);
x = malloc(cols*sizeof(double));
XPRSgetpresolvesol(prob,x,NULL,NULL,NULL);
for(i=0;i<cols;i++)
printf("Presolved x(%d) = %g\n",i,x[i]);
XPRSmipoptimize(prob,"");
```

Further information

1. If the problem has not been presolved, the solution in memory will be returned.
2. The solution to the original problem should be returned using the related function `XPRSgetlpsol`.
3. If called during a global callback the solution of the current node will be returned.
4. When an integer solution is found during a global search, it is always set up as a solution to the current node; therefore the integer solution is available as the current node solution and can be retrieved with `XPRSgetlpsol` and `XPRSgetpresolvesol`.

Related topics

`XPRSgetlpsol`, 5.3.

XPRSgetprimalray

Purpose

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

Synopsis

```
int XPRS_CC XPRSgetprimalray(XPRSprob prob, double dray[], int *hasRay);
```

Arguments

`prob`	The current problem.
`dray`	Double array of length `COLS` to hold the ray. May be NULL if not required.
`hasRay`	This variable will be set to 1 if the Optimizer is able to return a primal ray, 0 otherwise.

Example

The following code tries to retrieve a primal ray:

```
int cols;
double *primalRay;
int hasRay;
...
XPRSgetintattrib(prob, XPRS_COLS, &cols);
primalRay = malloc(cols*sizeof(double));
XPRSgetprimalray(prob, primalRay, &hasRay);
if(!hasRay) printf("Could not retrieve a primal ray\n");
```

Further information

1. It is possible to retrieve a primal ray only when, after solving an LP problem, the final status (`LPSTATUS`) is XPRS_LP_UNBOUNDED.
2. Primal rays are not post-solved. If the problem is in a presolved state, the primal ray that is returned will be for the presolved problem. If the problem was solved with presolve on and has been restored to the original state (the default behavior), this function will not be able to return a ray. To ensure that a primal ray can be obtained, it is recommended to solve a problem with presolve turned off (`PRESOLVE` = 0).

Related topics

`XPRSgetdualray`.

XPRSgetprobname

Purpose

Returns the current problem name.

Synopsis

```
int XPRS_CC XPRSgetprobname(XPRSprob prob, char *probname);
```

Arguments

`prob`	The current problem.
`probname`	A string of up to `MAXPROBNAMELENGTH` characters to contain the current problem name.

Related topics

`XPRSsetprobname`, `MAXPROBNAMELENGTH`.

XPRSgetpwlcons, XPRSgetpwlcons64

Purpose

Returns the piecewise linear constraints `y = f(x)` in a given range.

Synopsis

```
int XPRS_CC XPRSgetpwlcons(XPRSprob prob, int col[], int resultant[], int
      start[], double xval[], double yval[], int size, int *npoints, int
      first, int last);

int XPRS_CC XPRSgetpwlcons64(XPRSprob prob, int col[], int resultant[],
      XPRSint64 start[], double xval[], double yval[], XPRSint64 size,
      XPRSint64 *npoints, int first, int last);
```

Arguments

`prob`	The current problem.
`col`	Integer array which will be filled with the indices of the input variables `x`. It must be of length at least `last-first+1`. May be `NULL` if not required.
`resultant`	Integer array which will be filled with the indices of the output variables `y`. It must be of length at least `last-first+1`. May be `NULL` if not required.
`start`	Integer array which will be filled with the start indices of the different constraints in the breakpoint arrays. It must be of length at least `last-first+2`. The `x`-values of the breakpoints of piecewise linear constraint `i < last` will be given in `xval[start[i]]` to `xval[start[i+1]]`. May be `NULL` if not required.
`xval`	Double array of length `size` which will be filled with the `x`-values of the breakpoints. May be `NULL` if not required.
`yval`	Double array of length `size` which will be filled with the `y`-values of the breakpoints. May be `NULL` if not required.
`size`	Maximum number of breakpoints to be retrieved.
`npoints`	Pointer to return the number of breakpoints in the selected constraints. If the number of breakpoints is greater than `size`, then only `size` elements will be returned in the `xval` and `yval` arrays. May be `NULL` if not required.
`first`	First piecewise linear constraint in the range.
`last`	Last piecewise linear constraint in the range.

Example

The following example retrieves all variables and breakpoints in the first two piecewise linear constraints:

```
int *col;
int *resultant;
int *start;
double *xval;
double *yval;
int size;
int npoints;
...
XPRSgetpwlcons(prob, NULL, NULL, NULL, NULL, NULL, 0, &size, 0, 1);
col = (int*) malloc(2*sizeof(int));
resultant = (int*) malloc(2*sizeof(int));
start = (int*) malloc(3*sizeof(int));
xval = (double*) malloc(size*sizeof(double);
yval = (double*) malloc(size*sizeof(double);
XPRSgetpwlcons(prob, col, resultant, start, xval, yval, size, &npoints, 0, 1);
```

```
...
```

Further information

It is possible to obtain just the number of breakpoints in the range of piecewise linear constraints by calling this function with `size` set to `0`, in which case the required size for the breakpoint arrays will be returned in `npoints`.

Related topics

`XPRSaddpwlcons`, `XPRSdelpwlcons`.

XPRSgetqobj

Purpose

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

Synopsis

```
int XPRS_CC XPRSgetqobj(XPRSprob prob, int icol, int jcol, double *dval);
```

Arguments

`prob`	The current problem.
`icol`	Column index for the first variable in the quadratic term.
`jcol`	Column index for the second variable in the quadratic term.
`dval`	Pointer to a double value where the objective function coefficient is to be placed.

Example

The following returns the coefficient of the x_0^2 term in the objective function, placing it in the variable `value`:

```
double value;
...
XPRSgetqobj(prob,0,0,&value);
```

Further information

`dval` is the coefficient in the quadratic Hessian matrix. For example, if the objective function has the term $[3x_1x_2 + 3x_2x_1]/2$ the value retrieved by `XPRSgetqobj` is `3.0` and if the objective function has the term $[6x_1{}^2]/2$ the value retrieved by `XPRSgetqobj` is `6.0`.

Related topics

`XPRSgetmqobj`, `XPRSchgqobj`, `XPRSchgmqobj`.

XPRSgetqrowcoeff

Purpose

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

Synopsis

```
int XPRS_CC XPRSgetqrowcoeff (XPRSprob prob, int row, int icol, int jcol,
      double *dval);
```

Arguments

`prob`	The current problem.
`row`	The quadratic row where the coefficient is to be looked up.
`icol`	Column index for the first variable in the quadratic term.
`jcol`	Column index for the second variable in the quadratic term.
`dval`	Pointer to a double value where the objective function coefficient is to be placed.

Example

The following returns the coefficient of the x_0^2 term in the second row, placing it in the variable value :

```
double value;
...
XPRSgetqrowcoeff(prob,1,0,0,&value);
```

Further information

The coefficient returned corresponds to the Hessian of the constraint. That means the for constraint `x + [x^2 + 6 xy] <= 10` `XPRSgetqrowcoeff` would return `1` as the coefficient of `x^2` and `3` as the coefficient of `xy`.

Related topics

`XPRSloadqcqp`, `XPRSaddqmatrix`, `XPRSchgqrowcoeff`, `XPRSgetqrowqmatrix`, `XPRSgetqrowqmatrixtriplets`, `XPRSgetqrows`, `XPRSchgqobj`, `XPRSchgmqobj`, `XPRSgetqobj`.

XPRSgetqrowqmatrix

Purpose

Returns the nonzeros in a quadratic constraint coefficients matrix for the columns in a given range. To achieve maximum efficiency, `XPRSgetqrowqmatrix` returns the lower triangular part of this matrix only.

Synopsis

```
int XPRS_CC XPRSgetqrowqmatrix(XPRSprob prob, int irow, int mstart[], int
      mclind[], double dqe[], int size, int * nels, int first, int last);
```

Arguments

`prob`	The current problem.
`irow`	Index of the row for which the quadratic coefficients are to be returned.
`mstart`	Integer array which will be filled with indices indicating the starting offsets in the `mclind` and `dobjval` arrays for each requested column. It must be length of at least `last-first+2`. Column `i` starts at position `mstart[i]` in the `mrwind` and `dmatval` arrays, and has `mstart[i+1]-mstart[i]` elements in it. May be NULL if size is 0.
`mclind`	Integer array of length size which will be filled with the column indices of the nonzero elements in the lower triangular part of Q. May be NULL if size is 0.
`dqe`	Double array of length size which will be filled with the nonzero element values. May be NULL if size is 0.
`size`	Number of elements to be saved in mclind and dqe. If `size < *nels`, only `size` elements are written.
`nels`	Pointer to the integer where the number of nonzero elements in the queried columns will be returned. If the number of nonzero elements is greater than size, then only size elements will be returned. If `nels` is smaller than size, then only `nels` will be returned.
`first`	First column in the range.
`last`	Last column in the range.

Related topics

`XPRSloadqcqp`, `XPRSgetqrowcoeff`, `XPRSaddqmatrix`, `XPRSchgqrowcoeff`, `XPRSgetqrowqmatrixtriplets`, `XPRSgetqrows`, `XPRSchgqobj`, `XPRSchgmqobj`, `XPRSgetqobj`.

XPRSgetqrowqmatrixtriplets

Purpose

Returns the nonzeros in a quadratic constraint coefficients matrix as triplets (index pairs with coefficients). To achieve maximum efficiency, `XPRSgetqrowqmatrixtriplets` returns the lower triangular part of this matrix only.

Synopsis

```
int XPRS_CC XPRSgetqrowqmatrixtriplets(XPRSprob prob, int irow, int *
      nqelem, int mqcol1[], int mqcol2[], double dqe[]);
```

Arguments

`prob`	The current problem.
`irow`	Index of the row for which the quadratic coefficients are to be returned.
`nqelem`	Argument used to return the number of quadratic coefficients in the row.
`mqcol1`	First index in the triplets. May be NULL if not required.
`mqcol2`	Second index in the triplets. May be NULL if not required.
`dqe`	Coefficients in the triplets. May be NULL if not required.

Further information

If a row index of `-1` is used, the function returns the quadratic coefficients for the objective function.

Related topics

`XPRSloadqcqp`, `XPRSgetqrowcoeff`, `XPRSaddqmatrix`, `XPRSchgqrowcoeff`, `XPRSgetqrowqmatrix`, `XPRSgetqrows`, `XPRSchgqobj`, `XPRSchgmqobj`, `XPRSgetqobj`.

XPRSgetqrows

Purpose

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

Synopsis

```
int XPRS_CC XPRSgetqrows(XPRSprob prob, int * qmn, int qcrows[]);
```

Arguments

`prob`	The current problem.
`qmn`	Used to return the number of quadratic constraints in the matrix.
`qcrows`	Array of length `*qmn` used to return the indices of rows with quadratic coefficients in them. May be NULL if not required.

Related topics

`XPRSloadqcqp`, `XPRSgetqrowcoeff`, `XPRSaddqmatrix`, `XPRSchgqrowcoeff`, `XPRSgetqrowqmatrix`, `XPRSgetqrowqmatrixtriplets`, `XPRSchgqobj`, `XPRSchgmqobj`, `XPRSgetqobj`.

XPRSgetrhs

Purpose

Returns the right hand side elements for the rows in a given range.

Synopsis

```
int XPRS_CC XPRSgetrhs(XPRSprob prob, double rhs[], int first, int last);
```

Arguments

`prob`	The current problem.
`rhs`	Double array of length `last-first+1` where the right hand side elements are to be placed.
`first`	First row in the range.
`last`	Last row in the range.

Example

The following example retrieves the right hand side values of the problem:

```
int rows;
double *rhs;
...
XPRSgetintattrib(prob,XPRS_ROWS,&rows);
rhs = (double *) malloc(sizeof(double)*rows);
XPRSgetrhs(prob,rhs,0,rows-1);
```

Related topics

`XPRSchgrhs`, `XPRSchgrhsrange`, `XPRSgetrhsrange`.

XPRSgetrhsrange

Purpose

Returns the right hand side range values for the rows in a given range.

Synopsis

```
int XPRS_CC XPRSgetrhsrange(XPRSprob prob, double range[], int first, int
      last);
```

Arguments

`prob`	The current problem.
`range`	Double array of length `last-first+1` where the right hand side range values are to be placed.
`first`	First row in the range.
`last`	Last row in the range.

Example

The following returns right hand side range values for all rows in the matrix:

```
int rows;
double *range;
...
XPRSgetintattrib(prob,XPRS_ROWS,&rows);
range = malloc(rows*sizeof(double));
XPRSgetrhsrange(prob,range,0,rows);
```

Related topics

`XPRSchgrhs`, `XPRSchgrhsrange`, `XPRSgetrhs`, `XPRSrange`.

XPRSgetrowrange

Purpose

Returns the row ranges computed by XPRSrange.

Synopsis

```
int XPRS_CC XPRSgetrowrange(XPRSprob prob, double upact[], double loact[],
      double uup[], double udn[]);
```

Arguments

prob	The current problem.
upact	Double array of length ROWS for the upper row activities.
loact	Double array of length ROWS for the lower row activities.
uup	Double array of length ROWS for the upper row unit costs.
udn	Double array of length ROWS for the lower row unit costs.

Example

The following computes row ranges and returns them:

```
int rows;
double *upact, *loact, *uup, *udn;
...
XPRSrange(prob);
XPRSgetintattrib(prob,XPRS_ROWS,&rows);
upact = malloc(rows*sizeof(double));
loact = malloc(rows*sizeof(double));
uup   = malloc(rows*sizeof(double));
udn   = malloc(rows*sizeof(double));
...
XPRSgetrowrange(prob,upact,loact,uup,udn);
```

Further information

The activities and unit costs are obtained from the range file (*problem_name*.rng). The meaning of the upper and lower column activities and upper and lower unit costs in the ASCII range files is described in Appendix A.

Related topics

XPRSchgrhsrange, XPRSgetcolrange.

XPRSgetrows, XPRSgetrows64

Purpose

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

Synopsis

```
int XPRS_CC XPRSgetrows(XPRSprob prob, int mstart[], int mclind[], double
      dmatval[], int size, int *nels, int first, int last);

int XPRS_CC XPRSgetrows64(XPRSprob prob, XPRSint64 mstart[], int mclind[],
      double dmatval[], XPRSint64 size, XPRSint64 *nels, int first, int
      last);
```

Arguments

`prob`	The current problem.
`mstart`	Integer array which will be filled with the indices indicating the starting offsets in the `mrwind` and `dmatval` arrays for each requested row. It must be of length at least `last-first+2`. Row `i` starts at position `mstart[i]` in the `mrwind` and `dmatval` arrays, and has `mstart[i+1]-mstart[i]` elements in it. May be `NULL` if not required.
`mrwind`	Integer arrays of length `size` which will be filled with the column indices of the nonzero elements for each row. May be `NULL` if not required.
`dmatval`	Double array of length `size` which will be filled with the nonzero element values. May be `NULL` if not required.
`size`	Maximum number of elements to be retrieved.
`nels`	Pointer to the integer where the number of nonzero elements in the selected rows will be returned. If the number of nonzero elements is greater than `size`, then only `size` elements will be returned. If `nels` is smaller than `size`, then only `nels` will be returned.
`first`	First row in the range.
`last`	Last row in the range.

Example

The following example returns and displays at most six nonzero matrix entries in the first two rows:

```
int size=6, nels, nreturnedels, mstart[3], mrwind[6];
double dmatval[6];
...
XPRSgetrows(prob,mstart,mrwind,dmatval,size,&nels,0,1);

nreturnedels = nels > size ? size : nels;
for(i=0;i<nreturnedels;i++) printf("\t%2.1f\n",dmtval[i]);
```

Further information

It is possible to obtain just the number of elements in the range of columns by replacing `mstart`, `mrwind` and `dmatval` by `NULL`. In this case, `size` must be set to `0` to indicate that the length of arrays passed is `0`.

Related topics

`XPRSgetcols`, `XPRSgetrowrange`, `XPRSgetrowtype`.

XPRSgetrowtype

Purpose

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

Synopsis

```
int XPRS_CC XPRSgetrowtype(XPRSprob prob, char qrtype[], int first, int
        last);
```

Arguments

`prob`	The current problem.
`qrtype`	Character array of length `last-first+1` characters where the row types will be returned: `N` indicates a free constraint; `L` indicates a $\leq$ constraint; `E` indicates an = constraint; `G` indicates a $\geq$ constraint; `R` indicates a range constraint.
`first`	First row in the range.
`last`	Last row in the range.

Example

The following example retrieves row types into an array `qrtype` :

```
int rows;
char *qrtype;
...
XPRSgetintattrib(prob,XPRS_ROWS,&rows);
qrtype = (char *) malloc(sizeof(char)*rows);
XPRSgetrowtype(prob,qrtype,0,rows-1);
```

Related topics

`XPRSchgrowtype`, `XPRSgetrowrange`, `XPRSgetrows`.

XPRSgetscale

Purpose

Returns the the current scaling of the matrix.

Synopsis

```
int XPRS_CC XPRSgetscale(XPRSprob prob, int mrscal[], int mcscal[]);
```

Arguments

prob	The current problem.
mrscal	Integer array of size ROWS that will contain the powers of 2 with which the rows are currently scaled.
mcscal	Integer array of size COLS that will contain the powers of 2 with which the columns are currently scaled.

Related topics

XPRSscale (SCALE).

XPRSgetscaledinfeas

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

```
int XPRS_CC XPRSgetscaledinfeas(XPRSprob prob, int *npv, int *nps, int
        *nds, int *ndv, int mx[], int mslack[], int mdual[], int mdj[]);
```

Arguments

`prob`	The current problem.
`npv`	Number of primal infeasible variables.
`nps`	Number of primal infeasible rows.
`nds`	Number of dual infeasible rows.
`ndv`	Number of dual infeasible variables.
`mx`	Integer array of length `npv` where the primal infeasible variables will be returned. May be `NULL` if not required.
`mslack`	Integer array of length `nps` where the primal infeasible rows will be returned. May be `NULL` if not required.
`mdual`	Integer array of length `nds` where the dual infeasible rows will be returned. May be `NULL` if not required.
`mdj`	Integer array of length `ndv` where the dual infeasible variables will be returned. May be `NULL` if not required.

Error value

`422`	A solution is not available.

Related controls

Double

`FEASTOL`	Tolerance on RHS.
`OPTIMALITYTOL`	Reduced cost tolerance.

Example

In this example, `XPRSgetscaledinfeas` is first called with nulled integer arrays to get the number of infeasible entries. Then space is allocated for the arrays and the function is again called to fill them in.

```
int *mx, *mslack, *mdual, *mdj, npv, nps, nds, ndv;
...
XPRSgetscaledinfeas(prob, &npv, &nps, &nds, &ndv,
                    NULL, NULL, NULL, NULL);

mx = malloc(npv * sizeof(int));
mslack = malloc(nps * sizeof(int));
mdual = malloc(nds * sizeof(int));
mdj = malloc(ndv * sizeof(int));
XPRSgetscaledinfeas(prob, &npv, &nps, &nds, &ndv,
                    mx, mslack, mdual, mdj);
```

Further information

If any of the last four arguments are set to `NULL`, the corresponding number of infeasibilities is still returned.

Related topics

XPRSgetinfeas, XPRSgetiisdata, XPRSiisall, XPRSiisclear, XPRSiisfirst, XPRSiisisolations, XPRSiisnext, XPRSiisstatus, XPRSiiswrite, IIS.

XPRSgetstrattrib, XPRSgetstringattrib

Purpose

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

Synopsis

```
int XPRS_CC XPRSgetstrattrib(XPRSprob prob, int ipar, char *cval);

int XPRS_CC XPRSgetstringattrib(XPRSprob prob, int ipar, char *cgval, int
      cgvalsize, int* controlsize);
```

Arguments

`prob`	The current problem.
`ipar`	Problem attribute whose value is to be returned. A full list of all problem attributes may be found in 10, or from the list in the `xprs.h` header file.
`cval`	Pointer to a string where the value of the attribute (plus null terminator) will be returned.
`cgvalsize`	Maximum number of bytes to be written into the cgval argument.
`controlsize`	Returns the length of the string control including the null terminator.

Related topics

`XPRSgetdblattrib`, `XPRSgetintattrib`.

XPRSgetstrcontrol, XPRSgetstringcontrol

Purpose

Returns the value of a given string control parameters.

Synopsis

```
int XPRS_CC XPRSgetstrcontrol(XPRSprob prob, int ipar, char *cgval);

int XPRS_CC XPRSgetstringcontrol(XPRSprob prob, int ipar, char *cgval, int
      cgvalsize, int* controlsize);
```

Arguments

`prob`	The current problem.
`ipar`	Control parameter whose value is to be returned. A full list of all controls may be found in 9, or from the list in the `xprs.h` header file.
`cgval`	Pointer to a string where the value of the control (plus null terminator) will be returned.
`cgvalsize`	Maximum number of bytes to be written into the cgval argument.
`controlsize`	Returns the length of the string control including the null terminator.

Related topics

`XPRSgetdblcontrol`, `XPRSgetintcontrol`, `XPRSsetstrcontrol`.

XPRSgetub

Purpose

Returns the upper bounds for the columns in a given range.

Synopsis

```
int XPRS_CC XPRSgetub(XPRSprob prob, double ub[], int first, int last);
```

Arguments

`prob`	The current problem.
`ub`	Double array of length `last-first+1` where the upper bounds are to be placed.
`first`	First column in the range.
`last`	Last column in the range.

Example

The following example retrieves the upper bounds for the columns of the current problem:

```
int cols;
double *ub;
...
XPRSgetintattrib(prob, XPRS_COLS, &cols);
ub = (double *) malloc(sizeof(double)*ncol);
XPRSgetub(prob, ub, 0, ncol-1);
```

Further information

Values greater than or equal to `XPRS_PLUSINFINITY` should be interpreted as infinite; values less than or equal to `XPRS_MINUSINFINITY` should be interpreted as infinite and negative.

Related topics

`XPRSchgbounds`, `XPRSgetlb`.

XPRSgetunbvec

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

```
int XPRS_CC XPRSgetunbvec(XPRSprob prob, int *junb);
```

Arguments

`prob`	The current problem.
`junb`	Pointer to an integer where the vector causing the problem to be detected as being primal or dual unbounded will be returned. In the dual simplex case, the vector is the leaving row for which the dual simplex detected dual unboundedness. In the primal simplex case, the vector is the entering row `junb` (if `junb` is in the range `0` to `ROWS-1`) or column (variable) `junb-ROWS-SPAREROWS` (if `junb` is between `ROWS+SPAREROWS` and `ROWS+SPAREROWS+COLS-1`) for which the primal simplex detected primal unboundedness.

Error value

`91`	A current problem is not available.

Further information

When solving using the dual simplex method, if the problem is primal infeasible then `XPRSgetunbvec` returns the pivot row where dual unboundedness was detected. Also note that when solving using the dual simplex method, if the problem is primal unbounded then `XPRSgetunbvec` returns -1 since the problem is dual infeasible and not dual unbounded.

Related topics

`XPRSgetinfeas`, `XPRSlpoptimize`.

XPRSgetversion

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

```
int XPRS_CC XPRSgetversion(char *version);
```

Argument

`version`	Buffer long enough to hold the version string (plus a null terminator). This should be at least 16 characters.

Related controls

Integer

`VERSION`	The Optimizer version number

Example

The following calls `XPRSgetversion` to return version information at the start of the program:

```
char version[16];
XPRSgetversion(version);
printf("Xpress Optimizer version %s\n",version);
XPRSinit(NULL);
```

Further information

This function supersedes the `VERSION` control, which only returns the first two parts of the version number. Release 2004 versions of the Optimizer have a three-part version number.

Related topics

`XPRSinit`.

XPRSglobal GLOBAL

Purpose

Starts the global search for an integer solution after solving the LP relaxation with XPRSmaxim (MAXIM) or XPRSminim (MINIM) or continues a global search if it has been interrupted. This function is deprecated and might be removed in a future release. XPRSmipoptimize should be used instead.

Synopsis

```
int XPRS_CC XPRSglobal(XPRSprob prob);
GLOBAL
```

Argument

prob	The current problem.

Related controls

Integer

BACKTRACK	Node selection criterion.
BRANCHCHOICE	Once a global entity has been selected for branching, this control determines whether the branch with the minimum or maximum estimate is followed first.
BREADTHFIRST	Limit for node selection criterion.
COVERCUTS	Number of rounds of lifted cover inequalities at top node.
CPUTIME	1 for CPU time; 0 for elapsed time.
CUTDEPTH	Maximum depth in the tree at which cuts are generated.
CUTFREQ	Frequency at which cuts are generated in the tree search.
CUTSTRATEGY	Specifies the cut strategy.
DEFAULTALG	Algorithm to use with the tree search.
GOMCUTS	Number of rounds of Gomory cuts at the top node.
MAXMIPSOL	Maximum number of MIP solutions to find.
MAXNODE	Maximum number of nodes in Branch and Bound search.
MAXTIME	Maximum time allowed.
MIPLOG	Global print flag.
MIPPRESOLVE	Type of integer preprocessing to be performed.
MIPTHREADS	Number of threads used for parallel MIP search.
NODESELECTION	Node selection control.
REFACTOR	Indicates whether to re-factorize the optimal basis.
SBBEST	Number of infeasible global entities on which to perform strong branching.
SBITERLIMIT	Number of dual iterations to perform strong branching.
SBSELECT	The size of the candidate list of global entities for strong branching.
TREECOVERCUTS	Number of rounds of lifted cover inequalities in the tree.
TREEGOMCUTS	Number of rounds of Gomory cuts in the tree.
VARSELECTION	Node selection degradator estimate control.

Double

MIPABSCUTOFF	Cutoff set after an LP Optimizer command.
MIPABSSTOP	Absolute optimality stopping criterion.
MIPADDCUTOFF	Amount added to objective function to give new cutoff.
MIPRELCUTOFF	Percentage cutoff.
MIPRELSTOP	Relative optimality stopping criterion.
MIPTOL	Integer feasibility tolerance.
PSEUDOCOST	Default pseudo cost in node degradation estimation.

Example 1 (Library)

The following example inputs a problem `fred.mat`, solves the LP and the global problem before printing the solution to file.

```
XPRSreadprob(prob,"fred","");
XPRSmaxim(prob,"");
XPRSglobal(prob);
XPRSwriteprtsol(prob);
```

Example 2 (Console)

The equivalent set of commands for the Console Optimizer are:

```
READPROB fred
MAXIM
GLOBAL
WRITEPRTSOL
```

Further information

1. When an optimal LP solution has been found with `XPRSmaxim` (`MAXIM`) or `XPRSminim` (`MINIM`), the search for an integer solution is started using `XPRSglobal` (`GLOBAL`). In many cases `XPRSglobal` (`GLOBAL`) is to be called directly after `XPRSmaxim` (`MAXIM`)/`XPRSminim` (`MINIM`). In such circumstances this can be achieved slightly more efficiently using the `g` flag to `XPRSmaxim` (`MAXIM`)/`XPRSminim` (`MINIM`).
2. If a global search is interrupted and `XPRSglobal` (`GLOBAL`) is subsequently called again, the search will continue where it left off. To restart the search at the top node you need to call either `XPRSinitglobal` or `XPRSpostsolve` (`POSTSOLVE`).
3. The controls described for `XPRSmaxim` (`MAXIM`) and `XPRSminim` (`MINIM`) can also be used to control the `XPRSglobal` (`GLOBAL`) algorithm.
4. (*Console*) The global search may be interrupted by typing CTRL-C as long as the user has not already typed ahead.
5. A summary log of six columns of information is output every *n* nodes, where -*n* is the value of `MIPLOG` (see A.10).
6. Optimizer library users can check the final status of the global search using the `MIPSTATUS` problem attribute.
7. The Optimizer may create global files (used for storing parts of the tree when there is insufficient available memory) in excess of 2 GigaBytes. If your filing system does not support files this large, you can instruct the Optimizer to spread the data over multiple files by setting the `MAXGLOBALFILESIZE` control.

Related topics

`XPRSfixglobals` (`FIXGLOBALS`), `XPRSinitglobal`, `XPRSmaxim` (`MAXIM`)/`XPRSminim` (`MINIM`), A.10.

XPRSgoal GOAL

Purpose

This function is deprecated, and will be removed in future releases. Perform goal programming.

Synopsis

```
int XPRS_CC XPRSgoal(XPRSprob prob, const char *filename, const char
        *flags);
GOAL [filename] [-flags]
```

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 (a `.gol` extension will be added).
`flags`	Flags to pass to `XPRSgoal` (`GOAL`): `o` optimization process logs to be displayed; `l` treat integer variables as linear; `f` write output into a file `filename.grp`.

Example 1 (Library)

In the following example, goal programming is carried out on a problem, `goalex`, taking instructions from the file `gb1.gol`:

```
XPRSreadprob(prob,"goalex","");
XPRSgoal(prob,"gb1","fo");
```

Example 2 (Console)

Suppose we have a problem where the weight for objective function `OBJ1` is unknown and we wish to perform goal programming, maximizing this row and relaxing the resulting constraint by 5% of the optimal value, then the following sequence will solve this problem:

```
READPROB
GOAL
P
O
OBJ1
MAX
P
5
<empty line>
```

Further information

1. The command `XPRSgoal` (`GOAL`) used with objective functions allows the user to find solutions of problems with more than one objective function. `XPRSgoal` (`GOAL`) used with constraints enables the user to find solutions to infeasible problems. The goals are the constraints relaxed at the beginning to make the problem feasible. Then one can see how many of these relaxed constraints can be met, knowing the penalty of making the problem feasible (in the Archimedean case) or knowing which relaxed constraints will never be met (in the pre-emptive case).
2. (*Console*) If the optional `filename` is specified when `GOAL` is used, the responses to the prompts are read from `filename.gol`. If there is an invalid answer to a prompt, goal programming will stop and control will be returned to the Optimizer.
3. It is not always possible to use the output of one of the goal problems as an input for further study because the coefficients for the objective function, the right hand side and the row type may all have changed.
4. In the Archimedean/objective function option, the fixed value of the resulting objective function will be the linear combination of the right hand sides of the objective functions involved.

Related topics

7.

HELP

Purpose

Provides quick reference help for console users of the Optimizer.

Synopsis

```
HELP
HELP commands
HELP controls
HELP attributes
HELP [command-name]
HELP [control-name]
HELP [attribute-name]
```

Example

This command is used by calling it at the Console Optimizer command line:

```
HELP MAXTIME
```

Related topics

None.

IIS

Purpose

Provides the Irreducible Infeasible Set (IIS) functionality for the console.

Synopsis

`IIS [-flags]`

Arguments

`IIS`	Finds an IIS.
`IIS -a`	Performs an automated search for a set of independent IISs.
`IIS -c`	Resets the search for IISs (deletes already found ones).
`IIS -e [num`	fn] Writes a CSV file named `fn` containing the IIS data of IIS `num`.
`IIS -f`	Generate an approximation of an IIS only.
`IIS -i num`	Performs the isolation identification for IIS with ordinal number `num`.
`IIS -n`	Finds another (independent) IIS if any.
`IIS -p [num`	] Prints the IIS with ordinal number `num` to the screen.
`IIS -s`	Returns statistics on the IISs found.
`IIS -w [num`	fn type] Writes an LP or MPS file named `fn` containing the IIS subproblem of IIS `num` depending on the `type` flags.

Example 1 (Console)

This example reads in an infeasible problem, executes an automated search for the IISs, prints the IIS to the screen and then displays a summary on the results.

```
READPROB PROB.LP
IIS -a -s
```

Example 2 (Console)

This example reads in an infeasible problem, identifies an IIS and its isolations, then writes the IIS as an LP for easier viewing and as a CSV file to contain the supplementary information.

```
READPROB PROB.LP
IIS
IIS -i -p 1
IIS -w 1 "IIS.LP"  lp
IIS -e 1 "IIS.CSV"
```

Further information

1. The IISs are numbered from `1` to `NUMIIS`. If no IIS number is provided, the functions take the last IIS identified as default. When applicable, IIS `0` refers to the initial infeasible IIS (the IIS approximation).
2. A model may have several infeasibilities. Repairing a single IIS may not make the model feasible. For this reason the Optimizer attempts to find an IIS for each of the infeasibilities in a model. You may call the `IIS -n` function repeatedly, or use the `IIS -a` function to retrieve all IIS at once.
3. An IIS isolation is a special constraint or bound in an IIS. Removing an IIS isolation constraint or bound will remove all infeasibilities in the IIS without increasing the infeasibilities in any row or column outside the IIS, thus in any other IISs. The IIS isolations thus indicate the likely cause of each independent infeasibility and give an indication of which constraint or bound to drop or modify. It is not always possible to find IIS isolations. IIS isolations are only available for linear problems.
4. Generally, one should first look for rows or columns in the IIS which are both in isolation, and have a high dual multiplier relative to the others.
5. Initial infeasible subproblem: The subproblem identified after the sensitivity filter is referred to as initial infeasible subproblem. Its size is crucial to the running time of the deletion filter and it contains all the infeasibilities of the first phase simplex, thus if the corresponding rows and bounds are removed the problem becomes feasible
6. `IIS -f` performs the initial sensitivity analysis on rows and columns to reduce the problem size, and sets up the initial infeasible subproblem. This subproblem significantly speeds up the generation of IISs, however in itself it may serve as an approximation of an IIS, since its identification typically takes only a fraction of time compared to the identification of an IIS.
7. The `num` parameter cannot be zero for `IIS -i`: the concept of isolations is meaningless for the initial infeasible subproblem.
8. If `IIS -n [num]` is called, the return status is `1` if less than `num` IISs have been found and zero otherwise. The total number of IISs found is stored in `NUMIIS`.
9. The type flags passed to `IIS -w` are directly passed to the `WRITEPROB` command.
10. The LP or MPS files created by `IIS -w` corresponding to an IIS contain no objective function, since infeasibility is independent from the objective.
11. Please note that there are problems on the boundary of being infeasible or not. For such problems, feasibility or infeasibility often depends on tolerances or even on scaling. This phenomenon makes it possible that after writing an IIS out as an LP file and reading it back, it may report feasibility. As a first check it is advised to consider the following options:
    (a) Turn presolve off (e.g. in console `presolve = 0`) since the nature of an IIS makes it necessary that during their identification the presolve is turned off.
    (b) Use the primal simplex method to solve the problem (e.g. in console `lpoptimize -p`).
12. Note that the original sense of the original objective function plays no role in an IIS.
13. The supplementary information provided in the CSV file created by `IIS -e` is identical to that returned by the `XPRSgetiisdata` function.
14. The IIS approximation and the IISs generated so far are always available.

Related topics

`XPRSgetiisdata`, `XPRSiisall`, `XPRSiisclear`, `XPRSiisfirst`, `XPRSiisisolations`, `XPRSiisnext`, `XPRSiisstatus`, `XPRSiiswrite`.

XPRSiisall

Purpose

Performs an automated search for independent Irreducible Infeasible Sets (IIS) in an infeasible problem.

Synopsis

```
int XPRS_CC XPRSiisall(XPRSprob prob);
```

Argument

`prob`	The current problem.

Related controls

Integer

`MAXIIS`	Number of Irreducible Infeasible Sets to be found.

Example

This example searches for IISs and then questions the problem attribute `NUMIIS` to determine how many were found:

```
int iis;
...
XPRSiisall(prob);
XPRSgetintattrib(prob, XPRS_NUMIIS, &iis);
printf("number of IISs = %d\n", iis);
```

Further information

1. Calling `IIS -a` from the console has the same effect as this function.
2. A model may have several infeasibilities. Repairing a single IIS may not make the model feasible. For this reason the Optimizer can find an IIS for each of the infeasibilities in a model. If the control `MAXIIS` is set to a positive integer value then the `XPRSiisall` command will stop if `MAXIIS` IISs have been found. By default the control `MAXIIS` is set to `-1`, in which case an IIS is found for each of the infeasibilities in the model.
3. The problem attribute `NUMIIS` allows the user to recover the number of IISs found in a particular search. Alternatively, the `XPRSiisstatus` function may be used to retrieve the number of IISs found by `XPRSiisfirst` (`IIS`), `XPRSiisnext` (`IIS -n`) or `XPRSiisall` (`IIS -a`) functions.

Related topics

`XPRSgetiisdata`, `XPRSiisclear`, `XPRSiisfirst`, `XPRSiisisolations`, `XPRSiisnext`, `XPRSiisstatus`, `XPRSiiswrite`, `IIS`.

XPRSiisclear

Purpose

Resets the search for Irreducible Infeasible Sets (IIS).

Synopsis

```
int XPRS_CC XPRSiisclear(XPRSprob prob);
```

Argument

`prob`	The current problem.

Example

```
XPRSiisclear(prob);
```

Further information

1. Calling `IIS -c` from the console has the same effect as this function.
2. The information stored internally about the IISs identified by `XPRSiisfirst`, `XPRSiisnext` or `XPRSiisall` are cleared. Functions `XPRSgetiisdata`, `XPRSiisstatus`, `XPRSiiswrite` and `XPRSiisisolations` cannot be called until the IIS identification procedure is started again.
3. This function is automatically called by `XPRSiisfirst` and `XPRSiisall`

Related topics

`XPRSgetiisdata`, `XPRSiisall`, `XPRSiisfirst`, `XPRSiisisolations`, `XPRSiisnext`, `XPRSiisstatus`, `XPRSiiswrite`, `IIS`.

XPRSiisfirst

Purpose

Initiates a search for an Irreducible Infeasible Set (IIS) in an infeasible problem.

Synopsis

```
int XPRS_CC XPRSiisfirst(XPRSprob prob, int iismode, int *status_code);
```

Arguments

`prob`	The current problem.
`iismode`	The IIS search mode:
`0`	stops after finding the initial infeasible subproblem;
`1`	find an IIS, emphasizing simplicity of the IIS;
`2`	find an IIS, emphasizing a quick result.
`status_code`	The status after the search:
`0`	success;
`1`	if problem is feasible;
`2`	error (when the function returns nonzero).

Example

This looks for the first IIS.

```
XPRSiisfirst(myprob,1,&status);
```

Further information

1. Calling `IIS` from the console has the same effect as this function.
2. A model may have several infeasibilities. Repairing a single IIS may not make the model feasible. For this reason the Optimizer can find an IIS for each of the infeasibilities in a model. For the generation of several independent IISs use functions `XPRSiisnext` (`IIS -n`) or `XPRSiisall` (`IIS -a`).
3. IIS sensitivity filter: after an optimal but infeasible first phase primal simplex, it is possible to identify a subproblem containing all the infeasibilities (corresponding to the given basis) to reduce the size of the IIS working problem dramatically, i.e., rows with zero duals (thus with artificials of zero reduced cost) and columns that have zero reduced costs may be deleted. Moreover, for rows and columns with nonzero costs, the sign of the cost is used to relax equality rows either to less than or greater than equal rows, and to drop either possible upper or lower bounds on columns.
4. Initial infeasible subproblem: The subproblem identified after the sensitivity filter is referred to as initial infeasible subproblem. Its size is crucial to the running time of the deletion filter and it contains all the infeasibilities of the first phase simplex, thus if the corresponding rows and bounds are removed the problem becomes feasible.
5. `XPRSiisfirst` performs the initial sensitivity analysis on rows and columns to reduce the problem size, and sets up the initial infeasible subproblem. This subproblem significantly speeds up the generation of IISs, however in itself it may serve as an approximation of an IIS, since its identification typically takes only a fraction of time compared to the identification of an IIS.
6. The IIS approximation and the IISs generated so far are always available.

Related topics

`XPRSgetiisdata`, `XPRSiisall`, `XPRSiisclear`, `XPRSiisisolations`, `XPRSiisnext`, `XPRSiisstatus`, `XPRSiiswrite`, `IIS`.

XPRSiisisolations

Purpose

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

Synopsis

```
int XPRS_CC XPRSiisisolations(XPRSprob prob, int num);
```

Arguments

`prob`	The current problem.
`num`	The number of the IIS identified by either `XPRSiisfirst` (`IIS`), `XPRSiisnext` (`IIS -n`) or `XPRSiisall` (`IIS -a`) in which the isolations should be identified.

Example

This example finds the first IIS and searches for the isolations in that IIS.

```
XPRSiisfirst(prob,1,&status);
XPRSiisisolations (prob,1);
```

Further information

1. Calling `IIS -i [num]` from the console has the same effect as this function.
2. An IIS isolation is a special constraint or bound in an IIS. Removing an IIS isolation constraint or bound will remove all infeasibilities in the IIS without increasing the infeasibilities in any row or column outside the IIS, thus in any other IISs. The IIS isolations thus indicate the likely cause of each independent infeasibility and give an indication of which constraint or bound to drop or modify. It is not always possible to find IIS isolations.
3. Generally, one should first look for rows or columns in the IIS which are both in isolation, and have a high dual multiplier relative to the others.
4. The `num` parameter cannot be zero: the concept of isolations is meaningless for the initial infeasible subproblem.

Related topics

`XPRSgetiisdata`, `XPRSiisall`, `XPRSiisclear`, `XPRSiisfirst`, `XPRSiisnext`, `XPRSiisstatus`, `XPRSiiswrite`, `IIS`.

XPRSiisnext

Purpose

Continues the search for further Irreducible Infeasible Sets (IIS), or calls `XPRSiisfirst` (`IIS`) if no IIS has been identified yet.

Synopsis

```
int XPRS_CC XPRSiisnext(XPRSprob prob, int *status_code);
```

Arguments

`prob`	The current problem.
`status_code`	The status after the search:
`0`	success;
`1`	no more IIS could be found, or problem is feasible if no `XPRSiisfirst` call preceded;
`2`	on error (when the function returns nonzero).

Example

This looks for a further IIS.

```
XPRSiisnext(prob,&status_code);
```

Further information

1. Calling `IIS -n` from the console has the same effect as this function.
2. A model may have several infeasibilities. Repairing a single IIS may not make the model feasible. For this reason the Optimizer attempts to find an IIS for each of the infeasibilities in a model. You may call the `XPRSiisnext` function repeatedly, or use the `XPRSiisall` (`IIS -a`) function to retrieve all IIS at once.
3. This function is not affected by the control `MAXIIS`.
4. If the problem has been modified since the last call to `XPRSiisfirst` or `XPRSiisnext`, the generation process has to be started from scratch.

Related topics

`XPRSgetiisdata`, `XPRSiisall`, `XPRSiisclear`, `XPRSiisfirst`, `XPRSiisisolations`, `XPRSiisstatus`, `XPRSiiswrite`, `IIS`.

XPRSiisstatus

Purpose

Returns statistics on the Irreducible Infeasible Sets (IIS) found so far by `XPRSiisfirst` (`IIS`), `XPRSiisnext` (`IIS -n`) or `XPRSiisall` (`IIS -a`).

Synopsis

```
int XPRS_CC XPRSiisstatus(XPRSprob prob, int *iiscount, int rowsizes[], int
        colsizes[], double suminfeas[], int numinfeas[]);
```

Arguments

`prob`	The current problem.
`iiscount`	The number of IISs found so far.
`rowsizes`	Number of rows in the IISs.
`colsizes`	Number of bounds in the IISs.
`suminfeas`	The sum of infeasibilities in the IISs after the first phase simplex.
`numinfeas`	The number of infeasible variables in the IISs after the first phase simplex.

Example

This example first retrieves the number of IISs found so far, and then retrieves their main properties. Note that the arrays have size `count+1`, since the first index is reserved for the initial infeasible subset.

```
XPRSiisstatus(myprob,&count,NULL,NULL,NULL,NULL);
rowsizes  = malloc((count+1)*sizeof(int));
colsizes  = malloc((count+1)*sizeof(int));
suminfeas = malloc((count+1)*sizeof(double));
numinfeas = malloc((count+1)*sizeof(int));
XPRSiisstatus(myprob,&count,rowsizes,colsizes,suminfeas,numinfeas);
```

Further information

1. Calling `IIS -s` from the console has the same effect as this function.
2. All arrays should be of dimension `iiscount+1`. The arrays are `0` based, index 0 corresponding to the initial infeasible subproblem.
3. The arrays may be NULL if not required.
4. For the initial infeasible problem (at position 0) the subproblem size is returned (which may be different from the number of bounds), while for the IISs the number of bounds is returned (usually much smaller than the number of columns in the IIS).
5. Note that the values in `suminfeas` and `numinfeas` heavily depend on the actual basis where the simplex has stopped.
6. `iiscount` is set to `-1` if the search for IISs has not yet started.

Related topics

`XPRSgetiisdata`, `XPRSiisall`, `XPRSiisclear`, `XPRSiisfirst`, `XPRSiisisolations`, `XPRSiisnext`, `XPRSiiswrite`, `IIS`.

XPRSiiswrite

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

```
int XPRS_CC XPRSiiswrite(XPRSprob prob, int num, const char *fn, int type,
        const char *typeflags);
```

Arguments

`prob`	The current problem.
`num`	The ordinal number of the IIS to be written.
`fn`	The name of the file to be created.
`type`	Type of file to be created:
`0`	creates an lp/mps file containing the IIS as a linear programming problem;
`1`	creates a comma separated (csv) file containing the description and supplementary information on the given IIS.
`typeflags`	Flags passed to the `XPRSwriteprob` function.

Example

This writes the first IIS (if one exists and is already found) as an lp file.

```
XPRSiiswrite(prob,1,"iis.lp",0,"l")
```

Further information

1. Calling `IIS -w [num] fn` and `IIS -e [num] fn` from the console have the same effect as this function.
2. Please note that there are problems on the boundary of being infeasible or not. For such problems, feasibility or infeasibility often depends on tolerances or even on scaling. This phenomenon makes it possible that after writing an IIS out as an LP file and reading it back, it may report feasibility. As a first check it is advised to consider the following options:
   (a) turn presolve off (e.g. in console `presolve = 0`) since the nature of an IIS makes it necessary that during their identification the presolve is turned off.
   (b) use the primal simplex method to solve the problem (e.g. in console `LPOPTIMIZE -p`).
3. Note that the original sense of the original objective function plays no role in an IIS.
4. Even though an attempt is made to identify the most infeasible IISs first by the `XPRSiisfirst` (`IIS`), `XPRSiisnext` (`IIS -n`) and `XPRSiisall` (`IIS -a`) functions, it is also possible that an IIS becomes just infeasible in problems that are otherwise highly infeasible. In such cases, you may try to deal with the more stable IISs first, and consider to use the infeasibility breaker tool if only slight infeasibilities remain.
5. The LP or MPS files created by `XPRSiiswrite` corresponding to an IIS contain no objective function, since infeasibility is independent from the objective.

Related topics

`XPRSgetiisdata`, `XPRSiisall`, `XPRSiisclear`, `XPRSiisfirst`, `XPRSiisisolations`, `XPRSiisnext`, `XPRSiisstatus`, `IIS`.

XPRSinit

Purpose

Initializes the Optimizer library. This must be called before any other library routines.

Synopsis

```
int XPRS_CC XPRSinit(const char *xpress);
```

Argument

`xpress`	The directory where the FICO Xpress password file is located. Users should employ a value of `NULL` unless otherwise advised, allowing the standard initialization directories to be checked.

Example

The following is the usual way of calling `XPRSinit`:

```
if(XPRSinit(NULL)) printf("Problem with XPRSinit\n");
```

Further information

1. Whilst error checking should always be used on all library function calls, it is especially important to do so with the initialization functions, since a majority of errors encountered by users are caused at the initialization stage. Any nonzero return code indicates that no license could be found. In such circumstances the application should be made to exit. A return code of `32`, however, indicates that a student license has been found and the software will work, but with restricted functionality and problem capacity. It is possible to retrieve a message describing the error by calling `XPRSgetlicerrmsg`.
2. In multi-threaded applications where all threads are equal, `XPRSinit` may be called by each thread prior to using the library. Whilst the process of initialization will be carried out only once, this guarantees that the library functions will be available to each thread as necessary. In applications with a clear master thread, spawning other Optimizer threads, initialization need only be called by the master thread.

Related topics

`XPRScreateprob`, `XPRSfree`, `XPRSgetlicerrmsg`.

XPRSinitglobal

Purpose

Reinitializes the global tree search. By default if a global search is interrupted and called again the global search will continue from where it left off. If `XPRSinitglobal` is called after the first call to `XPRSmipoptimize`, the global search will start from the top node when `XPRSmipoptimize` is called again. This function is deprecated and might be removed in a future release. `XPRSpostsolve` should be used instead.

Synopsis

```
int XPRS_CC XPRSinitglobal(XPRSprob prob);
```

Argument

`prob`	The current problem.

Example

The following initializes the global search before attempting to solve the problem again:

```
XPRSinitglobal(prob);
XPRSmipoptimize(prob,"");
```

Related topics

`XPRSmipoptimize` (`MIPOPTIMIZE`).

XPRSinterrupt

Purpose
Interrupts the Optimizer algorithms.

Synopsis
```
int XPRS_CC XPRSinterrupt(XPRSprob prob, int reason);
```

Arguments

`prob`	The current problem.
`reason`	The reason for stopping. Possible reasons are: `XPRS_STOP_TIMELIMIT` time limit hit; `XPRS_STOP_CTRLC` control C hit; `XPRS_STOP_NODELIMIT` node limit hit; `XPRS_STOP_ITERLIMIT` iteration limit hit; `XPRS_STOP_MIPGAP` MIP gap is sufficiently small; `XPRS_STOP_SOLLIMIT` solution limit hit; `XPRS_STOP_USER` user interrupt.

Further information
The `XPRSinterrupt` command can be called from any callback.

Related topics
None.

XPRSloadbasis

Purpose

Loads a basis from the user's areas.

Synopsis

```
int XPRS_CC XPRSloadbasis(XPRSprob prob, const int rstatus[], const int
      cstatus[]);
```

Arguments

`prob`	The current problem.
`rstatus`	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: `0` slack, surplus or artificial is non-basic at lower bound; `1` slack, surplus or artificial is basic; `2` slack or surplus is non-basic at upper bound. `3` slack or surplus is super-basic.
`cstatus`	Integer array of length `COLS` containing the basis status of each of the columns in the constraint matrix. The status must be one of: `0` variable is non-basic at lower bound or superbasic at zero if the variable has no lower bound; `1` variable is basic; `2` variable is at upper bound; `3` variable is super-basic.

Example

This example loads a problem and then reloads a (previously optimized) basis from a similar problem to speed up the optimization:

```
XPRSreadprob(prob,"problem","");
XPRSloadbasis(prob,rstatus,cstatus);
XPRSlpoptimize(prob,"");
```

Further information

If the problem has been altered since saving an advanced basis, you may want to alter the basis as follows before loading it:

- Make new variables non-basic at their lower bound (`cstatus[icol]=0`), unless a variable has an infinite lower bound and a finite upper bound, in which case make the variable non-basic at its upper bound (`cstatus[icol]=2`);
- Make new constraints basic (`rstatus[jrow]=1`);
- Try not to delete basic variables, or non-basic constraints.

Related topics

`XPRSgetbasis`, `XPRSgetpresolvebasis`, `XPRSloadpresolvebasis`.

XPRSloadbranchdirs

Purpose

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

Synopsis

```
int XPRS_CC XPRSloadbranchdirs(XPRSprob prob, int ndirs, const int mcols[],
        const int mbranch[]);
```

Arguments

`prob`	The current problem.
`ndirs`	Number of directives.
`mcols`	Integer array of length `ndirs` containing the column numbers. A negative value indicates a set number (the first set being -1, the second -2, and so on).
`mbranch`	Integer array of length `ndirs` containing either 0 or 1 for the entities given in `mcols`. Entities for which mbranch is set to 1 will be branched on until fixed before a global feasible solution is returned. If `mbranch` is NULL, the branching directive will be set for all entities in `mcols`.

Related topics

`XPRSloaddirs`, `XPRSreaddirs`, A.6.

XPRSloadcuts

Purpose

Loads cuts from the cut pool into the matrix. Without calling XPRSloadcuts 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 XPRSdelcuts.

Synopsis

```
int XPRS_CC XPRSloadcuts(XPRSprob prob, int itype, int interp, int ncuts,
        const XPRScut mcutind[]);
```

Arguments

prob	The current problem.
itype	Cut type.
interp	The way in which the cut type is interpreted: -1 load all cuts; 1 treat cut types as numbers; 2 treat cut types as bit maps - load cut if any bit matches any bit set in itype; 3 treat cut types as bit maps - 0 load cut if all bits match those set in itype.
ncuts	Number of cuts to load.
mcutind	Array of length ncuts containing pointers to the cuts to be loaded into the matrix. These are pointers returned by either XPRSstorecuts or XPRSgetcpcutlist.

Further information

This function should be called only from within callback functions set by either XPRSaddcboptnode or XPRSaddcbcutmgr.

Related topics

XPRSaddcuts, XPRSdelcpcuts, XPRSdelcuts, XPRSgetcpcutlist, 5.9.

XPRSloaddelayedrows

Purpose

Specifies that a set of rows in the matrix will be treated as delayed rows during a global search. These are rows that must be satisfied for any integer solution, but will not be loaded into the active set of constraints until required.

Synopsis

```
int XPRS_CC XPRSloaddelayedrows(XPRSprob prob, int nrows, const int
      mrows[]);
```

Arguments

`prob`	The current problem.
`nrows`	The number of delayed rows.
`mrows`	An array of row indices to treat as delayed rows.

Example

This sets the first six matrix rows as delayed rows in the global problem `prob`.

```
int mrows[] = {0,1,2,3,4,5}
...
XPRSloaddelayedrows(prob,6,mrows);
XPRSmipoptimize(prob,"");
```

Further information

Delayed rows must be set up before solving the problem. Any delayed rows will be removed from the matrix after presolve and added to a special pool. A delayed row will be added back into the active matrix only when such a row is violated by an integer solution found by the Optimizer.

Related topics

`XPRSloadmodelcuts`.

XPRSloaddirs

Purpose

Loads directives into the matrix.

Synopsis

```
int XPRS_CC XPRSloaddirs(XPRSprob prob, int ndir, const int mcols[], const
      int mpri[], const char qbr[], const double dupc[], const double
      ddpc[]);
```

Arguments

`prob`	The current problem.
`ndir`	Number of directives.
`mcols`	Integer array of length `ndir` containing the column numbers. A negative value indicates a set number (the first set being `-1`, the second `-2`, and so on).
`mpri`	Integer array of length `ndir` containing the priorities for the columns or sets. Priorities must be between 0 and 1000, where columns/sets with smallest priority will be branched on first. May be `NULL` if not required.
`qbr`	Character array of length `ndir` specifying the branching direction for each column or set: `U` the entity is to be forced up; `D` the entity is to be forced down; `N` not specified. May be `NULL` if not required.
`dupc`	Double array of length `ndir` containing the up pseudo costs for the columns or sets. May be `NULL` if not required.
`ddpc`	Double array of length `ndir` containing the down pseudo costs for the columns or sets. May be `NULL` if not required.

Related topics

`XPRSgetdirs`, `XPRSloadpresolvedirs`, `XPRSreaddirs`.

XPRSloadglobal, XPRSloadglobal64

Purpose

Used to load a global problem in to 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

```
int XPRS_CC XPRSloadglobal(XPRSprob prob, const char *probname, int ncol,
      int nrow, const char qrtype[], const double rhs[], const double
      range[], const double obj[], const int mstart[], const int mnel[],
      const int mrwind[], const double dmatval[], const double dlb[], const
      double dub[], int ngents, int nsets, const char qgtype[], const int
      mgcols[], const double dlim[], const char qstype[], const int
      msstart[], const int mscols[], const double dref[]);

int XPRS_CC XPRSloadglobal64(XPRSprob prob, const char *probname, int ncol,
      int nrow, const char qrtype[], const double rhs[], const double
      range[], const double obj[], const XPRSint64 mstart[], const int
      mnel[], const int mrwind[], const double dmatval[], const double
      dlb[], const double dub[], int ngents, int nsets, const char
      qgtype[], const int mgcols[], const double dlim[], const char
      qstype[], const XPRSint64 msstart[], const int mscols[], const double
      dref[]);
```

Arguments

`prob`	The current problem.
`probname`	A string of up to `MAXPROBNAMELENGTH` characters containing a name for the problem.
`ncol`	Number of structural columns in the matrix.
`nrow`	Number of rows in the matrix not (including the objective row). Objective coefficients must be supplied in the `obj` array, and the objective function should not be included in any of the other arrays.
`qrtype`	Character array of length `nrow` containing the row types: `L` indicates a $\leq$ constraint; `E` indicates an = constraint; `G` indicates a $\geq$ constraint; `R` indicates a range constraint; `N` indicates a nonbinding constraint.
`rhs`	Double array of length `nrow` containing the right hand side coefficients. The right hand side value for a range row gives the **upper** bound on the row.
`range`	Double array of length `nrow` containing the range values for range rows. Values for all other rows will be ignored. May be `NULL` if not required. The lower bound on a range row is the right hand side value minus the range value. The sign of the range value is ignored - the absolute value is used in all cases.
`obj`	Double array of length `ncol` containing the objective function coefficients.
`mstart`	Integer array containing the offsets in the `mrwind` and `dmatval` arrays of the start of the elements for each column. This array is of length `ncol` or, if `mnel` is `NULL`, length `ncol+1`. If `mnel` is `NULL`, the extra entry of `mstart`, `mstart[ncol]`, contains the position in the `mrwind` and `dmatval` arrays at which an extra column would start, if it were present. In C, this value is also the length of the `mrwind` and `dmatval` arrays.
`mnel`	Integer array of length `ncol` containing the number of nonzero elements in each column. May be `NULL` if not required. This array is not required if the non-zero

	coefficients in the `mrwind` and `dmatval` arrays are continuous, and the `mstart` array has `ncol+1` entries as described above. It may be `NULL` if not required.
`mrwind`	Integer arrays containing the row indices for the nonzero elements in each column. If the indices are input contiguously, with the columns in ascending order, then the length of `mrwind` is `mstart[ncol-1]+mnel[ncol-1]` or, if `mnel` is `NULL`, `mstart[ncol]`.
`dmatval`	Double array containing the nonzero element values length as for `mrwind`.
`dlb`	Double array of length `ncol` containing the lower bounds on the columns. Use `XPRS_MINUSINFINITY` to represent a lower bound of minus infinity.
`dub`	Double array of length `ncol` containing the upper bounds on the columns. Use `XPRS_PLUSINFINITY` to represent an upper bound of plus infinity.
`ngents`	Number of binary, integer, semi-continuous, semi-continuous integer and partial integer entities.
`nsets`	Number of SOS1 and SOS2 sets.
`qgtype`	Character array of length `ngents` containing the entity types: `B` binary variables; `I` integer variables; `P` partial integer variables; `S` semi-continuous variables; `R` semi-continuous integer variables.
`mgcols`	Integer array of length `ngents` containing the column indices of the global entities.
`dlim`	Double array of length `ngents` 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.
`qstype`	Character array of length `nsets` containing the set types: `1` SOS1 type sets; `2` SOS2 type sets. May be `NULL` if not required.
`msstart`	Integer array containing the offsets in the `mscols` and `dref` arrays indicating the start of the sets. This array is of length `nsets+1`, the last member containing the offset where set `nsets+1` would start. May be `NULL` if not required.
`mscols`	Integer array of length `msstart[nsets]-1` containing the columns in each set. May be `NULL` if not required.
`dref`	Double array of length `msstart[nsets]-1` containing the reference row entries for each member of the sets. May be `NULL` if not required.

Related controls

Integer

`EXTRACOLS`	Number of extra columns to be allowed for.
`EXTRAELEMS`	Number of extra matrix elements to be allowed for.
`EXTRAMIPENTS`	Number of extra global entities to be allowed for.
`EXTRAPRESOLVE`	Number of extra elements to allow for in presolve.
`EXTRAROWS`	Number of extra rows to be allowed for.
`KEEPNROWS`	Status for nonbinding rows.
`SCALING`	Type of scaling.

Double

`MATRIXTOL`	Tolerance on matrix elements.
`SOSREFTOL`	Minimum gap between reference row entries.

Example

The following specifies an integer problem, `globalEx`, corresponding to:

minimize:	x + 2y		
subject to:	3x + 2y	$\leq$	400
	x + 3y	$\leq$	200

with both *x* and *y* integral:

```
char probname[]    = "globalEx";
int ncol = 2, nrow = 2;
char qrtype[]      = {'L','L'};
double rhs[]       = {400.0, 200.0};
int mstart[]       = {0, 2, 4};
int mrwind[]       = {0, 1, 0, 1};
double dmatval[]   = {3.0, 1.0, 2.0, 3.0};
double objcoefs[] = {1.0, 2.0};
double dlb[]       = {0.0, 0.0};
double dub[]       = {200.0, 200.0};

int ngents = 2;
int nsets = 0;
char qgtype[] = {"I","I"};
int mgcols[] = {0,1};
...
XPRSloadglobal(prob, probname, ncol, nrow, qrtype, rhs, NULL,
               objcoefs, mstart, NULL, mrwind,
               dmatval, dlb, dub, ngents, nsets, qgtype, mgcols,
               NULL, NULL, NULL, NULL, NULL);
```

Further information

1. The row and column indices follow the usual C convention of going from `0` to `nrow-1` and `0` to `ncol-1` respectively.
2. The double constants `XPRS_PLUSINFINITY` and `XPRS_MINUSINFINITY` are defined in the Optimizer library header file.
3. Semi-continuous lower bounds are taken from the `dlim` array. If this is `NULL` then they are given a default value of `1.0`. If a semi-continuous variable has a positive lower bound then this will be used as the semi-continuous lower bound and the lower bound on the variable will be set to zero.

Related topics

`XPRSaddsetnames`, `XPRSloadlp`, `XPRSloadqglobal`, `XPRSloadqp`, `XPRSreadprob`.

XPRSloadlp, XPRSloadlp64

Purpose

Enables the user to pass a matrix directly to the Optimizer, rather than reading the matrix from a file.

Synopsis

```
int XPRS_CC XPRSloadlp(XPRSprob prob, const char *probname, int ncol, int
      nrow, const char qrtype[], const double rhs[], const double range[],
      const double obj[], const int mstart[], const int mnel[], const int
      mrwind[], const double dmatval[], const double dlb[], const double
      dub[]);

int XPRS_CC XPRSloadlp64(XPRSprob prob, const char *probname, int ncol, int
      nrow, const char qrtype[], const double rhs[], const double range[],
      const double obj[], const XPRSint64 mstart[], const int mnel[], const
      int mrwind[], const double dmatval[], const double dlb[], const
      double dub[]);
```

Arguments

`prob`	The current problem.
`probname`	A string of up to `MAXPROBNAMELENGTH` characters containing a names for the problem.
`ncol`	Number of structural columns in the matrix.
`nrow`	Number of rows in the matrix (not including the objective). Objective coefficients must be supplied in the `obj` array, and the objective function should not be included in any of the other arrays.
`qrtype`	Character array of length `nrow` containing the row types: `L` indicates a $\leq$ constraint; `E` indicates an = constraint; `G` indicates a $\geq$ constraint; `R` indicates a range constraint; `N` indicates a nonbinding constraint.
`rhs`	Double array of length `nrow` containing the right hand side coefficients of the rows. The right hand side value for a range row gives the **upper** bound on the row.
`range`	Double array of length `nrow` containing the range values for range rows. Values for all other rows will be ignored. May be `NULL` if not required. The lower bound on a range row is the right hand side value minus the range value. The sign of the range value is ignored - the absolute value is used in all cases.
`obj`	Double array of length `ncol` containing the objective function coefficients.
`mstart`	Integer array containing the offsets in the `mrwind` and `dmatval` arrays of the start of the elements for each column. This array is of length `ncol` or, if `mnel` is `NULL`, length `ncol+1`. If `mnel` is `NULL`, the extra entry of `mstart`, `mstart[ncol]`, contains the position in the `mrwind` and `dmatval` arrays at which an extra column would start, if it were present. In C, this value is also the length of the `mrwind` and `dmatval` arrays.
`mnel`	Integer array of length `ncol` containing the number of nonzero elements in each column. May be `NULL` if not required. This array is not required if the non-zero coefficients in the `mrwind` and `dmatval` arrays are continuous, and the `mstart` array has `ncol+1` entries as described above.
`mrwind`	Integer array containing the row indices for the nonzero elements in each column. If the indices are input contiguously, with the columns in ascending order, the length of the `mrwind` is `mstart[ncol-1]+mnel[ncol-1]` or, if `mnel` is `NULL`, `mstart[ncol]`.
`dmatval`	Double array containing the nonzero element values; length as for `mrwind`.
`dlb`	Double array of length `ncol` containing the lower bounds on the columns. Use

	XPRS_MINUSINFINITY to represent a lower bound of minus infinity.
dub	Double array of length ncol containing the upper bounds on the columns. Use XPRS_PLUSINFINITY to represent an upper bound of plus infinity.

Related controls

Integer

EXTRACOLS	Number of extra columns to be allowed for.
EXTRAELEMS	Number of extra matrix elements to be allowed for.
EXTRAPRESOLVE	Number of extra elements to allow for in presolve.
EXTRAROWS	Number of extra rows to be allowed for.
KEEPNROWS	Status for nonbinding rows.
SCALING	Type of scaling.

Double

MATRIXTOL	Tolerance on matrix elements.

Example

Given an LP problem:

minimize:	x + y		
subject to:	2x	$\geq$	3
	x + 2y	$\geq$	3
	x + y	$\geq$	1

the following shows how this may be loaded into the Optimizer using XPRSloadlp:

```
char probname[] = "small";
int ncol = 2, nrow = 3;
char qrtype[]     = {'G','G','G'};
double rhs[]      = { 3 , 3 , 1 };
double obj[]      = { 1 , 1 };
int mstart[]      = { 0 , 3 , 5 };
int mrwind[]      = { 0 , 1 , 2 , 1 , 2 };
double dmatval[] = { 2 , 1 , 1 , 2 , 1 };
double dlb[]    = { 0 , 0 };
double dub[]    = {XPRS_PLUSINFINITY,XPRS_PLUSINFINITY};

XPRSloadlp(prob, probname, ncol, nrow, qrtype, rhs, NULL,
           obj, mstart, NULL, mrwind, dmatval, dlb, dub)
```

Further information

1. The row and column indices follow the usual C convention of going from 0 to nrow-1 and 0 to ncol-1 respectively.

2. The double constants XPRS_PLUSINFINITY and XPRS_MINUSINFINITY are defined in the Optimizer library header file.

3. For a range constraint, the value in the rhs array specifies the upper bound on the constraint, while the value in the range array specifies the range on the constraint. So a range constraint *j* is interpreted as:

$$rhs_j - |range_j| \leq \sum_i a_{ij} x_i \leq rhs_j$$

Related topics

XPRSloadglobal, XPRSloadqglobal, XPRSloadqp, XPRSreadprob.

XPRSloadlpsol

Purpose

Loads an LP solution for the problem into the Optimizer.

Synopsis

```
int XPRS_CC XPRSloadlpsol(XPRSprob prob, const double x[], const double
      slack[], const double dual[], const double dj[], int *status);
```

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.
`dual`	Optional: double array of length `ROWS` containing the values of dual variables.
`dj`	Optional: double array of length `COLS` containing the values of reduced costs.
`status`	Pointer to an `int` where the status will be returned. The status is one of: `0` Solution is loaded. `1` Solution is not loaded because the problem is in presolved status.

Example

This example loads a problem, loads a solution for the problem and then uses `XPRScrossoverlpsol` to find a basic optimal solution.

```
XPRSreadprob(prob, "problem", "");
XPRSloadlpsol(prob, x, NULL, dual, NULL, &status);
XPRScrossoverlpsol(prob, &status);
```

Further information

1. At least one of variables `x` and dual variables `dual` must be provided.
2. When variables `x` is `NULL`, the variables will be set to their bounds.
3. When slack variables `slack` is `NULL`, it will be computed from variables `x`. If slacks are provided, variables cannot be omitted.
4. When dual variables `dual` is `NULL`, both dual variables and reduced costs will be set to zero.
5. When reduced costs `dj` is `NULL`, it will be computed from dual variables `dual`. If reduced costs are provided, dual variables cannot be omitted.

Related topics

`XPRSgetlpsol`, `XPRScrossoverlpsol`.

XPRSloadmipsol

Purpose

Loads a starting MIP solution for the problem into the Optimizer.

Synopsis

```
int XPRS_CC XPRSloadmipsol(XPRSprob prob, const double dsol[], int
       *status);
```

Arguments

`prob`	The current problem.
`dsol`	Double array of length COLS (for the original problem and not the presolve problem) containing the values of the variables.
`status`	Pointer to an `int` where the status will be returned. The status is one of:
`-1`	Solution rejected because an error occurred;
`0`	Solution accepted. When loading a solution before a MIP solve, the solution is always accepted. See Further Information below.
`1`	Solution rejected because it is infeasible;
`2`	Solution rejected because it is cut off;
`3`	Solution rejected because the LP reoptimization was interrupted.

Example

This example loads a problem and then loads a solution found previously for the problem to help speed up the MIP search:

```
XPRSreadprob(prob,"problem","");
XPRSloadmipsol(prob,dsol,&status);
XPRSmipoptimize(prob,"");
```

Further information

1. When a solution is loaded before a MIP solve, the solution is placed in temporary storage until the MIP solve is started. Only after the MIP solve has commenced and any presolve has been applied, will the loaded solution be checked and possibly accepted as a new incumbent integer solution. There are no checks performed on the solution before the MIP solve and the returned status in `XPRSloadmipsol` will always be `0` for accepted.

2. Loaded solution values will automatically be adjusted to fit within the current problem bounds.

3. It is recommended to use `XPRSaddmipsol` instead of `XPRSloadmipsol`. `XPRSaddmipsol` can be called both before a solve, to load a starting solution, and during a MIP solve, to load new solutions within callbacks. `XPRSaddmipsol` also allows for loading of infeasible or partial solutions and comes with a callback to check the status of loaded solutions.

Related topics

`XPRSaddmipsol`, `XPRSgetmipsol`.

XPRSloadmodelcuts

Purpose

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

Synopsis

```
int XPRS_CC XPRSloadmodelcuts(XPRSprob prob, int nmod, const int mrows[]);
```

Arguments

`prob`	The current problem.
`nmod`	The number of model cuts.
`mrows`	An array of row indices to be treated as cuts.

Error value

268	Cannot perform operation on presolved matrix.

Example

This sets the first six matrix rows as model cuts in the global problem `myprob`.

```
int mrows[] = {0,1,2,3,4,5}
...
XPRSloadmodelcuts(prob,6,mrows);
XPRSmipoptimize(prob,"");
```

Further information

1. During presolve the model cuts are removed from the matrix and added to an internal cut pool. During the global search, the Optimizer will regularly check this cut pool for any violated model cuts and add those that cuts off a node LP solution.
2. The model cuts must be "true" model cuts, in the sense that they are redundant at the optimal MIP solution. The Optimizer does not guarantee to add all violated model cuts, so they must not be required to define the optimal MIP solution.

Related topics

5.9.

XPRSloadpresolvebasis

Purpose

Loads a presolved basis from the user's areas.

Synopsis

```
int XPRS_CC XPRSloadpresolvebasis(XPRSprob prob, const int rstatus[], const
      int cstatus[]);
```

Arguments

`prob` The current problem.

`rstatus` 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:
- `0` slack, surplus or artificial is non-basic at lower bound;
- `1` slack, surplus or artificial is basic;
- `2` slack or surplus is non-basic at upper bound.

`cstatus` Integer array of length `COLS` containing the basis status of each of the columns in the matrix. The status must be one of:
- `0` variable is non-basic at lower bound or superbasic at zero if the variable has no lower bound;
- `1` variable is basic;
- `2` variable is at upper bound;
- `3` variable is super-basic.

Example

The following example saves the presolved basis for one problem, loading it into another:

```
int rows, cols, *rstatus, *cstatus;
...
XPRSreadprob(prob,"myprob","");
XPRSmipoptimize(prob,"l");
XPRSgetintattrib(prob,XPRS_ROWS,&rows);
XPRSgetintattrib(prob,XPRS_COLS,&cols);
rstatus = malloc(rows*sizeof(int));
cstatus = malloc(cols*sizeof(int));
XPRSgetpresolvebasis(prob,rstatus,cstatus);
XPRSreadprob(prob2,"myotherprob","");
XPRSmipoptimize(prob2,"l");
XPRSloadpresolvebasis(prob2,rstatus,cstatus);
```

Related topics

`XPRSgetbasis`, `XPRSgetpresolvebasis`, `XPRSloadbasis`.

XPRSloadpresolvedirs

Purpose

Loads directives into the presolved matrix.

Synopsis

```
int XPRS_CC XPRSloadpresolvedirs(XPRSprob prob, int ndir, const int
      mcols[], const int mpri[], const char qbr[], const double dupc[],
      const double ddpc[]);
```

Arguments

`prob`	The current problem.
`ndir`	Number of directives.
`mcols`	Integer array of length `ndir` containing the column numbers. A negative value indicates a set number (`−1` being the first set, `−2` the second, and so on).
`mpri`	Integer array of length `ndir` containing the priorities for the columns or sets. May be `NULL` if not required.
`qbr`	Character array of length `ndir` specifying the branching direction for each column or set: `U` the entity is to be forced up; `D` the entity is to be forced down; `N` not specified. May be `NULL` if not required.
`dupc`	Double array of length `ndir` containing the up pseudo costs for the columns or sets. May be `NULL` if not required.
`ddpc`	Double array of length `ndir` containing the down pseudo costs for the columns or sets. May be `NULL` if not required.

Example

The following loads priority directives for column `0` in the matrix:

```
int mcols[] = {0}, mpri[] = {1};
...
XPRSmipoptimize(prob,"l");
XPRSloadpresolvedirs(prob,1,mcols,mpri,NULL,NULL,NULL);
XPRSmipoptimize(prob,"");
```

Related topics

`XPRSgetdirs`, `XPRSloaddirs`.

XPRSloadqcqp, XPRSloadqcqp64

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

```
int XPRS_CC XPRSloadqcqp(XPRSprob prob, const char * probname, int ncol,
      int nrow, const char qrtypes[], const double rhs[], const double
      range[], const double obj[], const int mstart[], const int mnel[],
      const int mrwind[], const double dmatval[], const double dlb[], const
      double dub[], int nqtr, const int mqcol1[], const int mqcol2[], const
      double dqe[], int qmn, const int qcrows[], const int qcnquads[],
      const int qcmqcol1[], const int qcmqcol2[], const double qcdqval[]);

int XPRS_CC XPRSloadqcqp64(XPRSprob prob, const char * probname, int ncol,
      int nrow, const char qrtypes[], const double rhs[], const double
      range[], const double obj[], const XPRSint64 mstart[], const int
      mnel[], const int mrwind[], const double dmatval[], const double
      dlb[], const double dub[], XPRSint64 nqtr, const int mqcol1[], const
      int mqcol2[], const double dqe[], int qmn, const int qcrows[], const
      XPRSint64 qcnquads[], const int qcmqcol1[], const int qcmqcol2[],
      const double qcdqval[]);
```

Arguments

`prob`	The current problem.
`probname`	A string of up to `MAXPROBNAMELENGTH` characters containing a name for the problem.
`ncol`	Number of structural columns in the matrix.
`nrow`	Number of rows in the matrix (not including the objective row). Objective coefficients must be supplied in the `obj` array, and the objective function should not be included in any of the other arrays.
`qrtype`	Character array of length `nrow` 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 `nrow` containing the right hand side coefficients of the rows. The right hand side value for a range row gives the **upper** bound on the row.
`range`	Double array of length `nrow` containing the range values for range rows. Values for all other rows will be ignored. May be NULL if there are no ranged constraints. The lower bound on a range row is the right hand side value minus the range value. The sign of the range value is ignored - the absolute value is used in all cases.
`obj`	Double array of length `ncol` containing the objective function coefficients.
`mstart`	Integer array containing the offsets in the `mrwind` and `dmatval` arrays of the start of the elements for each column. This array is of length `ncol` or, if `mnel` is NULL, `length ncol+1`. If `mnel` is NULL the extra entry of `mstart`, `mstart[ncol]`, contains the position in the `mrwind` and `dmatval` arrays at which an extra column would start, if it were present. In C, this value is also the length of the `mrwind` and `dmatval` arrays.
`mnel`	Integer array of length `ncol` containing the number of nonzero elements in each column. May be NULL if all elements are contiguous and `mstart[ncol]` contains the offset where the elements for column `ncol+1` would start. This array is not required if the non-zero coefficients in the `mrwind` and `dmatval` arrays are continuous, and the

	mstart array has ncol+1 entries as described above. It may be NULL if not required.
mrwind	Integer array containing the row indices for the nonzero elements in each column. If the indices are input contiguously, with the columns in ascending order, the length of the mrwind is mstart[ncol-1]+mnel[ncol-1] or, if mnel is NULL, mstart[ncol].
dmatval	Double array containing the nonzero element values; length as for mrwind.
dlb	Double array of length ncol containing the lower bounds on the columns. Use XPRS_MINUSINFINITY to represent a lower bound of minus infinity.
dub	Double array of length ncol containing the upper bounds on the columns. Use XPRS_PLUSINFINITY to represent an upper bound of plus infinity.
nqtr	Number of quadratic terms.
mqc1	Integer array of size nqtr containing the column index of the first variable in each quadratic term.
mqc2	Integer array of size nqtr containing the column index of the second variable in each quadratic term.
dqe	Double array of size nqtr containing the quadratic coefficients.
qmn	Number of rows containing quadratic matrices.
qcrows	Integer array of size qmn, containing the indices of rows with quadratic matrices in them. Note that the rows are expected to be defined in qrtype as type L.
qcnquads	Integer array of size qmn, containing the number of nonzeros in each quadratic constraint matrix.
qcmqcol1	Integer array of size nqcelem, where nqcelem equals the sum of the elements in qcnquads (i.e. the total number of quadratic matrix elements in all the constraints). It contains the first column indices of the quadratic matrices. Indices for the first matrix are listed from 0 to qcnquads[0]-1, for the second matrix from qcnquads[0] to qcnquads[0]+ qcnquads[1]-1, etc.
qcmqcol2	Integer array of size nqcelem, containing the second index for the quadratic constraint matrices.
qcdqval	Integer array of size nqcelem, containing the coefficients for the quadratic constraint matrices.

Related controls

Integer

EXTRACOLS	Number of extra columns to be allowed for.
EXTRAELEMS	Number of extra matrix elements to be allowed for.
EXTRAMIPENTS	Number of extra global entities to be allowed for.
EXTRAPRESOLVE	Number of extra elements to allow for in presolve.
EXTRAQCELEMENTS	Number of extra qcqp elements to be allowed for.
EXTRAQCROWS	Number of extra qcqp matrices to be allowed for.
EXTRAROWS	Number of extra rows to be allowed for.
KEEPNROWS	Status for nonbinding rows.
SCALING	Type of scaling.

Double

MATRIXTOL	Tolerance on matrix elements.

Example

To load the following problem presented in LP format:

```
minimize [ x^2 ]
s.t.
4 x + y <= 4
x + y + [z^2] <= 5
```

```
 [ x^2 + 2 x*y + y^2 + 4 y*z + z^2 ] <= 10
x + 2 y >= 8
 [ 3 y^2 ] <= 20
end
```

the following code may be used:

```
{
    int ncols = 3;
    int nrows = 5;
    char rowtypes[] = {'L','L','L','G','L'};
    double rhs[] = {4,5,10,8,20};
    double range[] = {0,0,0,0,0};
    double obj[] = {0,0,0,0,0};
    int mstart[] = {0,3,6,6};
    int* mnel = NULL;
    int mrind[] = {0,1,3,0,1,3};
    double dmatval[] = {4,1,1,1,1,2};
    double lb[] = {0,0,0};
    double ub[] = {XPRS_PLUSINFINITY,XPRS_PLUSINFINITY,
    XPRS_PLUSINFINITY};

    int nqtr = 1;
    int mqc1[] = {0};
    int mqc2[] = {0};
    double dqe[] = {1};

    int qmn = 3;
    int qcrows[] = {1,2,4};
    int qcnquads[] = {1,5,1};
    int qcmcol1[] = {2,0,0,1,1,2,1};
    int qcmcol2[] = {2,0,1,1,2,2,1};
    // ! to have 2xy define 1xy (1yx will be assumed to be implicitly present)
    double qcdqval[] = {1,9,1,8,2,7,3};
}

XPRSloadqcqp(xprob,"qcqp",ncols,nrows,rowtypes,rhs,range,obj,mstart,
mnel,mrind,dmatval,lb,ub,nqtr,mqc1,mqc2,dqe,qmn,qcrows,qcnquads,
qcmcol1,qcmcol2,qcdqval);

}
```

Further information

1. The objective function is of the form $c^T x + 0.5\ x^T Q x$ where Q is positive semi-definite for minimization problems and negative semi-definite for maximization problems. If this is not the case the optimization algorithms may converge to a local optimum or may not converge at all. Note that only the upper or lower triangular part of the Q matrix is specified.
2. All Q matrices in the constraints must be positive semi-definite. Note that only the upper or lower triangular part of the Q matrix is specified for constraints as well.
3. The row and column indices follow the usual C convention of going from `0` to `nrow-1` and `0` to `ncol-1` respectively.
4. The double constants `XPRS_PLUSINFINITY` and `XPRS_MINUSINFINITY` are defined in the Optimizer library header file.

Related topics

XPRSloadglobal, XPRSloadlp, XPRSloadqglobal, XPRSloadqp, XPRSreadprob.

XPRSloadqcqpglobal, XPRSloadqcqpglobal64

Purpose

Used to load a global, 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. 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

```
int XPRS_CC XPRSloadqcqpglobal(XPRSprob prob, const char * probname, int
       ncol, int nrow, const char qrtypes[], const double rhs[], const
       double range[], const double obj[], const int mstart[], const int
       mnel[], const int mrwind[], const double dmatval[], const double
       dlb[], const double dub[], int nqtr, const int mqcol1[], const int
       mqcol2[], const double dqe[], int qmn, const int qcrows[], const int
       qcnquads[], const int qcmqcol1[], const int qcmqcol2[], const double
       qcdqval[], const int ngents, const int nsets, const char qgtype[],
       const int mgcols[], const double dlim[], const char qstype[], const
       int msstart[], const int mscols[], const double dref[]);

int XPRS_CC XPRSloadqcqpglobal64(XPRSprob prob, const char * probname, int
       ncol, int nrow, const char qrtypes[], const double rhs[], const
       double range[], const double obj[], const XPRSint64 mstart[], const
       int mnel[], const int mrwind[], const double dmatval[], const double
       dlb[], const double dub[], XPRSint64 nqtr, const int mqcol1[], const
       int mqcol2[], const double dqe[], int qmn, const int qcrows[], const
       XPRSint64 qcnquads[], const int qcmqcol1[], const int qcmqcol2[],
       const double qcdqval[], const int ngents, const int nsets, const char
       qgtype[], const int mgcols[], const double dlim[], const char
       qstype[], const XPRSint64 msstart[], const int mscols[], const double
       dref[]);
```

Arguments

`prob`	The current problem.
`probname`	A string of up to `MAXPROBNAMELENGTH` characters containing a name for the problem.
`ncol`	Number of structural columns in the matrix.
`nrow`	Number of rows in the matrix (not including the objective row). Objective coefficients must be supplied in the `obj` array, and the objective function should not be included in any of the other arrays.
`qrtype`	Character array of length `nrow` 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 `nrow` containing the right hand side coefficients of the rows. The right hand side value for a range row gives the **upper** bound on the row.
`range`	Double array of length `nrow` containing the range values for range rows. Values for all other rows will be ignored. May be NULL if there are no ranged constraints. The lower bound on a range row is the right hand side value minus the range value. The sign of the range value is ignored - the absolute value is used in all cases.
`obj`	Double array of length `ncol` containing the objective function coefficients.

mstart	Integer array containing the offsets in the mrwind and dmatval arrays of the start of the elements for each column. This array is of length ncol or, if mnel is NULL, length ncol+1. If mnel is NULL the extra entry of mstart, mstart[ncol], contains the position in the mrwind and dmatval arrays at which an extra column would start, if it were present. In C, this value is also the length of the mrwind and dmatval arrays.
mnel	Integer array of length ncol containing the number of nonzero elements in each column. May be NULL if all elements are contiguous and mstart[ncol] contains the offset where the elements for column ncol+1 would start. This array is not required if the non-zero coefficients in the mrwind and dmatval arrays are continuous, and the mstart array has ncol+1 entries as described above. It may be NULL if not required.
mrwind	Integer array containing the row indices for the nonzero elements in each column. If the indices are input contiguously, with the columns in ascending order, the length of the mrwind is mstart[ncol-1]+mnel[ncol-1] or, if mnel is NULL, mstart[ncol].
dmatval	Double array containing the nonzero element values; length as for mrwind.
dlb	Double array of length ncol containing the lower bounds on the columns. Use XPRS_MINUSINFINITY to represent a lower bound of minus infinity.
dub	Double array of length ncol containing the upper bounds on the columns. Use XPRS_PLUSINFINITY to represent an upper bound of plus infinity.
nqtr	Number of quadratic terms.
mqc1	Integer array of size nqtr containing the column index of the first variable in each quadratic term.
mqc2	Integer array of size nqtr containing the column index of the second variable in each quadratic term.
dqe	Double array of size nqtr containing the quadratic coefficients.
qmn	Number of rows containing quadratic matrices.
qcrows	Integer array of size qmn, containing the indices of rows with quadratic matrices in them. Note that the rows are expected to be defined in qrtype as type L.
qcnquads	Integer array of size qmn, containing the number of nonzeros in each quadratic constraint matrix.
qcmqcol1	Integer array of size nqcelem, where nqcelem equals the sum of the elements in qcnquads (i.e. the total number of quadratic matrix elements in all the constraints). It contains the first column indices of the quadratic matrices. Indices for the first matrix are listed from 0 to qcnquads[0]-1, for the second matrix from qcnquads[0] to qcnquads[0]+ qcnquads[1]-1, etc.
qcmqcol2	Integer array of size nqcelem, containing the second index for the quadratic constraint matrices.
qcdqval	Integer array of size nqcelem, containing the coefficients for the quadratic constraint matrices.
ngents	Number of binary, integer, semi-continuous, semi-continuous integer and partial integer entities.
nsets	Number of SOS1 and SOS2 sets.
qgtype	Character array of length ngents containing the entity types: B binary variables; I integer variables; P partial integer variables; S semi-continuous variables; R semi-continuous integer variables.
mgcols	Integer array of length ngents containing the column indices of the global entities.
dlim	Double array of length ngents 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.
`qstype`	Character array of length `nsets` containing the set types: `1` SOS1 type sets; `2` SOS2 type sets. May be `NULL` if not required.
`msstart`	Integer array containing the offsets in the `mscols` and `dref` arrays indicating the start of the sets. This array is of length `nsets+1`, the last member containing the offset where set `nsets+1` would start. May be `NULL` if not required.
`mscols`	Integer array of length `msstart[nsets]-1` containing the columns in each set. May be `NULL` if not required.
`dref`	Double array of length `msstart[nsets]-1` containing the reference row entries for each member of the sets. May be `NULL` if not required.

Related controls

Integer

`EXTRACOLS`	Number of extra columns to be allowed for.
`EXTRAELEMS`	Number of extra matrix elements to be allowed for.
`EXTRAMIPENTS`	Number of extra global entities to be allowed for.
`EXTRAPRESOLVE`	Number of extra elements to allow for in presolve.
`EXTRAQCELEMENTS`	Number of extra `qcqp` elements to be allowed for.
`EXTRAQCROWS`	Number of extra `qcqp` matrices to be allowed for.
`EXTRAROWS`	Number of extra rows to be allowed for.
`KEEPNROWS`	Status for nonbinding rows.
`SCALING`	Type of scaling.

Double

`MATRIXTOL`	Tolerance on matrix elements.

Further information

1. The objective function is of the form $c^T x + 0.5\ x^T Qx$ where `Q` is positive semi-definite for minimization problems and negative semi-definite for maximization problems. If this is not the case the optimization algorithms may converge to a local optimum or may not converge at all. Note that only the upper or lower triangular part of the `Q` matrix is specified.

2. All `Q` matrices in the constraints must be positive semi-definite. Note that only the upper or lower triangular part of the `Q` matrix is specified for constraints as well.

3. The row and column indices follow the usual C convention of going from `0` to `nrow-1` and `0` to `ncol-1` respectively.

4. The double constants `XPRS_PLUSINFINITY` and `XPRS_MINUSINFINITY` are defined in the Optimizer library header file.

5. The row and column indices follow the usual C convention of going from `0` to `nrow-1` and `0` to `ncol-1` respectively.

6. The double constants `XPRS_PLUSINFINITY` and `XPRS_MINUSINFINITY` are defined in the Optimizer library header file.

7. Semi-continuous lower bounds are taken from the dlim array. If this is `NULL` then they are given a default value of `1.0`. If a semi-continuous variable has a positive lower bound then this will be used as the semi-continuous lower bound and the lower bound on the variable will be set to zero.

Related topics

`XPRSloadglobal`, `XPRSloadlp`, `XPRSloadqcqp`, `XPRSloadqglobal`, `XPRSloadqp`, XPRSreadprob.

XPRSloadqglobal, XPRSloadqglobal64

Purpose

Used to load a global problem with quadratic objective coefficients in to 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

```
int XPRS_CC XPRSloadqglobal(XPRSprob prob, const char *probname, int ncol,
      int nrow, const char qrtype[], const double rhs[], const double
      range[], const double obj[], const int mstart[], const int mnel[],
      const int mrwind[], const double dmatval[], const double dlb[], const
      double dub[], int nqtr, const int mqc1[], const int mqc2[], const
      double dqe[], const int ngents, const int nsets, const char qgtype[],
      const int mgcols[], const double dlim[], const char qstype[], const
      int msstart[], const int mscols[], const double dref[]);

int XPRS_CC XPRSloadqglobal64(XPRSprob prob, const char *probname, int
      ncol, int nrow, const char qrtype[], const double rhs[], const double
      range[], const double obj[], const XPRSint64 mstart[], const int
      mnel[], const int mrwind[], const double dmatval[], const double
      dlb[], const double dub[], XPRSint64 nqtr, const int mqc1[], const
      int mqc2[], const double dqe[], const int ngents, const int nsets,
      const char qgtype[], const int mgcols[], const double dlim[], const
      char qstype[], const XPRSint64 msstart[], const int mscols[], const
      double dref[]);
```

Arguments

`prob`	The current problem.
`probname`	A string of up to `MAXPROBNAMELENGTH` characters containing a name for the problem.
`ncol`	Number of structural columns in the matrix.
`nrow`	Number of rows in the matrix (not including the objective). Objective coefficients must be supplied in the `obj` array, and the objective function should not be included in any of the other arrays.
`qrtype`	Character array of length `nrow` containing the row type: `L` indicates a $\leq$ constraint; `E` indicates an = constraint; `G` indicates a $\geq$ constraint; `R` indicates a range constraint; `N` indicates a nonbinding constraint.
`rhs`	Double array of length `nrow` containing the right hand side coefficients. The right hand side value for a range row gives the **upper** bound on the row.
`range`	Double array of length `nrow` containing the range values for range rows. The values in the range array will only be read for `R` type rows. The entries for other type rows will be ignored. May be `NULL` if not required. The lower bound on a range row is the right hand side value minus the range value. The sign of the range value is ignored - the absolute value is used in all cases.
`obj`	Double array of length `ncol` containing the objective function coefficients.
`mstart`	Integer array containing the offsets in the `mrwind` and `dmatval` arrays of the start of the elements for each column. This array is of length `ncol` or, if `mnel` is `NULL`, length `ncol+1`.
`mnel`	Integer array of length `ncol` containing the number of nonzero elements in each

	column. May be NULL if not required. This array is not required if the non-zero coefficients in the mrwind and dmatval arrays are continuous, and the mstart array has ncol+1 entries as described above. It may be NULL if not required.
mrwind	Integer arrays containing the row indices for the nonzero elements in each column. If the indices are input contiguously, with the columns in ascending order, then the length of mrwind is mstart[ncol-1]+mnel[ncol-1] or, if mnel is NULL, mstart[ncol].
dmatval	Double array containing the nonzero element values length as for mrwind.
dlb	Double array of length ncol containing the lower bounds on the columns. Use XPRS_MINUSINFINITY to represent a lower bound of minus infinity.
dub	Double array of length ncol containing the upper bounds on the columns. Use XPRS_PLUSINFINITY to represent an upper bound of plus infinity.
nqtr	Number of quadratic terms.
mqc1	Integer array of size nqtr containing the column index of the first variable in each quadratic term.
mqc2	Integer array of size nqtr containing the column index of the second variable in each quadratic term.
dqe	Double array of size nqtr containing the quadratic coefficients.
ngents	Number of binary, integer, semi-continuous, semi-continuous integer and partial integer entities.
nsets	Number of SOS1 and SOS2 sets.
qgtype	Character array of length ngents containing the entity types: B binary variables; I integer variables; P partial integer variables; S semi-continuous variables; R semi-continuous integers.
mgcols	Integer array of length ngents containing the column indices of the global entities.
dlim	Double array of length ngents 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.
qstype	Character array of length nsets containing: 1 SOS1 type sets; 2 SOS2 type sets. May be NULL if not required.
msstart	Integer array containing the offsets in the mscols and dref arrays indicating the start of the sets. This array is of length nsets+1, the last member containing the offset where set nsets+1 would start. May be NULL if not required.
mscols	Integer array of length msstart[nsets]-1 containing the columns in each set. May be NULL if not required.
dref	Double array of length msstart[nsets]-1 containing the reference row entries for each member of the sets. May be NULL if not required.

Related controls

Integer

EXTRACOLS	Number of extra columns to be allowed for.
EXTRAELEMS	Number of extra matrix elements to be allowed for.
EXTRAMIPENTS	Number of extra global entities to be allowed for.
EXTRAPRESOLVE	Number of extra elements to allow for in presolve.
EXTRAROWS	Number of extra rows to be allowed for.

KEEPNROWS	Status for nonbinding rows.
SCALING	Type of scaling.

Double

MATRIXTOL	Tolerance on matrix elements.
SOSREFTOL	Minimum gap between reference row entries.

Example

Minimize $-6x_1 + 2x_1^2 - 2x_1x_2 + 2x_2^2$ subject to $x_1 + x_2 \leq 1.9$, where x_1 must be an integer:

```
int nrow = 1, ncol = 2, nquad = 3;
int mstart[] = {0, 1, 2};
int mrwind[] = {0, 0};
double dmatval[] = {1, 1};
double rhs[] = {1.9};
char qrtype[] = {'L'};
double lbound[] = {0, 0};
double ubound[] = {XPRS_PLUSINFINITY, XPRS_PLUSINFINITY};

double obj[] = {-6, 0};
int mqc1[] = {0, 0, 1};
int mqc2[] = {0, 1, 1};
double dquad[] = {4, -2, 4};

int ngents = 1, nsets = 0;
int mgcols[] = {0};
char qgtype[]={'I'};

double *primal, *dual;

primal = malloc(ncol*sizeof(double));
dual = malloc(nrow*sizeof(double));
...
XPRSloadqglobal(prob, "myprob", ncol, nrow, qrtype, rhs,
                NULL, obj, mstart, NULL, mrwind,
                dmatval, lbound, ubound, nquad, mqc1, mqc2,
                dquad, ngents, nsets, qgtype, mgcols, NULL,
                NULL, NULL, NULL, NULL)
```

Further information

1. The objective function is of the form `c'x+ 0.5 x'Qx` where `Q` is positive semi-definite for minimization problems and negative semi-definite for maximization problems. If this is not the case the optimization algorithms may converge to a local optimum or may not converge at all. Note that only the upper or lower triangular part of the `Q` matrix is specified.
2. The row and column indices follow the usual C convention of going from `0` to `nrow-1` and `0` to `ncol-1` respectively.
3. The double constants `XPRS_PLUSINFINITY` and `XPRS_MINUSINFINITY` are defined in the Optimizer library header file.

Related topics

XPRSaddsetnames, XPRSloadglobal, XPRSloadlp, XPRSloadqp, XPRSreadprob.

XPRSloadqp, XPRSloadqp64

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

```
int XPRS_CC XPRSloadqp(XPRSprob prob, const char *probname, int ncol, int
      nrow, const char qrtype[], const double rhs[], const double range[],
      const double obj[], const int mstart[], const int mnel[], const int
      mrwind[], const double dmatval[], const double dlb[], const double
      dub[], int nqtr, const int mqc1[], const int mqc2[], const double
      dqe[]);

int XPRS_CC XPRSloadqp64(XPRSprob prob, const char *probname, int ncol, int
      nrow, const char qrtype[], const double rhs[], const double range[],
      const double obj[], const XPRSint64 mstart[], const int mnel[], const
      int mrwind[], const double dmatval[], const double dlb[], const
      double dub[], XPRSint64 nqtr, const int mqc1[], const int mqc2[],
      const double dqe[]);
```

Arguments

`prob`	The current problem.
`probname`	A string of up to `MAXPROBNAMELENGTH` characters containing a names for the problem.
`ncol`	Number of structural columns in the matrix.
`nrow`	Number of rows in the matrix (not including the objective row). Objective coefficients must be supplied in the `obj` array, and the objective function should not be included in any of the other arrays.
`qrtype`	Character array of length `nrow` containing the row types: `L` indicates a $\leq$ constraint; `E` indicates an = constraint; `G` indicates a $\geq$ constraint; `R` indicates a range constraint; `N` indicates a nonbinding constraint.
`rhs`	Double array of length `nrow` containing the right hand side coefficients of the rows. The right hand side value for a range row gives the **upper** bound on the row.
`range`	Double array of length `nrow` containing the range values for range rows. Values for all other rows will be ignored. May be `NULL` if there are no ranged constraints. The lower bound on a range row is the right hand side value minus the range value. The sign of the range value is ignored - the absolute value is used in all cases.
`obj`	Double array of length `ncol` containing the objective function coefficients.
`mstart`	Integer array containing the offsets in the `mrwind` and `dmatval` arrays of the start of the elements for each column. This array is of length `ncol` or, if `mnel` is `NULL`, length `ncol+1`. If `mnel` is `NULL` the extra entry of `mstart`, `mstart[ncol]`, contains the position in the `mrwind` and `dmatval` arrays at which an extra column would start, if it were present. In C, this value is also the length of the `mrwind` and `dmatval` arrays.
`mnel`	Integer array of length `ncol` containing the number of nonzero elements in each column. May be `NULL` if all elements are contiguous and `mstart[ncol]` contains the offset where the elements for column `ncol+1` would start. This array is not required if the non-zero coefficients in the `mrwind` and `dmatval` arrays are continuous, and the `mstart` array has `ncol+1` entries as described above. It may be `NULL` if not required.
`mrwind`	Integer array containing the row indices for the nonzero elements in each column. If the indices are input contiguously, with the columns in ascending order, the length of the

	`mrwind` is `mstart[ncol-1]+mnel[ncol-1]` or, if `mnel` is `NULL`, `mstart[ncol]`.
`dmatval`	Double array containing the nonzero element values; length as for `mrwind`.
`dlb`	Double array of length `ncol` containing the lower bounds on the columns. Use `XPRS_MINUSINFINITY` to represent a lower bound of minus infinity.
`dub`	Double array of length `ncol` containing the upper bounds on the columns. Use `XPRS_PLUSINFINITY` to represent an upper bound of plus infinity.
`nqtr`	Number of quadratic terms.
`mqc1`	Integer array of size `nqtr` containing the column index of the first variable in each quadratic term.
`mqc2`	Integer array of size `nqtr` containing the column index of the second variable in each quadratic term.
`dqe`	Double array of size `nqtr` containing the quadratic coefficients.

Related controls

Integer

EXTRACOLS	Number of extra columns to be allowed for.
EXTRAELEMS	Number of extra matrix elements to be allowed for.
EXTRAPRESOLVE	Number of extra elements to allow for in presolve.
EXTRAROWS	Number of extra rows to be allowed for.
KEEPNROWS	Status for nonbinding rows.
SCALING	Type of scaling.

Double

MATRIXTOL	Tolerance on matrix elements.

Example

Minimize $-6x_1 + 2x_1^2 - 2x_1x_2 + 2x_2^2$ subject to $x_1 + x_2 \leq 1.9$:

```
int nrow = 1, ncol = 2, nquad = 3;
int mstart[] = {0, 1, 2};
int mrwind[] = {0, 0};
double dmatval[] = {1, 1};
double rhs[] = {1.9};
char qrtype[] = {'L'};
double lbound[] = {0, 0};
double ubound[] = {XPRS_PLUSINFINITY,XPRS_PLUSINFINITY};

double obj[] = {-6, 0};
int mqc1[] = {0, 0, 1};
int mqc2[] = {0, 1, 1};
double dquad[] = {4, -2, 4};

double *primal, *dual;

primal = malloc(ncol*sizeof(double));
dual = malloc(nrow*sizeof(double));
...
XPRSloadqp(prob, "example", ncol, nrow, qrtype, rhs,
           NULL, obj, mstart, NULL, mrwind, dmatval,
           lbound, ubound, nquad, mqc1, mqc2, dquad)
```

Further information

1. The objective function is of the form `c'x+ 0.5 x'Qx` where `Q` is positive semi-definite for minimization problems and negative semi-definite for maximization problems. If this is not the case the optimization algorithms may converge to a local optimum or may not converge at all. Note that only the upper or lower triangular part of the Q matrix is specified.
2. The row and column indices follow the usual C convention of going from `0` to `nrow-1` and `0` to `ncol-1` respectively.
3. The double constants `XPRS_PLUSINFINITY` and `XPRS_MINUSINFINITY` are defined in the Optimizer library header file.

Related topics

`XPRSloadglobal`, `XPRSloadlp`, `XPRSloadqglobal`, `XPRSreadprob`.

XPRSloadsecurevecs

Purpose

Allows the user to mark rows and columns in order to prevent the presolve removing these rows and columns from the matrix.

Synopsis

```
int XPRS_CC XPRSloadsecurevecs(XPRSprob prob, int nr, int nc, const int
      mrow[], const int mcol[]);
```

Arguments

`prob`	The current problem.
`nr`	Number of rows to be marked.
`nc`	Number of columns to be marked.
`mrow`	Integer array of length `nr` containing the rows to be marked. May be `NULL` if not required.
`mcol`	Integer array of length `nc` containing the columns to be marked. May be `NULL` if not required.

Example

This sets the first six rows and the first four columns to not be removed during presolve.

```
int mrow[] = {0,1,2,3,4,5};
int mcol[] = {0,1,2,3};
...
XPRSreadprob(prob,"myprob","");
XPRSloadsecurevecs(prob,6,4,mrow,mcol);
XPRSmipoptimize(prob,"");
```

Related topics

5.3.

XPRSlpoptimize LPOPTIMIZE

Purpose

This function begins a search for the optimal continuous (LP) solution. The direction of optimization is given by OBJSENSE. The status of the problem when the function completes can be checked using LPSTATUS. Any global entities in the problem will be ignored.

Synopsis

```
int XPRS_CC XPRSlpoptimize(XPRSprob prob, const char *flags);
LPOPTIMIZE [-flags]
```

Arguments

`prob`	The current problem.
`flags`	Flags to pass to `XPRSlpoptimize` (`LPOPTIMIZE`). The default is `""` or `NULL`, in which case the algorithm used is determined by the DEFAULTALG control. If the argument includes: `b` the model will be solved using the Newton barrier method; `p` the model will be solved using the primal simplex algorithm; `d` the model will be solved using the dual simplex algorithm; `n` (lower case `N`), the network part of the model will be identified and solved using the network simplex algorithm;

Further information

1. The algorithm used to optimize is determined by the DEFAULTALG control if no flags are provided. By default, the dual simplex is used for linear problems and the barrier is used for non-linear problems.
2. The `d` and `p` flags can be used with the `n` flag to complete the solution of the model with either the dual or primal algorithms once the network algorithm has solved the network part of the model.
3. The `b` flag cannot be used with the `n` flag.

Related topics

XPRSmipoptimize (MIPOPTIMIZE), 4.

XPRSmaxim, XPRSminim — MAXIM, MINIM

Purpose

Begins a search for the optimal LP solution. These functions are deprecated and might be removed in a future release. `XPRSlpoptimize` or `XPRSmipoptimize` should be used instead.

Synopsis

```
int XPRS_CC XPRSmaxim(XPRSprob prob, const char *flags);
int XPRS_CC XPRSminim(XPRSprob prob, const char *flags);
MAXIM [-flags]
MINIM [-flags]
```

Arguments

`prob`	The current problem.
`flags`	Flags to pass to `XPRSmaxim` (`MAXIM`) or `XPRSminim` (`MINIM`). The default is `""` or `NULL`, in which case the algorithm used is determined by the `DEFAULTALG` control. If the argument includes: `b` the model will be solved using the Newton barrier method; `p` the model will be solved using the primal simplex algorithm; `d` the model will be solved using the dual simplex algorithm; `l` (lower case `L`), the model will be solved as a linear model ignoring the discreteness of global variables; `n` (lower case `N`), the network part of the model will be identified and solved using the network simplex algorithm; `g` the global model will be solved, calling `XPRSglobal` (`GLOBAL`). Certain combinations of options may be used where this makes sense so, for example, `pg` will solve the LP with the primal algorithm and then go on to perform the global search.

Related controls

Integer

`AUTOPERTURB`	Whether automatic perturbation is performed.
`BARITERLIMIT`	Maximum number of Newton Barrier iterations.
`BARORDER`	Ordering algorithm for the Cholesky factorization.
`BARORDERTHREADS`	Maximum number of theads for the ordering algorithm.
`BAROUTPUT`	Newton barrier: level of solution output.
`BARTHREADS`	Max number of threads to run.
`BIGMMETHOD`	Specifies "Big M" method, or phaseI/phaseII.
`CACHESIZE`	Cache size in Kbytes for the Newton barrier.
`CPUTIME`	`1` for CPU time; `0` for elapsed time.
`CRASH`	Type of crash.
`CROSSOVER`	Newton barrier crossover control.
`DEFAULTALG`	Algorithm to use with the tree search.
`DENSECOLLIMIT`	Columns with this many elements are considered dense.
`DUALGRADIENT`	Pricing method for the dual algorithm.
`INVERTFREQ`	Invert frequency.
`INVERTMIN`	Minimum number of iterations between inverts.
`KEEPBASIS`	Whether to use previously loaded basis.
`LPITERLIMIT`	Iteration limit for the simplex algorithm.
`LPLOG`	Frequency and type of simplex algorithm log.
`MAXTIME`	Maximum time allowed.

PRESOLVE	Degree of presolving to perform.
PRESOLVEOPS	Specifies the operations performed during presolve.
PRICINGALG	Type of pricing to be used.
REFACTOR	Indicates whether to re-factorize the optimal basis.
TRACE	Control of the infeasibility diagnosis during presolve.

Double

BARDUALSTOP	Newton barrier tolerance for dual infeasibilities.
BARGAPSTOP	Newton barrier tolerance for relative duality gap.
BARPRIMALSTOP	Newton barrier tolerance for primal infeasibilities.
BARSTEPSTOP	Newton barrier minimal step size.
BIGM	Infeasibility penalty.
CHOLESKYTOL	Tolerance in the Cholesky decomposition.
ELIMTOL	Markowitz tolerance for elimination phase of presolve.
ETATOL	Tolerance on eta elements.
FEASTOL	Tolerance on RHS.
MARKOWITZTOL	Markowitz tolerance for the factorization.
MIPABSCUTOFF	Cutoff set after an LP Optimizer command. (Dual only)
OPTIMALITYTOL	Reduced cost tolerance.
PENALTY	Maximum absolute penalty variable coefficient.
PERTURB	Perturbation value.
PIVOTTOL	Pivot tolerance.
PPFACTOR	Partial pricing candidate list sizing parameter.
RELPIVOTTOL	Relative pivot tolerance.

Example 1 (Library)

```
XPRSmaxim(prob,"b");
```

This maximizes the current problem using the Newton barrier method.

Example 2 (Console)

```
MINIM -g
```

This minimizes the current problem and commences the global search.

Further information

1. The algorithm used to optimize is determined by the DEFAULTALG control. By default, the dual simplex is used for LP and MIP problems and the barrier is used for QP problems.
2. The `d` and `p` flags can be used with the `n` flag to complete the solution of the model with either the dual or primal algorithms once the network algorithm has solved the network part of the model.
3. The `b` flag cannot be used with the `n` flag.
4. The dual simplex algorithm is a two phase algorithm which can remove dual infeasibilities.
5. (*Console*) If the user prematurely terminates the solution process by typing CTRL-C, the iterative procedure will terminate at the first "safe" point.

Related topics

XPRSglobal (GLOBAL), XPRSreadbasis (READBASIS), XPRSgoal (GOAL), 4, A.8.

XPRSmipoptimize MIPOPTIMIZE

Purpose

This function begins a global search for the optimal MIP solution. The direction of optimization is given by `OBJSENSE`. The status of the problem when the function completes can be checked using `MIPSTATUS`.

Synopsis

```
int XPRS_CC XPRSmipoptimize(XPRSprob prob, const char *flags);
MIPOPTIMIZE [-flags]
```

Arguments

`prob`	The current problem.
`flags`	Flags to pass to `XPRSmipoptimize` (`MIPOPTIMIZE`), which specifies how to solve the initial continuous problem where the global entities are relaxed. If the argument includes: `b` the initial continuous relaxation will be solved using the Newton barrier method; `p` the initial continuous relaxation will be solved using the primal simplex algorithm; `d` the initial continuous relaxation will be solved using the dual simplex algorithm; `n` the network part of the initial continuous relaxation will be identified and solved using the network simplex algorithm; `l` stop after having solved the initial continous relaxation.

Further information

1. If the `l` flag is used, the Optimizer will stop immediately after solving the initial continuous relaxation. The status of the continuous solve can be checked with `LPSTATUS` and standard LP results are available, such as the objective value (`LPOBJVAL`) and solution (use `XPRSgetlpsol`), depending on `LPSTATUS`.
2. It is possible for the Optimizer to find integer solutions before solving the initial continuous relaxation, either through heuristics or by having the user load an initial integer solution. This can potentially result in the global search finishing before solving the continuous relaxation to optimality.
3. If the function returns without having completed the search for an optimal solution, the search can be resumed from where it stopped by calling `XPRSmipoptimize` again.
4. The algorithm used to reoptimize the continuous relaxations during the global search is given by `DEFAULTALG`. The default is to use the dual simplex algorithm.

Related topics

`XPRSlpoptimize` (`LPOPTIMIZE`), 4.

XPRSobjsa

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

```
int XPRS_CC XPRSobjsa(XPRSprob prob, int nels, const int mindex[], double
        lower[], double upper[]);
```

Arguments

`prob`	The current problem.
`nels`	Number of objective function coefficients whose sensitivity is sought.
`mindex`	Integer array of length `nels` containing the indices of the columns whose objective function coefficients sensitivity ranges are required.
`lower`	Double array of length `nels` where the objective function lower range values are to be returned.
`upper`	Double array of length `nels` where the objective function upper range values are to be returned.

Example

Here we obtain the objective function ranges for the three columns: `2`, `6` and `8`:

```
mindex[0] = 2; mindex[1] = 8; mindex[2] = 6;
XPRSobjsa(prob,3,mindex,lower,upper);
```

After which lower and upper contain:

```
lower[0] = 5.0; upper[0] = 7.0;
lower[1] = 3.8; upper[1] = 5.2;
lower[2] = 5.7; upper[2] = 1e+20;
```

Meaning that the current basis remains optimal when $5.0 \leq C_2 \leq 7.0$, $3.8 \leq C_8 \leq 5.2$ and $5.7 \leq C_6$, C_i being the objective coefficient of column `i`.

Further information

`XPRSobjsa` can only be called when an optimal solution to the current LP has been found. It cannot be used when the problem is MIP presolved.

Related topics

`XPRSrhssa`.

XPRSpivot

Purpose

Performs a simplex pivot by bringing variable `in` into the basis and removing `out`.

Synopsis

```
int XPRS_CC XPRSpivot(XPRSprob prob, int in, int out);
```

Arguments

`prob`	The current problem.
`in`	Index of row or column to enter basis.
`out`	Index of row or column to leave basis.

Error values

425	`in` is invalid (out of range or already basic).
426	`out` is invalid (out of range or not eligible, e.g. nonbasic, zero pivot, etc.).

Related controls

Double

PIVOTTOL	Pivot tolerance.
RELPIVOTTOL	Relative pivot tolerance.

Example

The following brings the 7th variable into the basis and removes the 5th:

```
XPRSpivot(prob,6,4)
```

Further information

Row indices are in the range `0` to `ROWS-1`, whilst columns are in the range `ROWS+SPAREROWS` to `ROWS+SPAREROWS+COLS-1`.

Related topics

`XPRSgetpivotorder`, `XPRSgetpivots`.

XPRSpostsolve POSTSOLVE

Purpose

Postsolve the current matrix when it is in a presolved state.

Synopsis

```
int XPRS_CC XPRSpostsolve(XPRSprob prob);
POSTSOLVE
```

Argument

`prob` The current problem.

Further information

A problem is left in a presolved state whenever a LP or MIP optimization does not complete. In these cases `XPRSpostsolve` (`POSTSOLVE`) can be called to get the problem back into its original state.

Related topics

`XPRSlpoptimize`, `XPRSmipoptimize`

XPRSpresolverow

Purpose

Presolves a row formulated in terms of the original variables such that it can be added to a presolved matrix.

Synopsis

```
int XPRS_CC XPRSpresolverow(XPRSprob prob, char qrtype, int nzo, const int
        mcolso[], const double dvalo[], double drhso, int maxcoeffs, int *
        nzp, int mcolsp[], double dvalp[], double * drhsp, int * status);
```

Arguments

`prob`	The current problem.
`qrtype`	The type of the row: `L` indicates a $\leq$ row; `G` indicates a $\geq$ row.
`nzo`	Number of elements in the `mcolso` and `dvalo` arrays.
`mcolso`	Integer array of length `nzo` containing the column indices of the row to presolve.
`dvalo`	Double array of length `nzo` containing the non-zero coefficients of the row to presolve.
`drhso`	The right-hand side constant of the row to presolve.
`maxcoeffs`	Maximum number of elements to return in the `mcolsp` and `dvalp` arrays.
`nzp`	Pointer to the integer where the number of elements in the `mcolsp` and `dvalp` arrays will be returned.
`mcolsp`	Integer array which will be filled with the column indices of the presolved row. It must be allocated to hold at least `COLS` elements.
`dvalp`	Double array which will be filled with the coefficients of the presolved row. It must be allocated to hold at least `COLS` elements.
`drhsp`	Pointer to the double where the presolved right-hand side will be returned.
`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.

Related controls

Integer

PRESOLVE	Turns presolve on or off.
PRESOLVEOPS	Selects the presolve operations.

Example

Suppose we want to add the row $2x_1 + x_2 \leq 1$ to our presolved matrix. This could be done in the following way:

```
int mindo[] = { 1, 2 };
int dvalo[] = { 2.0, 1.0 };
char qrtype = 'L';
double drhso = 1.0;
int nzp, status, mtype, mstart[2], *mindp;
double drhsp, *dvalp;
...
XPRSgetintattrib(prob, XPRS_COLS, &ncols);
mindp = (int*) malloc(ncols*sizeof(int));
```

```
dvalp = (double*) malloc(ncols*sizeof(double));
XPRSpresolverow(prob, qrtype, 2, mindo, dvalo, drhso, ncols,
                &nzp, mindp, dvalp, &drhsp, &status);
if (status >= 0) {
  mtype = 0;
  mstart[0] = 0; mstart[1] = nzp;
  XPRSaddcuts(prob, 1, &mtype, &qrtype, &drhsp, mstart, mindp,
              dvalp);
}
```

Further information

There are certain presolve operations that can prevent a row from being presolved exactly. If the row contains a coefficient for a column that was eliminated due to duplicate column reductions or singleton column reductions, the row might have to be relaxed to remain valid for the presolved problem. The relaxation will be done automatically by the `XPRSpresolverow` function, but a return status of `+1` will be returned. If it is not possible to relax the row, a status of `-2` will be returned instead. Likewise, it is possible that certain dual reductions prevents the row from being presolved. In such a case a status of `-3` will be returned instead.

If `XPRSpresolverow` will be used for presolving e.g. branching bounds or constraints, then dual reductions and duplicate column reductions should be disabled, by clearing the corresponding bits of `PRESOLVEOPS`. By clearing these bits, the default value for `PRESOLVEOPS` changes to 471.

If the user knows in advance which columns will have non-zero coefficients in rows that will be presolved, it is possible to protect these individual columns through the `XPRSloadsecurevecs` function. This way the Optimizer is left free to apply all possible reductions to the remaining columns.

Related topics

`XPRSaddcuts`, `XPRSloadsecurevecs`, `XPRSsetbranchcuts`, `XPRSstorecuts`.

PRINTRANGE

Purpose

Writes the ranging information to the screen. The binary range file (`.rng`) must already exist, created by `XPRSrange` (`RANGE`).

Synopsis

```
PRINTRANGE
```

Related controls

Integer

`MAXPAGELINES`	Number of lines between page breaks.

Double

`OUTPUTTOL`	Tolerance on print values.

Further information

See `WRITEPRTRANGE` for more information.

Related topics

`XPRSgetcolrange`, `XPRSgetrowrange`, `XPRSrange` (`RANGE`), `XPRSwriteprtsol`, `XPRSwriterange`, A.6.

PRINTSOL

Purpose

Writes the current solution to the screen.

Synopsis

`PRINTSOL`

Related controls

Integer

`MAXPAGELINES`	Number of lines between page breaks.

Double

`OUTPUTTOL`	Tolerance on print values.

Further information

See `WRITEPRTSOL` for more information.

Related topics

`XPRSgetlpsol`, `XPRSgetmipsol`, `XPRSwriteprtsol`.

QUIT

Purpose

Terminates the Console Optimizer, returning a zero exit code to the operating system. Alias for EXIT.

Synopsis

```
QUIT
```

Example

The command is called simply as:

```
QUIT
```

Further information

1. Fatal error conditions return nonzero exit values which may be of use to the host operating system. These are described in 11.
2. If you wish to return an exit code reflecting the final solution status, then use the `STOP` command instead.

Related topics

`STOP`, `XPRSsave` (`SAVE`).

XPRSrange RANGE

Purpose

Calculates the ranging information for a problem and saves it to the binary ranging file *problem_name*`.rng`.

Synopsis

```
int XPRS_CC XPRSrange(XPRSprob prob);
RANGE
```

Argument

`prob` The current problem.

Example 1 (Library)

This example computes the ranging information following optimization and outputs the solution to a file `leonor.rrt`:

```
XPRSreadprob(prob,"leonor","");
XPRSlpoptimize(prob,"");
XPRSrange(prob);
XPRSwriteprtrange(prob);
```

Example 2 (Console)

The following example is equivalent for the console, except the output is sent to the screen instead of a file:

```
READPROB leonor
LPOPTIMIZE
RANGE
PRINTRANGE
```

Further information

1. A basic optimal solution to the problem must be available, i.e. `XPRSlpoptimize` (`LPOPTIMIZE`) must have been called (with crossover used if the Newton Barrier algorithm is being used) and an optimal solution found.

2. The information calculated by `XPRSrange` (`RANGE`) enables the user to do sophisticated postoptimal analysis of the problem. In particular, the user may find the ranges over which the right hand sides can vary without the optimal basis changing, the ranges over which the shadow prices hold, and the activities which limit these changes. See functions `XPRSgetcolrange`, `XPRSgetrowrange`, `XPRSwriteprtrange` (`WRITEPRTRANGE`) and/or `XPRSwriterange` (`WRITERANGE`) to obtain the values calculated

3. It is not impossible to range on a MIP problem. The global entities should be fixed using `XPRSfixglobals` (`FIXGLOBALS`) first and the remaining LP resolved - see `XPRSfixglobals` (`FIXGLOBALS`).

Related topics

`XPRSgetcolrange`, `XPRSgetrowrange`, `XPRSwriteprtrange` (`WRITEPRTRANGE`), `XPRSwriterange` (`WRITERANGE`).

XPRSreadbasis READBASIS

Purpose

Instructs the Optimizer to read in a previously saved basis from a file.

Synopsis

```
int XPRS_CC XPRSreadbasis(XPRSprob prob, const char *filename, const char
       *flags);
READBASIS [-flags] [filename]
```

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. If omitted, the default *problem_name* is used with a `.bss` extension.
`flags`	Flags to pass to `XPRSreadbasis` (`READBASIS`): `i` output the internal presolved basis. `t` input a compact advanced form of the basis.

Example 1 (Library)

If an advanced basis is available for the current problem the Optimizer input might be:

```
XPRSreadprob(prob,"filename","");
XPRSreadbasis(prob,"","");
XPRSmipoptimize(prob,"");
```

This reads in a matrix file, inputs an advanced starting basis and maximizes the MIP.

Example 2 (Console)

An equivalent set of commands for the Console user may look like:

```
READPROB
READBASIS
MIPOPTIMIZE
```

Further information

1. The only check done when reading compact basis is that the number of rows and columns in the basis agrees with the current number of rows and columns.
2. `XPRSreadbasis` (`READBASIS`) will read the basis for the original problem even if the matrix has been presolved. The Optimizer will read the basis, checking that it is valid, and will display error messages if it detects inconsistencies.

Related topics

`XPRSloadbasis`, `XPRSwritebasis` (`WRITEBASIS`).

XPRSreadbinsol READBINSOL

Purpose

Reads a solution from a binary solution file.

Synopsis

```
int XPRS_CC XPRSreadbinsol(XPRSprob prob, const char *filename, const char
       *flags);
READBINSOL [-flags] [filename]
```

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. If omitted, the default *problem_name* is used with a `.sol` extension.
`flags`	Flags to pass to `XPRSreadbinsol` (`READBINSOL`): `m` load the solution as a solution for the MIP.

Example 1 (Library)

A previously saved solution can be loaded into memory and a print file created from it with the following commands:

```
XPRSreadprob(prob, "myprob", "");
XPRSreadbinsol(prob, "", "");
XPRSwriteprtsol(prob, "", "");
```

Example 2 (Console)

An equivalent set of commands to the above for console users would be:

```
READPROB
READBINSOL
WRITEPRTSOL
```

Related topics

`XPRSgetlpsol`, `XPRSgetmipsol`, `XPRSwritebinsol` (`WRITEBINSOL`), `XPRSwritesol` (`WRITESOL`), `XPRSwriteprtsol` (`WRITEPRTSOL`).

XPRSreaddirs READDIRS

Purpose

Reads a directives file to help direct the global search.

Synopsis

```
int XPRS_CC XPRSreaddirs(XPRSprob prob, const char *filename);
READDIRS [filename]
```

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. If omitted (or NULL), the default *problem_name* is used with a .dir extension.

Related controls

Double

PSEUDOCOST Default pseudo cost in node degradation estimation.

Example 1 (Library)

The following example reads in directives from the file sue.dir for use with the problem, steve:

```
XPRSreadprob(prob,"steve","");
XPRSreaddirs(prob,"sue");
XPRSmipoptimize(prob,"");
```

Example 2 (Console)

```
READPROB
READDIRS
MIPOPTIMIZE
```

This is the most usual form at the console. It will attempt to read in a directives file with the current problem name and an extension of .dir.

Further information

1. Directives cannot be read in after a model has been presolved, so unless presolve has been disabled by setting `PRESOLVE` to `0`, this command must be issued before `XPRSmipoptimize` (`MIPOPTIMIZE`).

2. Directives can be given relating to priorities, forced branching directions, pseudo costs and model cuts. There is a priority value associated with each global entity. The *lower* the number, the *more* likely the entity is to be selected for branching; the *higher*, the *less* likely. By default, all global entities have a priority value of `500` which can be altered with a priority entry in the directives file. In general, it is advantageous for the entity's priority to reflect its relative importance in the model. Priority entries with values in excess of `1000` are illegal and are ignored. A full description of the directives file format may be found in A.6.

3. By default, `XPRSmipoptimize` (`MIPOPTIMIZE`) will explore the branch expected to yield the best integer solution from each node, irrespective of whether this forces the global entity up or down. This can be overridden with an `UP` or `DN` entry in the directives file, which forces `XPRSmipoptimize` (`MIPOPTIMIZE`) to branch up first or down first on the specified entity.

4. Pseudo-costs are estimates of the unit cost of forcing an entity up or down. By default `XPRSmipoptimize` (`MIPOPTIMIZE`) uses dual information to calculate estimates of the unit up and down costs and these are added to the default pseudo costs which are set to the `PSEUDOCOST` control. The default pseudo costs can be overridden by a `PU` or `PD` entry in the directives file.

5. If model cuts are used, then the specified constraints are removed from the matrix and added to the Optimizer cut pool, and only put back in the matrix when they are violated by an LP solution at one of the nodes in the global search.

6. If creating a directives file by hand, wild cards can be used to specify several vectors at once, for example `PR x1* 2` will give all global entities whose names start with `x1` a priority of `2`.

Related topics

`XPRSloaddirs`, A.6.

XPRSreadprob READPROB

Purpose

Reads an (X)MPS or LP format matrix from file.

Synopsis

```
int XPRS_CC XPRSreadprob(XPRSprob prob, const char *probname, const char
        *flags);
READPROB [-flags] [probname]
```

Arguments

`prob`	The current problem.
`probname`	The path and file name from which the problem is to be read. Limited to `MAXPROBNAMELENGTH` characters. If omitted (console users only), the default *problem_name* is used with various extensions - see below.
`flags`	Flags to be passed: `l` only `probname.lp` is searched for; `z` read input file in gzip format from a `.gz` file [Console only]

Related controls

Integer

`EXTRACOLS`	Number of extra columns to be allowed for.
`EXTRAELEMS`	Number of extra matrix elements to be allowed for.
`EXTRAMIPENTS`	Number of extra global entities to be allowed for.
`EXTRAPRESOLVE`	Number of extra elements to allow for in presolve.
`EXTRAROWS`	Number of extra rows to be allowed for.
`KEEPNROWS`	Status for nonbinding rows.
`MPSECHO`	Whether MPS comments are to be echoed.
`MPSFORMAT`	Specifies format of MPS files.
`SCALING`	Type of scaling.

Double

`MATRIXTOL`	Tolerance on matrix elements.
`SOSREFTOL`	Minimum gap between reference row entries.

String

`MPSBOUNDNAME`	The active bound name.
`MPSOBJNAME`	Name of objective function row.
`MPSRANGENAME`	Name of range.
`MPSRHSNAME`	Name of right hand side.

Example 1 (Library)

```
XPRSreadprob(prob,"myprob","");
```

This instructs the Optimizer to read an MPS format matrix from the first file found out of `myprob.mat`, `myprob.mps` or (in LP format) `myprob.lp`.

Example 2 (Console)

```
READPROB -l
```

This instructs the Optimizer to read an LP format matrix from the file *problem_name* `.lp`.

Further information

1. If no flags are given, file types are searched for in the order: `.mat`, `.mps`, `.lp`. Matrix files are assumed to be in XMPS or MPS format unless their file extension is `.lp` in which case they must be LP files.

2. If `probname` has been specified, the problem name is changed to `probname`, ignoring any extension.

3. `XPRSreadprob` (`READPROB`) will take as the objective function the first `N` type row in the matrix, unless the string parameter `MPSOBJNAME` has been set, in which case the objective row sought will be the one named by `MPSOBJNAME`. Similarly, if non-blank, the string parameters `MPSRHSNAME`, `MPSBOUNDNAME` and `MPSRANGENAME` specify the right hand side, bound and range sets to be taken. For example:
   ```
   MPSOBJNAME="Cost"
   MPSRHSNAME="RHS 1"
   READPROB
   ```
   The treatment of `N` type rows other than the objective function depends on the `KEEPNROWS` control. If `KEEPNROWS` is `1` the rows and their elements are kept in memory; if it is `0` the rows are retained, but their elements are removed; and if it is `-1` the rows are deleted entirely. The performance impact of retaining such `N` type rows will be small unless the presolve has been disabled by setting `PRESOLVE` to `0` prior to optimization.

4. The Optimizer checks that the matrix file is in a legal format and displays error messages if it detects errors. When the Optimizer has read and verified the problem, it will display summary problem statistics.

5. By default, the `MPSFORMAT` control is set to `-1` and `XPRSreadprob` (`READPROB`) determines automatically whether the MPS files are in free or fixed format. If `MPSFORMAT` is set to `0`, fixed format is assumed and if it is set to `1`, free format is assumed. Fields in free format MPS files are delimited by one or more blank characters. The keywords NAME, ROWS, COLUMNS, QUADOBJ / QMATRIX, QCMATRIX, DELAYEDROWS, MODELCUTS, SETS, RHS, RANGES, BOUNDS and ENDATAmust start in column one and no vector name may contain blanks. If a special ordered set is specified with a reference row, its name may not be the same as that of a column. Note that numeric values which contain embedded spaces (for example after unary minus sign) will not be read correctly unless `MPSFORMAT` is set to `0`.

6. If the problem is not to be scaled automatically, set the parameter `SCALING` to `0` before issuing the `XPRSreadprob` (`READPROB`) command.

Related topics

`XPRSloadglobal`, `XPRSloadlp`, `XPRSloadqglobal`, `XPRSloadqp`.

XPRSreadslxsol READSLXSOL

Purpose

Reads an ASCII solution file (`.slx`) created by the `XPRSwriteslxsol` function.

Synopsis

```
int XPRS_CC XPRSreadslxsol(XPRSprob prob, const char *filename, const char
      *flags);
READSLXSOL -[flags] [filename]
```

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. If omitted, the default *problem_name* is used with a `.slx` extension.
`flags`	Flags to pass to `XPRSwriteslxsol` (`WRITESLXSOL`): `l` read the solution as an LP solution in case of a MIP problem; `m` read the solution as a solution for the MIP problem; `a` reads multiple MIP solutions from the `.slx` file and adds them to the MIP problem;

Example 1 (Library)

```
XPRSreadslxsol(prob,"lpsolution","");
```

This loads the solution to the MIP problem if the problem contains global entities, or otherwise loads it as an LP (barrier in case of quadratic problems) solution into the problem.

Example 2 (Console)

```
READSLXSOL lpsolution
```

Further information

1. When `XPRSreadslxsol` is called before a MIP solve, the loaded solutions will not be checked before calling `XPRSmipoptimize`. By default, only the last MIP solution read from the `.slx` file will be stored. Use the `a` flag to store all MIP solutions read from the file.
2. When using the `a` flag, read solutions will be queued similarly to the user of the `XPRSaddmipsol` function. Each name string given by the `NAME` field in the `.slx` file will be associated with the corresponding solution. Any registered `usersolnotify` callback will be fired when the solution has been checked, and will include the read name string as one of its arguments.
3. Refer to the Appendix on Log and File Formats for a description of the ASCII Solution (.slx) File format A.4.4.

Related topics

`XPRSreadbinsol` (`READBINSOL`), `XPRSwriteslxsol` (`WRITESLXSOL`), `XPRSwritebinsol` `WRITEBINSOL`, `XPRSreadbinsol` (`READBINSOL`), `XPRSaddmipsol`, `XPRSaddcbusersolnotify`.

XPRSrefinemipsol REFINEMIPSOL

Purpose

Executes the MIP solution refiner.

Synopsis

```
int XPRS_CC XPRSrefinemipsol(XPRSprob prob, int options, const char* flags,
      const double solution[], double refined_solution[], int*
      refinestatus);
REFINEMIPSOL
```

Arguments

`prob` The current problem.

`options` Refinement options:
- `0` Reducing MIP fractionality is priority.
- `1` Reducing LP infeasibility is priority

`flags` Flags passed to any optimization calls during refinement.

`solution` The MIP solution to refine. Must be a valid MIP solution.

`refined_solution` The refined MIP solution in case of success

`refinestatus` Refinement results:
- `0` An error has occurred
- `1` The solution has been refined
- `2` Current solution meets target criteria
- `3` Solution cannot be refined

Further information

The function provides a mechanism to refine the MIP solution by attempting to round any fractional global entity and by attempting to reduce LP infeasibility.

Related topics

REFINEOPS.

XPRSremovecbbariteration

Purpose

Removes a barrier iteration callback function previously added by `XPRSaddcbbariteration`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbbariteration(XPRSprob prob, void (XPRS_CC
      *f_bariteration)(XPRSprob prob, void* vContext, int* barrier_action),
      void* object);
```

Arguments

`prob` The current problem.

`f_bariteration` The callback function to remove. If NULL then all bariteration callback functions added with the given user-defined object value will be removed.

`object` The object value that the callback was added with. If NULL, then the object value will not be checked and all barrier iteration callbacks with the function pointer `f_bariteration` will be removed.

Related topics

`XPRSaddcbbariteration`.

XPRSremovecbcomputerestart

Purpose

Removes a computerestart callback function previously added by `XPRSaddcbcomputerestart`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbpcomputerestart(XPRSprob prob, void (XPRS_CC
      *f_computerestart)(XPRSprob prob, void* vContext), void* object);
```

Arguments

`prob`	The current problem.
`f_presolve`	The callback function to remove. If NULL then all computerestart callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all computerestart callbacks with the function pointer `f_computerestart` will be removed.

Related topics

`XPRSaddcbcomputerestart`.

XPRSremovecbpresolve

Purpose

Removes a presolve callback function previously added by `XPRSaddcbpresolve`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbpresolve(XPRSprob prob, void (XPRS_CC
      *f_presolve)(XPRSprob prob, void* vContext), void* object);
```

Arguments

`prob`	The current problem.
`f_presolve`	The callback function to remove. If NULL then all presolve callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all barrier iteration callbacks with the function pointer `f_presolve` will be removed.

Related topics

`XPRSaddcbpresolve`.

XPRSremovecbbarlog

Purpose

Removes a Newton barrier log callback function previously added by `XPRSaddcbbarlog`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbbarlog(XPRSprob prob, int (XPRS_CC
       *f_barlog)(XPRSprob prob, void* object), void* object);
```

Arguments

`prob`	The current problem.
`f_barlog`	The callback function to remove. If NULL then all barrier log callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all barrier log callbacks with the function pointer `f_barlog` will be removed.

Related topics

`XPRSaddcbbarlog`.

XPRSremovecbchgbranch

Purpose

Removes a variable branching callback function previously added by `XPRSaddcbchgbranch`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbchgbranch(XPRSprob prob, void (XPRS_CC
      *f_chgbranch)(XPRSprob prob, void* vContext, int* entity, int* up,
      double* estdeg), void* object);
```

Arguments

`prob`	The current problem.
`f_chgbranch`	The callback function to remove. If NULL then all variable branching callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all variable branching callbacks with the function pointer `f_chgbranch` will be removed.

Related topics

`XPRSaddcbchgbranch`.

XPRSremovecbchgbranchobject

Purpose

Removes a callback function previously added by `XPRSaddcbchgbranchobject`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbchgbranchobject(XPRSprob prob, void (XPRS_CC
      *f_chgbranchobject)(XPRSprob my_prob, void* my_object,
      XPRSbranchobject obranch, XPRSbranchobject* p_newobject), void*
      object);
```

Arguments

`prob`	The current problem.
`f_chgbranchobject`	The callback function to remove. If NULL then all branch object callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all branch object callbacks with the function pointer `f_chgbranchobject` will be removed.

Related topics

`XPRSaddcbchgbranchobject`

XPRSremovecbchecktime

Purpose

Removes a callback function previously added by `XPRSaddcbchecktime`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbchecktime(XPRSprob prob, int (XPRS_CC
      *f_checktime)(XPRSprob prob, void* object), void* object);
```

Arguments

`prob`	The current problem.
`f_checktime`	The callback function to remove. If NULL then all checktime callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all checktime callbacks with the function pointer `f_checktime` will be removed.

Related topics

`XPRSaddcbchecktime`

XPRSremovecbchgnode

Purpose

This function is deprecated and may be removed in future releases. Please use branching objects instead.

Removes a node selection callback function previously added by `XPRSaddcbchgnode`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbchgnode(XPRSprob prob, void (XPRS_CC
      *f_chgnode)(XPRSprob prob, void* object, int* nodnum), void* object);
```

Arguments

`prob`	The current problem.
`f_chgnode`	The callback function to remove. If NULL then all node selection callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all node selection callbacks with the function pointer `f_chgnode` will be removed.

Related topics

`XPRSaddcbchgnode`

XPRSremovecbcutlog

Purpose

Removes a cut log callback function previously added by `XPRSaddcbcutlog`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbcutlog(XPRSprob prob, int (XPRS_CC
       *f_cutlog)(XPRSprob prob, void* object), void* object);
```

Arguments

`prob`	The current problem.
`f_cutlog`	The callback function to remove. If NULL then all cut log callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all cut log callbacks with the function pointer `f_cutlog` will be removed.

Related topics

`XPRSaddcbcutlog`

XPRSremovecbcutmgr

Purpose

Removes a cut manager callback function previously added by `XPRSaddcbcutmgr`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbcutmgr(XPRSprob prob, int (XPRS_CC
       *f_cutmgr)(XPRSprob prob, void* object), void* object);
```

Arguments

`prob`	The current problem.
`f_cutmgr`	The callback function to remove. If NULL then all cut manager callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all cut manager callbacks with the function pointer `f_cutmgr` will be removed.

Related topics

`XPRSaddcbcutmgr`

XPRSremovecbdestroymt

Purpose

Removes a slave thread destruction callback function previously added by `XPRSaddcbdestroymt`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbdestroymt(XPRSprob prob, void (XPRS_CC
      *f_destroymt)(XPRSprob prob, void* vContext), void* object);
```

Arguments

`prob`	The current problem.
`f_destroymt`	The callback function to remove. If NULL then all thread destruction callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all thread destruction callbacks with the function pointer `f_destroymt` will be removed.

Related topics

`XPRSaddcbdestroymt`

XPRSremovecbestimate

Purpose

Removes an estimate callback function previously added by `XPRSaddcbestimate`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbestimate(XPRSprob prob, int (XPRS_CC
      *f_estimate)(XPRSprob prob, void* vContext, int* iglsel, int* iprio,
      double* degbest, double* degworst, double* curval, int* ifupx, int*
      nglinf, double* degsum, int* nbr), void* object);
```

Arguments

`prob`	The current problem.
`f_estimate`	The callback function to remove. If NULL then all integer solution callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all estimate callbacks with the function pointer `f_estimate` will be removed.

Related topics

`XPRSaddcbestimate`

XPRSremovecbgapnotify

Purpose

Removes a callback function previously added by `XPRSaddcbgapnotify`. The specified callback function will no longer be removed after it has been returned.

Synopsis

```
int XPRS_CC XPRSremovecbgapnotify(XPRSprob prob, void (XPRS_CC
      *f_gapnotify)(XPRSprob prob, void* vContext, double*
      newRelGapNotifyTarget, double* newAbsGapNotifyTarget, double*
      newAbsGapNotifyObjTarget, double* newAbsGapNotifyBoundTarget), void*
      p);
```

Arguments

`prob`	The current problem.
`f_gapnotify`	The callback function to remove. If `NULL` then all `gapnotify` callback functions added with the given user-defined pointer value will be removed.
`p`	The user-defined pointer value that the callback was added with. If `NULL` then the pointer value will not be checked and all the `gapnotify` callbacks with the function pointer `f_gapnotify` will be removed.

Related topics

`XPRSaddcbgapnotify`.

XPRSremovecbgloballog

Purpose

Removes a global log callback function previously added by `XPRSaddcbgloballog`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbgloballog(XPRSprob prob, int (XPRS_CC
      *f_globallog)(XPRSprob prob, void* vContext), void* object);
```

Arguments

`prob`	The current problem.
`f_globallog`	The callback function to remove. If NULL then all global log callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all global log callbacks with the function pointer `f_globallog` will be removed.

Example

The following code sets and removes a callback function:

```
XPRSsetintcontrol(prob, XPRS_MIPLOG, 3);
XPRSaddcbgloballog(prob, globalLog, NULL, 0);
XPRSmipoptimize(prob,"");
XPRSremovecbgloballog(prob,globalLog,NULL);
}
```

Related topics

`XPRSaddcbgloballog`

XPRSremovecbinfnode

Purpose

Removes a user infeasible node callback function previously added by `XPRSaddcbinfnode`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbinfnode(XPRSprob prob, void (XPRS_CC
      *f_infnode)(XPRSprob prob, void* object), void* object);
```

Arguments

`prob`	The current problem.
`f_infnode`	The callback function to remove. If NULL then all user infeasible node callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all user infeasible node callbacks with the function pointer `f_infnode` will be removed.

Related topics

`XPRSaddcbinfnode`

XPRSremovecbintsol

Purpose

Removes an integer solution callback function previously added by `XPRSaddcbintsol`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbintsol(XPRSprob prob, void (XPRS_CC
      *f_intsol)(XPRSprob prob, void* my_object), void* object);
```

Arguments

`prob`	The current problem.
`f_intsol`	The callback function to remove. If NULL then all integer solution callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all integer solution callbacks with the function pointer `f_intsol` will be removed.

Related topics

`XPRSaddcbintsol`

XPRSremovecblplog

Purpose

Removes a simplex log callback function previously added by `XPRSaddcblplog`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecblplog(XPRSprob prob, int (XPRS_CC
      *f_lplog)(XPRSprob prob, void* object), void* object);
```

Arguments

`prob`	The current problem.
`f_lplog`	The callback function to remove. If NULL then all lplog callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all lplog callbacks with the function pointer `f_lplog` will be removed.

Example

The following code sets and removes a callback function:

```
XPRSsetintcontrol(prob,XPRS_LPLOG,10);
XPRSaddcblplog(prob,lpLog,NULL,0);
XPRSreadprob(prob,"problem","");
XPRSlpoptimize(prob,"");
XPRSremovecblplog(prob,lpLog,NULL);
}
```

Related topics

`XPRSaddcblplog`

XPRSremovecbmessage

Purpose

Removes a message callback function previously added by `XPRSaddcbmessage`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbmessage(XPRSprob prob, void (XPRS_CC
      *f_message)(XPRSprob prob, void* vContext, const char* msg, int len,
      int msgtype), void* object);
```

Arguments

`prob`	The current problem.
`f_message`	The callback function to remove. If NULL then all message callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all message callbacks with the function pointer `f_message` will be removed.

Related topics

`XPRSaddcbmessage`

XPRSremovecbmipthread

Purpose

Removes a callback function previously added by `XPRSaddcbmipthread`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbmipthread(XPRSprob prob, void (XPRS_CC
      *f_mipthread)(XPRSprob master_prob, void* vContext, XPRSprob prob),
      void* object);
```

Arguments

`prob`	The current problem.
`f_mipthread`	The callback function to remove. If NULL then all variable branching callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all variable branching callbacks with the function pointer `f_mipthread` will be removed.

Related topics

`XPRSaddcbmipthread`

XPRSremovecbnewnode

Purpose

Removes a new-node callback function previously added by `XPRSaddcbnewnode`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbnewnode(XPRSprob prob, void (XPRS_CC
      *f_newnode)(XPRSprob my_prob, void* my_object, int parentnode, int
      newnode, int branch), void* object);
```

Arguments

`prob`	The current problem.
`f_newnode`	The callback function to remove. If NULL then all separation callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all separation callbacks with the function pointer `f_newnode` will be removed.

Related topics

`XPRSaddcbnewnode`

XPRSremovecbnodecutoff

Purpose

Removes a node-cutoff callback function previously added by `XPRSaddcbnodecutoff`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbnodecutoff(XPRSprob prob, void (XPRS_CC
      *f_nodecutoff)(XPRSprob my_prob, void *my_object, int nodnum), void*
      object);
```

Arguments

`prob`	The current problem.
`f_nodecutoff`	The callback function to remove. If NULL then all node-cutoff callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all node-cutoff callbacks with the function pointer `f_nodecutoff` will be removed.

Related topics

`XPRSaddcbnodecutoff`

XPRSremovecboptnode

Purpose

Removes a node-optimal callback function previously added by `XPRSaddcboptnode`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecboptnode(XPRSprob prob, void (XPRS_CC
      *f_optnode)(XPRSprob my_prob, void *my_object, int *feas), void*
      object);
```

Arguments

`prob`	The current problem.
`f_optnode`	The callback function to remove. If NULL then all node-optimal callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all node-optimal callbacks with the function pointer `f_optnode` will be removed.

Related topics

`XPRSaddcboptnode`

XPRSremovecbpreintsol

Purpose

Removes a pre-integer solution callback function previously added by `XPRSaddcbpreintsol`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbpreintsol(XPRSprob prob, void (XPRS_CC
      *f_preintsol)(XPRSprob my_prob, void *my_object, int soltype, int
      *ifreject, double *cutoff), void* object);
```

Arguments

`prob`	The current problem.
`f_preintsol`	The callback function to remove. If NULL then all user infeasible node callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all user infeasible node callbacks with the function pointer `f_preintsol` will be removed.

Related topics

`XPRSaddcbpreintsol`

XPRSremovecbprenode

Purpose

Removes a preprocess node callback function previously added by `XPRSaddcbprenode`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbprenode(XPRSprob prob, void (XPRS_CC
      *f_prenode)(XPRSprob prob, void* my_object, int* nodinfeas), void*
      object);
```

Arguments

`prob`	The current problem.
`f_prenode`	The callback function to remove. If NULL then all preprocess node callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all preprocess node callbacks with the function pointer `f_prenode` will be removed.

Related topics

`XPRSaddcbprenode`

XPRSremovecbsepnode

Purpose

This function is deprecated and may be removed in future releases. Please use branching objects instead.

Removes a pre-integer solution callback function previously added by `XPRSaddcbsepnode`. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbsepnode(XPRSprob prob, int (XPRS_CC
       *f_sepnode)(XPRSprob prob, void* vContext, int ibr, int iglsel, int
       ifup, double curval), void* object);
```

Arguments

`prob`	The current problem.
`f_sepnode`	The callback function to remove. If NULL then all separation callback functions added with the given user-defined object value will be removed.
`object`	The object value that the callback was added with. If NULL, then the object value will not be checked and all separation callbacks with the function pointer `f_sepnode` will be removed.

Related topics

XPRSaddcbsepnode

XPRSremovecbusersolnotify

Purpose

Removes a user solution notification callback previously added by XPRSaddcbusersolnotify. The specified callback function will no longer be called after it has been removed.

Synopsis

```
int XPRS_CC XPRSremovecbusersolnotify(XPRSprob prob, void (XPRS_CC
      *f_usersolnotify)(XPRSprob my_prob, void* my_object, const char*
      solname, int status), void* object);
```

Arguments

`prob`	The current problem.
`f_usersolnotify`	The callback function to remove. If `NULL` then all user solution notification callback functions added with the given user defined object value will be removed.
`object`	The object value that the callback was added with. If `NULL`, then the object value will not be checked and all integer solution callbacks with the function pointer `f_usersolnotify` will be removed.

Related topics

XPRSaddcbusersolnotify.

XPRSrepairinfeas

Purpose

Provides a simplified interface for XPRSrepairweightedinfeas.

Synopsis

```
int XPRS_CC XPRSrepairinfeas (XPRSprob prob, int *scode, char pflags, char
      oflags, char gflags, double lrp, double grp, double lbp, double ubp,
      double delta);
```

Arguments

prob	The current problem.
scode	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.
pflags	The type of penalties created from the preferences: c each penalty is the reciprocal of the preference (default); s the penalties are placed in the scaled problem.
oflags	Controls the second phase of optimization: o use the objective sense of the original problem (default); x maximize the relaxed problem using the original objective; f skip optimization regarding the original objective; n minimize the relaxed problem using the original objective; i if the relaxation is infeasible, generate an irreducible infeasible subset for the analys of the problem; a if the relaxation is infeasible, generate all irreducible infeasible subsets for the analys of the problem.
gflags	Specifies if the global search should be done: g do the global search (default); l solve as a linear model ignoring the discreteness of variables.
lrp	Preference for relaxing the less or equal side of row.
grp	Preference for relaxing the greater or equal side of a row.
lbp	Preferences for relaxing lower bounds.
ubp	Preferences for relaxing upper bounds.
delta	The relaxation multiplier in the second phase -1. For console use `-d` value. A positive value means a relative relaxation by multiplying the first phase objective with (`delta-1`), while a negative value means an absolute relaxation, by adding `abs(delta)` to the first phase objective.

Related controls

Integer

DEFAULTALG	Forced algorithm selection (default for repairinfeas is primal).

Example

```
READPROB MYPROB.LP
REPAIRINFEAS -a -d 0.002
```

Further information

1. A row or bound is relaxed by introducing a new nonnegative variable that will contain the infeasibility of the row or bound. Suppose for example that row $a^T x = b$ is relaxed from below. Then a new variable (infeasibility breaker) $s \geq 0$ is added to the row, which becomes $a^T x + s = b$. Observe that $a^T x$ may now take smaller values than b. To minimize such violations, the weighted sum of these new variables is minimized.
2. A preference of 0 results in the row or bound not being relaxed.
3. A negative preference indicates that a quadratic penalty cost should be applied. This can specified on a per constraint side or bound basis.
4. Note that the set of preferences are scaling independent.
5. If a feasible solution is identified for the relaxed problem, with a sum of violations p, then the sum of violations is restricted to be no greater than (1+delta)p, and the problem is optimized with respect to the original objective function. A nonzero delta increases the freedom of the original problem.
6. Note that on some problems, slight modifications of delta may affect the value of the original objective drastically.
7. The default value for delta in the console is 0.001.
8. Note that because of their special associated modeling properties, binary and semi-continuous variables are not relaxed.
9. The default algorithm for the first phase is the simplex algorithm, since the primal problem can be efficiently warm started in case of the extended problem. These may be altered by setting the value of control DEFAULTALG.
10. If pflags is set such that each penalty is the reciprocal of the preference, the following rules are applied while introducing the auxiliary variables:

Preference	Affects	Relaxation	Cost if pref.>0	Cost if pref.<0
lrp	= rows	$a^T x$ − aux_var = b	1/lrp*aux_var	1/lrp*aux_var^2
lrp	<= rows	$a^T x$ − aux_var <= b	1/lrp*aux_var	1/lrp*aux_var^2
grp	= rows	$a^T x$ + aux_var = b	1/grp*aux_var	1/grp*aux_var^2
grp	>= rows	$a^T x$ + aux_var >= b	1/grp*aux_var	1/grp*aux_var^2
ubp	upper bounds	x_i − aux_var <= u	1/ubp*aux_var	1/ubp*aux_var^2
lbp	lower bounds	x_i + aux_var >= l	1/lbp*aux_var	1/lbp*aux_var^2

11. If an irreducible infeasible set (IIS) has been identified, the generated IIS(s) are accessible through the IIS retrieval functions, see NUMIIS and XPRSgetiisdata.

Related topics

XPRSrepairweightedinfeas, 6.1.4.

XPRSrepairweightedinfeas

Purpose

By relaxing a set of selected constraints and bounds of an infeasible problem, it attempts to identify a 'solution' that violates the selected set of constraints and bounds minimally, while satisfying all other constraints and bounds. Among such solution candidates, it selects one that is optimal regarding to the original objective function. For the console version, see `REPAIRINFEAS`.

Synopsis

```
int XPRS_CC XPRSrepairweightedinfeas(XPRSprob prob, int * scode, const
      double lrp_array[], const double grp_array[], const double
      lbp_array[], const double ubp_array[], char phase2, double delta,
      const char *optflags);
```

Arguments

`prob` The current problem.

`scode` The status after the relaxation:
- `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.

`lrp_array` Array of size `ROWS` containing the preferences for relaxing the less or equal side of row.

`grp_array` Array of size `ROWS` containing the preferences for relaxing the greater or equal side of a row.

`lbp_array` Array of size `COLS` containing the preferences for relaxing lower bounds.

`ubp_array` Array of size `COLS` containing preferences for relaxing upper bounds.

`phase2` Controls the second phase of optimization:
- `o` use the objective sense of the original problem (default);
- `x` maximize the relaxed problem using the original objective;
- `f` skip optimization regarding the original objective;
- `n` minimize the relaxed problem using the original objective;
- `i` if the relaxation is infeasible, generate an irreducible infeasible subset for the analys of the problem;
- `a` if the relaxation is infeasible, generate all irreducible infeasible subsets for the analys of the problem.

`delta` The relaxation multiplier in the second phase -1.

`optflags` Specifies flags to be passed to the Optimizer.

Related controls

Double

`PENALTYVALUE` The weighted sum of violations if a solution is identified to the relaxed problem.

Further information

1. A row or bound is relaxed by introducing a new nonnegative variable that will contain the infeasibility of the row or bound. Suppose for example that row $a^T x = b$ is relaxed from below. Then a new variable ('infeasibility breaker') $s \geq 0$ is added to the row, which becomes $a^T x + s = b$. Observe that $a^T x$ may now take smaller values than b. To minimize such violations, the weighted sum of these new variables is minimized.
2. A preference of 0 results in the row or bound not being relaxed. The higher the preference, the more willing the modeller is to relax a given row or bound.
3. The weight of each infeasibility breaker in the objective minimizing the violations is 1/p, where `p` is the preference associated with the infeasibility breaker. Thus the higher the preference is, the lower a penalty is associated with the infeasibility breaker while minimizing the violations.
4. If a feasible solution is identified for the relaxed problem, with a sum of violations `p`, then the sum of violations is restricted to be no greater than (`1+delta`)`p`, and the problem is optimized with respect to the original objective function. A nonzero delta increases the freedom of the original problem.
5. Note that on some problems, slight modifications of delta may affect the value of the original objective drastically.
6. The default value for delta in the console is 0.001.
7. Note that because of their special associated modeling properties, binary and semi-continuous variables are not relaxed.
8. If `pflags` is set such that each penalty is the reciprocal of the preference, the following rules are applied while introducing the auxiliary variables:

Pref. array	Affects	Relaxation	Cost if pref.>0	Cost if pref.<0
lrp_array	= rows	$a^T x$ - aux_var = b	1/lrp*aux_var	1/lrp*aux_var^2
lrp_array	<= rows	$a^T x$ - aux_var <= b	1/lrp*aux_var	1/lrp*aux_var^2
grp_array	= rows	$a^T x$ + aux_var = b	1/grp*aux_var	1/grp*aux_var^2
grp_array	>= rows	$a^T x$ + aux_var >= b	1/grp*aux_var	1/grp*aux_var^2
ubp_array	upper bounds	x_i - aux_var <= u	1/ubp*aux_var	1/ubp*aux_var^2
lbp_array	lower bounds	x_i + aux_var >= l	1/lbp*aux_var	1/lbp*aux_var^2

9. If an irreducible infeasible set (IIS) has been identified, the generated IIS(s) are accessible through the IIS retrieval functions, see `NUMIIS` and `XPRSgetiisdata`.

Related topics

`XPRSrepairinfeas` (`REPAIRINFEAS`), `XPRSrepairweightedinfeasbounds`, 6.1.4.

XPRSrepairweightedinfeasbounds REPAIRINFEAS

Purpose

An extended version of XPRSrepairweightedinfeas that allows for bounding the level of relaxation allowed.

Synopsis

```
int XPRS_CC XPRSrepairweightedinfeasbounds(XPRSprob prob, int * scode,
      const double lrp_array[], const double grp_array[], const double
      lbp_array[], const double ubp_array[], const double lrb_array[],
      const double grb_array[], const double lbb_array[], const double
      ubb_array[], char phase2, double delta, const char *optflags);
REPAIRINFEAS -[pflags] -[oflags] -[gflags] -[lrp value] -[grp value] -[lbp
      value] -[ubp value] -[lrb value] -[grb value] -[lbb value] -[ubb
      value] -[d value] -[r]
```

Arguments

`prob`	The current problem.
`scode`	The status after the relaxation: `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.
`lrp_array`	Array of size `ROWS` containing the preferences for relaxing the less or equal side of row. For the console use `-lrp` value.
`grp_array`	Array of size `ROWS` containing the preferences for relaxing the greater or equal side of a row. For the console use `-grp` value.
`lbp_array`	Array of size `COLS` containing the preferences for relaxing lower bounds. For the console use `-lbp` value.
`ubp_array`	Array of size `COLS` containing preferences for relaxing upper bounds. For the console use `-ubp` value.
`lrb_array`	Array of size `ROWS` containing the upper bounds on the amount the less or equal side of a row can be relaxed. For the console use `-lrb` value.
`grb_array`	Array of size `ROWS` containing the upper bounds on the amount the greater or equal side of a row can be relaxed. For the console use `-grb` value.
`lbb_array`	Array of size `COLS` containing the upper bounds on the amount the lower bounds can be relaxed. For the console use `-lbb` value.
`ubb_array`	Array of size `COLS` containing the upper bounds on the amount the upper bounds can be relaxed. For the console use `-ubb` value.
`phase2`	Controls the second phase of optimization: `o` use the objective sense of the original problem (default); `x` maximize the relaxed problem using the original objective; `f` skip optimization regarding the original objective; `n` minimize the relaxed problem using the original objective; `i` if the relaxation is infeasible, generate an irreducible infeasible subset for the analys of the problem; `a` if the relaxation is infeasible, generate all irreducible infeasible subsets for the analys of the problem.
`delta`	The relaxation multiplier in the second phase -1.
`optflags`	Specifies flags to be passed to the Optimizer.

`r` If a summary of the violated variables and constraints should be printed after the relaxed solution is determined.

Related controls

Double

PENALTYVALUE The weighted sum of violations if a solution is identified to the relaxed problem.

Further information

1. The console command REPAIRINFEAS assumes that all preferences are 1 by default. Use the options `-lrp`, `-grp`, `-lbp` or `-ubp` to change them. The default limit on the maximum allowed relaxation per row or bound in plus infinity.
2. A row or bound is relaxed by introducing a new nonnegative variable that will contain the infeasibility of the row or bound. Suppose for example that row $a^T x = b$ is relaxed from below. Then a new variable ('infeasibility breaker') $s \geq 0$ is added to the row, which becomes $a^T x + s = b$. Observe that $a^T x$ may now take smaller values than b. To minimize such violations, the weighted sum of these new variables is minimized.
3. A preference of 0 results in the row or bound not being relaxed. The higher the preference, the more willing the modeller is to relax a given row or bound.
4. A negative preference indicates that a quadratic penalty cost should be applied. This can specified on a per constraint side or bound basis.
5. If a feasible solution is identified for the relaxed problem, with a sum of violations `p`, then the sum of violations is restricted to be no greater than (`1+delta`)`p`, and the problem is optimized with respect to the original objective function. A nonzero delta increases the freedom of the original problem.
6. Note that on some problems, slight modifications of delta may affect the value of the original objective drastically.
7. The default value for delta in the console is 0.001.
8. Note that because of their special associated modeling properties, binary and semi-continuous variables are not relaxed.
9. Given any row `j` with preferences `lrp=lrp_array[j]` and `grp=grp_array[j]`, or variable `i` with bound preferences `ubp=ubp_array[i]` and `lbp=lbp_array[i]`, the following rules are applied while introducing the auxiliary variables:

Preference	Affects	Relaxation	Cost if pref.>0	Cost if pref.<0
lrp	= rows	$a^T x$ - aux_var = b	1/lrp*aux_var	1/lrp*aux_var^2
lrp	<= rows	$a^T x$ - aux_var <= b	1/lrp*aux_var	1/lrp*aux_var^2
grp	= rows	$a^T x$ + aux_var = b	1/grp*aux_var	1/grp*aux_var^2
grp	>= rows	$a^T x$ + aux_var >= b	1/grp*aux_var	1/grp*aux_var^2
ubp	upper bounds	x_i - aux_var <= u	1/ubp*aux_var	1/ubp*aux_var^2
lbp	lower bounds	x_i + aux_var >= l	1/lbp*aux_var	1/lbp*aux_var^2

10. Only positive bounds are applied; a zero or negative bound is ignored and the amount of relaxation allowed for the corresponding row or bound is not limited. The effect of a zero bound on a row or bound would be equivalent with not relaxing it, and can be achieved by setting its preference array value to zero instead, or not including it in the preference arrays.
11. If an irreducible infeasible set (IIS) has been identified, the generated IIS(s) are accessible through the IIS retrieval functions, see NUMIIS and XPRSgetiisdata.

Related topics

XPRSrepairinfeas (REPAIRINFEAS), 6.1.4.

XPRSrestore RESTORE

Purpose

Restores the Optimizer's data structures from a file created by XPRSsave (SAVE). Optimization may then recommence from the point at which the file was created.

Synopsis

```
int XPRS_CC XPRSrestore(XPRSprob prob, const char *probname, const char
      *flags);
RESTORE [probname] [flags]
```

Arguments

prob	The current problem.	
probname	A string of up to MAXPROBNAMELENGTH characters containing the problem name.	
flags	f	Force the restoring of a save file even if it is from a different version.

Example 1 (Library)

```
XPRSrestore(prob,"","");
```

Example 2 (Console)

```
RESTORE
```

Further information

1. This routine restores the data structures from the file *problem_name*`.svf` that was created by a previous execution of `XPRSsave` (`SAVE`). Note that `.svf` files are particular to the release of the Optimizer used to create them. They can only be read using the same release Optimizer as used to create them.
2. (*Console*) The main use for XPRSsave (SAVE) and `XPRSrestore` (`RESTORE`) is to enable the user to interrupt a long optimization run using CTRL-C, and save the Optimizer status with the ability to restart it later from where it left off. It might also be used to save the optimal status of a problem when the user then intends to implement several uses of XPRSalter (ALTER) on the problem, re-optimizing each time from the saved status.
3. The use of the 'f' flag is not recommended and can cause unexpected results.

Related topics

XPRSalter (ALTER), XPRSsave (SAVE).

XPRSrhssa

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

```
int XPRS_CC XPRSrhssa(XPRSprob prob, int nels, const int mindex[], double
       lower[], double upper[]);
```

Arguments

`prob`	The current problem.
`nels`	The number of RHS coefficients for which sensitivity ranges are required.
`mindex`	Integer array of length `nels` containing the indices of the rows whose RHS coefficients sensitivity ranges are required.
`lower`	Double array of length `nels` where the RHS lower range values are to be returned.
`upper`	Double array of length `nels` where the RHS upper range values are to be returned.

Example

Here we obtain the RHS function ranges for the three columns: `2`, `6` and `8`:

```
mindex[0] = 2; mindex[1] = 8; mindex[2] = 6;
XPRSrhssa(prob,3,mindex,lower,upper);
```

After which lower and upper contain:

```
lower[0] = 5.0; upper[0] = 7.0;
lower[1] = 3.8; upper[1] = 5.2;
lower[2] = 5.7; upper[2] = 1e+20;
```

Meaning that the current basis remains optimal when $5.0 \leq \text{rhs}_2$, $3.8 \leq \text{rhs}_8 \leq 5.2$ and $5.7 \leq \text{rhs}_6$, rhs_i being the RHS coefficient of row `i`.

Further information

`XPRSrhssa` can only be called when an optimal solution to the current LP has been found. It cannot be used when the problem is MIP presolved.

Related topics

`XPRSobjsa`.

XPRSsave, XPRSsaveas 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

```
int XPRS_CC XPRSsave(XPRSprob prob);
int XPRS_CC XPRSsaveas(XPRSprob prob, const char *probname);
SAVE
```

Arguments

`prob`	The current problem.
`probname`	The name of the file (without .svf) to save to.

Example 1 (Library)

```
XPRSsave(prob);
```

Example 2 (Console)

```
SAVE
```

Further information

1. The data structures are written to the file *problem_name*`.svf`. Optimization may recommence from the same point when the data structures are restored by a call to `XPRSrestore` (`RESTORE`). Note that the `.svf` files created are particular to the release of the Optimizer used to create them. They can only be read using the same release Optimizer as used to create them.
2. The function `XPRSsaveas` is equivalent to `XPRSsave` with the exception of allowing to adjust the name of the file created. The name of the file must not be greater than `MAXPROBNAMELENGTH`.

Related topics

`XPRSrestore` (`RESTORE`).

XPRSscale SCALE

Purpose

Re-scales the current matrix.

Synopsis

```
int XPRS_CC XPRSscale(XPRSprob prob, const int mrscal[], const int
        mcscal[]);
SCALE
```

Arguments

`prob`	The current problem.
`mrscal`	Integer array of size `ROWS` containing the powers of `2` with which to scale the rows, or `NULL` if not required.
`mcscal`	Integer array of size `COLS` containing the powers of `2` with which to scale the columns, or `NULL` if not required.

Related controls

Integer

`SCALING` Type of scaling.

Example 1 (Library)

```
XPRSreadprob(prob,"jovial","");
XPRSalter(prob,"serious");
XPRSscale(prob,NULL,NULL);
XPRSlpoptimize(prob,"");
```

This reads the MPS file `jovial.mat`, modifies it according to instructions in the file `serious.alt`, rescales the matrix and seeks the minimum objective value.

Example 2 (Console)

The equivalent set of commands for the Console user would be:

```
READPROB jovial
ALTER serious
SCALE
LPOPTIMIZE
```

Further information

1. If `mrscal` and `mcscal` are both non-`NULL` then they will be used to scale the matrix. Otherwise the matrix will be scaled according to the control `SCALING`. This routine may be useful when the current matrix has been modified by calls to routines such as `XPRSalter` (`ALTER`), `XPRSchgmcoef` and `XPRSaddrows`.
2. `XPRSscale` (`SCALE`) cannot be called if the current matrix is presolved.

Related topics

`XPRSalter` (`ALTER`), `XPRSreadprob` (`READPROB`).

XPRSsetbranchbounds

Purpose

Specifies the bounds previously stored using `XPRSstorebounds` that are to be applied in order to branch on a user global entity. This routine can only be called from the user separate callback function, `XPRSaddcbsepnode`.

Synopsis

```
int XPRS_CC XPRSsetbranchbounds(XPRSprob prob, void *mindex);
```

Arguments

`prob`	The current problem.
`mindex`	Pointer previously defined in a call to `XPRSstorebounds` that references the stored bounds to be used to separate the node.

Example

This example defines a user separate callback function for the global search:

```
XPRSaddcbsepnode(prob,nodeSep,NULL,0);
```

where the function `nodeSep` is defined as follows:

```
int nodeSep(XPRSprob prob, void *obj, int ibr, int iglsel,
            int ifup, double curval)
{
  void *index;
  double dbd;

  if( ifup )
  {
    dbd = ceil(curval);
    XPRSstorebounds(prob, 1, &iglsel, "L", &dbd, &index);
  }
  else
  {
    dbd = floor(curval);
    XPRSstorebounds(prob, 1, &iglsel, "U", &dbd, &index);
  }
  XPRSsetbranchbounds(prob, index);
  return 0;
}
```

Related topics

`XPRSloadcuts`, `XPRSaddcbestimate`, `XPRSaddcbsepnode`, `XPRSstorebounds`, Section 5.9.

XPRSsetbranchcuts

Purpose

Specifies the pointers to cuts in the cut pool that are to be applied in order to branch on a user global entity. This routine can only be called from the user separate callback function, XPRSaddcbsepnode.

Synopsis

```
int XPRS_CC XPRSsetbranchcuts(XPRSprob prob, int ncuts, const XPRScut
      mindex[]);
```

Arguments

prob	The current problem.
ncuts	Number of cuts to apply.
mindex	Array containing the pointers to the cuts in the cut pool that are to be applied. Typically obtained from XPRSstorecuts.

Related topics

XPRSgetcpcutlist, XPRSaddcbestimate, XPRSaddcbsepnode, XPRSstorecuts, Section 5.9.

XPRSsetcheckedmode

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

```
int XPRS_CC XPRSsetcheckedmode(int checked_mode);
```

Argument

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

Related topics

`XPRSgetcheckedmode`.

XPRSsetdblcontrol

Purpose

Sets the value of a given double control parameter.

Synopsis

```
int XPRS_CC XPRSsetdblcontrol(XPRSprob prob, int ipar, double dsval);
```

Arguments

`prob`	The current problem.
`ipar`	Control parameter whose value is to be set. A full list of all controls may be found in 9, or from the list in the `xprs.h` header file.
`dsval`	Value to which the control parameter is to be set.

Related topics

`XPRSgetdblcontrol`, `XPRSsetintcontrol`, `XPRSsetstrcontrol`.

XPRSsetdefaultcontrol SETDEFAULTCONTROL

Purpose

Sets a single control to its default value.

Synopsis

```
int XPRS_CC XPRSsetdefaultcontrol(XPRSprob prob, int ipar);
SETDEFAULTCONTROL controlname
```

Arguments

`prob`	The current problem.
`ipar`	Integer, double or string control parameter whose default value is to be set.
`controlname`	Integer, double or string control parameter whose default value is to be set.

Example

The following turns off presolve to solve a problem, before resetting it to its default value and solving it again:

```
XPRSsetintcontrol(prob, XPRS_PRESOLVE, 0);
XPRSmipoptimize(prob, "");
XPRSwriteprtsol(prob);
XPRSsetdefaultcontrol(prob, XPRS_PRESOLVE);
XPRSmipoptimize(prob, "");
```

Further information

A full list of all controls may be found in Chapter 9, or from the list in the `xprs.h` header file.

Related topics

`XPRSsetdefaults`, `XPRSsetintcontrol`, `XPRSsetdblcontrol`, `XPRSsetstrcontrol`.

XPRSsetdefaults SETDEFAULTS

Purpose

Sets all controls to their default values. Must be called before the problem is read or loaded by `XPRSreadprob`, `XPRSloadglobal`, `XPRSloadlp`, `XPRSloadqglobal`, `XPRSloadqp`.

Synopsis

```
int XPRS_CC XPRSsetdefaults(XPRSprob prob);
SETDEFAULTS
```

Argument

`prob`	The current problem.

Example

The following turns off presolve to solve a problem, before resetting the control defaults, reading it and solving it again:

```
XPRSsetintcontrol(prob, XPRS_PRESOLVE, 0);
XPRSmipoptimize(prob, "");
XPRSwriteprtsol(prob);
XPRSsetdefaults(prob);
XPRSreadprob(prob);
XPRSmipoptimize(prob, "");
```

Related topics

`XPRSsetdefaultcontrol`, `XPRSsetintcontrol`, `XPRSsetdblcontrol`, `XPRSsetstrcontrol`.

XPRSsetindicators

Purpose

Specifies that a set of rows in the matrix will be treated as indicator constraints, during a global search. An indicator constraint is made of a `condition` and a `linear constraint`. The `condition` is of the type "`bin = value`", where `bin` is a binary variable and `value` is either 0 or 1. The `linear constraint` is any linear row. During global search, a row configured as an indicator constraint is enforced only when condition holds, that is only if the indicator variable `bin` has the specified value.

Synopsis

```
int XPRS_CC XPRSsetindicators(XPRSprob prob, int nrows, const int mrows[],
      const int inds[], const int comps[]);
```

Arguments

`prob`	The current problem.
`nrows`	The number of indicator constraints.
`mrows`	Integer array of length `nrows` containing the indices of the rows that define the linear constraint part for the indicator constraints.
`inds`	Integer array of length `nrows` containing the column indices of the indicator variables.
`comps`	Integer array of length `nrows` with the complement flags: `0` not an indicator constraint (in this case the corresponding entry in the `inds` array is ignored); `1` for indicator constraints with condition "`bin = 1`"; `-1` for indicator constraints with condition "`bin = 0`";

Example

This sets the first two matrix rows as indicator rows in the global problem prob; the first row controlled by condition `x4=1` and the second row controlled by condition `x5=0` (assuming `x4` and `x5` correspond to columns indices 4 and 5).

```
int mrows[] = {0,1};
int inds[] = {4,5};
int comps[] = {1,-1};

...
XPRSsetindicators(prob,2,mrows,inds,comps);
XPRSmipoptimize(prob,"");
```

Further information

Indicator rows must be set up before solving the problem. Any indicator row will be removed from the matrix after presolve and added to a special pool. An indicator row will be added back into the active matrix only when its associated condition holds. An indicator variable can be used in multiple indicator rows and can also appear in normal rows and in the objective function.

Related topics

`XPRSgetindicators`, `XPRSdelindicators`.

XPRSsetintcontrol, XPRSsetintcontrol64

Purpose

Sets the value of a given integer control parameter.

Synopsis

```
int XPRS_CC XPRSsetintcontrol(XPRSprob prob, int ipar, int isval);

int XPRS_CC XPRSsetintcontrol64(XPRSprob prob, int ipar, XPRSint64 isval);
```

Arguments

`prob`	The current problem.
`ipar`	Control parameter whose value is to be set. A full list of all controls may be found in 9, or from the list in the `xprs.h` header file.
`isval`	Value to which the control parameter is to be set.

Example

The following sets the control `PRESOLVE` to `0`, turning off the presolve facility prior to optimization:

```
XPRSsetintcontrol(prob, XPRS_PRESOLVE, 0);
XPRSlpoptimize(prob, "");
```

Further information

Some of the integer control parameters, such as `SCALING`, are bitmaps, with each bit controlling different behavior. Bit `0` has value `1`, bit `1` has value `2`, bit `2` has value `4`, and so on.

Related topics

`XPRSgetintcontrol`, `XPRSsetdblcontrol`, `XPRSsetstrcontrol`.

XPRSsetlogfile SETLOGFILE

Purpose

This directs all Optimizer output to a log file.

Synopsis

```
int XPRS_CC XPRSsetlogfile(XPRSprob prob, const char *filename);
SETLOGFILE 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. If set to `NULL`, redirection of the output will stop and all screen output will be turned back on (except for library users where screen output is always turned off).

Example

The following directs output to the file `logfile.log`:

```
XPRSinit(NULL);
XPRScreateprob(&prob);
XPRSsetlogfile(prob,"logfile.log");
```

Further information

1. It is recommended that a log file be set up for each problem being worked on, since it provides a means for obtaining any errors or warnings output by the Optimizer during the solution process.
2. If output is redirected with `XPRSsetlogfile` all screen output will be turned off.
3. Alternatively, an output callback can be defined using `XPRSaddcbmessage`, which will be called every time a line of text is output. To discard all output messages the `OUTPUTLOG` integer control can be set to `0`.

Related topics

`XPRSaddcbmessage`.

XPRSsetmessagestatus

Purpose

Manages suppression of messages.

Synopsis

```
int XPRS_CC XPRSsetmessagestatus(XPRSprob prob, int errcode, int status);
```

Arguments

`prob`	The problem for which message `errcode` is to have its suppression status changed; pass `NULL` if the message should have the status apply globally to all problems.
`errcode`	The id number of the message. Refer to the section 11 for a list of possible message numbers.
`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.

Example

Attempting to optimize a problem that has no matrix loaded gives error 91. The following code uses `XPRSsetmessagestatus` to suppress the error message:

```
XPRScreateprob(&prob);
XPRSsetmessagestatus(prob,91,0);
XPRSlpoptimize(prob,"");
```

Further information

If a message is suppressed globally then the message can only be enabled for any problem once the global suppression is removed with a call to `XPRSsetmessagestatus` with `prob` passed as `NULL`.

Related topics

`XPRSgetmessagestatus`.

XPRSsetprobname SETPROBNAME

Purpose

Sets the current default problem name. This command is rarely used.

Synopsis

```
int XPRS_CC XPRSsetprobname(XPRSprob prob, const char *probname);
SETPROBNAME probname
```

Arguments

`prob`	The current problem.
`probname`	A string of up to `MAXPROBNAMELENGTH` characters containing the problem name.

Example

```
READPROB bob
LPOPTIMIZE
SETPROBNAME jim
READPROB
```

The above will read the problem `bob` and then read the problem `jim`.

Related topics

`XPRSreadprob` (`READPROB`), `XPRSgetprobname`, `MAXPROBNAMELENGTH`.

XPRSsetstrcontrol

Purpose

Used to set the value of a given string control parameter.

Synopsis

```
int XPRS_CC XPRSsetstrcontrol(XPRSprob prob, int ipar, const char *csval);
```

Arguments

`prob`	The current problem.
`ipar`	Control parameter whose value is to be set. A full list of all controls may be found in 9, or from the list in the `xprs.h` header file.
`csval`	A string containing the value to which the control is to be set (plus a null terminator).

Example

The following sets the control `MPSOBJNAME` to `"Profit"`:

```
XPRSsetstrcontrol(prob, XPRS_MPSOBJNAME, "Profit");
```

Related topics

`XPRSgetstrcontrol`, `XPRSsetdblcontrol`, `XPRSsetintcontrol`.

STOP

Purpose

Terminates the Console Optimizer, returning an exit code to the operating system. This is useful for batch operations.

Synopsis

```
STOP
```

Example

The following example inputs a matrix file, `lama.mat`, runs a global optimization on it and then exits:

```
READPROB lama
MIPOPTIMIZE
STOP
```

Further information

This command may be used to terminate the Optimizer as with the `QUIT` command. It sets an exit value which may be inspected by the host operating system or invoking program.

Related topics

`QUIT`.

XPRSstorebounds

Purpose

Stores bounds for node separation using user separate callback function.

Synopsis

```
int XPRS_CC XPRSstorebounds(XPRSprob prob, int nbnds, const int mcols[],
        const char qbtype[], const double dbds[], void **mindex);
```

Arguments

`prob`	The current problem.
`nbnds`	Number of bounds to store.
`mcols`	Array containing the column indices.
`qbtype`	Array containing the bounds types: `U` indicates an upper bound; `L` indicates a lower bound.
`dbds`	Array containing the bound values.
`mindex`	Pointer that the user will use to reference the stored bounds for the Optimizer in `XPRSsetbranchbounds`.

Example

This example defines a user separate callback function for the global search:

```
XPRSaddcbsepnode(prob,nodeSep,void,0);
```

where the function `nodeSep` is defined as follows:

```
int nodeSep(XPRSprob prob, void *obj int ibr, int iglsel,
            int ifup, double curval)
{
  void *index;
  double dbd;

  if( ifup )
  {
    dbd = ceil(curval);
    XPRSstorebounds(prob, 1, &iglsel, "L", &dbd, &index);
  }
  else
  {
    dbd = floor(curval);
    XPRSstorebounds(prob, 1, &iglsel, "U", &dbd, &index);
  }
  XPRSsetbranchbounds(prob, index);
  return 0;
}
```

Related topics

`XPRSsetbranchbounds`, `XPRSaddcbestimate`, `XPRSaddcbsepnode`.

XPRSstorecuts, XPRSstorecuts64

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 XPRSloadcuts or XPRSsetbranchcuts before they become active.

Synopsis

```
int XPRS_CC XPRSstorecuts(XPRSprob prob, int ncuts, int nodupl, const int
      mtype[], const char qrtype[], const double drhs[], const int
      mstart[], XPRScut mindex[], const int mcols[], const double
      dmatval[]);

int XPRS_CC XPRSstorecuts64(XPRSprob prob, int ncuts, int nodupl, const int
      mtype[], const char qrtype[], const double drhs[], const XPRSint64
      mstart[], XPRScut mindex[], const int mcols[], const double
      dmatval[]);
```

Arguments

`prob`	The current problem.
`ncuts`	Number of cuts to add.
`nodupl`	`0` do not exclude duplicates from the cut pool; `1` duplicates are to be excluded from the cut pool; `2` duplicates are to be excluded from the cut pool, ignoring cut type.
`mtype`	Integer array of length `ncuts` containing the cut types. The cut types can be any integer and are used to identify the cuts.
`qrtype`	Character array of length `ncuts` containing the row types: `L` indicates a $\leq$ row; `E` indicates an = row; `G` indicates a $\geq$ row.
`drhs`	Double array of length `ncuts` containing the right hand side elements for the cuts.
`mstart`	Integer array containing offsets into the `mcols` and `dmtval` arrays indicating the start of each cut. This array is of length `ncuts+1` with the last element `mstart[ncuts]` being where cut `ncuts+1` would start.
`mindex`	Array of length `ncuts` where the pointers to the cuts will be returned.
`mcols`	Integer array of length `mstart[ncuts]` containing the column indices in the cuts.
`dmatval`	Double array of length `mstart[ncuts]` containing the matrix values for the cuts.

Related controls

Double

MATRIXTOL	Tolerance on matrix elements.

Further information

1. XPRSstorecuts can be used to eliminate duplicate cuts. If the nodupl parameter is set to 1, the cut pool will be checked for duplicate cuts with a cut type identical to the cuts being added. If a duplicate cut is found the new cut will only be added if its right hand side value makes the cut stronger. If the cut in the pool is weaker than the added cut it will be removed unless it has been applied to an active node of the tree. If nodupl is set to 2 the same test is carried out on all cuts, ignoring the cut type.

2. XPRSstorecuts returns a list of the cuts added to the cut pool in the mindex array. If the cut is not added to the cut pool because a stronger cut exits a NULL will be returned. The mindex array can be passed directly to XPRSloadcuts or XPRSsetbranchcuts to load the most recently stored cuts into the matrix.

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

Related topics

XPRSloadcuts XPRSsetbranchcuts, XPRSaddcbestimate, XPRSaddcbsepnode, 5.9.

XPRSstrongbranch

Purpose

Performs strong branching iterations on all specified bound changes. For each candidate bound change, `XPRSstrongbranch` 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

```
int XPRS_CC XPRSstrongbranch(XPRSprob prob, const int nbnds, const int
       mbndind[], const char cbndtype[], const double dbndval[], const int
       itrlimit, double dsobjval[], int msbstatus[]);
```

Arguments

`prob`	The current problem.
`nbnds`	Number of bound changes to try.
`mbndind`	Integer array of size `nbnds` containing the indices of the columns on which the bounds will change.
`cbndtype`	Character array of length `nbnds` indicating the type of bound to change: `U` indicates change the upper bound; `L` indicates change the lower bound; `B` indicates change both bounds, i.e. fix the column.
`dbndval`	Double array of length `nbnds` giving the new bound values.
`itrlimit`	Maximum number of LP iterations to perform for each bound change.
`dsobjval`	Objective value of each LP after performing the strong branching iterations.
`msbstatus`	Status of each LP after performing the strong branching iterations, as detailed for the `LPSTATUS` attribute.

Example

Suppose that the current LP relaxation has two integer columns (columns `0` and `1` which are fractionals at 0.3 and 1.5, respectively, and we want to perform strong branching in order to choose which to branch on. This could be done in the following way:

```
int mbndind[] = { 0, 0, 1, 1 };
char cbndtype[] = "LULU";
double dbndval[] = {1, 0, 2, 1};
double dsobjval[4];
int msbstatus[4];
...
XPRSstrongbranch(prob, 4, mbndind, cbndtype, dbndval, 1000,
                 dsobjval, msbstatus);
```

Further information

Prior to calling XPRSstrongbranch, the current LP problem must have been solved to optimality and an optimal basis must be available.

XPRSstrongbranchcb

Purpose

Performs strong branching iterations on all specified bound changes. For each candidate bound change, `XPRSstrongbranchcb` 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

```
int XPRS_CC XPRSstrongbranchcb(XPRSprob prob, const int nbnds, const int
      mbndind[], const char cbndtype[], const double dbndval[], const int
      itrlimit, double dsbobjval[], int msbstatus[], int (XPRS_CC
      *sbsolvecb)(XPRSprob prob, void* vContext, int ibnd), void*
      vContext);
```

Arguments

`prob`	The current problem.
`nbnds`	Number of bound changes to try.
`mbndind`	Integer array of size `nbnds` containing the indices of the columns on which the bounds will change.
`cbndtype`	Character array of length `nbnds` indicating the type of bound to change: `U` indicates change the upper bound; `L` indicates change the lower bound; `B` indicates change both bounds, i.e. fix the column.
`dbndval`	Double array of length `nbnds` giving the new bound values.
`itrlimit`	Maximum number of LP iterations to perform for each bound change.
`dsobjval`	Objective value of each LP after performing the strong branching iterations.
`msbstatus`	Status of each LP after performing the strong branching iterations, as detailed for the `LPSTATUS` attribute.
`sbsolvecb`	Function to be called after each strong branch has been reoptimized.
`vContext`	User context to be provided for `sbsolvecb`.
`ibnd`	The index of bound for which `sbsolvecb` is called.

Further information

Prior to calling `XPRSstrongbranchcb`, the current LP problem must have been solved to optimality and an optimal basis must be available.

`XPRSstrongbranchcb` is an extension to `XPRSstrongbranch`. If identical input arguments are provided both will return identical results, the difference being that for the case of `XPRSstrongbranchcb` the `sbsolvecb` function is called at the end of each LP reoptimization.

For each branch optimized, the LP can be interrogated: the LP status of the branch is available through checking `LPSTATUS`, and the objective function value is available through `LPOBJVAL`. It is possible to access the full current LP solution by using `XPRSgetlpsol`.

TUNE

Purpose

This command can start a tuner session for the current problem. In this case, 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`. This command can also handle the input and output of tuner method files.

Synopsis

```
TUNE [-flags] [subcommand [filename]]
```

Arguments

`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`); `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.
`subcommand`	Subcommand to pass to `TUNE` for handling tuner method files. It can be one of: `pm / printmethod` Print the tuner method on the console. `wm / writemethod` Write the tuner method to a file. `rm / readmethod` Read the tuner method from a file. `probset` Tune a set of problems. `mipset` Tune a set of MIP problems. `lpset` Tune a set of LP problems.
`filename`	Tuner method file or problem set file. This is an optional argument of the subcommand.

Related controls

Integer

`TUNERHISTORY`	Whether to reuse and append to previous tuner result.
`TUNERMAXTIME`	Maximum total time allowed for the tuner.
`TUNERMETHOD`	Selects a factory tuner method.
`TUNERMODE`	Enable or disable the tuner.
`TUNEROUTPUT`	Whether to write tuner result and logs to file system.
`TUNERPERMUTE`	Number of permutations to solve with each control setting.
`TUNERTARGET`	Defines the criterion by which individual runs are compared.
`TUNERTHREADS`	Number of threads to be used by the tuner.

String

`TUNERMETHODFILE`	A file which contains a user-defined tuner method.
`TUNEROUTPUTPATH`	The root path for all tuner result output.
`TUNERSESSIONNAME`	When defined, will override the problem name within the tuner.

Example 1 (Console)

```
TUNE -l
```

This tunes the current problem as an LP problem.

Example 2 (Console)

```
TUNE pm
```

```
TUNE printmethod
```

Both commands print the tuner method to the console.

Example 3 (Console)

```
TUNE rm method
```

```
TUNE readmethod  method
```

Both commands read the tuner method from the `method.xtm` file.

Example 4 (Console)

```
TUNE wm method
```

```
TUNE writemethod method
```

Both commands write the tuner method to the `method.xtm` file.

Example 5 (Console)

```
TUNE probset problem.set
```

Tune a set of problems defined by the `problem.set` file.

Example 6 (Console)

```
TUNE lpset problem.set
```

Tune a set of LP problems defined by the `problem.set` file.

Further information

1. When both flags and subcommand are provided with the `TUNE` command, the subcommand will be ignored.
2. Please refer to Section 5.12 for a detailed guide of how to use the tuner.
3. Please refer to Section 5.12.8 for more information about tuning a set of problems.

XPRStune

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

```
int XPRS_CC XPRStune(XPRSprob prob, const char *flags);
```

Arguments

`prob`	The current problem.
`flags`	Flags to pass to `XPRStune`, 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`); `d` will use the dual simplex method; `p` will use the primal simplex method; `b` will use the barrier method; `n` will use the network simplex method.

Example

```
XPRStune(prob, "dp");
```

This tunes the current problem. The problem type is automatically determined. If it is an LP problem, it will be solved with a concurrent run of the dual and primal simplex method. If it is a MIP problem, the initial LP relaxation of the MIP will be solved with a concurrent run of primal and dual simplex.

Further information

1. Please refer to command `TUNE` for a list of related controls.
2. Please refer to Section 5.12 for a detailed guide of how to use the tuner.

XPRStunerreadmethod

Purpose

This function loads a user defined tuner method from the given file.

Synopsis

```
int XPRS_CC XPRStunerreadmethod(XPRSprob prob, const char* methodfile);
```

Arguments

`prob`	The current problem.
`methodfile`	The method file name, from which the tuner can load a user-defined tuner method.

Example

```
XPRStunerreadmethod(prob, "method.xtm");
```

This loads the tuner method from the `method.xtm` file.

Further information

Please refer to Section 5.12.2 for more information about the tuner method, and Appendix A.9 for the format of the tuner method file.

XPRStunerwritemethod

Purpose

This function writes the current tuner method to a given file or prints it to the console.

Synopsis

```
int XPRS_CC XPRStunerwritemethod(XPRSprob prob, const char* methodfile);
```

Arguments

`prob`	The current problem.
`methodfile`	The method file name, to which the tuner will write the current tuner method. If the input is `stdout` or `STDOUT`, then the tuner will print the method to the console instead.

Example 1 (Library)

```
XPRStunerwritemethod(prob, "method.xtm");
```

This writes the tuner method to the `method.xtm` file.

Example 2 (Library)

```
XPRStunerwritemethod(prob, "stdout");
```

This prints the tuner method to the console.

Further information

Please refer to Section 5.12.2 for more information about the tuner method, and Appendix A.9 for the format of the tuner method file.

XPRSunloadprob

Purpose

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

```
int XPRS_CC XPRSunloadprob(XPRSprob prob);
```

Argument

`prob`	The current problem.

Related topics

`XPRSreadprob`, `XPRSloadlp`, `XPRSloadglobal`, `XPRSloadqglobal`, `XPRSloadqp`.

XPRSwritebasis WRITEBASIS

Purpose

Writes the current basis to a file for later input into the Optimizer.

Synopsis

```
int XPRS_CC XPRSwritebasis(XPRSprob prob, const char *filename, const char
        *flags);
WRITEBASIS [-flags] [filename]
```

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. If omitted, the default *problem_name* is used with a `.bss` extension.
`flags`	Flags to pass to `XPRSwritebasis` (`WRITEBASIS`): `i` output the internal presolved basis. `t` output a compact advanced form of the basis. `n` output basis file containing current solution values. `h` output values in single precision. `p` obsolete flag (now default behavior).

Example 1 (Library)

After an LP has been solved it may be desirable to save the basis for future input as an advanced starting point for other similar problems. This may save significant amounts of time if the LP is complex. The Optimizer input commands might then be:

```
XPRSreadprob(prob, "myprob", "");
XPRSlpoptimize(prob, "");
XPRSwritebasis(prob, "", "");
```

This reads in a matrix file, maximizes the LP and saves the basis. Loading a basis for a MIP problem can disable some MIP presolve operations which can result in a large increase in solution times so it is generally not recommended.

Example 2 (Console)

An equivalent set of commands to the above for console users would be:

```
READPROB
LPOPTIMIZE
WRITEBASIS
```

Further information

1. The `t` flag is only useful for later input to a similar problem using the `t` flag with `XPRSreadbasis` (`READBASIS`).

2. If the Newton barrier algorithm has been used for optimization then crossover must have been performed before there is a valid basis. This basis can then only be used for restarting the simplex (primal or dual) algorithm.

3. `XPRSwritebasis` (`WRITEBASIS`) will output the basis for the original problem even if the matrix has been presolved.

Related topics

`XPRSgetbasis`, `XPRSreadbasis` (`READBASIS`).

XPRSwritebinsol WRITEBINSOL

Purpose

Writes the current MIP or LP solution to a binary solution file for later input into the Optimizer.

Synopsis

```
int XPRS_CC XPRSwritebinsol(XPRSprob prob, const char *filename, const char
      *flags);
WRITEBINSOL [-flags] [filename]
```

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. If omitted, the default *problem_name* is used with a `.sol` extension.
`flags`	Flags to pass to `XPRSwritebinsol` (`WRITEBINSOL`): `x` output the LP solution.

Example 1 (Library)

After an LP has been solved or a MIP solution has been found the solution can be saved to file. If a MIP solution exists it will be written to file unless the -x flag is passed to `XPRSwritebinsol` (`WRITEBINSOL`) in which case the LP solution will be written. The Optimizer input commands might then be:

```
XPRSreadprob(prob, "myprob", "");
XPRSmipoptimize(prob, "");
XPRSwritebinsol(prob, "", "");
```

This reads in a matrix file, maximizes the MIP and saves the last found MIP solution.

Example 2 (Console)

An equivalent set of commands to the above for console users would be:

```
READPROB
MIPOPTIMIZE
WRITEBINSOL
```

Related topics

`XPRSgetlpsol`, `XPRSgetmipsol`, `XPRSreadbinsol` (`READBINSOL`), `XPRSwritesol` (`WRITESOL`), `XPRSwriteprtsol` (`WRITEPRTSOL`).

XPRSwritedirs WRITEDIRS

Purpose

Writes the global search directives from the current problem to a directives file.

Synopsis

```
int XPRS_CC XPRSwritedirs(XPRSprob prob, const char *filename);
WRITEDIRS [filename]
```

Arguments

`prob`	The current problem.
`filename`	A string of up to `MAXPROBNAMELENGTH` characters containing the file name to which the directives should be written. If omitted (or NULL), the default *problem_name* is used with a `.dir` extension.

Further information

If the problem has been presolved, only the directives for columns in the presolved problem will be written to file.

Related topics

`XPRSloaddirs`, A.6.

XPRSwriteprob WRITEPROB

Purpose

Writes the current problem to an MPS or LP file.

Synopsis

```
int XPRS_CC XPRSwriteprob(XPRSprob prob, const char *filename, const char
        *flags);
WRITEPROB [-flags] [filename]
```

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. If omitted, the default *problem_name* is used with a `.mps` extension, unless the `l` flag is used in which case the extension is `.lp`.
`flags`	Flags, which can be one or more of the following: `o` one element per line; `n` scaled; `s` scrambled vector names; `l` output in LP format; `p` full precision of numerical values (obsolete as this is now default behavior). `t` omit the Xpress header in LP format;

Example

The following example outputs the current problem in LP format with scrambled vector names to the file *problem_name*`.lp`.

```
XPRSwriteprob(prob, "", "ls");
```

Further information

1. If `XPRSloadlp`, `XPRSloadglobal`, `XPRSloadqglobal` or `XPRSloadqp` is used to obtain a matrix then there is no association between the objective function and the `N` rows in the matrix and so a separate `N` row (called `__OBJ___`) is created when you do an `XPRSwriteprob` (`WRITEPROB`). Also if you do an `XPRSreadprob` (`READPROB`) and then change either the objective row or the `N` row in the matrix corresponding to the objective row, you lose the association between the two and the `__OBJ___` row is created when you do an `XPRSwriteprob` (`WRITEPROB`). To remove the objective row from the matrix when doing an `XPRSreadprob` (`READPROB`), set `KEEPNROWS` to `-1` before `XPRSreadprob` (`READPROB`).
2. **Warning:** If `XPRSreadprob` (`READPROB`) is used to input a problem, then the input file will be overwritten by `XPRSwriteprob` (`WRITEPROB`) if a new filename is not specified.

Related topics

`XPRSreadprob` (`READPROB`).

XPRSwriteprtrange WRITEPRTRANGE

Purpose

Writes the ranging information to a fixed format ASCII file, *problem_name*`.rrt`. The binary range file (`.rng`) must already exist, created by `XPRSrange` (`RANGE`).

Synopsis

```
int XPRS_CC XPRSwriteprtrange(XPRSprob prob);
WRITEPRTRANGE
```

Argument

`prob`	The current problem.

Related controls

Integer

`MAXPAGELINES`	Number of lines between page breaks.

Double

`OUTPUTTOL`	Tolerance on print values.

Example 1 (Library)

The following example solves the LP problem and then calls `XPRSrange` (`RANGE`) before outputting the result to file for printing:

```
XPRSreadprob(prob, "myprob", "");
XPRSlpoptimize(prob, "");
XPRSrange(prob);
XPRSwriteprttange(prob);
```

Example 2 (Console)

An equivalent set of commands for the Console user would be:

```
READPROB
LPOPTIMIZE
RANGE
WRITEPRTRANGE
```

Further information

1. (*Console*) There is an equivalent command `PRINTRANGE` which outputs the same information to the screen. The format is the same as that output to file by `XPRSwriteprtrange` (`WRITEPRTRANGE`), except that the user is permitted to enter a response after each screen if further output is required.
2. The fixed width ASCII format created by this command is not as readily useful as that produced by `XPRSwriterange` (`WRITERANGE`). The main purpose of `XPRSwriteprtrange` (`WRITEPRTRANGE`) is to create a file that can be printed. The format of this fixed format range file is described in Appendix A.

Related topics

`XPRSgetcolrange`, `XPRSgetrowrange`, `XPRSrange` (`RANGE`), `XPRSwriteprtsol`, `XPRSwriterange`, A.6.

XPRSwriteprtsol WRITEPRTSOL

Purpose

Writes the current solution to a fixed format ASCII file, *problem_name* `.prt`.

Synopsis

```
int XPRS_CC XPRSwriteprtsol(XPRSprob prob, const char *filename, const char
      *flags);
WRITEPRTSOL [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. If omitted, the default *problem_name* will be used. The extension `.prt` will be appended.
`flags`	Flags for `XPRSwriteprtsol` (`WRITEPRTSOL`) are: `x` write the LP solution instead of the current MIP solution.

Related controls

Integer

`MAXPAGELINES`	Number of lines between page breaks.

Double

`OUTPUTTOL`	Tolerance on print values.

Example 1 (Library)

This example shows the standard use of this function, outputting the solution to file immediately following optimization:

```
XPRSreadprob(prob, "myprob", "");
XPRSlpoptimize(prob, "");
XPRSwriteprtsol(prob, "", "");
```

Example 2 (Console)

```
READPROB
LPOPTIMIZE
PRINTSOL
```

are the equivalent set of commands for Console users who wish to view the output directly on screen.

Further information

1. (*Console*) There is an equivalent command `PRINTSOL` which outputs the same information to the screen. The format is the same as that output to file by `XPRSwriteprtsol` (`WRITEPRTSOL`), except that the user is permitted to enter a response after each screen if further output is required.

2. The fixed width ASCII format created by this command is not as readily useful as that produced by `XPRSwritesol` (`WRITESOL`). The main purpose of `XPRSwriteprtsol` (`WRITEPRTSOL`) is to create a file that can be sent directly to a printer. The format of this fixed format ASCII file is described in Appendix A.

3. To create a prt file for a previously saved solution, the solution must first be loaded with the `XPRSreadbinsol` (`READBINSOL`) function.

Related topics

`XPRSgetlpsol`, `XPRSgetmipsol`, `XPRSreadbinsol` `XPRSwritebinsol`, `XPRSwriteprtrange`, `XPRSwritesol`, A.4.

XPRSwriterange WRITERANGE

Purpose

Writes the ranging information to a CSV format ASCII file, *problem_name*`.rsc` (and `.hdr`). The binary range file (`.rng`) must already exist, created by `XPRSrange` (`RANGE`) and an associated header file.

Synopsis

```
int XPRS_CC XPRSwriterange(XPRSprob prob, const char *filename, const char
        *flags);
WRITERANGE [filename] [-flags]
```

Arguments

`prob`	The current problem.
`filename`	A string of up to `MAXPROBNAMELENGTH` characters containing the file name to which the ranging information is to be written. If omitted, the default *problem_name* will be used. The extensions `.hdr` and `.rsc` will be appended to the filename.
`flags`	Flags to control which optional fields are output: `s` sequence number; `n` name; `t` type; `b` basis status; `a` activity; `c` cost (column), slack (row). If no flags are specified, all fields are output.

Related controls

Double

`OUTPUTTOL`	Tolerance on print values.

String

`OUTPUTMASK`	Mask to restrict the row and column names output to file.

Example 1 (Library)

At its most basic, the usage of `XPRSwriterange` (`WRITERANGE`) is similar to that of `XPRSwriteprtrange` (`WRITEPRTRANGE`), except that the output is intended as input to another program. The following example shows its use:

```
XPRSreadprob(prob, "myprob", "");
XPRSlpoptimize(prob, "");
XPRSrange(prob);
XPRSwriterange(prob, "", "");
```

Example 2 (Console)

```
RANGE
WRITERANGE -nbac
```

This example would output just the name, basis status, activity, and cost (for columns) or slack (for rows) for each vector to the file *problem_name*`.rsc`. It would also output a number of other fields of ranging information which cannot be enabled/disabled by the user.

Further information

1. The following fields are always present in the `.rsc` file, in the order specified. See the description of the ASCII range files in Appendix A for details of their interpretation.For rows, the lower and upper cost entries are zero. If a limiting process or activity does not exist, the field is blank, delimited by double quotes.
   - lower activity
   - unit cost down
   - upper cost (or lower profit if maximizing)
   - limiting process down
   - status of down limiting process
   - upper activity
   - unit cost up
   - lower cost (or upper profit if maximizing)
   - limiting process up
   - status of up limiting process
2. The control `OUTPUTMASK` may be used to control which vectors are reported to the ASCII file. Only vectors whose names match `OUTPUTMASK` are output. This is set to "`????????`" by default, so that all vectors are output.

Related topics

`XPRSgetlpsol`, `XPRSgetmipsol`, `XPRSwriteprtrange` (`WRITEPRTRANGE`), `XPRSrange` (`RANGE`), `XPRSwritesol` (`WRITESOL`), A.6.

XPRSwriteslxsol 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 `XPRSreadslxsol` function.

Synopsis

```
int XPRS_CC XPRSwriteslxsol(XPRSprob prob, const char *filename, const char
      *flags);
WRITESLXSOL -[flags] [filename]
```

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. If omitted, the default *problem_name* is used with a `.slx` extension.
`flags`	Flags to pass to `XPRSwriteslxsol` (`WRITESLXSOL`): `l` write the LP solution in case of a MIP problem; `m` write the MIP solution; `p` use full precision for numerical values; `d` LP solution only: including dual variables; `s` LP solution only: including slack variables; `r` LP solution only: including reduced cost.

Example 1 (Library)

```
XPRSwriteslxsol(prob,"lpsolution","");
```

This saves the MIP solution if the problem contains global entities, or otherwise saves the LP (barrier in case of quadratic problems) solution of the problem.

Example 2 (Console)

```
WRITESLXSOL lpsolution
```

Which is equivalent to the library example above.

Related topics

`XPRSreadslxsol` (`READSLXSOL`, `XPRSwriteprtsol` (`WRITEPRTSOL`), `XPRSwritebinsol` (`WRITEBINSOL`), `XPRSreadbinsol` (`READBINSOL`).

XPRSwritesol WRITESOL

Purpose

Writes the current solution to a CSV format ASCII file, *problem_name*`.asc` (and `.hdr`).

Synopsis

```
int XPRS_CC XPRSwritesol(XPRSprob prob, const char *filename, const char
        *flags);
WRITESOL [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. If omitted, the default *problem_name* will be used. The extensions `.hdr` and `.asc` will be appended.
`flags`	Flags to control which optional fields are output: `s` sequence number; `n` name; `t` type; `b` basis status; `a` activity; `c` cost (columns), slack (rows); `l` lower bound; `u` upper bound; `d` dj (column; reduced costs), dual value (rows; shadow prices); `r` right hand side (rows). If no flags are specified, all fields are output. Additional flags: `p` outputs in full precision; `q` only outputs vectors with nonzero optimum value; `x` output the current LP solution instead of the MIP solution.

Related controls

Double

OUTPUTTOL	Tolerance on print values.

String

OUTPUTMASK	Mask to restrict the row and column names output to file.

Example 1 (Library)

In this example the basis status is output (along with the sequence number) following optimization:

```
XPRSreadprob(prob, "richard", "");
XPRSlpoptimize(prob, "");
XPRSwritesol(prob, "", "sb");
```

Example 2 (Console)

Suppose we wish to produce files containing

- the names and values of variables starting with the letter `X` which are nonzero and
- the names, values and right hand sides of constraints starting with `CO2`.

The Optimizer commands necessary to do this are:

```
OUTPUTMASK = "X???????"
WRITESOL XVALS -naq
OUTPUTMASK = "CO2?????"
WRITESOL CO2 -nar
```

Further information

1. The command produces two readable files: `filename.hdr` (the solution header file) and `filename.asc` (the CSV foramt solution file). The header file contains summary information, all in one line. The ASCII file contains one line of information for each row and column in the problem. Any fields appearing in the `.asc` file will be in the order the flags are described above. The order that the flags are specified by the user is irrelevant.

2. Additionally, the mask control `OUTPUTMASK` may be used to control which names are reported to the ASCII file. Only vectors whose names match `OUTPUTMASK` are output. `OUTPUTMASK` is set by default to "`????????`", so that all vectors are output.

Related topics

`XPRSgetlpsol`, `XPRSgetmipsol`, `XPRSwriterange` (`WRITERANGE`), `XPRSwriteprtsol` (`WRITEPRTSOL`).

CHAPTER 9

Control Parameters

Various controls exist within the Optimizer to govern the solution procedure and the form of output. The majority of these take integer values and act as switches between various types of behavior. The tolerances on values are double precision, and there are a few controls which are character strings, setting names to structures. Any of these may be altered by the user to enhance performance of the Optimizer. However, it should be noted that the default values provided have been found to work well in practice over a range of problems and caution should be exercised if they are changed.

9.1 Retrieving and Changing Control Values

Console Xpress users may obtain control values by issuing the control name at the Optimizer prompt, >, and hitting the RETURN key. Controls may be set using the assignment syntax:

control_name = *new_value*

where *new_value* is an integer value, double or string as appropriate. For character strings, the name must be enclosed in single quotes and all eight characters must be given.

Users of the FICO Xpress Libraries are provided with the following set of functions for setting and obtaining control values:

XPRSgetintcontrol	XPRSgetdblcontrol	XPRSgetstrcontrol
XPRSsetintcontrol	XPRSsetdblcontrol	XPRSsetstrcontrol

It is an important point that the controls as listed in this chapter *must* be prefixed with XPRS_ to be used with the FICO Xpress Libraries and failure to do so will result in an error. An example of their usage is as follows:

```
XPRSgetintcontrol(prob, XPRS_PRESOLVE, &presolve);
printf("The value of PRESOLVE is %d\n", presolve);
XPRSsetintcontrol(prob, XPRS_PRESOLVE, 1-presolve);
printf("The value of PRESOLVE is now %d\n", 1-presolve);
```

ALGAFTERCROSSOVER

Description The algorithm to be used for the final clean up step after the crossover.

Type Integer

Values

`1`	Automatically determined.
`2`	Dual simplex.
`3`	Primal simplex.
`4`	Concurrent.

Default value `1`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`) (when the barrier is used), `XPRScrossoverlpsol`.

ALGAFTERNETWORK

Description The algorithm to be used for the clean up step after the network simplex solver.

Type Integer

Values

`-1`	Automatically determined.
`2`	Dual simplex.
`3`	Primal simplex.

Default value `-1`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`) (when the barrier is used).

AUTOSCALING

Description Whether the Optimizer should automatically select between different scaling algorithms. If the `SCALING` control is set, no automatic scaling will be applied.

Values

`-1`	Automatic.
`0`	Disabled.
`1`	Cautious strategy. Non-standard scaling will only be selected if it appears to be clearly superior.
`2`	Moderate strategy.
`3`	Aggressive strategy. Standard scaling will only be selected if it appears to be clearly superior.

Default value -1

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `SCALING`, `OBJSCALEFACTOR`.

AUTOPERTURB

Description Simplex: This indicates whether automatic perturbation is performed. If this is set to `1`, the problem will be perturbed whenever the simplex method encounters an excessive number of degenerate pivot steps, thus preventing the Optimizer being hindered by degeneracies.

Type	Integer	
Values	0	No perturbation performed.
	1	Automatic perturbation is performed.
Default value	1	
Affects routines	XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).	

BACKTRACK

Description	Branch and Bound: Specifies how to select the next node to work on when a full backtrack is performed.	
Type	Integer	
Values	-1	Automatically determined.
	1	Unused.
	2	Select the node with the best estimated solution.
	3	Select the node with the best bound on the solution.
	4	Select the deepest node in the search tree (equivalent to depth-first search).
	5	Select the highest node in the search tree (equivalent to breadth-first search).
	6	Select the earliest node created.
	7	Select the latest node created.
	8	Select a node randomly.
	9	Select the node whose LP relaxation contains the fewest number of infeasible global entities.
	10	Combination of 2 and 9.
	11	Combination of 2 and 4.
	12	Combination of 3 and 4.
Default value	3	
Note	Note When two nodes are rated the same according to the BACKTRACK selection, a secondary rating is performed using the method set by BACKTRACKTIE.	
Affects routines	XPRSmipoptimize (MIPOPTIMIZE).	
See also	BACKTRACKTIE.	

BACKTRACKTIE

Description	Branch and Bound: Specifies how to break ties when selecting the next node to work on when a full backtrack is performed. The options are the same as for the BACKTRACK control.
Type	Integer

Values	`-1`	Default selection.
	`1`	Unused.
	`2`	Select the node with the best estimated solution.
	`3`	Select the node with the best bound on the solution.
	`4`	Select the deepest node in the search tree (equivalent to depth-first search).
	`5`	Select the highest node in the search tree (equivalent to breadth-first search).
	`6`	Select the earliest node created.
	`7`	Select the latest node created.
	`8`	Select a node randomly.
	`9`	Select the node whose LP relaxation contains the fewest number of infeasible global entities.
	`10`	Combination of 2 and 9.
	`11`	Combination of 2 and 4.
	`12`	Combination of 3 and 4.

Default value `-1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `BACKTRACK`.

BARALG

Description This control determines which barrier algorithm is to be used to solve the problem.

Type Integer

Values	`-1`	Determined automatically.
	`0`	Unused.
	`1`	Use the infeasible-start barrier algorithm.
	`2`	Use the homogeneous self-dual barrier algorithm.
	`3`	Start with 2 and optionally switch to 1 during the execution.

Default value `-1`

Note The automatic setting uses 1 for LP and QP problems and 3 for QCQP problems. Usually the detection of primal or dual infeasibility is more robust with settings 2 or 3, therefore, it is advantageous to use one of these values if the model is presumably infeasible.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`), `XPRSminim` (`MINIM`), `XPRSmaxim` (`MAXIM`), `XPRSglobal` (`GLOBAL`).

BARCRASH

Description Newton barrier: This determines the type of crash used for the crossover. During the crash procedure, an initial basis is determined which attempts to speed up the crossover. A good choice at this stage will significantly reduce the number of iterations required to crossover to an optimal solution. The possible values increase proportionally to their time-consumption.

Type Integer

Values

`0`	Turns off all crash procedures.
`1-6`	Available strategies with 1 being conservative and 6 being aggressive.

Default value `4`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

BARDUALSTOP

Description Newton barrier: This is a convergence parameter, representing the tolerance for dual infeasibilities. If the difference between the constraints and their bounds in the dual problem falls below this tolerance in absolute value, optimization will stop and the current solution will be returned.

Type Double

Default value `0` (determine automatically)

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

BARFREESCALE

Description Defines how the barrier algorithm scales free variables.

Type Double

Default value `1e-6`

Note When using smaller values the barrier algorithm scales free variables more aggressively which can improve performance but may impact numerical stability.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`).

See also `SCALING`.

BARGAPSTOP

Description Newton barrier: This is a convergence parameter, representing the tolerance for the relative duality gap. When the difference between the primal and dual objective function values falls below this tolerance, the Optimizer determines that the optimal solution has been found.

Type Double

Default value `0` (determine automatically)

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

BARGAPTARGET

Description Newton barrier: The target tolerance for the relative duality gap. The barrier algorithm will keep iterating until either BARGAPTARGET is satisfied or until no further improvements are possible. In the latter case, if BARGAPSTOP is satisfied, it will declare the problem optimal.

Type Double

Default value 0 (determine automatically)

Note When a solution returned by the barrier algorithm has not converged tightly enough for an application, for example if the dual solution is not accurate enough or crossover is taking too long, setter BARGAPTARGET to a small value often resolves the problem, without the risk of the solve failing due to a complementarity level not being numerically achievable. Typical suggested values can be between 1^{-10} and 1^{-18}.

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

BARFAILITERLIMIT

Description Newton barrier: The maximum number of consecutive iterations that fail to improve the solution in the barrier algorithm.

Type Integer

Values

0	Determined automatically
>0	Maximum number of consecutive barrier iterations allowed without progress.

Default value 0

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

BARINDEFLIMIT

Description Newton Barrier. This limits the number of consecutive indefinite barrier iterations that will be performed. The optimizer will try to minimize (resp. maximize) a QP problem even if the Q matrix is not positive (resp. negative) semi-definite. However, the optimizer may detect that the Q matrix is indefinite and this can result in the optimizer not converging. This control specifies how many indefinite iterations may occur before the optimizer stops and reports that the problem is indefinite. It is usual to specify a value greater than one, and only stop after a series of indefinite matrices, as the problem may be found to be indefinite incorrectly on a few iterations for numerical reasons.

Type Integer

Default value 15

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

BARITERLIMIT

Description Newton barrier: The maximum number of iterations. While the simplex method usually performs a number of iterations which is proportional to the number of constraints (rows) in a problem, the barrier method standardly finds the optimal solution to a given accuracy after a number of iterations which is independent of the problem size. The penalty is rather that the time for each iteration increases with the size of the problem. `BARITERLIMIT` specifies the maximum number of iterations which will be carried out by the barrier.

Type Integer

Default value `500`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

BARKERNEL

Description Newton barrier: Defines how centrality is weighted in the barrier algorithm.

Type Double

Values

`>=+1.0`	Increases the emphasis on centrality when larger value is set.
`<=-1.0`	Selects a value adaptively in every iteration from [+1, -BARKERNEL].

Default value `0.0`

Note Increasing this parameter may increase the number of iterations, therefore the recommended range is [1,2] and [-2,-1].

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`).

See also `BARALG`.

BAROBJSCALE

Description Defines how the barrier scales the objective.

Type Double

Values

`-1`	Let the optimizer decide.
`0`	Scale by geometric mean.
`>=0`	Scale such that the largest objective coefficient's largest element does not exceed this number. In quadratic problems, the quadratic diagonal is used as reference valuses instead of the linear objective.

Default value `-1`

Note The scaling perfromed by the barrier is applied on top of any other scaling in the problem and only affects the barrier solve.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`).

See also `SCALING`.

BARORDER

Description Newton barrier: This controls the Cholesky factorization in the Newton-Barrier.

Type Integer

Values

0	Choose automatically.
1	Minimum degree method. This selects diagonal elements with the smallest number of nonzeros in their rows or columns.
2	Minimum local fill method. This considers the adjacency graph of nonzeros in the matrix and seeks to eliminate nodes that minimize the creation of new edges.
3	Nested dissection method. This considers the adjacency graph and recursively seeks to separate it into non-adjacent pieces.

Default value 0

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

BARORDERTHREADS

Description If set to a positive integer it determines the number of concurrent threads for the sparse matrix ordering algorithm in the Newton-barrier method.

Type Integer

Default value 0 (determine automatically)

Note Larger values than `BARCORES` will be automatically reduced to the value of `BARCORES`.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `BARORDER`, `BARCORES`.

BAROUTPUT

Description Newton barrier: This specifies the level of solution output provided. Output is provided either after each iteration of the algorithm, or else can be turned off completely by this parameter.

Type Integer

Values

0	No output.
1	At each iteration.

Default value 1

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

BARPRESOLVEOPS

Description Newton barrier: This controls the Newton-Barrier specific presolve operations.

Type Integer

Values

0	Use standard presolve.
1	Extra effort is spent in barrier specific presolve.
2	Do full matrix eliminations (reduce matrix size).

Default value 0

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

BARPRIMALSTOP

Description Newton barrier: This is a convergence parameter, indicating the tolerance for primal infeasibilities. If the difference between the constraints and their bounds in the primal problem falls below this tolerance in absolute value, the Optimizer will terminate and return the current solution.

Type Double

Default value 0 (determine automatically)

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

BARREGULARIZE

Description This control determines how the barrier algorithm applies regularization on the KKT system.

Type Integer

Values

-1	Determined automatically.

Values

Bit	Meaning
0	Standard regularization is turned on/off.
1	Reduced regularization is turned on/off. This option reduces the perturbation effect of the standard regularization.
2	Forces to keep dependent rows in the KKT system.
3	Forces to preserve degenerate rows in the KKT system.

Default value -1

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE), XPRSminim (MINIM), XPRSmaxim (MAXIM), XPRSglobal (GLOBAL).

BARRHSSCALE

Description	Defines how the barrier scales the right hand side.
Type	Double
Values	`-1` Let the optimizer decide. `0` Scale by geometric mean. `>=0` Scale such that the largest right hand side coefficient's largest element does not exceed this number.
Default value	`-1`
Note	The scaling perfromed by the barrier is applied on top of any other scaling in the problem and only affects the barrier solve.
Affects routines	`XPRSlpoptimize` (`LPOPTIMIZE`).
See also	`SCALING`.

BARSOLUTION

Description	This determines whether the barrier has to decide which is the best solution found or return the solution computed by te last barrier iteration.
Type	Integer
Values	`-1` (callback only: do not save current soulution as the best one). `0` return the best solution found (in callback: let the barrier decide the current solution is the best or not). `1` return the last barrier iteration (in callback: save current solution as the best solution so far).
Default value	`0`
Affects routines	The barrier algorithm.

BARSTART

Description	Newton barrier: Controls the computation of the starting point for the barrier algorithm.
Type	Integer
Values	`-1` Uses the available solution for warm-start. `0` Determine automatically. `1` Uses simple heuristics to compute the starting point based on the magnitudes of the matrix entries. `2` Uses the pseudoinverse of the constraint matrix to determine primal and dual initial solutions. Less sensitive to scaling and numerically more robust, but in several case less efficient than 1. `3` Uses the unit starting point for the homogeneous self-dual barrier algorithm.

Default value 0

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

BARSTARTWEIGHT

Description Newton barrier: This sets a weight for the warm-start point when warm-start is set for the barrier algorithm. Using larger weight gives more emphasis for the supplied starting point.

Type Double

Default value 0.85

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

See also BARSTART.

BARSTEPSTOP

Description Newton barrier: A convergence parameter, representing the minimal step size. On each iteration of the barrier algorithm, a step is taken along a computed search direction. If that step size is smaller than BARSTEPSTOP, the Optimizer will terminate and return the current solution.

Type Double

Default value 1.0E-16

Note If the barrier method is making small improvements on BARGAPSTOP on later iterations, it may be better to set this value higher, to return a solution after a close approximation to the optimum has been found.

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

BARTHREADS

Description If set to a positive integer it determines the number of threads implemented to run the Newton-barrier algorithm. If the value is set to the default value (-1), the THREADS control will determine the number of threads used.

Type Integer

Default value -1(determined by the THREADS control)

Note There is a practical upper limit of 50 on the number of parallel threads the optimizer will create.

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

See also MIPTHREADS, CONCURRENTTHREADS, THREADS.

BARCORES

Description If set to a positive integer it determines the number of physical CPU cores assumed to be present in the system by the barrier algorithm. If the value is set to the default value (-1), Xpress will automatically detect the number of cores.

Type Integer

Default value -1(automatically detected)

Note The control is provided for cross-hardware reproducibility purposes. The count does not include logical cores created by Hyper-Threading.

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

See also BARTHREADS.

BIGM

Description The infeasibility penalty used if the "Big M" method is implemented.

Type Double

Default value Dependent on the matrix characteristics.

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

BIGMMETHOD

Description Simplex: This specifies whether to use the "Big M" method, or the standard phase I (achieving feasibility) and phase II (achieving optimality). In the "Big M" method, the objective coefficients of the variables are considered during the feasibility phase, possibly leading to an initial feasible basis which is closer to optimal. The side-effects involve possible round-off errors due to the presence of the "Big M" factor in the problem.

Type Integer

Values

0	For phase I / phase II.
1	If "Big M" method to be used.

Default value 1

Note Reset by XPRSreadprob (READPROB), XPRSloadglobal, XPRSloadlp, XPRSloadqglobal and XPRSloadqp.

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

BRANCHCHOICE

Description Once a global entity has been selected for branching, this control determines which of the branches is solved first.

Type Integer

Values

0	Minimum estimate branch first.
1	Maximum estimate branch first.
2	If an incumbent solution exists, solve the branch satisfied by that solution first. Otherwise solve the minimum estimate branch first (option 0).
3	Solve first the branch that forces the value of the branching variable to move farther away from the value it had at the root node. If the branching entity is not a simple variable, solve the minimum estimate branch first (option 0).

Default value 0

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

BRANCHDISJ

Description Branch and Bound: Determines whether the optimizer should attempt to branch on general split disjunctions during the branch and bound search.

Type Integer

Values

-1	Automatic selection of the strategy.
0	Disabled.
1	Cautious strategy. Disjunctive branches will be created only for general integers with a wide range.
2	Moderate strategy.
3	Aggressive strategy. Disjunctive branches will be created for both binaries and integers.

Default value -1

Note Note Split disjunctions are a special form of disjunctions that can be written as

$\sum_j m_j x_j \leq m_0 \vee \sum_j m_j x_j \geq m_0 + 1$

The split disjunctions created by the optimizer will use a combination of binary or integer variables x_j, with integer coefficients m_j.

Split disjunctions for branching will always be created with a default priority value of 400 instead of the default value of 500 for regular entity branches.

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

BRANCHSTRUCTURAL

Description Branch and Bound: Determines whether the optimizer should search for special structure in the problem to branch on during the branch and bound search.

Type Integer

Values

-1	Automatically determined.
0	Disabled.
1	Enabled.

Default value -1

Note Structural branches will often involve branching on more than a single global entity at a time. As a result of a structural branch, a parent node could therefore end up with more than two child nodes, unlike the standard single entity branches.

Structural branches will always be created with a default priority value of 400 instead of the default value of 500 for regular entity branches.

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

BREADTHFIRST

Description The number of nodes to include in the best-first search before switching to the local first search (`NODESELECTION` = 4).

Type Integer

Default value 11

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

CACHESIZE

Description Newton Barrier: L2 or L3 (see notes) cache size in kB (kilobytes) of the CPU. On Intel (or compatible) platforms a value of -1 may be used to determine the cache size automatically. If the CPU model is new then the cache size may not be correctly detected by an older release of the software.

Type Integer

Default value -1

Note Specifying the correct cache size can give a significant performance advantage with the Newton barrier algorithm. If the size is unknown, it is better to specify a smaller size.

If the size cannot be determined automatically, a default size of `128kB` is assumed.

Where present, the L3 cache size should be chosen rather than the L2 cache size.

For multi-core CPUs, the cache is shared between a subset of the cores. The Optimizer will divide the `CACHESIZE` value by the number of cores sharing the cache if >1 Barrier threads are running.

Where the CPU is described as having multiple caches ie. `2x6M` then the correct cache size to use is `6M` not `12M`.

Examples:

Intel Core 2 Duo E6400 (2M Cache, 2.13GHz), `CACHESIZE=2048` Intel Xeon x5570 (8M Cache, 2.93GHz), `CACHESIZE=8192` Intel Core 2 QX6700 (2x4M Cache, 2.93 GHz), `CACHESIZE=4096`

If in doubt, please contact Support for advice.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `L1CACHE`.

CALLBACKFROMMASTERTHREAD

Description Branch and Bound: specifies whether the MIP callbacks should only be called on the master thread.

Type Integer

Values

`0`	Invoke callbacks on worker threads during parallel MIP;
`1`	Only ever invoke a callback on the thread that called `XPRSmipoptimize`.

Default value 0

Affects routines `XPRSmipoptimize`.

CHOLESKYALG

Description Newton barrier: type of Cholesky factorization used.

Type Integer

Values

Bit	Meaning
0	matrix blocking: 0: automatic setting; 1: manual setting.
1	if manual selection of matrix blocking: 0: multi-pass; 1: single-pass.
2	nonseparable QP relaxation: 0: off; 1: on.
3	corrector weight: 0: automatic setting; 1: manual setting.
4	if manual selection of corrector weight: 0: off; 1: on.
5	refinement: 0: automatic setting; 1: manual setting.
6	preconditioned conjugate gradient method (PCGM): 0: PCGM off; 1: PCGM on.
7	Preconditioned quasi minimal residual (QMR) to refine solution: 0: QMR off; 1: QMR on.
8	Perform refinement on the augmented system 0: off; 1: on.
9	Force highest accuracy in refinement 0: off; 1: on.

Default value `-1 (automatic)`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

CHOLESKYTOL

Description Newton barrier: The tolerance for pivot elements in the Cholesky decomposition of the normal equations coefficient matrix, computed at each iteration of the barrier algorithm. If the absolute value of the pivot element is less than or equal to `CHOLESKYTOL`, it merits special treatment in the Cholesky decomposition process.

Type Double

Default value `1.0E-15`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

CLAMPING

Description This control allows the pushing the final solution values to be always within bounds.

Type Integer

Values

Value	
-1	Determined automatically.

Values

Bit	Meaning
0	Adjust primal solution to always be within primal bounds. Slacks if provided will be adjusted accordingly.
1	Adjust primal slack values to always be within constraint bounds.
2	Adjust dual solution to always be within the dual bounds implied by the slacks. Reduced costs, if provided, will be adjusted accordingly.
3	Adjust reduced costs to always be within dual bounds implied by the primal solution.

Default value 0

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`), `XPRSminim` (`MINIM`), `XPRSmaxim` (`MAXIM`), `XPRSglobal` (`GLOBAL`).

COMPUTEEXECSERVICE

Description Selects the Insight execution service that will be used for solving remote optimizations.

Type String

Default value Empty string

Note Set to the name of the execution service you want to use.

Note When an empty string, the Insight server's default execution service will be used.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

CONFLICTCUTS

Description Branch and Bound: Specifies how cautious or aggressive the optimizer should be when searching for and applying conflict cuts. Conflict cuts are in-tree cuts derived from nodes found to be infeasible or cut off, which can be used to cut off other branches of the search tree.

Type Integer

Values

Value	
-1	Automatic.
0	Disable conflict cuts.
1	Cautious application of conflict cuts.
2	Medium application of conflict cuts.
3	Aggressive application of conflict cuts.

Default value -1

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`)

CONCURRENTTHREADS

Description Determines the number of threads used by the concurrent solver.

Type Integer

Values

`-1`	Determined automatically
`>0`	Number of threads to use.

Default value `-1`

Note Please refer to section 5.10.1 for a detailed description of the concurrent solver.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`).

See also `DETERMINISTIC`, `DUALTHREADS`, `BARTHREADS`, `THREADS`.

CORESPERCPU

Description Used to override the detected value of the number of cores on a CPU. The cache size (either detected or specified via the `CACHESIZE` control) used in Barrier methods will be divided by this amount, and this scaled-down value will be the amount of cache allocated to each Barrier thread

Type Integer

Default value `-1`

Affects routines `CACHESIZE`

COVERCUTS

Description Branch and Bound: The number of rounds of lifted cover inequalities at the top node. A lifted cover inequality is an additional constraint that can be particularly effective at reducing the size of the feasible region without removing potential integral solutions. The process of generating these can be carried out a number of times, further reducing the feasible region, albeit incurring a time penalty. There is usually a good payoff from generating these at the top node, since these inequalities then apply to every subsequent node in the tree search.

Type Integer

Default value `-1` — determined automatically.

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

CPUPLATFORM

Description Newton Barrier: Selects the AMD or Intel x86 vectorization instruction set that Barrier should run optimized code for.

Type Integer

Values

-2	Highest supported [Generic, SSE2, AVX or AVX2].
-1	Highest supported solve path consistent code [Generic, SSE2 or AVX].
0	Use generic code compatible with all CPUs.
1	Use SSE2 optimized code.
2	Use AVX optimized code.
3	Use AVX2 optimized code.

Default value -1

Note Generic code, SSE2 and AVX optimized code will all result in the same solution path. Using AVX2 might result in a different solution path.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

CPUTIME

Description How time should be measured when timings are reported in the log and when checking against time limits

Type Integer

Values

-1	Disable the timer.
0	Use elapsed time.
1	Use process time.

Default value 0

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

CRASH

Description Simplex: This determines the type of crash used when the algorithm begins. During the crash procedure, an initial basis is determined which is as close to feasibility and triangularity as possible. A good choice at this stage will significantly reduce the number of iterations required to find an optimal solution. The possible values increase proportionally to their time-consumption.

Type Integer

Values	`0`	Turns off all crash procedures.
	`1`	For singletons only (one pass).
	`2`	For singletons only (multi pass).
	`3`	Multiple passes through the matrix considering slacks.
	`4`	Multiple ($\leq$ `10`) passes through the matrix but only doing slacks at the very end.
	`n>10`	As for value `4` but performing at most *n* - `10` passes.

Default value `2`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

CROSSOVER

Description Newton barrier: This control determines whether the barrier method will cross over to the simplex method when at optimal solution has been found, to provide an end basis (see `XPRSgetbasis`, `XPRSwritebasis`) and advanced sensitivity analysis information (see `XPRSrange`).

Type Integer

Values	`-1`	Determined automatically.
	`0`	No crossover.
	`1`	Primal crossover first.
	`2`	Dual crossover first.

Default value `-1`

Note The full primal and dual solution is available whether or not crossover is used. The crossover must not be disabled if the barrier is used to reoptimize nodes of a MIP. By default crossover will not be performed on QP and MIQP problems.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

CROSSOVERACCURACYTOL

Description Newton barrier: This control determines how crossover adjusts the default relative pivot tolerance. When re-inversion is necessary, crossover will compare the recalculated working basic solution with the assumed ones just before re-inversion took place. If the error is above this threshold, crossover will adjust the relative pivot tolerance to address the build-up of numerical inaccuracies.

Type Double

Default value `1e-6`

Note The full primal and dual solution is available whether or not crossover is used. The crossover must not be disabled if the barrier is used to reoptimize nodes of a MIP. By default crossover will not be performed on QP and MIQP problems.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

CROSSOVERITERLIMIT

Description	Newton barrier: The maximum number of iterations that will be performed in the crossover procedure before the optimization process terminates.
Type	Integer
Default value	`2147483647`
Affects routines	`XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).
See also	`CROSSOVER`.

CROSSOVEROPS

Description Newton barrier: a bit vector for adjusting the behavior of the crossover procedure.

Type Integer

Values

Bit	Meaning
0	Returned solution when the crossover terminates prematurely: 0: Return the last basis from the crossover; 1: Return the barrier solution.
1	Select the crossover stages to be performed: 0: Perform both crossover stages; 1: Skip second crossover stage.
2	Set crossover behaviour: 0: Force to perform all pivots; 1: Skip pivots that are numerically less reliable.
3	Set crossover behaviour: 0: Perform standard crossover; 1: Perform a slower, but numerically more careful crossover.

Default value `0`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `CROSSOVER`.

CROSSOVERTHREADS

Description	Determines the maximum number of threads that parallel crossover is allowed to use. If `CROSSOVERTHREADS` is set to the default value (`-1`), the `BARTHREADS` control will determine the number of threads used.
Type	Integer
Default value	`-1` (determined by the `BARTHREADS` control)
Affects routines	`XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).
See also	`BARTHREADS`, `CONCURRENTTHREADS`, `THREADS`.

CSTYLE

Description This control is deprecated, and will be removed from future versions of the optimizer. The control was used for numbering arrays.

Type Integer

Values

0	Indicates that the FORTRAN convention should be used for arrays (i.e. starting from 1).
1	Indicates that the C convention should be used for arrays (i.e. starting from 0).

Default value 1

Affects routines All library routines which take arrays as arguments.

CUTDEPTH

Description Branch and Bound: Sets the maximum depth in the tree search at which cuts will be generated. Generating cuts can take a lot of time, and is often less important at deeper levels of the tree since tighter bounds on the variables have already reduced the feasible region. A value of 0 signifies that no cuts will be generated.

Type Integer

Default value -1 — determined automatically.

Note Does not affect cutting on the root node.

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also CUTFREQ.

CUTFACTOR

Description Limit on the number of cuts and cut coefficients the optimizer is allowed to add to the matrix during global search. The cuts and cut coefficients are limited by CUTFACTOR times the number of rows and coefficients in the initial matrix.

Type Double

Values

Bit	Meaning
-1	Let the optimizer decide on the maximum amount of cuts based on CUTSTRATEGY.
>=0	Multiple of number of rows and coefficients to use.

Default value -1

Note A value of 0.0 prevents cuts from being added, and a value of e.g. 1.0 will allow the problem to grow to twice the initial number of rows and coefficients.

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also CUTSTRATEGY.

CUTFREQ

Description	Branch and Bound: This specifies the frequency at which cuts are generated in the tree search. If the depth of the node modulo `CUTFREQ` is zero, then cuts will be generated.
Type	Integer
Default value	`-1` — determined automatically.
Affects routines	`XPRSmipoptimize` (`MIPOPTIMIZE`).

CUTSTRATEGY

Description	Branch and Bound: This specifies the cut strategy. A more aggressive cut strategy, generating a greater number of cuts, will result in fewer nodes to be explored, but with an associated time cost in generating the cuts. The fewer cuts generated, the less time taken, but the greater subsequent number of nodes to be explored.
Type	Integer
Values	`-1` Automatic selection of the cut strategy. `0` No cuts. `1` Conservative cut strategy. `2` Moderate cut strategy. `3` Aggressive cut strategy.
Default value	`-1`
Affects routines	`XPRSmipoptimize` (`MIPOPTIMIZE`).

CUTSELECT

Description	A bit vector providing detailed control of the cuts created for the root node of a global solve. Use `TREECUTSELECT` to control cuts during the tree search.
Type	Integer

Values	Bit	Meaning
	5	Clique cuts.
	6	Mixed Integer Rounding (MIR) cuts.
	7	Lifted cover cuts.
	8	Turn on row aggregation for MIR cuts.
	11	Flow path cuts.
	12	Implication cuts.
	13	Turn on automatic Lift-and-Project cutting strategy.
	14	Disable cutting from cut rows.
	15	Lifted GUB cover cuts.
	16	Zero-half cuts.
	17	Indicator constraint cuts.

Default value -1

Note The default value is -1 which enables all bits. Any bits not listed in the above table should be left in their default 'on' state, since the interpretation of such bits might change in future versions of the optimizer.

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also COVERCUTS, GOMCUTS, TREECUTSELECT.

DEFAULTALG

Description This selects the algorithm that will be used to solve the LP if no algorithm flag is passed to the optimization routines.

Type Integer

Values	1	Automatically determined.
	2	Dual simplex.
	3	Primal simplex.
	4	Newton barrier.

Default value 1

Note Please note that this will affect how the MIP node LP problems are solved during the global search. To change how the root LP is solved only, please use the appropriate flags to XPRSlpoptimize or XPRSmipoptimize.

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

DENSECOLLIMIT

Description Newton barrier: Columns with more than DENSECOLLIMIT elements are considered to be dense. Such columns will be handled specially in the Cholesky factorization of this matrix.

Type Integer

Default value 0 — determined automatically.

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

DETERMINISTIC

Description Branch and Bound: Specifies whether the parallel MIP search should be deterministic.

Type Integer

Values

0	Use non-deterministic parallel MIP.
1	Use deterministic parallel MIP.

Default value 1

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also MIPTHREADS.

DUALGRADIENT

Description Simplex: This specifies the dual simplex pricing method.

Type Integer

Values

-1	Determine automatically.
0	Devex.
1	Steepest edge.
2	Direct steepest edge.
3	Sparse Devex.

Default value -1

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

See also PRICINGALG.

DUALIZE

Description This specifies whether presolve should form the dual of the problem.

Type Integer

Values

-1	Determine automatically.
0	Solve the primal problem.
1	Solve the dual problem.

Default value -1

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

See also DUALIZEOPS

DUALIZEOPS

Description Bit-vector control for adjusting the behavior when a problem is dualized.

Type Integer

Values

Bit	Meaning
0	Swap the simplex algorithm to run. If dual simplex is selected for the original problem then primal simplex will be run on the dualized problem, and simiarly if primal simplex is selected.

Default value 1 (bit 0 is set)

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

See also DUALIZE

DUALPERTURB

Description The factor by which the problem will be perturbed prior to optimization by dual simplex. A value of 0.0 results in no perturbation prior to optimization. DUALPERTURB, if set to a non-negative value, overrules the value of PERTURB. The control PERTURB is deprecated, the use of PRIMALPERTURB and DUALPERTURB is advised instead.

Note the interconnection to the AUTOPERTURB control. If AUTOPERTURB is set to 1, the decision whether to perturb or not is left to the Optimizer. When the problem is automatically perturbed in dual simplex, however, the value of DUALPERTURB will be used for perturbation.

Type Double

Default value -1 — determined automatically.

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

See also AUTOPERTURB, PERTURB, PRIMALPERTURB.

DUALSTRATEGY

Description This bit-vector control specifies the dual simplex strategy.

Type Integer

Values

Bit	Meaning
0	Switch to primal when re-optimization goes dual infeasible and numerically unstable.
1	When dual intend to switch to primal, stop the solve instead of switching to primal.
2	Use more aggressive cut-off in MIP search.
3	Use dual simplex to remove cost perturbations.
4	Enable more aggressive dual pivoting strategy.
5	Keep using dual simplex even when it's numerically unstable.

Default value 1

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

DUALTHREADS

Description Determines the maximum number of threads that dual simplex is allowed to use. If DUALTHREADS is set to the default value (-1), the THREADS control will determine the number of threads used.

Type Integer

Default value -1 (determined by the THREADS control)

Note When solving a linear MIP, the dual simplex algorithm will use multiple threads only when solving the initial LP relaxation or when reoptimizing between rounds of cuts on the root node.

The parallel dual simplex algorithm differs from the sequential dual simplex algorithm and might follow a different solve path. For DUALTHREADS > 1 the solve path is independent of the number of threads used, although the practical limit for observing performance benefits is around DUALTHREADS = 8.

Affects routines XPRSlpoptimize (LPOPTIMIZE).

See also CONCURRENTTHREADS, THREADS.

EIGENVALUETOL

Description A quadratic matrix is considered not to be positive semi-definite, if its smallest eigenvalue is smaller than the negative of this value.

Type Double

Default value 1E-6

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE), CHECKCONVEXITY.

See also IFCHECKCONVEXITY.

ELIMFILLIN

Description Amount of fill-in allowed when performing an elimination in presolve .

Type Integer

Default value 10

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

ELIMTOL

Description The Markowitz tolerance for the elimination phase of the presolve.

Type Double

Default value `0.001`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

ETATOL

Description Tolerance on eta elements. During each iteration, the basis inverse is premultiplied by an elementary matrix, which is the identity except for one column - the eta vector. Elements of eta vectors whose absolute value is smaller than `ETATOL` are taken to be zero in this step.

Type Double

Default value `1.0E-13`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`), `XPRSbtran`, `XPRSftran`.

EXTRACOLS

Description The initial number of extra columns to allow for in the matrix. If columns are to be added to the matrix, then, for maximum efficiency, space should be reserved for the columns before the matrix is input by setting the `EXTRACOLS` control. If this is not done, resizing will occur automatically, but more space may be allocated than the user actually requires.

Type Integer

Default value `0`

Affects routines `XPRSreadprob` (`READPROB`), `XPRSloadglobal`, `XPRSloadlp`, `XPRSloadqglobal`, `XPRSloadqp`.

See also `EXTRAROWS`, `EXTRAELEMS`, `EXTRAMIPENTS`.

EXTRAELEMS

Description The initial number of extra matrix elements to allow for in the matrix, including coefficients for cuts. If rows or columns are to be added to the matrix, then, for maximum efficiency, space should be reserved for the extra matrix elements before the matrix is input by setting the `EXTRAELEMS` control. If this is not done, resizing will occur automatically, but more space may be allocated than the user actually requires.

Type Integer

Default value Hardware/platform dependent.

Affects routines XPRSreadprob (READPROB), XPRSloadglobal, XPRSloadlp, XPRSloadqglobal, XPRSloadqp.

See also EXTRACOLS, EXTRAROWS.

EXTRAMIPENTS

Description The initial number of extra global entities to allow for.

Type Integer

Default value 0

Affects routines XPRSreadprob (READPROB), XPRSloadglobal, XPRSloadqglobal.

EXTRAPRESOLVE

Description This control no longer has any effect and will be removed from future releases. Use PRESOLVEMAXGROW to limit the number of non-zero coefficients in the presolved problem.

Type Integer

Default value 0

Affects routines XPRSreadprob (READPROB), XPRSloadglobal, XPRSloadlp, XPRSloadqglobal, XPRSloadqp.

EXTRAQCELEMENTS

Description This control is deprecated, and will be removed from future versions of the optimizer.

Type Integer

Default value 0

Affects routines XPRSreadprob (READPROB), XPRSloadqcqp.

See also EXTRAELEMS, EXTRAMIPENTS, EXTRAROWS, EXTRAQCROWS.

EXTRAQCROWS

Description This control is deprecated, and will be removed from future versions of the optimizer.

Type Integer

Default value 0

Affects routines XPRSreadprob (READPROB), XPRSloadqcqp.

See also EXTRAELEMS, EXTRAMIPENTS, EXTRAROWS, EXTRAQCELEMENTS.

EXTRAROWS

Description The initial number of extra rows to allow for in the matrix, including cuts. If rows are to be added to the matrix, then, for maximum efficiency, space should be reserved for the rows before the matrix is input by setting the EXTRAROWS control. If this is not done, resizing will occur automatically, but more space may be allocated than the user actually requires.

Type Integer

Default value 0

Affects routines XPRSreadprob (READPROB), XPRSloadglobal, XPRSloadlp, XPRSloadqglobal, XPRSloadqp.

See also EXTRACOLS.

EXTRASETELEMS

Description The initial number of extra elements in sets to allow for in the matrix. If sets are to be added to the matrix, then, for maximum efficiency, space should be reserved for the set elements before the matrix is input by setting the EXTRASETELEMS control. If this is not done, resizing will occur automatically, but more space may be allocated than the user actually requires.

Type Integer

Default value 0

Affects routines XPRSreadprob (READPROB), XPRSloadglobal, XPRSloadlp, XPRSloadqglobal, XPRSloadqp.

See also EXTRAMIPENTS, EXTRASETS.

EXTRASETS

Description The initial number of extra sets to allow for in the matrix. If sets are to be added to the matrix, then, for maximum efficiency, space should be reserved for the sets before the matrix is input by setting the EXTRASETS control. If this is not done, resizing will occur automatically, but more space may be allocated than the user actually requires.

Type Integer

Default value 0

Affects routines XPRSreadprob (READPROB), XPRSloadglobal, XPRSloadlp, XPRSloadqglobal, XPRSloadqp.

See also EXTRAMIPENTS, EXTRASETELEMS.

FEASIBILITYPUMP

Description Branch and Bound: Decides if the Feasibility Pump heuristic should be run at the top node.

Type Integer

Values

-1	Automatic.
0	Turned off.
1	Always try the Feasibility Pump.
2	Try the Feasibility Pump only if other heuristics have failed to find an integer solution.

Default value -1

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

FEASTOL

Description This tolerance determines when a solution is treated as feasible. If the amount by which a constraint's activity violates its right-hand side or ranged bound is less in absolute magnitude than FEASTOL, then the constraint is treated as satisfied. Similarly, if the amount by which a column violates its bounds is less in absolute magnitude than FEASTOL, those bounds are also treated as satisfied.

Type Double

Default value 1.0E-06

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE), XPRSgetinfeas.

FEASTOLPERTURB

Description This tolerance determines how much a feasible primal basic solution is allowed to be perturbed when performing basis changes. The tolerance FEASTOL is always considered as an upper limit for the perturbations, but in some cases smaller value can be more desirable.

Type Double

Default value 1.0E-06

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE), XPRSgetinfeas.

FEASTOLTARGET

Description This specifies the target feasibility tolerance for the solution refiner.

Type Double

Default value 0 — use the value specified by FEASTOL.

Note Zero and negative values are ignored, and the value of FEASTOL is used.

Note Use very small values like 1e-100 to state the refinement should continue as long as an improvement is made. Use very large values like 1e+100 to disable only this aspect of the refiner.

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

See also REFINEOPS, LPREFINEITERLIMIT, OPTIMALITYTOLTARGET.

FORCEOUTPUT

Description Certain names in the problem object may be incompatible with different file formats (such as names containing spaces for LP files). If the optimizer might be unable to read back a problem because of non-standard names, it will first attempt to write it out using an extended naming convention. If the names would not be possible to extend so that they would be reproducible and recognizable, it will give an error message and won't create the file. If the optimizer might be unable to read back a problem because of non-standard names, it will give an error message and won't create the file. This option may be used to force output anyway.

Type Integer

Values

0	Check format compatibility, and in case of failure try to extend names so that they are reproducible and recognizable.
1	Force output using problem names as is.
2	Always use 'x(' original name ')' in LP files to create a representation that can be read by Xpress. Default for problem having spaces in names
3	Substitute spaces by the '_' character in LP files

Default value 0

Affects routines XPRSwriteprob (WRITEPROB).

FORCEPARALLELDUAL

Description Dual simplex: specifies whether the dual simplex solver should always use the parallel simplex algorithm. By default, when using a single thread, the dual simplex solver will execute a dedicated sequential simplex algorithm.

Type Integer

Values

0	Disabled.
1	Enabled. Force the dual simplex solver to use the parallel algorithm.

Default value 0

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

See also THREADS, DUALTHREADS.

GENCONSABSTRANSFORMATION

Description This control specifies the reformulation method for absolute value general constraints at the beginning of the search.

Type Integer

Values

-1	Automatic.
0	Use a formulation based on indicator constraints.
1	Use a formulation based on SOS1-contraints.

Default value -1

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

GENCONSDUALREDUCTIONS

Description This parameter specifies whether dual reductions should be applied to reduce the number of columns and rows added when transforming general constraints to MIP structs.

Type Integer

Values

0	Disabled. No dual reductions, add columns and rows.
1	Enabled. Only add neccessary columns and rows, drop those implied by the objective sense.

Default value 1

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also MIPDUALREDUCTIONS.

GLOBALFILEBIAS

Description This control has been deprecated and no longer has any effect. In older versions of Xpress, it could be used to influence how much Xpress would write tree search data to the global file in preference to using in-memory data compression.

Type Double

Default value 0.5

See also GLOBALFILEUSAGE, TREEMEMORYLIMIT, TREEMEMORYSAVINGTARGET.

GLOBALFILELOGINTERVAL

Description This control sets the interval between progress messages output while writing tree data to the global file, in seconds. The solve is slowed greatly while data is being written to the global file and this output allows the user to see how much progress is being made.

Type Integer

Default value `60`

See also TREEDIAGNOSTICS.

GOMCUTS

Description Branch and Bound: The number of rounds of Gomory or lift-and-project cuts at the top node.

Type Integer

Default value `-1` — determined automatically.

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also TREEGOMCUTS, LNPBEST, LNPITERLIMIT.

HEURBEFORELP

Description Branch and Bound: Determines whether primal heuristics should be run before the initial LP relaxation has been solved.

Type Integer

Values

`-1`	Automatic - let the optimizer decide if heuristics should be run.
`0`	Disabled.
`1`	Enabled.

Default value `-1`

Note It is possible that a heuristic will find an optimal integer solution that will result in the LP relaxation solution being cut off. If the problem is solved with the "`l`" flag to XPRSmipoptimize (i.e., stop after solving the LP relaxation), then LPSTATUS might be returned as `XPRS_LP_CUTOFF` or `XPRS_LP_CUTOFF_IN_DUAL`. If dedicated heuristic threads are enabled through the HEURTHREADS control, then the initial heuristics will be run in parallel with the LP solve, instead of before.

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also HEURSTRATEGY, HEURTHREADS.

HEURDEPTH

Description Branch and Bound: Sets the maximum depth in the tree search at which heuristics will be used to find MIP solutions. It may be worth stopping the heuristic search for solutions after a certain depth in the tree search. A value of 0 signifies that heuristics will not be used. This control no longer has any effect and will be removed from future releases.

Type Integer

Default value `-1`

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

HEURDIVEITERLIMIT

Description Branch and Bound: Simplex iteration limit for reoptimizing during the diving heuristic.

Type Double

Values

`>=1`	Fixed iteration limit.
`0`	No iteration limit.
`<0`	Automatic selection of the iteration limit based on the problem size. The absolute value is used as a multiplier on the automatic selection.

Default value `-1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `HEURSTRATEGY`.

HEURDIVERANDOMIZE

Description The level of randomization to apply in the diving heuristic. The diving heuristic uses priority weights on rows and columns to determine the order in which to e.g. round fractional columns, or the direction in which to round them. This control determines by how large a random factor these weights should be changed.

Type Double

Values `0.0-1.0` Amount of randomization (0.0=none, 1.0=full)

Default value `0.0`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `HEURDIVESTRATEGY`, `HEURDIVESPEEDUP`.

HEURDIVESOFTROUNDING

Description Branch and Bound: Enables a more cautious strategy for the diving heuristic, where it tries to push binaries and integer variables to their bounds using the objective, instead of directly fixing them. This can be useful when the default diving heuristics fail to find any feasible solutions.

Type Integer

Values

`-1`	Automatic selection.
`0`	Do not use soft rounding.
`1`	Cautious use of the soft rounding strategy.
`2`	More aggressive use of the soft rounding strategy.

Default value `-1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `HEURDIVESTRATEGY`.

HEURDIVESPEEDUP

Description Branch and Bound: Changes the emphasis of the diving heuristic from solution quality to diving speed.

Type Integer

Values

`-2`	Automatic selection biased towards quality
`-1`	Automatic selection biased towards speed.
`0-4`	manual emphasis bias from emphasis on quality (0) to emphasis on speed (4).

Default value `-1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `HEURDIVESTRATEGY`.

HEURDIVESTRATEGY

Description Branch and Bound: Chooses the strategy for the diving heuristic.

Type Integer

Values

`-1`	Automatic selection of strategy.
`0`	Disables the diving heuristic.
`1-18`	Available pre-set strategies for rounding infeasible global entities and reoptimizing during the heuristic dive.

Default value `-1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `HEURSTRATEGY`.

HEURFORCESPECIALOBJ

Description Branch and Bound: This specifies whether local search heuristics without objective or wit an auxiliary objective should always be used, despite the automatic selection of the Optimiezr. By default, they will only be run on small problems and when no solution has been found yet.

Type Integer

Values

`0`	Disabled.
`1`	Enabled. Run special objective heuristics on large problems and even if incumbent exists.

Default value `0`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `HEURSTRATEGY`, `HEURSEARCHROOTSELECT`, `HEURSEARCHTREESELECT`.

HEURFREQ

Description Branch and Bound: This specifies the frequency at which heuristics are used in the tree search. Heuristics will only be used at a node if the depth of the node is a multiple of `HEURFREQ`.

Type Integer

Default value `-1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

HEURMAXSOL

Description Branch and Bound: This specifies the maximum number of heuristic solutions that will be found in the tree search. This control no longer has any effect and will be removed from future releases.

Type Integer

Default value `-1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

HEURNODES

Description Branch and Bound: This specifies the maximum number of nodes at which heuristics are used in the tree search. This control no longer has any effect and will be removed from future releases.

Type Integer

Default value `-1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

HEURSEARCHEFFORT

Description Adjusts the overall level of the local search heuristics.

Type Double

Default value `1.0`

Note `HEURSEARCHEFFORT` is used as a multiplier on the default amount of work the local search heuristics should do. A higher value means the local search heuristics will be run more often and that they are allowed to search larger neighborhoods.

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also HEURSTRATEGY, HEURSEARCHROOTSELECT, HEURSEARCHTREESELECT.

HEURSEARCHFREQ

Description Branch and Bound: This specifies how often the local search heuristic should be run in the tree.

Type Integer

Values

-1	Automatic.
0	Disabled in the tree.
n>0	Number of nodes between each run.

Default value -1

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also HEURSEARCHROOTCUTFREQ.

HEURSEARCHROOTCUTFREQ

Description How frequently to run the local search heuristic during root cutting. This is given as how many cut rounds to perform between runs of the heuristic. Set to zero to avoid applying the heuristic during root cutting.

Branch and Bound: This specifies how often the local search heuristic should be run in the tree.

Type Integer

Values

-1	Automatic.
0	Disabled heuristic during cuting.
n>0	Number of cutting rounds between each run.

Default value -1

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also HEURSEARCHFREQ.

HEURSEARCHROOTSELECT

Description A bit vector control for selecting which local search heuristics to apply on the root node of a global solve. Use HEURSEARCHTREESELECT to control local search heuristics during the tree search.

Type Integer

Values

Bit	Meaning
0	Local search with a large neighborhood. Potentially slow but is good for finding solutions that differs significantly from the incumbent.
1	Local search with a small neighborhood centered around a node LP solution.
2	Local search with a small neighborhood centered around an integer solution. This heuristic will often provide smaller, incremental improvements to an incumbent solution.
3	Local search with a neighborhood set up through the combination of multiple integer solutions.
4	*Unused*
5	Local search without an objective function. Called seldom and only when no feasible solution is available.
6	Local search with an auxiliary objective function. Called seldom and only when no feasible solution is available.

Default value 117

Note Some of the local search heuristics will benefit from having an existing incumbent solution, but it is not required. An initial solution can also be provided by the user through either XPRSloadmipsol or XPRSreadbinsol.

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also HEURSTRATEGY, HEURSEARCHTREESELECT, HEURSEARCHEFFORT.

HEURSEARCHTREESELECT

Description A bit vector control for selecting which local search heuristics to apply during the tree search of a global solve. Use HEURSEARCHROOTSELECT to control local search heuristics on the root node.

Type Integer

Values

Bit	Meaning
0	Local search with a large neighborhood. Potentially slow but is good for finding solutions that differs significantly from the incumbent.
1	Local search with a small neighborhood centered around a node LP solution.
2	Local search with a small neighborhood centered around an integer solution. This heuristic will often provide smaller, incremental improvements to an incumbent solution.
3	Local search with a neighborhood set up through the combination of multiple integer solutions.
4	*Unused*
5	Local search without an objective function. Called seldom and only when no feasible solution is available.
6	Local search with an auxiliary objective function. Called seldom and only when no feasible solution is available.

Default value 17

Note Some of the local search heuristics will benefit from having an existing incumbent solution, but it is not required. An initial solution can also be provided by the user through either XPRSloadmipsol or XPRSaddmipsol.

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also HEURSTRATEGY, HEURSEARCHROOTSELECT, HEURSEARCHEFFORT.

HEURSTRATEGY

Description Branch and Bound: This specifies the heuristic strategy. On some problems it is worth trying more comprehensive heuristic strategies by setting HEURSTRATEGY to 2 or 3.

Type Integer

Values

-1	Automatic selection of heuristic strategy.
0	No heuristics.
1	Basic heuristic strategy.
2	Enhanced heuristic strategy.
3	Extensive heuristic strategy.

Default value -1

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

HEURTHREADS

Description Branch and Bound: The number of threads to dedicate to running heuristics on the root node.

Type Integer

Values

-1	Automatically determined from the THREADS control.
0	Disabled. Heuristics will be run sequentially with the root LP solve and cutting.
>=1	Number of root threads to dedicate to parallel heuristics.

Default value 0

Note When heuristic threads are enable, the heuristics will be run in parallel with the initial LP solve, if possible, and in parallel with the root cutting.

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also THREADS.

HISTORYCOSTS

Description Branch and Bound: How to update the pseudo cost for a global entity when a strong branch or a regular branch is applied.

Type Integer

Values

`-1`	Automatically determined.
`0`	No update.
`1`	Initialize using only regular branches from the root to the current node.
`2`	Same as 1, but initialize with strong branching results as well.
`3`	Initialize using any regular branching or strong branching information from all nodes solves before the current node.

Default value `-1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `SBBEST`, `SBESTIMATE`, `SBSELECT`

IFCHECKCONVEXITY

Description Determines if the convexity of the problem is checked before optimization. Applies to quadratic, mixed integer quadratic and quadratically constrained problems. Checking convexity takes some time, thus for problems that are known to be convex it might be reasonable to switch the checking off.

Type Integer

Values

`0`	Turn off convexity checking.
`1`	Turn on convexity checking.

Default value `1`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `EIGENVALUETOL`

INDLINBIGM

Description Indicator constraints can be internally converted to regular rows (i.e. linearized) using a BigM coefficient whenever the BigM coefficient is smaller or equal to this value.

Type Double

Default value `1.0E+05`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

INDPRELINBIGM

Description During presolve, indicator constraints can be internally replaced with regular rows (i.e. linearized) using a BigM coefficient whenever the BigM coefficient is smaller or equal to this value.

Type Double

Default value `100.0`

Note Replacing an indicator constraint with a BigM row has a side effect on tolerances. In the indicator constraint form, the constraint part is satisfied with FEASTOL tolerance; while after changing it to BigM form, the constraint also includes the binary indicator variable (with a coefficient up to INDPRELINBIGM and an integrality tolerance of MIPTOL), therefore the constraint part of the indicator contraint is satisfied with tolerance FEASTOL+MIPTOL*INDPRELINBIGM.

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

INVERTFREQ

Description Simplex: The frequency with which the basis will be inverted. The basis is maintained in a factorized form and on most simplex iterations it is incrementally updated to reflect the step just taken. This is considerably faster than computing the full inverted matrix at each iteration, although after a number of iterations the basis becomes less well-conditioned and it becomes necessary to compute the full inverted matrix. The value of `INVERTFREQ` specifies the maximum number of iterations between full inversions.

Type Integer

Default value `-1` — the frequency is determined automatically.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

INVERTMIN

Description Simplex: The minimum number of iterations between full inversions of the basis matrix. See the description of `INVERTFREQ` for details.

Type Integer

Default value `3`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

KEEPBASIS

Description Simplex: This determines which basis to use for the next iteration. The choice is between using that determined by the crash procedure at the first iteration, or using the basis from the last iteration.

Type Integer

Values

`0`	Problem optimization starts from the first iteration, i.e. the previous basis is ignored.
`1`	The previously loaded basis (last in memory) should be used.
`2`	Use the previous basis only if it is valid for the current problem (the number of basic variables must match the number of rows).

Default value `1`

Note This gets reset to the default value after optimization has started.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

KEEPNROWS

Description Status for nonbinding rows.

Type Integer

Values

`-1`	Delete N type rows from the matrix.
`0`	Delete elements from N type rows leaving empty N type rows in the matrix.
`1`	Keep N type rows.

Default value `-1`

Affects routines `XPRSreadprob` (`READPROB`), `XPRSloadglobal`, `XPRSloadlp`, `XPRSloadqglobal`, `XPRSloadqp`.

L1CACHE

Description Newton barrier: L1 cache size in kB (kilo bytes) of the CPU. On Intel (or compatible) platforms a value of -1 may be used to determine the cache size automatically.

Type Integer

Default value Hardware/platform dependent.

Note Specifying the correct L1 cache size can give a significant performance advantage with the Newton barrier algorithm.

If the size is unknown, it is better to specify a smaller size.

If the size cannot be determined automatically on Intel (or compatible) platforms, a default size of 8 kB is assumed.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

LINELENGTH

Description This control is deprecated.

Type Integer

LNPBEST

Description Number of infeasible global entities to create lift-and-project cuts for during each round of Gomory cuts at the top node (see GOMCUTS).

Type Integer

Default value 50

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

LNPITERLIMIT

Description Number of iterations to perform in improving each lift-and-project cut.

Type Integer

Default value -1 — determined automatically.

Note By setting the number to zero a Gomory cut will be created instead.

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

LPFLAGS

Description A bit-vector control which defines the algorithm for solving an LP problem or the initial LP relaxation of a MIP problem.

Type Integer

Values

Bit	Meaning
0	Use the dual simplex method.
1	Use the primal simplex method.
2	Use the barrier method.
3	Use the network simplex method.

Default value 0

Note Setting bit 0, 1, 2, 3 of this control will have the same effect of passing flags d, p, b, n to XPRSmipoptimize or XPRSlpoptimize. When more than one bit are set, then the LP problem will be solved with the concurrent solver. When this control is set and flags are passed at the same time, the flags will overrule the value of the control.

This control can be tuned.

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

LPITERLIMIT

Description Simplex: The maximum number of iterations that will be performed before the optimization process terminates. For MIP problems, this is the maximum total number of iterations over all nodes explored by the Branch and Bound method.

Type Integer

Default value `2147483645`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

LPREFINEITERLIMIT

Description This specifies the simplex iteration limit the solution refiner can spend in attempting to increase the accuracy of an LP solution.

Type Integer

Default value `-1` — determined automatically.

Note The solution refiner iteratively attempts to increase the accuracy of the solution until either both `FEASTOLTARGET` and `OPTIMALITYTOLTARGET` is satisfied, or accuracy cannot further be increased, or the effort limit determined by LPREFINEITERLIMIT is exhausted.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `REFINEOPS`, `FEASTOLTARGET`, `OPTIMALITYTOLTARGET`.

LOCALCHOICE

Description Controls when to perform a local backtrack between the two child nodes during a dive in the branch and bound tree.

Type Integer

Values

`1`	Never backtrack from the first child, unless it is dropped (infeasible or cut off).
`2`	Always solve both child nodes before deciding which child to continue with.
`3`	Automatically determined.

Default value `1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

LPFOLDING

Description Simplex and barrier: whether to fold an LP problem before solving it.

Type Integer

Values

-1	Automatic.
0	Disable LP folding.
1	Enable LP folding. Attempt to fold all LP problems and MIP initial relaxations.

Default value -1

Affects routines XPRSlpoptimize (LPOPTIMIZE).

LPLOG

Description Simplex: The frequency at which the simplex log is printed.

Type Integer

Values

n<0	Detailed output every -*n* iterations.
0	Log displayed at the end of the optimization only.
n>0	Summary output every *n* iterations.

Default value 100

Note This control only has an effect if LPLOGSTYLE is set to 0.

Affects routines XPRSlpoptimize (LPOPTIMIZE).

See also A.8.

LPLOGDELAY

Description Time interval between two LP log lines.

Type Double

Default value 1.0

Note This control only has an effect if LPLOGSTYLE is set to 1.

Affects routines XPRSlpoptimize (LPOPTIMIZE).

LPLOGSTYLE

Description Simplex: The style of the simplex log.

Type Integer

Values

0	Simplex log is printed based on simplex iteration count, at a fixed frequency as specified by the LPLOG control.
1	Simplex Log is printed based on an estimation of elapsed time, determined by an internal deterministic timer.

Default value 1

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE), XPRSminim (MINIM), XPRSmaxim (MAXIM), XPRSglobal (GLOBAL).

LPTHREADS

Description This control is deprecated, and is provided for compatibility purposes. Please use CONCURRENTTHREADS instead.

Type Integer

Default value -1

Note The value of this control is mirrored by the CONCURRENTTHREADS control.

See also CONCURRENTTHREADS

MARKOWITZTOL

Description The Markowitz tolerance used for the factorization of the basis matrix.

Type Double

Default value 0.01

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

MATRIXTOL

Description The zero tolerance on matrix elements. If the value of a matrix element is less than or equal to MATRIXTOL in absolute value, it is treated as zero.

Type Double

Default value 1.0E-09

Affects routines XPRSreadprob (READPROB), XPRSloadglobal, XPRSloadlp, XPRSloadqglobal, XPRSloadqp, XPRSalter (ALTER), XPRSaddcols, XPRSaddcuts, XPRSaddrows, XPRSchgcoef, XPRSchgmcoef, XPRSstorecuts.

MAXCHECKSONMAXCUTTIME

Description This control is intended for use where optimization runs that are terminated using the MAXCUTTIME control are required to be reproduced exactly. This control is necessary because of the inherent difficulty in terminating algorithmic software in a consistent way using temporal criteria. The control value relates to the number of times the optimizer checks the MAXCUTTIME criterion up to and including the check when the termination of cutting was activated. To use the control the user first must obtain the value of the

CHECKSONMAXCUTTIME attribute after the run returns. This attribute value is the number of times the optimizer checked the MAXCUTTIME criterion during the last call to the optimization routine XPRSmipoptimize. Note that this attribute value will be negative if the optimizer terminated cutting on the MAXCUTTIME criterion. To ensure accurate reproduction of a run the user should first ensure that MAXCUTTIME is set to its default value or to a large value so the run does not terminate again on MAXCUTTIME and then simply set the control MAXCHECKSONMAXCUTTIME to the absolute value of the CHECKSONMAXCUTTIME value.

Type Integer

Values

0	Not active.
n>0	The number of times the optimizer should check the MAXCUTTIME criterion before triggering a termination.

Default value 0

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

MAXCHECKSONMAXTIME

Description This control is intended for use where optimization runs that are terminated using the MAXTIME control are required to be reproduced exactly. This control is necessary because of the inherent difficulty in terminating algorithmic software in a consistent way using temporal criteria. The control value relates to the number of times the optimizer checks the MAXTIME criterion up to and including the check when the termination was activated. To use the control the user first must obtain the value of the CHECKSONMAXTIME attribute after the run returns. This attribute value is the number of times the optimizer checked the MAXTIME criterion during the last call to the optimization routine XPRSmipoptimize. Note that this attribute value will be negative if the optimizer terminated on the MAXTIME criterion. To ensure that a reproduction of a run terminates in the same way the user should first ensure that MAXTIME is set to its default value or to a large value so the run does not terminate again on MAXTIME and then simply set the control MAXCHECKSONMAXTIME to the absolute value of the CHECKSONMAXTIME value.

Type Integer

Values

0	Not active.
n>0	The number of times the optimizer should check the MAXTIME criterion before triggering a termination.

Default value 0

Affects routines XPRSmipoptimize (MIPOPTIMIZE)

MAXMCOEFFBUFFERELEMS

Description The maximum number of matrix coefficients to buffer before flushing into the internal representation of the problem. Buffering coefficients can offer a significant performance gain when you are building a matrix using XPRSchgcoef or XPRSchgmcoef, but can lead to a significant memory overhead for large matrices, which this control allows you to influence.

Type Integer

Default value `2147483647`

Affects routines XPRSchgcoef, XPRSchgmcoef.

MAXCUTTIME

Description The maximum amount of time allowed for generation of cutting planes and reoptimization. The limit is checked during generation and no further cuts are added once this limit has been exceeded.

Type Integer

Values

0	No time limit.
n>0	Stop cut generation after *n* seconds.

Default value 0

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

MAXGLOBALFILESIZE

Description The maximum size, in megabytes, to which the global file may grow, or 0 for no limit. When the global file reaches this limit, a second global file will be created. Useful if you are using a filesystem that puts a maximum limit on the size of a file.

Type Integer

Default value 0

See also GLOBALFILESIZE.

MAXIIS

Description This function controls the number of Irreducible Infeasible Sets to be found using the XPRSiisall (IIS -a).

Type Integer

Values

-1	Search for all IIS.
0	Do not search for IIS.
n>0	Search for the first *n* IIS.

Default value -1

Note The function XPRSiisnext is not affected.

Affects routines XPRSiisall (IIS-a).

MAXIMPLIEDBOUND

Description Presolve: When tighter bounds are calculated during MIP preprocessing, only bounds whose absolute value are smaller than `MAXIMPLIEDBOUND` will be applied to the problem.

Type Double

Default value `1.0E+08`

Note For numerically challenging MIP problems, it can sometimes help make the solve more stable by reducing the value of `MAXIMPLIEDBOUND` to something smaller - e.g. 1.0E+06. It is not recommended to increase this parameter beyond the default of 1.0E+08.

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

MAXLOCALBACKTRACK

Description Branch-and-Bound: How far back up the current dive path the optimizer is allowed to look for a local backtrack candidate node.

Type Integer

Default value `-1`

Note If this control is set to *k*, then the candidate set of nodes for a local backtrack will consist of all active nodes in the subtree rooted at height *k* above the current node. For example, a setting of `1` will result in only sibling nodes of the current node being considered.

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `LOCALCHOICE`.

MAXMEMORYHARD

Description This control sets the maximum amount of memory in megabytes the optimizer should allocate. If this limit is exceeded, the solve will terminate. This control is designed to make the optimizer stop in a controlled manner, so that the problem object is valid once termination occurs. The solve state will be set to incomplete. This is different to an out of memory condition in which case the optimizer returns an error. The optimizer may still allocate memory once the limit is exceeded to be able to finsish the operations and stop in a controlled manner. When `RESOURCESTRATEGY` is enabled, the control also has the same effect as `MAXMEMORYSOFT` and will cause the optimizer to try preserving memory when possible.

Type Integer

Default value `0 (no limit)`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

MAXMEMORYSOFT

Description When RESOURCESTRATEGY is enabled, this control sets the maximum amount of memory in megabytes the optimizer targets to allocate. This may change the solving path, but will not cause the solve to terminate early. To set a hard version of the same, please set MAXMEMORYHARD.

Type Integer

Default value `0 (no limit)`

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

MAXMIPTASKS

Description Branch-and-Bound: The maximum number of tasks to run in parallel during a MIP solve.

Type Integer

Values

-1	Task limit determined automatically from MIPTHREADS.
>0	Fixed task limit.

Default value -1

Note The MIP solver will create smaller tasks from individual active nodes or based on local search heuristics. These are tasks that will be executed in parallel by the number of threads set by MIPTHREADS.

Note If MAXMIPTASKS is set to a fixed, positive value, the branch-and-bound tree nodes will always be solved in the same deterministic way, independent of the actual number of executing threads implied by MIPTHREADS.

How a MIP is solved will still depend on the number of threads used for solving the continuous relaxation and therefore on the settings for the controls BARTHREADS, DUALTHREADS and CONCURRENTTHREADS).

To obtain a MIP solve that is completely independent of the number of threads, it is sufficient to set MAXMIPTASKS, FORCEPARALLELDUAL and BARTHREADS. The concurrent LP solver should be avoided in this case.

Note The number of MIP tasks that can be defined for a 32-bit system is limited to 32 for performance reasons.

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also MIPTHREADS, THREADS, DUALTHREADS, BARTHREADS, CONCURRENTTHREADS, FORCEPARALLELDUAL.

MAXMIPSOL

Description Branch and Bound: This specifies a limit on the number of integer solutions to be found by the Optimizer. It is possible that during optimization the Optimizer will find the same objective solution from different nodes. However, MAXMIPSOL refers to the total number of integer solutions found, and not necessarily the number of distinct solutions.

Type Integer

Default value 0

Note Setting MAXMIPSOL=1 can alter the solution path as this will put the emphasis on finding any feasible solution by triggering additional heuristics.

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

MAXNODE

Description Branch and Bound: The maximum number of nodes that will be explored.

Type Integer

Default value 2147483647

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

MAXPAGELINES

Description Number of lines between page breaks in printable output.

Type Integer

Default value 23

Affects routines XPRSwriteprtsol (WRITEPRTSOL), XPRSwriteprtrange (WRITEPRTRANGE).

MAXSCALEFACTOR

Description This determines the maximum scaling factor that can be applied during scaling. The maximum is provided as an exponent of a power of 2.

Type Integer

Values 0-64 The maximum is provided an exponent of a power of 2.

Default value 64

Affects routines XPRSloadglobal, XPRSloadlp, XPRSloadqglobal, XPRSloadqp, XPRSreadprob (READPROB), XPRSscale (SCALE).

See also SCALING.

MAXTIME

Description The maximum time in seconds that the Optimizer will run before it terminates, including the problem setup time and solution time. For MIP problems, this is the total time taken to solve all the nodes.

Type Integer

Values

`0`	No time limit.
`n>0`	If an integer solution has been found, stop MIP search after *n* seconds, otherwise continue until an integer solution is finally found.
`n<0`	Stop in LP or MIP search after *n* seconds.

Default value `0`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

MIPABSCUTOFF

Description Branch and Bound: If the user knows that they are interested only in values of the objective function which are better than some value, this can be assigned to `MIPABSCUTOFF`. This allows the Optimizer to ignore solving any nodes which may yield worse objective values, saving solution time. When a MIP solution is found a new cut off value is calculated and the value can be obtained from the `CURRMIPCUTOFF` attribute. The value of `CURRMIPCUTOFF` is calculated using the `MIPRELCUTOFF` and `MIPADDCUTOFF` controls.

Type Double

Default value `1.0E+40` (for minimization problems); `-1.0E+40` (for maximization problems).

Note `MIPABSCUTOFF` can also be used to stop the dual algorithm.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `MIPRELCUTOFF`, `MIPADDCUTOFF`.

MIPABSGAPNOTIFY

Description Branch and bound: if the `gapnotify` callback has been set using `XPRSaddcbgapnotify`, then this callback will be triggered during the global search when the absolute gap reaches or passes the value you set of the `MIPRELGAPNOTIFY` control.

Type Double

Default value `-1.0`

Affects routines `XPRSaddcbgapnotify`, `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `MIPRELGAPNOTIFY`, `MIPABSGAPNOTIFYOBJ`, `MIPABSGAPNOTIFYBOUND`

MIPABSGAPNOTIFYBOUND

Description Branch and bound: if the gapnotify callback has been set using XPRSaddcbgapnotify, then this callback will be triggered during the global search when the best bound reaches or passes the value you set of the MIPRELGAPNOTIFYBOUND control.

Type Double

Default value 1.0E+40 (for minimization problems); -1.0E+40 (for maximization problems)

Affects routines XPRSaddcbgapnotify, XPRSmipoptimize (MIPOPTIMIZE).

See also MIPRELGAPNOTIFY, MIPABSGAPNOTIFYOBJ, MIPABSGAPNOTIFY

MIPABSGAPNOTIFYOBJ

Description Branch and bound: if the gapnotify callback has been set using XPRSaddcbgapnotify, then this callback will be triggered during the global search when the best solution value reaches or passes the value you set of the MIPRELGAPNOTIFYOBJ control.

Type Double

Default value -1.0E+40 (for minimization problems); 1.0E+40 (for maximization problems)

Affects routines XPRSaddcbgapnotify, XPRSmipoptimize (MIPOPTIMIZE).

See also MIPRELGAPNOTIFY, MIPABSGAPNOTIFY, MIPABSGAPNOTIFYBOUND

MIPABSSTOP

Description Branch and Bound: The absolute tolerance determining whether the global search will continue or not. It will terminate if

$|\mathtt{MIPOBJVAL} - \mathtt{BESTBOUND}| \leq \mathtt{MIPABSSTOP}$

where MIPOBJVAL is the value of the best solution's objective function, and BESTBOUND is the current best solution bound. For example, to stop the global search when a MIP solution has been found and the Optimizer can guarantee it is within 100 of the optimal solution, set MIPABSSTOP to 100.

Type Double

Default value 0.0

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also MIPRELSTOP, MIPADDCUTOFF.

MIPADDCUTOFF

Description Branch and Bound: The amount to add to the objective function of the best integer solution found to give the new CURRMIPCUTOFF. Once an integer solution has been found whose objective function is equal to or better than CURRMIPCUTOFF, improvements on this value may not be interesting unless they are better by at least a certain amount. If MIPADDCUTOFF is nonzero, it will be added to CURRMIPCUTOFF each time an integer solution is found which is better than this new value. This cuts off sections of the tree whose solutions would not represent substantial improvements in the objective function, saving processor time. The control MIPABSSTOP provides a similar function but works in a different way.

Type Double

Default value `-1.0E-05`

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also MIPRELCUTOFF, MIPABSSTOP, MIPABSCUTOFF.

MIPCOMPONENTS

Description Determines whether disconnected components in a MIP should be solved as separate MIPs. There can be significant performence benefits from solving disconnected components individual instead of being part of the main branch-and-bound search.

Type Integer

Values

`-1`	Automatic - let the solver decide.
`0`	Disable solving disconnected components separately.
`1`	Solve disconnected components separately.

Default value `-1`

Note If there are no constraints linking two variables, either directly or indirectly through other variables, they are said to belong to two separate disconnected components. When a problem contains disconnected components of signficant size, it can be advantageous to solve each component as a separate MIP. When significant disconnected components are detected in the problem, the solver will switch to a different solve mode where each component is solved separately. This switch will happen after the root node processing has completed and when the solve is about to enter the branch-and-bound search.

Note Solving disconnected components separately is not compatible with many callbacks that can be used for modifying the branch-and-bound search. Setting any of the following callbacks will automatically disable the separate solving of disconnected components: XPRSaddcboptnode, XPRSaddcbprenode, XPRSaddcbcutmgr, XPRSaddcbestimate, XPRSaddcbsepnode, XPRSaddcbchgbranch, XPRSaddcbchgbranchobject

Note Solving disconnected components separately is not compatible with concurrent MIP solves. If concurrent MIP solves has been turned off, disconnected components will be solved as part of the standard branch-and-bound search in each concurrent solve.

Note Disabling MIP dual reductions through MIPDUALREDUCTIONS will also disable the separate solve of disconnected components.

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also PRECOMPONENTS, MIPCONCURRENTSOLVES, XPRSaddcboptnode, XPRSaddcbprenode, XPRSaddcbcutmgr, XPRSaddcbestimate, XPRSaddcbsepnode, XPRSaddcbchgbranch, XPRSaddcbchgbranchobject

MIPCONCURRENTNODES

Description Sets the node limit for when a winning solve is selected when concurrent MIP solves are enabled. When multiple MIP solves are started, they each run up to the MIPCONCURRENTNODES node limit and only one winning solve is selected for contuinuing the search with.

Type Integer

Values

-1	Automatic - let the solver decide on a node limit.
>0	Number of nodes each concurrent solve should complete before a winner is selected.

Default value -1

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also MIPCONCURRENTSOLVES

MIPCONCURRENTSOLVES

Description Selects the number of concurrent solves to start for a MIP. Each solve will use a unique random seed for its random number generator, but will otherwise apply the same user controls. The first concurrent solve to complete will have solved the MIP and all the concurrent solves will be terminated at this point. Using concurrent solves can be advantageous when a MIP displays a high level of *performance variability*.

Type Integer

Values

-1	Enabled. The number of concurrent solves depends on MIPTHREADS.
0, 1	Disabled
n>1	Enabled. The number of concurrent solves to start is given by n.

Default value 0

Note A node limit is imposed on each concurrent solve, through MIPCONCURRENTNODES. When a concurrent solve reaches this node limit, it will be suspended until all concurrent solves have reached the limit. At this point a winner will be declared, based on which solve made the most progress towards optimality and only the winning solve will continue, using all threading resources. If a concurrent solve completes its MIP search before reaching the node limit, all solves will be stopped.

Note Concurrent solves are not compatible with many callbacks that can be used for modifying the branch-and-bound search. Setting any of the following callbacks will automatically disable concurrent solves: XPRSaddcboptnode, XPRSaddcbprenode, XPRSaddcbcutmgr, XPRSaddcbestimate, XPRSaddcbsepnode, XPRSaddcbchgbranch, XPRSaddcbchgbranchobject

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also MIPCONCURRENTNODES, XPRSaddcboptnode, XPRSaddcbprenode, XPRSaddcbcutmgr, XPRSaddcbestimate, XPRSaddcbsepnode, XPRSaddcbchgbranch, XPRSaddcbchgbranchobject

MIPDUALREDUCTIONS

Description Branch and Bound: Limits operations that can reduce the MIP solution space.

Type Integer

Values

2	Allow dual reductions on continuous variables only.
1	Allow all dual reductions.
0	Prevent all dual reductions.

Default value 1

Note The MIPDUALREDUCTIONS control, when set to a value different from 1 will adjust the values of other controls in order to prevent MIP solver operations that can result in the removal of dominated solutions. For example, dual reductions during preprocessing attempts to remove dominated solutions based on objective arguments, assuming that all constraints are known to the Optimizer. If a problem is detected to have symmetries, the solver might also remove some symmetrical solutions from the search space. In both cases, the set of feasible MIP solutions might be reduced. With default settings, it is only guaranteed that at least one optimal solution remains.

Note When attempting to collect the *n*-best solutions, it is recommended to set MIPDUALREDUCTIONS=2. This will ensure that the only solutions missed by the enumeration are those that only differ from an existing solution in the values of the continuous variables.

Note Advanced users that maintain external constraints, which are applied dynamically to the problem using callbacks during a branch-and-bound solve, it is recommended to set MIPDUALREDUCTIONS=0. This ensures that any solution to the original problem that satisfies all of the user's external constraints maps to a feasible solution in the presolved space.

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also MIPPRESOLVE, PRECOMPONENTS, PRESOLVEOPS, SYMMETRY.

MIPFRACREDUCE

Description Branch and Bound: Specifies how often the optimizer should run a heuristic to reduce the number of fractional integer variables in the node LP solutions.

Type Integer

Values

-1	Automatic.
0	Disabled.
1	Run before and after cutting on the root node.
2	Run also during root cutting.
3	Run also during the tree search.

Default value `-1`

Note This heuristic is only applicable to problems that are dual degenerate. These are problems that contain multiple solutions with identical objective function value. The more dual degenerate a problem is, the more likely it will be for this heuristic to have an improving effect.

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

MIPKAPPAFREQ

Description Branch and Bound: Specifies how frequently the basis condition number (also known as kappa) should be calculated during the branch-and-bound search.

Type Integer

Values

- `0` Do not calculate condition numbers.
- `1` Calculate conditions numbers on every node, including after each round of root cutting.
- `n>1` Calculate a condition number once per node of every n'th level of the branch-and-bound tree.

Default value `0`

Note The condition number is calculated as the norm of the basis matrix multiplied by the norm of its inverse. This uses the Froebenius norm.

Note A summary will be printed at the end of the solve, summarizing the collected condition numbers collected:

Statistic	Meaning
`Nodes kappa stable`	No. of stable sampled nodes (kappa < 10^7)
`Nodes kappa suspicius`	No. of suspicious sampled nodes ($10^7 \leq$ kappa < 10^{10})
`Nodes kappa unstable`	No. of unstable sampled nodes ($10^{10} \leq$ kappa < 10^{13})
`Nodes kappa ill-posed`	No. of ill-posed sampled nodes ($10^{13} \leq$ kappa)
`Largest kappa seen`	The largest condition number calculated through all sampled nodes.
`Kappa attention level`	A measure of how ill-posed the problem is (between 0 and 1).

Affects routines `ATTENTIONLEVEL`, `MAXKAPPA`, `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also .

MIPLOG

Description Global print control.

Type Integer

Values

`-n`	Print out summary log at each n^{th} node.
`0`	No printout in global.
`1`	Only print out summary statement at the end.
`2`	Print out detailed log at all solutions found.
`3`	Print out detailed log at each node.

Default value `-100`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also A.10.

MIPPRESOLVE

Description Branch and Bound: Type of integer processing to be performed. If set to `0`, no processing will be performed.

Type Integer

Values

Bit	Meaning
`0`	Reduced cost fixing will be performed at each node. This can simplify the node before it is solved, by deducing that certain variables' values can be fixed based on additional bounds imposed on other variables at this node.
`1`	Primal reductions will be performed at each node. Uses constraints of the node to tighten the range of variables, often resulting in fixing their values. This greatly simplifies the problem and may even determine optimality or infeasibility of the node before the simplex method commences.
`2`	[Unused] This bit is no longer used to control probing. Refer to the integer control `PREPROBING` for setting probing level during presolve.
`3`	If node preprocessing is allowed to change bounds on continuous columns.
`4`	Dual reductions will be performed at each node.
`5`	Allow global (non-bound) tightening of the problem during the tree search.
`6`	The objective function will be used to find reductions at each node.
`7`	Allow the branch-and-bound tree search to be restarted if it appears to be advantageous.
`8`	Allow that symmetry is used to presolve the node problem.

Default value -257

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also 5.3, `PRESOLVE`, `PRESOLVEOPS`, `PREPROBING`.

MIPRAMPUP

Description Controls the strategy used by the parallel MIP solver during the ramp-up phase of a branch-and-bound tree search.

Type Integer

Values	`-1`	Automatically determined.
	`0`	No special treatment during the ramp-up phase. Always run with the maximal number of tasks.
	`1`	Limit the number of tasks until the initial dives have completed.

Default value -1

Note The branch-and-bound tree search starts from the single root node, and only through branching on this root node and the resulting child nodes, are enough active nodes created to produce sufficient tasks to keep all MIP workers busy. This is referred to as the ramp-up phase of a parallel MIP.

In a typical MIP solve, the solutions found during the initial dives will typically provide a significant improvement over the root heuristic solutions. It can therefore be advantageous to let these initial dives run as fast as possible, by limiting resource contention. This can be accomplished by restricting the number of parallel tasks and thereby reducing the memory bus contention. The `MIPRAMPUP` control can be used to turn this initial task restriction of a parallel MIP solve on or off.

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `MIPTHREADS`, `MAXMIPTASKS`.

MIPRESTART

Description Branch and Bound: controls strategy for in-tree restarts.

Type Integer

Values	`-1`	Determined automatically.
	`0`	Disable in-tree restarts.
	`1`	Allow in-tree restarts.

Default value `-1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

MIPRESTARTGAPTHRESHOLD

Description Branch and Bound: Initial gap threshold to delay in-tree restart. The restart is delayed initially if the gap, given as a fraction between 0 and 1, is below this threshold. The optimizer adjusts the threshold every time a restart is delayed. Note that there are other criteria that can delay or prevent a restart.

Type Double

Default value `0.02`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `MIPRESTART`

MIQCPALG

Description This control determines which algorithm is to be used to solve mixed integer quadratic constrained and mixed integer second order cone problems.

Type Integer

Values

-1	Determined automatically.
0	Use the barrier algorithm in the branch and bound algorithm.
1	Use outer approximations in the branch and bound algorithm.

Default value -1

Affects routines XPRSmipoptimize (MIPOPTIMIZE), XPRSminim (MINIM), XPRSmaxim (MAXIM), XPRSglobal (GLOBAL).

MIPREFINEITERLIMIT

Description This defines an effort limit expressed as simplex iterations for the MIP solution refiner. The limit is per reoptimizations in the MIP refiner.

Type Integer

Default value -1 — determined automatically.

Affects routines XPRSrefinemipsol (REFINEMIPSOL).

MIPRELCUTOFF

Description Branch and Bound: Percentage of the LP solution value to be added to the value of the objective function when an integer solution is found, to give the new value of CURRMIPCUTOFF. The effect is to cut off the search in parts of the tree whose best possible objective function would not be substantially better than the current solution. The control MIPRELSTOP provides a similar functionality but works in a different way.

Type Double

Default value 1.0E-04

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also MIPABSCUTOFF, MIPADDCUTOFF, MIPRELSTOP.

MIPRELGAPNOTIFY

Description Branch and bound: if the gapnotify callback has been set using XPRSaddcbgapnotify, then this callback will be triggered during the global search when the relative gap reaches or passes the value you set of the MIPRELGAPNOTIFY control.

Type Double

Default value `-1.0`

Affects routines XPRSaddcbgapnotify, XPRSmipoptimize (MIPOPTIMIZE).

See also MIPABSGAPNOTIFY, MIPABSGAPNOTIFYOBJ, MIPABSGAPNOTIFYBOUND

MIPRELSTOP

Description Branch and Bound: This determines when the global search will terminate. Global search will stop if:
|MIPOBJVAL − BESTBOUND| ≤ MIPRELSTOP x max(|BESTBOUND|,|MIPOBJVAL|)
where MIPOBJVAL is the value of the best solution's objective function and BESTBOUND is the current best solution bound. For example, to stop the global search when a MIP solution has been found and the Optimizer can guarantee it is within 5% of the optimal solution, set MIPRELSTOP to 0.05.

Type Double

Default value `0.0001`

Note This control is a stopping criteria only and different values of the control will not affect the solution path before termination. Unlike other stopping criteria, like time and node count, termination on MIPRELSTOP will cause the final solution to be declared optimal and the problem to be returned to its original state.

Note Tolerances, such as MIPRELCUTOFF and MIPABSCUTOFF, determine how much the objective value of a new MIP solution has to differ from the incumbent for it to be accepted. These controls therefore also influence the final gap at the end of a MIP solve.

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also MIPABSSTOP, MIPRELCUTOFF.

MIPTERMINATIONMETHOD

Description Branch and Bound: How a MIP solve should be stopped on early termination when there are still active tasks in the system. This can happen when, for example, a time or node limit is reached.

Type Integer

Values

0	Terminate tasks at the earliest opportunity. This can result in some unfinished node solves being discarded, although never integer solutions.
1	Allow tasks to complete their current work but prevent new tasks from being started.

Default value `0`

Note With MIPTERMINATIONMETHOD=0, termination will be quick but the returned state of the MIP solve will not include any work done by interrupted tasks. In particular, it is possible that some user callbacks (not *intsol* or *preintsol*) will have been fired for nodes that are discarded at termination. A user program that relies on the firing of callbacks being completely deterministic should therefore set MIPTERMINATIONMETHOD=1, which will produce a slower termination, but guaranteed deterministic firing of all user callbacks.

Note Irrespective of the choice of MIPTERMINATIONMETHOD, a MIP solve will always be returned in a deterministic state when DETERMINISTIC=1.

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also DETERMINISTIC, MAXMIPTASKS, MIPTHREADS, THREADS.

MIPTHREADS

Description If set to a positive integer it determines the number of threads implemented to run the parallel MIP code. If MIPTHREADS is set to the default value (-1), the THREADS control will determine the number of threads used.

Type Integer

Default value -1 (determined by the THREADS control)

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also DETERMINISTIC, MAXMIPTASKS, HEURTHREADS, THREADS.

MIPTOL

Description Branch and Bound: This is the tolerance within which a decision variable's value is considered to be integral.

Type Double

Default value 5.0E-06

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

MIPTOLTARGET

Description Target MIPTOL value used by the automatic MIP solution refiner as defined by REFINEOPS. Negative and zero values are ignored.

Type Double

Default value 0.0

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

MPS18COMPATIBLE

Description Provides compatibility of MPS file output for older MPS readers.

Type Integer

Values `Bit 0` Do not write objective sense (`OBJSENSE` section).
`Bit 1` Fixed binaries are written as fixed only (unless used as a base variable for an indicator constraint).

Default value 0

Affects routines `XPRSwriteprob` (`WRITEPROB`)

MPSBOUNDNAME

Description The bound name sought in the MPS file. As with all string controls, this is of length 64 characters plus a null terminator, `\0`.

Type String

Default value 64 blanks

Affects routines `XPRSreadprob` (`READPROB`).

MPSECHO

Description Determines whether comments in MPS matrix files are to be printed out during matrix input.

Type Integer

Values

0	MPS comments are *not* to be echoed.
1	MPS comments *are* to be echoed.

Default value 0

Affects routines `XPRSreadprob` (`READPROB`).

MPSFORMAT

Description Specifies the format of MPS files.

Type Integer

Values

-1	To determine the file type automatically.
0	For fixed format.
1	If MPS files are assumed to be in free format by input.

Default value 1

Note Setting MPSFORMAT to 0 or -1 disables XSLPreadprob in case Xpress NonLinear is used.

Affects routines `XPRSalter` (`ALTER`), `XPRSreadbasis` (`READBASIS`), `XPRSreadprob` (`READPROB`).

MPSOBJNAME

Description The objective function name sought in the MPS file. As with all string controls, this is of length 64 characters plus a null terminator, `\0`.

Type String

Default value 64 blanks

Affects routines `XPRSreadprob` (`READPROB`).

MPSRANGENAME

Description The range name sought in the MPS file. As with all string controls, this is of length 64 characters plus a null terminator, `\0`.

Type String

Default value 64 blanks

Affects routines `XPRSreadprob` (`READPROB`).

MPSRHSNAME

Description The right hand side name sought in the MPS file. As with all string controls, this is of length 64 characters plus a null terminator, `\0`.

Type String

Default value 64 blanks

Affects routines `XPRSreadprob` (`READPROB`).

MUTEXCALLBACKS

Description Branch and Bound: This determines whether the callback routines are mutexed from within the optimizer.

Type Integer

Values

`0`	Callbacks are not mutexed.
`1`	Callbacks are mutexed.

Default value 1

Note If the users' callbacks take a significant amount of time it may be preferable not to mutex the callbacks. In this case the user must ensure that their callbacks are threadsafe.

Affects routines `XPRSaddcboptnode`, `XPRSaddcbinfnode`, `XPRSaddcbintsol`, `XPRSaddcbnodecutoff`, `XPRSaddcbprenode`.

NETCUTS

Description Determines the addition of multi-commodity network cuts to a problem. The parameter is defined as a bit string, and values 1, 2, 4 can be summed up if the user wants more classes of cuts to be added.

Type Integer

Values

`-1`	Automatically determined.
`0`	Do not add these cuts.
`1`	Add cut-set inequalities.
`2`	Add node cut-set inequalities, i.e., cut-set inequalities that are based on a network cut defined on a single network node.
`4`	Add lifted flow-cover inequalities.

Default value `0`

Note If the user wants to add both cut-set inequalities and lifted flow-cover inequalities but not node cut-set inequalities, the value of the control should be set to 1+4=5.

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

NODEPROBINGEFFORT

Description Adjusts the overall level of node probing.

Type Double

Default value `1.0`

Note `NODEPROBINGEFFORT` is used as a multiplier on the default amount of work node probing should do. Setting the control to zero disables node probing.

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

NODESELECTION

Description Branch and Bound: This determines which nodes will be considered for solution once the current node has been solved.

Type Integer

Values

`1`	*Local first*: Choose between descendant and sibling nodes if available; choose from all outstanding nodes otherwise.
`2`	*Best first*: Choose from all outstanding nodes.
`3`	*Local depth first*: Choose between descendant and sibling nodes if available; choose from the deepest nodes otherwise.
`4`	*Best first, then local first*: Best first is used for the first `BREADTHFIRST` nodes, after which local first is used.
`5`	*Pure depth first*: Choose from the deepest outstanding nodes.

Default value Dependent on the matrix characteristics.

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

NUMERICALEMPHASIS

Description How much emphasis to place on numerical stability instead of solve speed.

Type Integer

Values

-1	Automatic. The emphasis might be influenced by the setting of other controls.
0	Emphasize speed.
1	Mild emphasis on numerical stability.
2	Medium emphasis on numerical stability.
3	Strong emphasis on numerical stability.

Default value -1

Affects routines XPRSmipoptimize (MIPOPTIMIZE), XPRSlpoptimize (LPOPTIMIZE).

OBJSCALEFACTOR

Description Custom global objective scaling factor, expressed as a power of 2. When set, it overwrites the automatic global objective scaling factor. A value of 0 means no objective scaling. This control is applied for the full solve, and is independent of any extra scaling that may occur specifically for the barrier or simplex solvers. As it is a power of 2, to scale by 16, set the value of the control to 4.

Type Double

Default value 0

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

OPTIMALITYTOL

Description Simplex: This is the zero tolerance for reduced costs. On each iteration, the simplex method searches for a variable to enter the basis which has a negative reduced cost. The candidates are only those variables which have reduced costs less than the negative value of OPTIMALITYTOL.

Type Double

Default value 1.0E-06

Affects routines XPRSgetinfeas, XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

OPTIMALITYTOLTARGET

Description This specifies the target optimality tolerance for the solution refiner.

Type Real

Default value `0` — use the value specified by `OPTIMALITYTOL`.

Note Zero and negative values are ignored, and the value of `OPTIMALITYTOL` is used.

Note Use very small values like 1e-100 to state the refinement should continue as long as an improvement is made. Use very large values like 1e+100 to disable only this aspect of the refiner.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `REFINEOPS`, `LPREFINEITERLIMIT`, `FEASTOLTARGET`.

OUTPUTCONTROLS

Description This control toggles the printing of all control settings at the beginning of the search. This includes the printing of controls that have been explicitly assigned to their default value. All unset controls are omitted as they keep their default value.

Type Integer

Values

`0`	Turn off printing of user-specified control settings.
`1`	Print controls.

Default value `1`

Note Setting `OUTPUTCONTROLS` to 0 has no effect on the function `XPRSdumpcontrols`

OUTPUTLOG

Description This controls the level of output produced by the Optimizer during optimization. In the Console Optimizer, `OUTPUTLOG` controls which messages are sent to the screen (`stdout`). When using the Optimizer library, no output is sent to the screen. If the user wishes output to be displayed, they must define a callback function and print messages to the screen themselves. In this case, `OUTPUTLOG` controls which messages are sent to the user output callback.

Type Integer

Values

`0`	Turn all output off.
`1`	Print all messages.
`3`	Print error and warning messages.
`4`	Print error messages only.

Default value `1`

Affects routines `XPRSaddcbmessage`, `XPRSsetlogfile`.

OUTPUTMASK

Description Mask to restrict the row and column names written to file. As with all string controls, this is of length 64 characters plus a null terminator, `\0`.

Type String

Default value 64 '?'s

Affects routines `XPRSwriterange` (`WRITERANGE`), `XPRSwritesol` (`WRITESOL`).

OUTPUTTOL

Description Zero tolerance on print values.

Type Double

Default value `1.0E-05`

Affects routines `XPRSwriteprtrange` (`WRITEPRTRANGE`), `XPRSwriteprtsol` (`WRITEPRTSOL`), `XPRSwriterange` (`WRITERANGE`), `XPRSwritesol` (`WRITESOL`).

PENALTY

Description Minimum absolute penalty variable coefficient. `BIGM` and `PENALTY` are set by the input routine (`XPRSreadprob` (`READPROB`)) but may be reset by the user prior to `XPRSlpoptimize` (`LPOPTIMIZE`).

Type Double

Default value Dependent on the matrix characteristics.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

PERTURB

Description This control is deprecated and will be removed from future versions of the Optimizer. The use of `PRIMALPERTURB` and `DUALPERTURB` is advised instead. The control was used to give a factor by which the problem will be perturbed prior to optimization by either simplex algorithm.

Type Double

Default value `0.0`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `AUTOPERTURB`, `PERTURB`, `PRIMALPERTURB`.

PIVOTTOL

Description Simplex: The zero tolerance for matrix elements. On each iteration, the simplex method seeks a nonzero matrix element to pivot on. Any element with absolute value less than `PIVOTTOL` is treated as zero for this purpose.

Type Double

Default value `1.0E-09`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`), `XPRSpivot`.

PPFACTOR

Description The partial pricing candidate list sizing parameter.

Type Double

Default value `1.0`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

PREANALYTICCENTER

Description Determines if analytic centers should be computed and used for variable fixing and the generation of alternative reduced costs (-1: Auto 0: Off, 1: Fixing, 2: Redcost, 3: Both)

Type Integer

Values

`-1`	Automatic.
`0`	Disable analytic center presolving.
`1`	Use analytic center for variable fixing only.
`2`	Use analytic center for reduced cost computation only.
`3`	Use analytic centers for both, variable fixing and reduced cost computation.

Default value `-1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

PREBASISRED

Description Determines if a lattice basis reduction algorithm should be attempted as part of presolve

Type Integer

Values
`-1` Automatic.
`0` Disable basis reduction.
`1` Enable basis reduction.

Default value `0`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

PREBNDREDCONE

Description Determines if second order cone constraints should be used for inferring bound reductions on variables when solving a MIP.

Type Integer

Values
`-1` Automatic.
`0` Disable bound reductions from second order cone constraints.
`1` Enable bound reductions from second order cone constraints.

Default value `-1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `PREBNDREDQUAD`, `MIQCPALG`.

PREBNDREDQUAD

Description Determines if convex quadratic contraints should be used for inferring bound reductions on variables when solving a MIP.

Type Integer

Values
`-1` Automatic.
`0` Disable bound reductions from quadratic constraints.
`1` Enable bound reductions from quadratic constraints.

Default value `-1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `PREBNDREDCONE`, `MIQCPALG`.

PRECOEFELIM

Description Presolve: Specifies whether the optimizer should attempt to recombine constraints in order to reduce the number of non zero coefficients when presolving a mixed integer problem.

Type Integer

Values

`0`	Disabled.
`1`	Remove as many coefficients as possible.
`2`	Cautious eliminations. Will not perform a reduction if it might destroy problem structure useful to e.g. heuristics or cutting.

Default value `2`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `PRESOLVE`, `PRESOLVEOPS`.

PRECOMPONENTS

Description Presolve: determines whether small independent components should be detected and solved as individual subproblems during root node processing.

Type Integer

Values

`-1`	Automatically determined.
`0`	Disable detection of independent components.
`1`	Enable detection of independent components.

Default value `-1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `PRESOLVE`, `PRESOLVEOPS`.

PRECOMPONENTSEFFORT

Description Presolve: adjusts the overall effort for the independent component presolver. This control affects working limits for the subproblem solving as well as thresholds when it is called. Increase to put more emphasis on component presolving.

Type Double

Default value 1.0

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `PRECOMPONENTS`.

PRECONEDECOMP

Description Presolve: decompose regular and rotated cones with more than two elements and apply Outer Approximation on the resulting components.

Type Integer

Values	`-1`	Automatically determined.
	`0`	Disable cone decomposition.
	`1`	Enable cone decomposition by replacing large cones with small ones in the presolved problem.
	`2`	Similar to 1, plus decomposition is enabled even if the cone variable is fixed.
	`3`	Cones are decomposed within the Outer Approximation domain only, i.e., the problem maintains the original cones.

Default value `-1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `PRESOLVE`, `PRESOLVEOPS`.

PRECONVERTSEPARABLE

Description Presolve: reformulate problem with non-diagonal quadratic objective and/or constraints as diagonal quadratic or second-order conic constraints.

Type Integer

Values	`-1`	Automatically determined.
	`0`	Disable reformulation.
	`1`	Enable reformulation to diagonal quadratic constraints.
	`2`	Similar to 1, plus reduction to second-order cones.
	`3`	Similar to 2, plus the objective function is converted to a constraint and treated as a quadratic constraint.

Default value `-1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `PRESOLVE`.

Note This control is only used in MIQPs and MIQCQPs, and has no effect when used on continuous quadratic problems.

PREDOMCOL

Description Presolve: Determines the level of dominated column removal reductions to perform when presolving a mixed integer problem. Only binary columns will be checked.

Type Integer

Values	`-1`	Automatically determined.
	`0`	Disabled.
	`1`	Cautious strategy.
	`2`	All candidate binaries will be checked for domination.

Default value `-1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `PRESOLVE`, `PRESOLVEOPS`.

PREDOMROW

Description Presolve: Determines the level of dominated row removal reductions to perform when presolving a problem.

Type Integer

Values

`-1`	Automatically determined.
`0`	Disabled.
`1`	Cautious strategy.
`2`	Medium strategy.
`3`	Aggressive strategy. All candidate row combinations will be considered.

Default value `-1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`), `XPRSlpoptimize` (`LPOPTIMIZE`).

See also `PRESOLVE`, `PRESOLVEOPS`.

PREDUPROW

Description Presolve: Determines the type of duplicate rows to look for and eliminate when presolving a problem.

Type Integer

Values

`-1`	Automatically determined.
`0`	Do not eliminate duplicate rows.
`1`	Eliminate only rows that are identical in all variables.
`2`	Same as option 1 plus eliminate duplicate rows with simple penalty variable expressions. (MIP only).
`3`	Same as option 2 plus eliminate duplicate rows with more complex penalty variable expressions. (MIP only).

Default value `-1`

Note Duplicate rows can also be disabled by clearing the corresponding bit of the `PRESOLVEOPS` integer control.

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`), `XPRSlpoptimize` (`LPOPTIMIZE`).

See also `PRESOLVE`, `PRESOLVEOPS`.

PREELIMQUAD

Description Presolve: Allows for elimination of quadratic variables via doubleton rows.

Type Integer

Values

-1	Automatically determined.
0	Do not eliminate duplicate rows.
1	Eliminate at least one quadratic variable for each doubleton row.

Default value -1

Affects routines XPRSmipoptimize (MIPOPTIMIZE), XPRSlpoptimize (LPOPTIMIZE).

See also PRESOLVE, PRESOLVEOPS.

PREIMPLICATIONS

Description Presolve: Determines whether to use implication structures to remove redundant rows. If implication sequences are detected, they might also be used in probing.

Type Integer

Values

-1	Automatically determined.
0	Do not use implications for sparsification.
1	Use implications to remove reduandant rows.

Default value -1

Affects routines XPRSmipoptimize (MIPOPTIMIZE), XPRSlpoptimize (LPOPTIMIZE).

See also PRESOLVE, PRESOLVEOPS, PREPROBING.

PRELINDEP

Description Presolve: Determines whether to check for and remove linearly dependent equality constraints when presolving a problem.

Type Integer

Values

-1	Automatically determined.
0	Do not check for linearly dependent equality constraints.
1	Check for and remove linearly dependent equality constraints.

Default value -1

Affects routines XPRSmipoptimize (MIPOPTIMIZE), XPRSlpoptimize (LPOPTIMIZE).

See also PRESOLVE, PRESOLVEOPS.

PREOBJCUTDETECT

Description Presolve: Determines whether to check for constraints that are parallel or near parallel to a linear objective function, and which can safely be removed. This reduction applies to MIPs only.

Type Integer

Values

0	Disable check and reductions.
1	Enable check and reductions.

Default value 1

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also PRESOLVE, PRESOLVEOPS.

PREPERMUTE

Description This bit vector control specifies whether to randomly permute rows, columns and global information when starting the presolve. With the default value 0, no permutation will take place.

Type Integer

Values

Bit	Meaning
0	Permute rows.
1	Permute columns.
2	Permute global information. This bit only affects MIP problems.

Default value 0

Note Random permutations enable trying out different solution paths when solving a problem. The random seed for the permutations can be set using PREPERMUTESEED. When both PRESORT and PREPERMUTE are enabled, it will sort and then permute the problem.

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

See also PREPERMUTESEED, PRESORT, PRESOLVE, MIPPRESOLVE.

PREPERMUTESEED

Description This control sets the seed for the pseudo-random number generator for permuting the problem when starting the presolve. This control only has effects when PREPERMUTE is enabled.

Type Integer

Default value 1

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

See also PREPERMUTE, PRESOLVE, MIPPRESOLVE.

PREPROBING

Description Presolve: Amount of probing to perform on binary variables during presolve. This is done by fixing a binary to each of its values in turn and analyzing the implications.

Type Integer

Values

`-1`	Let the optimizer decide on the amount of probing.
`0`	Disabled.
`+1`	Light probing — only few implications will be examined.
`+2`	Full probing — all implications for all binaries will be examined.
`+3`	Full probing and repeat as long as the problem is significantly reduced.

Default value `-1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `PRESOLVE`.

PREPROTECTDUAL

Description Presolve: specifies whether the presolver should protect a given dual solution by maintaining the same level of dual feasibility. Enabling this control often results in a worse presolved model. This control only expected to be optionally enabled before calling `XPRScrossoverlpsol`.

Type Integer

Values

`0`	Disabled.
`1`	Enabled. Protect the dual solution during presolve.

Default value `0`

Affects routines `XPRScrossoverlpsol`

PRESOLVE

Description This control determines whether presolving should be performed prior to starting the main algorithm. Presolve attempts to simplify the problem by detecting and removing redundant constraints, tightening variable bounds, etc. In some cases, infeasibility may even be determined at this stage, or the optimal solution found.

Type Integer

Values	-1	Presolve applied, but a problem will not be declared infeasible if primal infeasibilities are detected. The problem will be solved by the LP optimization algorithm, returning an infeasible solution, which can sometimes be helpful.
	0	Presolve not applied.
	1	Presolve applied.
	2	Presolve applied, but redundant bounds are not removed. This can sometimes increase the efficiency of the barrier algorithm.
	3	Presolve is applied, and bounds detected to be redundant are always removed.

Default value 1

Note Memory for presolve is dynamically resized. If the Optimizer runs out of memory for presolve, an error message (245) is produced. Presolve settings 2 and 3 can sometimes make the barrier solves more efficient.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also 5.3, `PRESOLVEOPS`.

PRESOLVEMAXGROW

Description Limit on how much the number of non-zero coefficients is allowed to grow during presolve, specified as a ratio of the number of non-zero coefficients in the original problem.

Type Double

Default value 0.1

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

PRESOLVEOPS

Description This bit vevctor control specifies the operations which are performed during the presolve.

Type Integer

Values

Bit	Meaning
0	Singleton column removal.
1	Singleton row removal.
2	Forcing row removal.
3	Dual reductions.
4	Redundant row removal.
5	Duplicate column removal.
6	Duplicate row removal.
7	Strong dual reductions.
8	Variable eliminations.
9	No IP reductions.
10	No semi-continuous variable detection.
11	No advanced IP reductions.
12	No eliminations on integers.
14	Linearly dependant row removal.
15	No integer variable and SOS detection.

Default value 511 (bits 0 — 8 incl. are set)

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE), XPRSpresolverow.

See also 5.3, PRESOLVE, MIPPRESOLVE.

PRESOLVEPASSES

Description Number of reduction rounds to be performed in presolve

Type Integer

Default value 1

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

See also 5.3, PRESOLVE.

PRESORT

Description This bit vector control specifies whether to sort rows, columns and global information by their names when starting the presolve. With the default value 0, no sorting will take place.

Type Integer

Values

Bit	Meaning
0	Sort rows.
1	Sort columns.
2	Sort global information. This bit only affects MIP problems.

Default value 0

Note Sorting a problem by names can help obtain the same solution path when the rows, columns or global information of the problem is rearranged. It is recommended to enable all three bits when sorting a problem. When both `PRESORT` and `PREPERMUTE` are enabled, it will sort and then permute the problem.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `PREPERMUTE`, `PRESOLVE`, `MIPPRESOLVE`.

PRICINGALG

Description Simplex: This determines the primal simplex pricing method. It is used to select which variable enters the basis on each iteration. In general Devex pricing requires more time on each iteration, but may reduce the total number of iterations, whereas partial pricing saves time on each iteration, but may result in more iterations.

Type Integer

Values

`-1`	Partial pricing.
`0`	Determined automatically.
`1`	Devex pricing.
`2`	Steepest edge.
`3`	Steepest edge with unit initial weights.

Default value `0`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `DUALGRADIENT`.

PRIMALOPS

Description Primal simplex: allows fine tuning the variable selection in the primal simplex solver.

Type Integer

Values

Bit	Meaning
`0`	Use aggressive dj scaling.
`1`	Conventional dj scaling.
`2`	Use reluctant switching back to partial pricing.
`3`	Use dynamic switching between cheap and expensive pricing strategies.
`4`	Keep solving even after potential cycling is detected.

Default value `-1`

Note If both bits 0 and 1 are both set or unset then the dj scaling strategy is determined automatically.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `PRICINGALG`.

PRIMALPERTURB

Description The factor by which the problem will be perturbed prior to optimization by primal simplex. A value of `0.0` results in no perturbation prior to optimization. `PRIMALPERTURB`, if set to a non-negative value, overrules the value of `PERTURB`. The control `PERTURB` is deprecated, the use of `PRIMALPERTURB` and `DUALPERTURB` is advised instead.

Note the interconnection to the `AUTOPERTURB` control. If `AUTOPERTURB` is set to `1`, the decision whether to perturb or not is left to the Optimizer. When the problem is automatically perturbed in primal simplex, however, the value of `PRIMALPERTURB` will be used for perturbation.

Type Double

Default value `-1` — determined automatically.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `AUTOPERTURB`, `DUALPERTURB`, `PERTURB`.

PRIMALUNSHIFT

Description Determines whether primal is allowed to call dual to unshift.

Type Integer

Values

`0`	Allow the dual algorithm to be used to unshift.
`1`	Don't allow the dual algorithm to be used to unshift.

Default value `0`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `PRIMALOPS`, `PRICINGALG`, `DUALSTRATEGY`.

PSEUDOCOST

Description Branch and Bound: The default pseudo cost used in estimation of the degradation associated with an unexplored node in the tree search. A pseudo cost is associated with each integer decision variable and is an estimate of the amount by which the objective function will be worse if that variable is forced to an integral value.

Type Double

Default value `0.01`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`), `XPRSreaddirs` (`READDIRS`).

PWLDUALREDUCTIONS

Description This parameter specifies whether dual reductions should be applied to reduce the number of columns, rows and SOS-constraints added when transforming piecewise linear objectives and constraints to MIP structs.

Type Integer

Values

0	Disabled. No dual reductions, add all columns, rows and SOS-constraints.
1	Enabled. Only add neccessary columns, rows and sets, drop those implied by the objective sense.

Default value 1

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also MIPDUALREDUCTIONS.

PWLNONCONVEXTRANSFORMATION

Description This control specifies the reformulation method for piecewise linear constraints at the beginning of the search.

Note that the chosen formulation will only be used if MIP entities are necessary but not if presolve detected that a convex reformulation is possible. Furthermore, the binary formulation will only be applied to piecewise linear constraints with bounded input variable, otherwise the SOS2-formulation will be used.

Type Integer

Values

-1	Automatic.
0	Use a formulation based on SOS2-constraints.
1	Use a formulation based on binary variables.

Default value -1

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

QCCUTS

Description Branch and Bound: Limit on the number of rounds of outer approximation cuts generated for the root node, when solving a mixed integer quadratic constrained or mixed integer second order conic problem with outer approximation.

Type Integer

Default value -1 — determined automatically.

Note This control only has an effect for problems with quadratic or second order cone constraints, and only if outer approximation has not been disabled by setting MIQCPALG to 0.

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also TREEQCCUTS.

QCROOTALG

Description This control determines which algorithm is to be used to solve the root of a mixed integer quadratic constrained or mixed integer second order cone problem, when outer approximation is used.

Type Integer

Values

-1	Determined automatically.
0	Use the barrier algorithm.
1	Use the dual simplex on a relaxation of the problem constructed using outer approximation.

Default value -1

Note This control only has an effect if MIQCPALG is set to 1.

Affects routines XPRSmipoptimize (MIPOPTIMIZE), XPRSminim (MINIM), XPRSmaxim (MAXIM), XPRSglobal (GLOBAL).

QSIMPLEXOPS

Description Controls the behavior of the quadratic simplex solvers.

Type Integer

Values

Bit	Meaning
0	Force traditional primal first phase.
1	Force BigM primal first phase.
2	Force traditional dual first phase.
3	Force BigM dual first phase.
4	Always use artificial bounds in dual.
5	Use original problem basis only when warmstarting the KKT.
6	Skip the primal bound flips for ranged primals (might cause more trouble than good if the bounds are very large).
7	Also do the single pivot crash.
8	Do not apply aggressive perturbation in dual.

Default value 0

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

QUADRATICUNSHIFT

Description Determines whether an extra solution purification step is called after a solution found by the quadratic simplex (either primal or dual).

Type Integer

Values

`-1`	Determined automatically.
`0`	No purification step.
`1`	Always do the purification step.

Default value `-1`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

RANDOMSEED

Description Sets the initial seed to use for the pseudo-random number generator in the Optimizer. The sequence of random numbers is always reset using the seed when starting a new optimization run.

Type Integer

Default value `1`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

REFACTOR

Description Indicates whether the optimization should restart using the current representation of the factorization in memory.

Type Integer

Values

`-1`	Automatic.
`0`	Do not refactor on reoptimizing.
`1`	Refactor on reoptimizing.

Default value `-1`

Note In the tree search, the optimal bases at the nodes are not refactorized by default, but the optimal basis for an LP problem will be refactorized. If you are repeatedly solving LPs with few changes then it is more efficient to set `REFACTOR` to `0`.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

REFINEOPS

Description This specifies when the solution refiner should be executed to reduce solution infeasibilities. The refiner will attempt to satisfy the target tolerances for all original linear constraints before presolve or scaling has been applied.

Type Integer

Values

Bit	Meaning
0	Run the solution refiner on an optimal solution of a non-global problem.
1	Run the solution refiner when a new solution is found during a global search. The refiner will be applied to the presolved solution before any post-solve operations are applied.
3	Run the solution refiner on each node of the MIP search.
4	Run the solution refiner on an optimal solution before postsolve on a non-global problem.
5	Apply the iterative refiner to refine the solution.
6	Use higher precision in the iterative refinement.
7	If set, the iterative refiner will use the primal simplex algorithm.
8	If set, the iterative refiner will use the dual simplex algorithm.
9	Refine MIP solutions such that rounding them keeps the problem feasible when reoptimized.
10	Attempt to refine MIP solutions such that rounding them keeps the problem feasible when reoptimized, but accept integers solutions even if refinement fails.

Default value `19` (bits `0`, `1` and `4` are set)

Note If neither the 7th nor 8th bit is set, the refiner will use the primal simplex if the primal violations are larger than the dual violations, otherwise it will use the dual simplex.

If both the 7th and 8th bit are set then the refiner will split the problem into a primal feasible and dual feasible part, and solve the first with primal simplex and the second with dual simplex.

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `LPREFINEITERLIMIT`, `FEASTOLTARGET`, `OPTIMALITYTOLTARGET`, `MIPTOLTARGET`.

RELAXTREEMEMORYLIMIT

Description When the memory used by the branch and bound search tree exceeds the target specified by the `TREEMEMORYLIMIT` control, the optimizer will try to reduce this by writing nodes to the global file. In rare cases, usually where the solve has many millions of very small nodes, the tree structural data (which cannot be written to the global file) will grow large enough to approach or exceed the tree's memory target. When this happens, optimizer performance can degrade greatly as the solver makes heavy use of the global file in preference to memory. To prevent this, the solver will automatically relax the tree memory limit when it detects this case; the `RELAXTREEMEMORYLIMIT` control specifies the proportion of the previous memory limit by which to relax it. Set `RELAXTREEMEMORYLIMIT` to `0.0` to force the Xpress Optimizer to never relax the tree memory limit in this way.

Type Double

Note While setting higher values of `RELAXTREEMEMORYLIMIT` can improve performance significantly for a small number of models in low memory situations, the user is advised to use the `TREEMEMORYLIMIT` control to tune the memory usage of the branch and bound tree, according to the solve characteristics of their problem, rather than increasing `RELAXTREEMEMORYLIMIT`.

Default value `0.1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `TREEMEMORYLIMIT`.

RELPIVOTTOL

Description Simplex: At each iteration a pivot element is chosen within a given column of the matrix. The relative pivot tolerance, `RELPIVOTTOL`, is the size of the element chosen relative to the largest possible pivot element in the same column.

Type Double

Default value `1.0E-06`

Affects routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`), `XPRSpivot`.

REPAIRINDEFINITEQ

Description Controls if the optimizer should make indefinite quadratic matrices positive definite when it is possible.

Type Integer

Values

`0`	Repair if possible.
`1`	Do not repair.

Default value `1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

REPAIRINFEASMAXTIME

Description Overall time limit for the repairinfeas tool

Type Integer

Values

`0`	No time limit.
`n>0`	If an integer solution has been found, stop MIP search after *n* seconds, otherwise continue until an integer solution is finally found.
`n<0`	Stop in LP or MIP search after *n* seconds.

Default value 0

Note This control affects the total runtime of repairinfeas, as opposed to MAXTIME which affects the individual solves reapirinfeas carries out.

Affects routines XPRSrepairinfeas (REPAIRINFEAS).

RESOURCESTRATEGY

Description Controls whether the optimizer is allowed to make nondeterministic decisions if memory is running low in an effort to preserve memory and finish the solve. Available memory (or container limits) are automatically detected but can also be changed by MAXMEMORYSOFT and MAXMEMORYHARD

Type Integer

Values

1	Allow the optimizer to change the solve path if necessary to preserve memory when getting close to one of the memory limits.

Default value 0

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

ROOTPRESOLVE

Description Determines if presolving should be performed on the problem after the global search has finished with root cutting and heuristics.

Type Integer

Values

-1	Let the optimizer decide if the problem should be presolved again.
0	Disabled.
+1	Always presolve the root problem.

Default value -1

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also PRESOLVE.

SBBEST

Description Number of infeasible global entities to initialize pseudo costs for on each node.

Type Integer

Values

-1	determined automatically.
0	disable strong branching.
n>0	perform strong branching on up to *n* entities at each node.

Default value -1

Note By default, strong branching will be performed only for infeasible global entities whose pseudo costs have not otherwise been initialized (see `HISTORYCOSTS`).

If `SBBEST` is set to zero, the control `HISTORYCOSTS` will also be treated as zero and no past branching or strong branching information will be used in the global entity selection.

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `SBITERLIMIT`, `SBSELECT`, `SBEFFORT`, `HISTORYCOSTS`.

SBEFFORT

Description Adjusts the overall amount of effort when using strong branching to select an infeasible global entity to branch on.

Type Double

Default value `1.0`

Note `SBEFFORT` is used as a multiplier on other strong branching related controls, and affects the values used for `SBBEST`, `SBSELECT` and `SBITERLIMIT` when those are set to automatic.

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `SBBEST`, `SBITERLIMIT`, `SBSELECT`.

SBESTIMATE

Description Branch and Bound: How to calculate pseudo costs from the local node when selecting an infeasible global entity to branch on. These pseudo costs are used in combination with local strong branching and history costs to select the branch candidate.

Type Integer

Values

`-1`	Automatically determined.
`1-6`	Different variants of local pseudo costs.

Default value `-1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `SBBEST`, `SBITERLIMIT`, `SBSELECT`, `HISTORYCOSTS`.

SBITERLIMIT

Description Number of dual iterations to perform the strong branching for each entity.

Type Integer

Default value `-1` — determined automatically.

Note This control can be useful to increase or decrease the amount of effort (and thus time) spent performing strong branching at each node. Setting `SBITERLIMIT=0` will disable dual strong branch iterations. Instead, the entity at the head of the candidate list will be selected for branching.

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `SBBEST`, `SBSELECT`.

SBSELECT

Description The size of the candidate list of global entities for strong branching.

Type Integer

Values

`-2`	Automatic (low effort).
`-1`	Automatic (high effort).
`n>=0`	Include *n* entities in the candidate list (but always at least `SBBEST` candidates).

Default value `-2`

Note Before strong branching is applied on a node of the branch and bound tree, a list of candidates is selected among the infeasible global entities. These entities are then evaluated based on the local LP solution and prioritized. Strong branching will then be applied to the `SBBEST` candidates. The evaluation is potentially expensive and for some problems it might improve performance if the size of the candidate list is reduced.

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `SBBEST`, `SBEFFORT`, `SBESTIMATE`.

SCALING

Description This bit vector control determines how the Optimizer will rescale a model internally before optimization. If set to `0`, no scaling will take place.

Type Integer

Values

Bit	Meaning
0	Row scaling.
1	Column scaling.
2	Row scaling again.
3	Maximum.
4	Curtis-Reid.
5	0: scale by geometric mean. 1: scale by maximum element.
6	Treat big-M rows as normal rows.
7	Scale objective function for the simplex method.
8	Exclude the quadratic part of constraint when calculating scaling factors.
9	Scale before presolve.
10	Do not scale rows up.
11	Do not scale columns down.
12	Do not apply automatic global objective scaling.
13	RHS scaling.
14	Disable aggressive quadratic scaling.
15	Enable explicit linear slack scaling.

Default value 163

Note Setting `SCALING` to `0` will preserve the current scaling of the problem. Note that the Optimizer might automatically select a different scaling strategy, when the control `AUTOSCALING` is not disabled. However, if `SCALING` is set to any value by the user, `AUTOSCALING` will be ignored.

Affects routines `XPRSlpoptimize`, `XPRSlpoptimize`, `XPRSmipoptimize`, `XPRSscale` (`SCALE`).

See also 6.3.1, `MAXSCALEFACTOR`, `OBJSCALEFACTOR`, `AUTOSCALING`.

SIFTING

Description Determines whether to enable sifting algorithm with the dual simplex method.

Type Integer

Values

-1	Automatically determined.
0	Disable sifting with the dual simplex method.
1	Enable sifting with the dual simplex method.

Default value -1

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`), `XPRSlpoptimize` (`LPOPTIMIZE`).

SIFTSWITCH

Description Determines which algorithm to use for solving the subproblems during sifting.

Type Integer

Values	-1	Dual simplex.
	0	Barrier.
	>0	Use the barrier algorithm while the number of dual infeasibilities is larger than this value, otherwise use dual simplex.

Default value -1

Affects routines XPRSmipoptimize (MIPOPTIMIZE), XPRSlpoptimize (LPOPTIMIZE).

SLEEPONTHREADWAIT

Description Determines if the threads should be put into a wait state when waiting for work.

Type Integer

Values	Bit	Meaning
	-1	Automatically determined depending on the CPU the Optimizer is running on.
	0	Keep the threads busy when waiting for work.
	1	Put the threads into a wait state when waiting for work.

Default value -1

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

SOSREFTOL

Description The minimum relative gap between the ordering values of elements in a special ordered set. The gap divided by the absolute value of the larger of the two adjacent values must be at least SOSREFTOL.

Type Double

Default value 1.0E-06

Note This tolerance must not be set lower than 1.0E-06.

Affects routines XPRSloadglobal, XPRSloadqglobal, XPRSreadprob (READPROB).

SYMMETRY

Description Adjusts the overall amount of effort for symmetry detection.

Type Integer

Values	0	No symmetry detection.
	1	Conservative effort.
	2	Intensive symmetry search.

Default value 1

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also SYMSELECT.

SYMSELECT

Description Adjusts the overall amount of effort for symmetry detection.

Type Integer

Values

0	Search the whole matrix (otherwise the 0, 1 and −1 coefficients only).
1	Search all entities (otherwise binaries only).

Default value −1

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also SYMMETRY.

THREADS

Description The default number of threads used during optimization.

Type Integer

Values

−1	Determined automatically based on hardware configuration.
>0	Number of threads to use.

Default value −1

Note The value may be changed for specific parts of the optimization by the CONCURRENTTHREADS, MIPTHREADS and BARTHREADS controls.

Affects routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

See also DETERMINISTIC, MIPTHREADS, BARTHREADS, CONCURRENTTHREADS.

TRACE

Description Display the infeasibility diagnosis during presolve. If non-zero, an explanation of the logical deductions made by presolve to deduce infeasibility or unboundedness will be displayed on screen or sent to the message callback function.

Type Integer

Default value 0

Note Presolve is sometimes able to detect infeasibility and unboundedness in problems. The set of deductions made by presolve can allow the user to diagnose the cause of infeasibility or unboundedness in their problem. However, not all infeasibility or unboundedness can be detected and diagnosed in this way.

Affects routines XPRSlpoptimize (LPOPTIMIZE).

TREECOMPRESSION

Description When writing nodes to the gloal file, the optimizer can try to use data-compression techniques to reduce the size of the global file on disk. The TREECOMPRESSION control determines the strength of the data-compression algorithm used; higher values give superior data-compression at the affect of decreasing performance, while lower values compress quicker but not as effectively. Where TREECOMPRESSION is set to 0, no data compression will be used on the global file.

Type Integer

Default value 2

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also TREEMEMORYLIMIT.

TREECOVERCUTS

Description Branch and Bound: The number of rounds of lifted cover inequalities generated at nodes other than the top node in the tree. Compare with the description for COVERCUTS. A value of -1 indicates the number of rounds is determined automatically.

Type Integer

Default value −1

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

TREECUTSELECT

Description A bit vector providing detailed control of the cuts created during the tree search of a global solve. Use CUTSELECT to control cuts on the root node.

Type Integer

Values

Bit	Meaning
5	Clique cuts.
6	Mixed Integer Rounding (MIR) cuts.
7	Lifted cover cuts.
8	Turn on row aggregation for MIR cuts.
11	Flow path cuts.
12	Implication cuts.
13	Turn on automatic Lift and Project cutting strategy.
14	Disable cutting from cut rows.
15	Lifted GUB cover cuts.
16	Zero-half cuts.
17	Indicator constraint cuts.

Default value `-257`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `COVERCUTS`, `GOMCUTS`, `CUTSELECT`.

TREEDIAGNOSTICS

Description A bit vector providing control over how various tree-management-related messages get printed in the global logfile during the branch-and-bound search.

Type Integer

Values

Bit	Meaning
0	Output regular summaries of current tree memory usage.
1	Output messages whenever tree data is being written to global file.
1	Output progress messages while tree data is being written to the global flie, at an interval controlled by the `GLOBALFILELOGINTERVAL` control.

Default value `7`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `MIPLOG`, `PEAKTOTALTREEMEMORYUSAGE`, `GLOBALFILELOGINTERVAL`.

TREEGOMCUTS

Description Branch and Bound: The number of rounds of Gomory cuts generated at nodes other than the first node in the tree. Compare with the description for `GOMCUTS`. A value of -1 indicates the number of rounds is determined automatically.

Type Integer

Default value `-1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

TREEMEMORYLIMIT

Description A soft limit, in megabytes, for the amount of memory to use in storing the branch and bound search tree. This doesn't include memory used for presolve, heuristics, solving the LP relaxation, etc. When set to 0 (the default), the optimizer will calculate a limit automatically based on the amount of free physical memory detected in the machine. When the memory used by the branch and bound tree exceeds this limit, the optimizer will try to reduce the memory usage by writing lower-rated sections of the tree to a file called the "global file". Though the solve can continue if it cannot bring the tree memory usage below the specified limit, performance will be inhibited and a message will be printed to the log.

Type Integer

Default value 0 (calculate limit automatically)

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also TREEMEMORYSAVINGTARGET, TREECOMPRESSION, TREEDIAGNOSTICS.

TREEMEMORYSAVINGTARGET

Description When the memory used by the branch-and-bound search tree exceeds the limit specified by the TREEMEMORYLIMIT control, the optimizer will try to save memory by writing lower-rated sections of the tree to the global file. The target amount of memory to save will be enough to bring memory usage back below the limit, plus enough extra to give the tree room to grow. The TREEMEMORYSAVINGTARGET control specifies the extra proportion of the tree's size to try to save; for example, if the tree memory limit is 1000Mb and TREEMEMORYSAVINGTARGET is 0.1, when the tree size exceeds 1000Mb the optimizer will try to reduce the tree size to 900Mb. Reducing the value of TREEMEMORYSAVINGTARGET will cause less extra nodes of the tree to be written to the global file, but will result in the memory saving routine being triggered more often (as the tree will have less room in which to grow), which can reduce performance. Increasing the value of TREEMEMORYSAVINGTARGET will cause additional, more highly-rated nodes, of the tree to be written to the global file, which can cause performance issues if these nodes are required later in the solve.

Type Double

Default value 0.4

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also TREEMEMORYLIMIT

TREEPRESOLVE

Description Determines the amount of full presolving to apply to nodes of the branch-and-bound tree search.

Type Integer

Values

-1	Let the optimizer decide how often to presolve nodes.
0	Disabled.
1	Cautious strategy — presolve only when significant reductions are possible.
2	Medium strategy.
3	Aggressive strategy — presolve frequently.

Default value -1

Note The presolving of nodes will restrict the handling of user cuts. If a node in the tree has been presolved, it will not be possible to call XPRSloadcuts to load user cuts from the global pool into the node or any of its descendants. Any user cuts already loaded into a node problem will automatically be presolved with the node, but will afterwards appear as new user cuts. User cuts can still be added to a presolved node or its descendants, but any such cut must be presolved to match the node it is being loaded into. A cut can be presolved by calling XPRSpresolverow, and this should be done immediately before calling XPRSaddrows to load the cut, to ensure that the cut is being presolved to match the current node.

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also PRESOLVE.

TREEPRESOLVE_KEEPBASIS

Description Determines what to do with the existing basis when re-presolving a node of the branch-and-bound tree.

Type Integer

Values

0	The current basis is ignored and the LP relaxation of the presolved problem will be solved from scratch.
1	Attempt to presolve the current node basis and use it to warm-start the LP solve after the presolve. This can restrict some presolve reductions but should reduce the time for solving the LP relaxation.
2	Drop the basis during presolve, but attempt to create a valid warm-start basis based on the parent node solution.

Default value -1

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also TREEPRESOLVE.

TREEQCCUTS

Description Branch and Bound: Limit on the number of rounds of outer approximation cuts generated for nodes other than the root node, when solving a mixed integer quadratic constrained or mixed integer second order conic problem with outer approximation.

Type Integer

Default value -1 — determined automatically.

Note This control only has an effect for problems with quadratic or second order cone constraints, and only if outer approximation has not been disabled by setting MIQCPALG to 0.

Affects routines XPRSmipoptimize (MIPOPTIMIZE).

See also QCCUTS.

TUNERHISTORY

Description Tuner: Whether to reuse and append to previous tuner results of the same problem.

Type Integer

Values

0	Discard any previous tuner results.
1	Append new results to the previous tuner results, but do not reuse them.
2	Reuse the previous results and append new results to it.

Default value 2

Note Please refer to Section 5.12.5 for more information about reusing tuner results.

This control only has an effect on the tuner. This control cannot be tuned.

Affects routines `XPRStune` (`TUNE`).

TUNERMAXTIME

Description Tuner: The maximum time in seconds that the tuner will run before it terminates.

Type Integer

Values

`0`	No time limit.
`n>0`	Stop the tuner after `n` seconds.

Default value 0

Note This control only has an effect on the tuner. This control cannot be tuned.

Affects routines `XPRStune` (`TUNE`).

TUNERMETHOD

Description Tuner: Selects a factory tuner method. A tuner method consists of a list of controls with different settings that the tuner will evaluate and try to combine.

Type Integer

Values

`-1`	Automatically determined. The tuner will select the default method based on the problem type.
`0`	Select the default LP tuner method.
`1`	Select the default MIP tuner method.
`2`	Select a more comprehensive MIP tuner method.
`3`	Select a root-focus MIP tuner method.
`4`	Select a tree-focus MIP tuner method.
`5`	Select a simple MIP tuner method.
`6`	Select the default SLP tuner method.
`7`	Select the default MISLP tuner method.
`8`	Select a MIP tuner method focussed on primal heuristics.

Default value -1

Note If the tuner has already loaded a user-defined tuner method, then it will not load any factory tuner method.

Please refer to Section 5.12.2 for more information about the tuner method, and Appendix A.9 for the format of the tuner method file.

This control only has an effect on the tuner. This control cannot be tuned.

Affects routines `XPRStune` (`TUNE`).

TUNERMETHODFILE

Description Tuner: Defines a file from which the tuner can read user-defined tuner method.

Type String

Default value (empty)

Note If the tuner has already loaded a tuner method via `XPRStunerreadmethod`, then it will not check this control. Otherwise, when this control is defined and a tuner method can be succesfully loaded from this file, then the tuner will not load any factory tuner method.

Please refer to Section 5.12.2 for more information about the tuner method, and Appendix A.9 for the format of the tuner method file.

This control only has an effect on the tuner. This control cannot be tuned.

Affects routines `XPRStune` (`TUNE`).

TUNERMODE

Description Tuner: Whether to always enable the tuner or disable it.

Type Integer

Values

`-1`	No effect.
`0`	Always disable the tuner. `XPRStune` (`TUNE`) will have no effect.
`1`	Always enable the tuner. `XPRSmipoptimize` (`MIPOPTIMIZE`), `XPRSlpoptimize` (`LPOPTIMIZE`), etc. will call the tuner before solving the problem.

Default value `-1`

Note This control cannot be tuned.

Affects routines `XPRStune` (`TUNE`), `XPRSmipoptimize` (`MIPOPTIMIZE`), `XPRSlpoptimize` (`LPOPTIMIZE`).

TUNEROUTPUT

Description Tuner: Whether to output tuner results and logs to the file system.

Type Integer

Values

`0`	Don't output to the file system.
`1`	Output results and logs to the file system.

Default value `1`

Note Please refer to Section 5.12.3 for more information about the tuner output.

This control only has an effect on the tuner. This control cannot be tuned.

Affects routines `XPRStune` (`TUNE`).

TUNEROUTPUTPATH

Description Tuner: Defines a root path to which the tuner writes the result file and logs.

Type String

Default value tuneroutput

Note This control only defines the root path for the tuner output. For each problem, the tuner result will be output to a subfolder underneath this path. For example, by default, the tuner result for a problem called `prob` will be located at `tuneroutput/prob/`

Please refer to Section 5.12.3 for more information about the tuner output.

This control only has an effect on the tuner. This control cannot be tuned.

Affects routines `XPRStune` (`TUNE`).

TUNERPERMUTE

Description Tuner: Defines the number of permutations to solve for each control setting.

Type Integer

Values

`0`	Solve the original problem only for each setting.
`n>0`	Solve the original problem and `n` permuted problems for each setting.

Default value 0

Note Please refer to Section 5.12.7 for more information about tuner problem permutations.

This control only has an effect on the tuner. This control cannot be tuned.

Affects routines `XPRStune` (`TUNE`).

TUNERROOTALG

Description This control is deprecated, and is provided for compatibility purposes. Please use `LPFLAGS` instead. It is a bit-vector control which defines the algorithm for solving an LP problem or the initial LP relaxation of a MIP problem within the tuner specifically.

Type Integer

Values

Bit	Meaning
0	Use the dual simplex method.
1	Use the primal simplex method.
2	Use the barrier method.
3	Use the network simplex method.

Default value 0

Note Setting bit 0, 1, 2, 3 of this control will have the same effect of passing flags `d`, `p`, `b`, `n` to `XPRSmipoptimize` or `XPRSlpoptimize`. When more than one bit are set, then the LP problem will be solved with the concurrent solver.

This control only has an effect on the tuner.

This control can be tuned.

Affects routines `XPRStune` (`TUNE`).

TUNERSESSIONNAME

Description Tuner: Defines a session name for the tuner.

Type String

Default value (empty)

Note When defined, the session name will override the problem name within the tuner. For example, if this control is set to `session`, then the tuner result for a problem will be located at `tuneroutput/session/`

This control can be useful when the problem name is randomly generated.

Please refer to Section 5.12.3 for more information about the tuner output.

This control only has an effect on the tuner. This control cannot be tuned.

Affects routines `XPRStune` (`TUNE`).

TUNERTARGET

Description Tuner: Defines the tuner target – what should be evaluated when comparing two runs with different control settings.

Type Integer

Values

`-1`	Automatically determined. The tuner will choose the default target based on problem type.
`0`	Solution time then gap. (MIP/MISLP default)
`1`	Solution time then best bound.
`2`	Solution time then best integer solution.
`3`	The primal dual integral.
`4`	Time only. (LP/SLP default)
`5`	SLP objective only. (SLP/MISLP choice)
`6`	SLP validation number only. (SLP/MISLP choice)
`7`	Gap only.
`8`	Best bound only.
`9`	Best integer solution only.

Default value `-1`

Note Please refer to Section 5.12.4 for more information about tuner targets.

This control only has an effect on the tuner. This control cannot be tuned.

Affects routines XPRStune (TUNE).

TUNERTHREADS

Description Tuner: the number of threads used by the tuner.

Type Integer

Values

-1	Choose automaticlly.
1	The tuner will run in sequential.
n>1	The tuner will run in parallel with n threads.

Default value 1

Note Setting this control will not affect number of threads used by each individual run. It is recommended to have the product of TUNERTHREADS and THREADS less or equal to the number of system threads.

When setting TUNERTHREADS=-1, the tuner will automatically use as many threads as the number of logical processors detected.

Please refer to Section 5.12.6 for more information about tuner with multiple threads.

This control only has an effect on the tuner. This control cannot be tuned.

Affects routines XPRStune (TUNE).

TUNERVERBOSE

Description Tuner: whether the tuner should prints detailed information for each run.

Type Integer

Values

1	Print extra information.
0	Print less information.

Default value 1

Note Please refer to Section 5.12.6 for more information about tuner with multiple threads.

This control only has an effect on the tuner. This control cannot be tuned.

Affects routines XPRStune (TUNE).

USERSOLHEURISTIC

Description Determines how much effort to put into running a local search heuristic to find a feasible integer solution from a partial or infeasible user solution.

Type Integer

Values

`-1`	Automatically determined.
`0`	Search heuristic disabled.
`1`	Light effort.
`2`	Moderate effort.
`3`	High effort.

Default value `-1`

Note When a partial or infeasible user solution is added with `XPRSaddmipsol`, a local search heuristic will be applied to the problem in an attempt to find a feasible, integer solution that either completes the partial solution or is close to the infeasible solution. Whether to run such a heuristic, or how much effort to put into the heuristic can be controlled by this `USERSOLHEURISTIC` parameter.

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `HEURSEARCHROOTSELECT`, `HEURSEARCHTREESELECT`.

VARSELECTION

Description Branch and Bound: This determines the formula used to calculate the estimate of each integer variable, and thus which integer variable is selected to be branched on at a given node. The variable selected to be branched on is the one with the maximum estimate.

Type Integer

Values

`-1`	Determined automatically.
`1`	The minimum of the 'up' and 'down' pseudo costs.
`2`	The 'up' pseudo cost plus the 'down' pseudo cost.
`3`	The maximum of the 'up' and 'down' pseudo costs, plus twice the minimum of the 'up' and 'down' pseudo costs.
`4`	The maximum of the 'up' and 'down' pseudo costs.
`5`	The 'down' pseudo cost.
`6`	The 'up' pseudo cost.
`7`	A weighted combination of the 'up' and 'down' pseudo costs, where the weights depend on how fractional the variable is.
`8`	The product of the 'up' and 'down' pseudo costs.

Default value `-1`

Affects routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

VERSION

Description The Optimizer version number, e.g. `1301` meaning release 13.01.

Type Integer

Default value Software version dependent

CHAPTER 10

Problem Attributes

During the optimization process, various properties of the problem being solved are stored and made available to users of the FICO Xpress Libraries in the form of *problem attributes*. These can be accessed in much the same manner as for the controls. Examples of problem attributes include the sizes of arrays, for which library users may need to allocate space before the arrays themselves are retrieved. A full list of the attributes available and their types may be found in this chapter.

10.1 Retrieving Problem Attributes

Library users are provided with the following three functions for obtaining the values of attributes:

XPRSgetintattrib	XPRSgetdblattrib	XPRSgetstrattrib

Much as for the controls previously, it should be noted that the attributes as listed in this chapter *must* be prefixed with XPRS_ to be used with the FICO Xpress Libraries and failure to do so will result in an error. An example of their usage is the following which returns and prints the optimal value of the objective function after the linear problem has been solved:

```
XPRSgetdblattrib(prob, XPRS_LPOBJVAL, &lpobjval);

printf("The objective value is %2.1f\n", lpobjval);
```

ACTIVENODES

Description Number of outstanding nodes.

Type Integer

Set by routines XPRSmipoptimize (MIPOPTIMIZE), XPRSinitglobal.

ALGORITHM

Description The algorithm the optimizer currently is running / was running just before completition.

Type Integer

Values

1	No LP optimization yet.
2	Dual simplex.
3	Primal simplex.
4	Newton barrier.
5	Network simplex.

Note If the barrier with crossover is used, the value of `ALGORITHM` during the crossover and the final clean up will reflect the algorithm used, but will be reset to barrier once the optimization is complete.

ATTENTIONLEVEL

Description A measure between 0 and 1 for how numerically unstable the problem is. The attention level is based on a weighted combination of the number of basis condition numbers exceeding certain thresholds. It considers all nodes sampled by `MIPKAPPAFREQ`, with a setting of 1 being the most frequent sampling rate. The higher the attention level, the worse conditioned is the problem.

Type Double

Set by routines `XPRSmipoptimize`.

See also `MAXKAPPA`, `MIPKAPPAFREQ`, `PREDICTEDATTLEVEL`.

AVAILABLEMEMORY

Description The amount of heap memory detected by Xpress as free.

Type Integer

Note On 64bit systems this is a 64bit integer, use `XPRSgetintattrib64` to retrieve its value.

See also `PEAKMEMORY`, `CURRENTMEMORY`, `TOTALMEMORY`

BARAASIZE

Description Number of nonzeros in AA^T.

Type Integer

Set by routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

BARCGAP

Description Convergence criterion for the Newton barrier algorithm.

Type Double

Set by routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

BARCONDA

Description Absolute condition measure calculated in the last iteration of the barrier algorithm.

Type Double

Set by routines The barrier algorithm.

BARCONDD

Description Condition measure calculated in the last iteration of the barrier algorithm.

Type Double

Set by routines The barrier algorithm.

BARCROSSOVER

Description Indicates whether or not the basis crossover phase has been entered.

Type Integer

Values

`0` the crossover phase has not been entered.

`1` the crossover phase has been entered.

Set by routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

BARDENSECOL

Description Number of dense columns found in the matrix.

Type Integer

Set by routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

BARDUALINF

Description Sum of the dual infeasibilities for the Newton barrier algorithm.

Type Double

Set by routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

BARDUALOBJ

Description Dual objective value calculated by the Newton barrier algorithm.

Type Double

Set by routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

BARITER

Description Number of Newton barrier iterations.

Type Integer

Set by routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

BARLSIZE

Description Number of nonzeros in L resulting from the Cholesky factorization.

Type Integer

Set by routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

BARPRIMALINF

Description Sum of the primal infeasibilities for the Newton barrier algorithm.

Type Double

Set by routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

BARPRIMALOBJ

Description Primal objective value calculated by the Newton barrier algorithm.

Type Double

Set by routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

BARSING

Description Number of linearly dependent binding constraints at the optimal barrier solution. These results in singularities in the Cholesky decomposition during the barrier that may cause numerical troubles. Larger dependence means more chance for numerical difficulties.

Type Double

Set by routines The barrier algorithm.

BARSINGR

Description Regularized number of linearly dependent binding constraints at the optimal barrier solution. These results in singularities in the Cholesky decomposition during the barrier that may cause numerical troubles. Larger dependence means more chance for numerical difficulties.

Type Double

Set by routines The barrier algorithm.

BESTBOUND

Description Value of the best bound determined so far by the global search.

Type Double

Set by routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

BOUNDNAME

Description Active bound name.

Type String

Set by routines `XPRSreadprob`.

BRANCHVALUE

Description The value of the branching variable at a node of the Branch and Bound tree.

Type Double

Set by routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

BRANCHVAR

Description The branching variable at a node of the Branch and Bound tree.

Type Integer

Set by routines XPRSmipoptimize (MIPOPTIMIZE).

CALLBACKCOUNT_CUTMGR

Description This attribute counts the number of times the cut manager callback set by XPRSaddcbcutmgr has been called for the current node, including the current callback call. The value of this attribute should only be used from within the cut manager callback.

Type Integer

Set by routines XPRSmipoptimize (MIPOPTIMIZE).

CALLBACKCOUNT_OPTNODE

Description This attribute counts the number of times the optimal node callback set by XPRSaddcboptnode has been called for the current node, including the current callback call. The value of this attribute should only be used from within the optimal node callback.

Type Integer

Set by routines XPRSmipoptimize (MIPOPTIMIZE).

CHECKSONMAXCUTTIME

Description This attribute is used to set the value of the MAXCHECKSONMAXCUTTIME control. Its value is the number of times the optimizer checked the MAXCUTTIME criterion during the last call to the optimization routine XPRSmipoptimize. If a run terminates cutting operations on the MAXCUTTIME criterion then the attribute is the negative of the number of times the optimizer checked the MAXCUTTIME criterion up to and including the check when the termination was activated. Note that the attribute is set to zero at the beginning of each call to an optimization routine.

Type Integer

Set by routines XPRSmipoptimize (MIPOPTIMIZE).

CHECKSONMAXTIME

Description This attribute is used to set the value of the MAXCHECKSONMAXTIME control. Its value is the number of times the optimizer checked the MAXTIME criterion during the last call to the optimization routine XPRSmipoptimize. If a run terminates on the MAXTIME criterion then the attribute is the negative of the number of times the optimizer checked the MAXTIME criterion up to and including the check when the termination was activated. Note that the attribute is set to zero at the beginning of each call to an optimization routine.

Type Integer

Set by routines XPRSmipoptimize (MIPOPTIMIZE).

COLS

Description Number of columns (i.e. variables) in the matrix.

Type Integer

Note If the matrix is in a presolved state, this attribute returns the number of columns in the **presolved** matrix. If you require the value for the original matrix then use the ORIGINALCOLS attribute instead. The PRESOLVESTATE attribute can be used to test if the matrix is presolved or not. See also 5.3.

Set by routines XPRSloadglobal, XPRSloadlp, XPRSloadqglobal, XPRSloadqp, XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE) XPRSreadprob.

COMPUTEEXECUTIONS

Description The number of solves executed on a compute server.

Type Integer

Set by routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE), XPRStune, XPRSrepairinfeas, XPRSiisfirst, XPRSiisnext, XPRSiisall.

CONEELEMS

Description Number of second order cone coefficients in the problem.

Type Integer

Note If the matrix is in a presolved state, this attribute returns the number of the second order (including rotated second order) cone coefficients in the **presolved** matrix. Second order conic quadratic constraints are automaticly detected at optimization time, and this attribute is not set before optimizing the problem.

Set by routines Optimizing the problem.

CONES

Description Number of second order and rotated second order cones in the problem.

Type Integer

Note If the matrix is in a presolved state, this attribute returns the number of second order (including rotated second order) cones in the **presolved** matrix. Conic quadratic constraints are automaticly detected at optimization time, and this attribute is not set before optimizing the problem.

Set by routines Optimizing the problem.

CORESDETECTED

Description Number of logical cores detected by the optimizer, which is the total number of threads the hardware can execute across all CPUs.

Type Integer

Values `>=1` Detected number of logical cores.

Note The optimizer will automatically use as many solver threads as the number of logical cores detected.

If the detection fails, the optimizer will default to using a single thread only.

Set by routines `XPRSinit`.

See also `THREADS`, `CORESPERCPUDETECTED`, `CPUSDETECTED`, `PHYSICALCORESDETECTED`, `PHYSICALCORESPERCPUDETECTED`.

CORESPERCPUDETECTED

Description Number of logical cores per CPU unit detected by the optimizer, which is the number of threads each CPU can execute.

Type Integer

Values `>=1` Detected number of logical cores per CPU unit.

Set by routines `XPRSinit`.

See also `THREADS`, `CORESDETECTED`, `CPUSDETECTED`, `PHYSICALCORESDETECTED`, `PHYSICALCORESPERCPUDETECTED`.

CPUSDETECTED

Description Number of CPU units detected by the optimizer.

Type Integer

Values `>=1` Detected number of CPU units.

Set by routines `XPRSinit`.

See also `THREADS`, `CORESDETECTED`, `CORESPERCPUDETECTED`, `PHYSICALCORESDETECTED`, `PHYSICALCORESPERCPUDETECTED`.

CURRENTMEMORY

Description The amount of dynamically allocated heap memory by the problem being solved.

Type Integer

Note On 64bit systems this is a 64bit integer, use `XPRSgetintattrib64` to retrieve its value.

See also `PEAKMEMORY`, `CURRENTMEMORY`, `TOTALMEMORY`

CURRENTNODE

Description The unique identifier of the current node in the tree search.

Type Integer

Note The root node is always identified as node 1.

Set by routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `PARENTNODE`.

CURRMIPCUTOFF

Description The current MIP cut off.

Type Double

Set by routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `MIPABSCUTOFF`.

CUTS

Description Number of cuts being added to the matrix.

Type Integer

Set by routines XPRSaddcuts, XPRSdelcpcuts, XPRSdelcuts, XPRSloadcuts, XPRSloadmodelcuts.

DUALINFEAS

Description Number of dual infeasibilities.

Type Integer

Note If the matrix is in a presolved state, this attribute returns the number of dual infeasibilities in the **presolved** matrix. If you require the value for the original matrix, make sure you obtain the value when the matrix is not presolved. The PRESOLVESTATE attribute can be used to test if the matrix is presolved or not. See also 5.3.

Set by routines XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE).

See also PRIMALINFEAS.

ELEMS

Description Number of matrix nonzeros (elements).

Type Integer

Note If the matrix is in a presolved state, this attribute returns the number of matrix nonzeros in the **presolved** matrix. If you require the value for the original matrix, make sure you obtain the value when the matrix is not presolved. The PRESOLVESTATE attribute can be used to test if the matrix is presolved or not. See also 5.3.

Set by routines XPRSloadglobal, XPRSloadlp, XPRSloadqglobal, XPRSloadqp, XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE), XPRSreadprob.

ERRORCODE

Description The most recent Optimizer error number that occurred. This is useful to determine the precise error or warning that has occurred, after an Optimizer function has signalled an error by returning a non-zero value. The return value itself is **not** the error number. Refer to the section 11.2 for a list of possible error numbers, the errors and warnings that they indicate, and advice on what they mean and how to resolve them. A short error message may be obtained using XPRSgetlasterror, and all messages may be intercepted using the user output callback function; see XPRSaddcbmessage.

Type Integer

Set by routines Any.

GENCONCOLS

Description Number of input variables in general constraints in the problem.

Type Integer

Note The total number of input variables in MIN/MAX/AND/OR/ABS constraints in the problem.

Set by routines `XPRSaddgencons`

GENCONS

Description Number of general constraints in the problem.

Type Integer

Note The total number of MIN/MAX/AND/OR/ABS constraints in the problem.

Set by routines `XPRSaddgencons`

GENCONVALS

Description Number of constant values in general constraints in the problem.

Type Integer

Note The total number of constant values in MIN/MAX constraints in the problem.

Set by routines `XPRSaddgencons`

GLOBALFILESIZE

Description The allocated size of the global file, in megabytes. Because data can be removed from the global file during the branch and bound search, the size of the global file is usually greater than the amount of data currently within it (represented by the `GLOBALFILEUSAGE` attribute).

Type Integer

See also `GLOBALFILEUSAGE`.

GLOBALFILEUSAGE

Description The number of megabytes of data from the branch-and-bound tree that have been saved to the global file. Note that the actual allocated size of the global file (represented by the `GLOBALFILESIZE` control) may be greater than this value.

Type Integer

Set by routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `GLOBALFILESIZE`, `GLOBALFILEBIAS`, `TREEMEMORYLIMIT`.

INDICATORS

Description Number of indicator constrains in the problem.

Type Integer

Note When the matrix is in a presolved state, the indicator constraints are stored in a special pool and not part of the matrix. Otherwise the indicator constraints are rows of the matrix and their details can be retrieved with the `XPRSgetindicators` function. The `PRESOLVESTATE` attribute can be used to test if the matrix is presolved or not. See also 5.3.

Set by routines `XPRSsetindicators`, `XPRSdelindicators`, `XPRSreadprob`.

LPOBJVAL

Description Value of the objective function of the last LP solved.

Type Double

Set by routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `MIPOBJVAL`, `OBJRHS`.

LPSTATUS

Description LP solution status.

Type Integer

Values

0	Unstarted (XPRS_LP_UNSTARTED).
1	Optimal (XPRS_LP_OPTIMAL).
2	Infeasible (XPRS_LP_INFEAS).
3	Objective worse than cutoff (XPRS_LP_CUTOFF).
4	Unfinished (XPRS_LP_UNFINISHED).
5	Unbounded (XPRS_LP_UNBOUNDED).
6	Cutoff in dual (XPRS_LP_CUTOFF_IN_DUAL).
7	Problem could not be solved due to numerical issues. (XPRS_LP_UNSOLVED).
8	Problem contains quadratic data, which is not convex (XPRS_LP_NONCONVEX).

Note The possible return values are defined as constants in the Optimizer C header file and VB .bas file.

Set by routines XPRSlpoptimize (LPOPTIMIZE).

See also MIPSTATUS.

MATRIXNAME

Description The matrix name.

Type String

Note This is the name read from the MATRIX field in an MPS matrix, and is *not* related to the problem name used in the Optimizer. Use XPRSgetprobname to get the problem name.

Set by routines XPRSreadprob, XPRSsetprobname.

MAXABSDUALINFEAS

Description Maximum calculated absolute dual infeasibility in the unscaled original problem.

Type Double

Set by routines XPRSlpoptimize, XPRSmipoptimize.

MAXABSPRIMALINFEAS

Description Maximum calculated absolute primal infeasibility in the unscaled original problem.

Type Double

Set by routines XPRSlpoptimize, XPRSmipoptimize, XPRSrefinemipsol.

MAXKAPPA

Description Largest basis condition number (also known as kappa) calculated through all nodes sampled by `MIPKAPPAFREQ`.

Type Double

Set by routines `XPRSmipoptimize`.

See also `MIPKAPPAFREQ`.

MAXMIPINFEAS

Description Maximum integer fractionality in the solution.

Type Double

Set by routines `XPRSmipoptimize`.

MAXPROBNAMELENGTH

Description Maximum size of the problem name and also the maximum allowed length of the file or path string for any function that accepts such an argument.

Type Integer

Set by routines `XPRSgetprobname`, `XPRSsetprobname`.

MAXRELDUALINFEAS

Description Maximum calculated relative dual infeasibility in the unscaled original problem.

Type Double

Set by routines `XPRSlpoptimize`, `XPRSmipoptimize`.

MAXRELPRIMALINFEAS

Description Maximum calculated relative primal infeasibility in the unscaled original problem.

Type Double

Set by routines `XPRSlpoptimize`, `XPRSmipoptimize`.

MIPBESTOBJVAL

Description Objective function value of the best integer solution found.

Type Double

Set by routines XPRSmipoptimize (MIPOPTIMIZE).

See also MIPOBJVAL.

MIPENTS

Description Number of global entities (i.e. binary, integer, semi-continuous, partial integer, and semi-continuous integer variables) but excluding the number of special ordered sets.

Type Integer

Note If the matrix is in a presolved state, this attribute returns the number of global entities in the **presolved** matrix. If you require the value for the original matrix, make sure you obtain the value when the matrix is not presolved. The PRESOLVESTATE attribute can be used to test if the matrix is presolved or not. See also 5.3.

Set by routines XPRSaddcols, XPRSchgcoltype, XPRSdelcols, XPRSloadglobal, XPRSloadqglobal, XPRSreadprob.

See also SETS.

MIPINFEAS

Description Number of integer infeasibilities, including violations of special ordered sets, at the current node.

Type Integer

Set by routines XPRSmipoptimize (MIPOPTIMIZE).

See also PRIMALINFEAS.

MIPOBJVAL

Description Objective function value of the last integer solution found.

Type Double

Set by routines XPRSmipoptimize (MIPOPTIMIZE).

See also MIPBESTOBJVAL.

MIPSOLNODE

Description Node at which the last integer feasible solution was found.

Type Integer

Set by routines XPRSmipoptimize (MIPOPTIMIZE).

MIPSOLS

Description Number of integer solutions that have been found.

Type Integer

Set by routines XPRSmipoptimize (MIPOPTIMIZE).

MIPSTATUS

Description Global (MIP) solution status.

Type Integer

Values

0	Problem has not been loaded (XPRS_MIP_NOT_LOADED).
1	Global search incomplete - the initial continuous relaxation has not been solved and no integer solution has been found (XPRS_MIP_LP_NOT_OPTIMAL).
2	Global search incomplete - the initial continuous relaxation has been solved and no integer solution has been found (XPRS_MIP_LP_OPTIMAL).
3	Global search incomplete - no integer solution found (XPRS_MIP_NO_SOL_FOUND).
4	Global search incomplete - an integer solution has been found (XPRS_MIP_SOLUTION).
5	Global search complete - no integer solution found (XPRS_MIP_INFEAS).
6	Global search complete - integer solution found (XPRS_MIP_OPTIMAL).
7	Global search incomplete - the initial continuous relaxation was found to be unbounded. A solution may have been found (XPRS_MIP_UNBOUNDED).

Note The possible return values are defined as constants in the Optimizer C header file and VB .bas file.

Set by routines XPRSmipoptimize (MIPOPTIMIZE).

See also LPSTATUS.

MIPTHREADID

Description The ID for the MIP thread.

Type Integer

Note The first MIP thread has ID 0 and is the same as the main thread. All other threads are new threads and are destroyed when the global search is halted.

Set by routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `MIPTHREADS`.

NAMELENGTH

Description The length (in 8 character units) of row and column names in the matrix. To allocate a character array to store names, you must allow `8*NAMELENGTH+1` characters per name (the `+1` allows for the string terminator character).

Type Integer

Set by routines `XPRSloadglobal`, `XPRSloadlp`, `XPRSloadqglobal`, `XPRSloadqp`, `XPRSreadprob`.

NODEDEPTH

Description Depth of the current node.

Type Integer

Set by routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

NODES

Description Number of nodes solved so far in the global search. A node is counted as solved when it is either dropped or branched on.

Type Integer

Note The root node has depth 1.

Set by routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

NUMIIS

Description Number of IISs found.

Type Integer

Set by routines `IIS`, `XPRSiisfirst`, `XPRSiisnext`, `XPRSiisall`.

OBJNAME

Description Active objective function row name.

Type String

Set by routines XPRSreadprob.

OBJRHS

Description Fixed part of the objective function.

Type Double

Note If the matrix is in a presolved state, this attribute returns the fixed part of the objective in the **presolved** matrix. If you require the value for the original matrix, make sure you obtain the value when the matrix is not presolved. The PRESOLVESTATE attribute can be used to test if the matrix is presolved or not. See also 5.3. If an MPS file contains an objective function coefficient in the RHS then the negative of this will become OBJRHS.

Set by routines XPRSchgobj.

See also LPOBJVAL.

OBJSENSE

Description Sense of the optimization being performed.

Type Double

Values

-1.0	For maximization problems.
1.0	For minimization problems.

Note The objective sense of a problem can be changed using XPRSchgobjsense.

Set by routines XPRSchgobjsense (CHGOBJSENSE).

ORIGINALCOLS

Description Number of columns (i.e. variables) in the original matrix before presolving.

Type Integer

Note If you require the value for the presolved matrix then use the COLS attribute.

Set by routines XPRSloadglobal, XPRSloadlp, XPRSloadqglobal, XPRSloadqp, XPRSreadprob.

ORIGINALGENCONS

Description Number of general constraints in the original problem before presolving.

Type Integer

Note If you require the value for the presolved problem then use the `GENCONS` attribute.

Set by routines `XPRSaddgencons`.

ORIGINALGENCONCOLS

Description Number of input variables in general constraints in the original problem before presolving.

Type Integer

Note If you require the value for the presolved problem then use the `GENCONCOLS` attribute.

Set by routines `XPRSaddgencons`.

ORIGINALGENCONVALS

Description Number of constant values in general constraints in the original problem before presolving.

Type Integer

Note If you require the value for the presolved problem then use the `GENCONVALS` attribute.

Set by routines `XPRSaddgencons`.

ORIGINALINDICATORS

Description Number of indicator constraints in the original matrix before presolving.

Type Integer

Note If you require the value for the presolved matrix then use the `INDICATORS` attribute.

Set by routines `XPRSloadglobal`, `XPRSloadlp`, `XPRSloadqglobal`, `XPRSloadqp`, `XPRSreadprob`.

ORIGINALMIPENTS

Description Number of global entities (i.e. binary, integer, semi-continuous, partial integer, and semi-continuous integer variables) but excluding the number of special ordered sets in the original matrix before presolving.

Type Integer

Note If you require the value for the presolved matrix then use the MIPENTS attribute. .

Set by routines XPRSloadglobal, XPRSloadlp, XPRSloadqglobal, XPRSloadqp, XPRSreadprob.

ORIGINALPWLS

Description Number of piecewise linear constraints in the original problem before presolving.

Type Integer

Note If you require the value for the presolved problem then use the PWLCONS attribute.

Set by routines XPRSaddpwlcons.

ORIGINALPWLPOINTS

Description Number of breakpoints of piecewise linear constraints in the original problem before presolving.

Type Integer

Note If you require the value for the presolved problem then use the PWLPOINTS attribute.

Set by routines XPRSaddpwlcons.

ORIGINALQCONSTRAINTS

Description Number of rows with quadratic coefficients in the original matrix before presolving.

Type Integer

Note If you require the value for the presolved matrix then use the QCONSTRAINTS attribute.

Set by routines XPRSloadglobal, XPRSloadlp, XPRSloadqglobal, XPRSloadqp, XPRSreadprob.

ORIGINALQCELEMS

Description Number of quadratic row coefficients in the original matrix before presolving.

Type Integer

Note If you require the value for the presolved matrix then use the QCELEMS attribute.

Set by routines XPRSloadglobal, XPRSloadlp, XPRSloadqglobal, XPRSloadqp, XPRSreadprob.

ORIGINALQELEMS

Description Number of quadratic elements in the original matrix before presolving.

Type Integer

Note If you require the value for the presolved matrix then use the `QELEMS` attribute.

Set by routines `XPRSloadglobal`, `XPRSloadlp`, `XPRSloadqglobal`, `XPRSloadqp`, `XPRSreadprob`.

ORIGINALSETMEMBERS

Description Number of variables within special ordered sets (set members) in the original matrix before presolving.

Type Integer

Note If you require the value for the presolved matrix then use the `SETMEMBERS` attribute.

Set by routines `XPRSloadglobal`, `XPRSloadlp`, `XPRSloadqglobal`, `XPRSloadqp`, `XPRSreadprob`.

ORIGINALSETS

Description Number of special ordered sets in the original matrix before presolving.

Type Integer

Note If you require the value for the presolved matrix then use the `SETS` attribute.

Set by routines `XPRSloadglobal`, `XPRSloadlp`, `XPRSloadqglobal`, `XPRSloadqp`, `XPRSreadprob`.

ORIGINALROWS

Description Number of rows (i.e. constraints) in the original matrix before presolving.

Type Integer

Note If you require the value for the presolved matrix then use the `ROWS` attribute.

Set by routines `XPRSaddrows`, `XPRSdelrows`, `XPRSloadglobal`, `XPRSloadlp`, `XPRSloadqglobal`, `XPRSloadlp`, `XPRSreadprob`.

PARENTNODE

Description The parent node of the current node in the tree search.

Type Integer

Set by routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

PEAKMEMORY

Description An estimate of the peak amount of dynamically allocated heap memory by the problem.

Type Integer

Note On 64bit systems this is a 64bit integer, use `XPRSgetintattrib64` to retrieve its value.

See also `CURRENTMEMORY`,`SYSTEMMEMORY`

PEAKTOTALTREEMEMORYUSAGE

Description The peak size, in megabytes, that the branch-and-bound search tree reached during the solve. Note that this value will include the uncompressed size of any compressed data and the size of any data saved to the global file.

Type Integer

Set by routines `XPRSmipoptimize`.

See also `TREEMEMORYUSAGE`.

PENALTYVALUE

Description The weighted sum of violations in the solution to the relaxed problem identified by the infeasibility repair function.

Type Double

Set by routines `XPRSrepairinfeas` (`REPAIRINFEAS`), `XPRSrepairweightedinfeas`.

PHYSICALCORESDETECTED

Description The total number of physical cores across all CPUs detected by the optimizer.

Type Integer

Values `>=1` Detected number of physical cores.

Set by routines `XPRSinit`.

See also `CORESDETECTED`, `CORESPERCPUDETECTED`, `CPUSDETECTED`, `PHYSICALCORESPERCPUDETECTED`.

PHYSICALCORESPERCPUDETECTED

Description The number of physical cores per CPU detected by the optimizer.

Type Integer

Values `>=1` Detected number of physical cores per CPU.

Set by routines `XPRSinit`.

See also `CORESDETECTED`, `CORESPERCPUDETECTED`, `CPUSDETECTED`, `PHYSICALCORESPERCPUDETECTED`.

PREDICTEDATTLEVEL

Description A measure between 0 and 1 to predict how numerically unstable the current MIP solve can be expected to be. After the root LP solve, a machine learning model is used to predict the actual `ATTENTIONLEVEL` which will only be computed if `MIPKAPPAFREQ` is set to a nonzero value. If the predicted attention level exceeds a value of 0.1, a message will be printed to the log.

Type Double

Set by routines `XPRSmipoptimize`.

See also `ATTENTIONLEVEL`, `MAXKAPPA`.

PRESOLVEINDEX

Description Presolve: The row or column index on which presolve detected a problem to be infeasible or unbounded.

Type Integer

Note Row indices are in the range 0 to `ROWS-1`, and column indices are in the range `ROWS+SPAREROWS` to `ROWS+SPAREROWS+COLS-1`.

Set by routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

PRESOLVESTATE

Description Problem status as a bit map.

Type Integer

Values

Bit	Meaning
0	Problem has been loaded.
1	Problem has been LP presolved.
2	Problem has been MIP presolved.
7	Solution in memory is valid.

Note Other bits are reserved.

Set by routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

PRIMALDUALINTEGRAL

Description Value of the primal-dual integral.

Type Double

Note This attribute represents the integral of the primal-dual gap over time. It measures the convergence of the best (dual) bound `BESTBOUND` and the primal bound `MIPBESTOBJVAL` over the whole solving time. Lower values are better. For details on the primal(-dual) integral see Berthold: *Measuring the impact of primal heuristics*, OR Letters 41(6), pp. 611-614, 2013.

Set by routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `BESTBOUND`, `MIPBESTOBJVAL`.

PRIMALINFEAS

Description Number of primal infeasibilities.

Type Integer

Note If the matrix is in a presolved state, this attribute returns the number of primal infeasibilities in the **presolved** matrix. If you require the value for the original matrix, make sure you obtain the value when the matrix is not presolved. The `PRESOLVESTATE` attribute can be used to test if the matrix is presolved or not. See also 5.3.

Set by routines `XPRSlpoptimize` (`LPOPTIMIZE`).

See also `SUMPRIMALINF`, `DUALINFEAS`, `MIPINFEAS`.

PWLCONS

Description Number of piecewise linear constraints in the problem.

Type Integer

Set by routines XPRSaddpwlcons

PWLPOINTS

Description Number of breakpoints of piecewise linear constraints in the problem.

Type Integer

Note The total number of breakpoints over all piecewise linear constraints.

Set by routines XPRSaddpwlcons

QCELEMS

Description Number of quadratic row coefficients in the matrix.

Type Integer

Note If the matrix is in a presolved state, this attribute returns the number of quadratic row coefficients in the **presolved** matrix.

Set by routines XPRSaddqmatrix, XPRSchgqrowcoeff, XPRSgetqrowqmatrixtriplets, XPRSloadqcqp.

QCONSTRAINTS

Description Number of rows with quadratic coefficients in the matrix.

Type Integer

Note If the matrix is in a presolved state, this attribute returns the number of rows with quadratic coefficients in the **presolved** matrix.

Set by routines XPRSaddqmatrix, XPRSchgqrowcoeff, XPRSgetqrowqmatrixtriplets, XPRSloadqcqp.

QELEMS

Description Number of quadratic elements in the matrix.

Type Integer

Note If the matrix is in a presolved state, this attribute returns the number of quadratic elements in the **presolved** matrix. If you require the value for the original matrix, make sure you obtain the value when the matrix is not presolved. The PRESOLVESTATE attribute can be used to test if the matrix is presolved or not. See also 5.3.

Set by routines XPRSchgmqobj, XPRSchgqobj, XPRSloadqglobal, XPRSloadqp.

RANGENAME

Description Active range name.

Type String

Set by routines XPRSreadprob.

RHSNAME

Description Active right hand side name.

Type String

Set by routines XPRSreadprob.

ROWS

Description Number of rows (i.e. constraints) in the matrix.

Type Integer

Note If the matrix is in a presolved state, this attribute returns the number of rows in the **presolved** matrix. If you require the value for the original matrix then use the ORIGINALROWS attribute instead. The PRESOLVESTATE attribute can be used to test if the matrix is presolved or not. See also 5.3.

Set by routines XPRSaddrows, XPRSdelrows, XPRSloadglobal, XPRSloadlp, XPRSloadqglobal, XPRSloadlp, XPRSlpoptimize (LPOPTIMIZE), XPRSmipoptimize (MIPOPTIMIZE), XPRSreadprob.

SIMPLEXITER

Description Number of simplex iterations performed.

Type Integer

Set by routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

SETMEMBERS

Description Number of variables within special ordered sets (set members) in the matrix.

Type Integer

Note If the matrix is in a presolved state, this attribute returns the number of variables within special ordered sets in the **presolved** matrix. If you require the value for the original matrix, make sure you obtain the value when the matrix is not presolved. The `PRESOLVESTATE` attribute can be used to test if the matrix is presolved or not. See also 5.3.

Set by routines `XPRSloadglobal`, `XPRSloadqglobal`, `XPRSreadprob`.

See also `SETS`.

SETS

Description Number of special ordered sets in the matrix.

Type Integer

Note If the matrix is in a presolved state, this attribute returns the number of special ordered sets in the **presolved** matrix. If you require the value for the original matrix, make sure you obtain the value when the matrix is not presolved. The `PRESOLVESTATE` attribute can be used to test if the matrix is presolved or not. See also 5.3.

Set by routines `XPRSloadglobal`, `XPRSloadqglobal`, `XPRSreadprob`.

See also `SETMEMBERS`, `MIPENTS`.

SPARECOLS

Description Number of spare columns in the matrix.

Type Integer

Set by routines `XPRSloadglobal`, `XPRSloadlp`, `XPRSloadqglobal`, `XPRSloadqp`, `XPRSreadprob`.

SPAREELEMS

Description Number of spare matrix elements in the matrix.

Type Integer

Set by routines `XPRSloadglobal`, `XPRSloadlp`, `XPRSloadqglobal`, `XPRSloadqp`, `XPRSreadprob`.

SPAREMIPENTS

Description Number of spare global entities in the matrix.

Type Integer

Set by routines `XPRSloadglobal`, `XPRSloadlp`, `XPRSloadqglobal`, `XPRSloadqp`, `XPRSreadprob`.

SPAREROWS

Description Number of spare rows in the matrix.

Type Integer

Set by routines `XPRSloadglobal`, `XPRSloadlp`, `XPRSloadqglobal`, `XPRSloadqp`, `XPRSreadprob`.

SPARESETELEMS

Description Number of spare set elements in the matrix.

Type Integer

Set by routines `XPRSloadglobal`, `XPRSloadlp`, `XPRSloadqglobal`, `XPRSloadqp`, `XPRSreadprob`.

SPARESETS

Description Number of spare sets in the matrix.

Type Integer

Set by routines `XPRSloadglobal`, `XPRSloadlp`, `XPRSloadqglobal`, `XPRSloadqp`, `XPRSreadprob`.

STOPSTATUS

Description Status of the optimization process.

Type Integer

Note Possible values are:

Value	Description
XPRS_STOP_NONE	no interruption - the solve completed normally
XPRS_STOP_TIMELIMIT	time limit hit
XPRS_STOP_CTRLC	control C hit
XPRS_STOP_NODELIMIT	node limit hit
XPRS_STOP_ITERLIMIT	iteration limit hit
XPRS_STOP_MIPGAP	MIP gap is sufficiently small
XPRS_STOP_SOLLIMIT	solution limit hit
XPRS_STOP_USER	user interrupt.

Set by routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

SUMPRIMALINF

Description Scaled sum of primal infeasibilities.

Type Double

Note If the matrix is in a presolved state, this attribute returns the scaled sum of primal infeasibilities in the **presolved** matrix. If you require the value for the original matrix, make sure you obtain the value when the matrix is not presolved. The `PRESOLVESTATE` attribute can be used to test if the matrix is presolved or not. See also 5.3.

Set by routines `XPRSlpoptimize` (`LPOPTIMIZE`).

See also `PRIMALINFEAS`.

SYSTEMMEMORY

Description The amount of non problem specific memory used by the solver.

Type Integer

Note On 64bit systems this is a 64bit integer, use `XPRSgetintattrib64` to retrieve its value.

See also `CURRENTMEMORY`, `PEAKMEMORY`

TIME

Description Time spent solving the problem as measured by the optimizer.

Type Integer

Set by routines `XPRSlpoptimize` (`LPOPTIMIZE`), `XPRSmipoptimize` (`MIPOPTIMIZE`).

TOTALMEMORY

Description The amount of dynamically allocated heap memory by the optimizer, including all problems currently exsisting.

Type Integer

Note On 64bit systems this is a 64bit integer, use `XPRSgetintattrib64` to retrieve its value.

See also `PEAKMEMORY`, `CURRENTMEMORY`

TREECOMPLETION

Description Estimation of the relative completion of the search tree as fraction between 0 and 1. Its accuracy mainly depends on the level of degeneracy of a problem and the balancedness of the search tree.

Type Double

Set by routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

TREEMEMORYUSAGE

Description The amount of physical memory, in megabytes, currently being used to store the branch-and-bound search tree.

Type Integer

Set by routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

See also `TREEMEMORYLIMIT`, `GLOBALFILEUSAGE`.

TREERESTARTS

Description Number of in-tree restarts performed.

Type Integer

Set by routines `XPRSmipoptimize` (`MIPOPTIMIZE`).

UUID

Description Universally Unique Identifier for the problem instance.

Type String

XPRESSVERSION

Description The Xpress verion number.

Type String

Note The version number of Xpress.

Chapter 11

Return Codes and Error Messages

11.1 Optimizer Return Codes

The table below shows the possible return codes from the subroutine library functions. See also the **MIP Solution Pool Reference Manual** for MIP Solution Pool Errors.

Return Code	Description
0	Subroutine completed successfully.
1[a]	Bad input encountered.
2[a]	Bad or corrupt file - unrecoverable.
4[a]	Memory error.
8[a]	Corrupt use.
16[a]	Program error.
32	Subroutine not completed successfully, possibly due to invalid argument.
128	Too many users.
a - Unrecoverable error.	

When the Optimizer terminates after the `STOP` command, it may set an exit code that can be tested by the operating system or by the calling program. The exit code is set as follows:

Return Code	Description
0	Program terminated normally (with `STOP`).
63	LP optimization unfinished.
64	LP feasible and optimal.
65	LP infeasible.
66	LP unbounded.
67	IP optimal solution found.
68	IP search incomplete but an IP solution has been found.
69	IP search incomplete, no IP solution found.
70	IP infeasible.
99	LP optimization not started.
255	Xpress Optimizer has not been initialized.

11.2 Optimizer Error and Warning Messages

Following a premature exit, the Optimizer can be interrogated as necessary to obtain more information about the specific error or warning which occurred. Library users may return a description of errors or warnings as they are encountered using the function `XPRSgetlasterror`. This function returns information related to the error code, held in the problem attribute `ERRORCODE`. For Console users the value of this attribute is output to the screen as errors or warnings are encountered. For Library users it must be retrieved using:

```
XPRSgetintattrib(prob,XPRS_ERRORCODE,&errorcode);
```

The following list contains values of `ERRORCODE` and a possible resolution of the error or warning.

3 ***Extension not allowed - ignored.***

The specified extension is not allowed. The Optimizer ignores the extension and truncates the filename.

4 ***Column <col> has no upper bound.***

Column <col> cannot be at its upper bound in the supplied basis since it does not have one. A new basis will be created internally where column <col> will be at its lower bound while the rest of the columns and rows maintain their basic/non-basic status.

5 ***Error on `.`<ext> file.***

An error has occurred on the `.` <ext> file. Please make sure that there is adequate disk space for the file and that it has not become corrupted.

6 ***No match for column <col> in matrix.***

Column <col> has not been defined in the `COLUMNS` section of the matrix and cannot be used in subsequent sections. Please check that the spelling of <col> is correct and that it is not written outside the field reserved for column names.

7 ***Empty matrix. Please increase `EXTRAROWS`.***

There are too few rows or columns. Please increase `EXTRAROWS` before input, or make sure there is at least one row in your matrix and try to read it again.

9 ***Error on read of basis file.***

The basis file `.BSS` is corrupt. Please make sure that there is adequate disk space for the file and that it has not been corrupted.

11 ***Not allowed - solution not optimal.***

The operation you are trying to perform is not allowed unless the solution is optimal. Please call `XPRSmaxim` (`MAXIM`) or `XPRSminim` (`MINIM`) to optimize the problem and make sure the process is completed. If the control `LPITERLIMIT` has been set, make sure that the optimal solution can be found within the maximum number of iterations allowed.

18 ***Bound conflict for column <col>.***

Specified upper bound for column <col> is smaller that the specified lower bound. Please change one or both bounds to solve the conflict and try again.

19 ***Eta overflow straight after invert - unrecoverable.***

There is not enough memory for eta arrays. Either increase the virtual paging space or the physical memory.

20 ***Insufficient memory for array <array>.***

There is not enough memory for an internal data structure. Either increase the virtual paging space or the physical memory.

21 ***Unidentified section The command is not recognized by the Optimizer.***

Please check the spelling and try again. Please refer to the Reference Manual for a list of valid commands.

29 ***Input aborted.***

Input has encountered too many problems in reading your matrix and it has been aborted. This message will be preceded by other error messages whose error numbers will give information about the nature of each of the problems. Please correct all errors and try again.

36 ***Linear Optimizer only***

You are only authorized to use the Linear Optimizer. Please contact your local sales office to discuss upgrading to the IP Optimizer if you wish to use this command.

38 ***Invalid option.***

One of the options you have specified is incorrect. Please check the input option and retype the command. A list of valid options for each command can be found in 8.

41 ***Global error - contact the Xpress support team.***

Internal error. Please contact your local support office.

45 ***Failure to open global file - aborting. (Perhaps disk is full).***

The Optimizer cannot open the `.GLB` file. This usually occurs when your disk is full. If this is not the case it means that the `.GLB` file has been corrupted.

50 ***Inconsistent basis.***

Internal basis held in memory has been corrupted. Please contact your local support office.

52 ***Too many nonzero elements.***

The number of matrix elements exceeds the maximum allowed. If you have the Hyper version then increase your virtual page space or physical memory. If you have purchased any other version of the software please contact your local sales office to discuss upgrading if you wish to read matrices with this number of elements.

56 ***Reference row entries too close for set <set> member <col>.***

The coefficient of column <col> in the constraint being used as reference row for set <set> is too close to the coefficient of some other column in the reference row. Please make sure the coefficients in the reference row differ enough from one another. One way of doing this is to create a non computational constraint (`N` type) that contains all the variables members of the set <set> and then assign coefficients whose distance from each other is of at least 1 unit.

58 ***Duplicate element for column <col> row <row>.***

The coefficient for column <col> appears more than once in row <row>. The elements are added together but please make sure column <col> only has one coefficient in <row> to avoid this warning message.

61 ***Unexpected EOF on workfile.***

An internal workfile has been corrupted. Please make sure that there is adequate disk space and try again. If the problem persists please contact your local support office.

64 ***Error closing file <file>.***

The Optimizer could not close file <file>. Please make sure that the file exists and that it is not being used by another application.

65 ***Fatal error on read from workfile <file> - program aborted.***

An internal workfile has been corrupted. Please make sure that your disk has enough space and try again. If the problem persists please contact your local support office.

66 ***Unable to open file <file>.***

The Optimizer has failed to open the file <file>. Please make sure that the file exists and there is adequate disk space.

67 ***Error on read of file <file>.***

The Optimizer has failed to read the file <file>. Please make sure that the file exists and that it has not been corrupted.

71 ***Not a basic vector: <vector>.***

Dual value of row or column <vector> cannot be analyzed because the vector is not basic.

72 ***Not a non-basic vector: <vector>.***

Activity of row or column <vector> cannot be analyzed because the vector is basic.

73 ***Problem has too many rows. The maximum is <num>.***

The Optimizer cannot input your problem since the number of rows exceeds <num>, the maximum allowed. If you have purchased any other than the Hyper version of the software please contact your local sales office to discuss upgrading it to solve larger problems.

76 ***Illegal priority: entity <ent> value <num>.***

Entity <ent> has been assigned an invalid priority value of <num> in the directives files and this priority will be ignored. Please make sure that the priority value lies between 0 and 1000 and that it is written inside the corresponding field in the `.DIR` file.

77 ***Illegal set card <line>.***

The set definition in line <line> of the `.MAT` or .MPS file creates a conflict. Please make sure that the set has a correct type and has not been already defined. Please refer to the Reference Manual for a list of valid set types.

80 ***File creation error.***

The Optimizer cannot create a file. Please make sure that these is adequate disk space and that the volume is not corrupt.

81 ***Fatal error on write to workfile <file> - program aborted.***

The Optimizer cannot write to the file <file>. Please make sure that there is adequate disk space and that the volume is not corrupt.

83 ***Fatal error on write to file - program aborted.***

The Optimizer cannot write to an internal file. Please make sure that there is adequate disk space and that the volume is not corrupt.

84 ***Input line too long. Maximum line length is <num>***

A line in the `.MAT` or `.MPS` file has been found to be too long. Please reduce the length to be less or equal than <num> and input again.

85 ***File not found: <file>.***

The Optimizer cannot find the file <file>. Please check the spelling and that the file exists. If this file has to be created by the Optimizer, make sure that the process which creates the file has been performed.

89 ***No optimization has been attempted.***

The operation you are trying to perform is not allowed unless the solution is optimal. Please call XPRSmaxim (MAXIM) or XPRSminim (MINIM) to optimize the problem and make sure the process is completed. If you have set the control LPITERLIMIT make sure that the optimal solution can be found within the maximum number of iterations allowed.

91 ***No problem has been input.***

An operation has been attempted that requires a problem to have been input. Please make sure that XPRSreadprob (READPROB) is called and that the problem has been loaded successfully before trying again.

97 ***Split vector <vector>.***

The declaration of column <vector> in the COLUMN section of the .MAT or .MPS file must be done in contiguous line. It is not possible to interrupt the declaration of a column with lines corresponding to a different vector.

98 ***At line <num> no match for row <row>.***

A non existing row <row> is being used at line number <num> of the .MAT or .MPS file. Please check spelling and make sure that <row> is defined in the ROWS section.

102 ***Eta file space exceeded - optimization aborted.***

The Optimizer requires more memory. Please increase your virtual paging space or physical memory and try to optimize again.

107 ***Too many global entities at column <col>.***

The Optimizer cannot input your problem since the number of global entities exceeds the maximum allowed. If you have the Hyper version then increase your virtual page space or physical memory. If you have purchased any other version of the software please contact your local sales office to discuss upgrading it to solve larger problems.

111 ***Duplicate row <row> - ignored.***

Row <row> is used more than once in the same section. Only the first use is kept and subsequent ones are ignored.

112 ***Postoptimal analysis not permitted on presolved problems.***

Re-optimize with PRESOLVE = 0. An operation has been attempted on the presolved problem. Please optimize again calling XPRSmaxim (MAXIM), XPRSminim (MINIM) with the l flag or turning presolve off by setting PRESOLVE to 0.

113 ***Unable to restore version <ver> save files.***

The svf file was created by a different version of the Optimizer and cannot be restored with this version.

114 ***Fatal error - pool hash table full at vector <vector>.***

Internal error. Please contact your local support office.

120 ***Problem has too many rows and columns. The maximum is <num>***

The Optimizer cannot input your problem since the number of rows plus columns exceeds the maximum allowed. If you have purchased any other than the Hyper version of the software please contact your local sales office to discuss upgrading it to solver larger problems.

122 ***Corrupt solution file.***

Solution file .SOL could not be accessed. Please make sure that there is adequate disk space and that the file is not being used by another process.

124 ***Invalid parameter value passed to <function>. Parameter value <param_name> is not allowed***

A parameter lookup by name has failed. The provided parameter name does not match any parameters in Xpress.

127 ***Not found: <vector>.***

An attempt has been made to use a row or column <vector> that cannot be found in the problem. Please check spelling and try again.

128 ***Cannot load directives for problem with no global entities.***

The problem does not have global entities and so directives cannot be loaded.

129 ***Access denied to problem state : '<name>' (<routine>).***

The user is not licensed to have set or get access to problem control (or attribute) <name>. The routine used for access was <routine>.

130 ***Bound type illegal <type>.***

Illegal bound type <type> has been used in the basis file `.BSS`. A new basis will be created internally where the column with the illegal bound type will be at its lower bound and the rest of the columns and rows will maintain their basic/non-basic status. Please check that you are using `XPRSreadbasis` (`READBASIS`) with the `t` flag to read compact format basis.

131 ***No column: <col>.***

Column <col> used in basis file `.BSS` does not exist in the problem. A new basis will be created internally from where column <col> will have been removed and the rest of columns and rows will maintain their basic/non-basic status.

132 ***No row: <row>.***

Row <row> used in basis file `.BSS` does not exist in the problem. A new basis will be created internally from where row <row> will have been removed and the rest of columns and rows will maintain their basic/non-basic status.

136 ***Cannot access control <control_name> via attribute routine <function>***

When accessing controls and attributes, the API function called must be matched appropriately to the type (double, int, string) and access type (control / attribute) of the parameter.

137 ***Bad internal state found in 'Struct' lookup : <parameter_table> (<parameter_name>)***

A parameter provided could not be found in the parameters table. This is an internal error, please contact FICO support.

140 ***Basis lost - recovering.***

The number of rows in the problem is not equal to the number of basic rows + columns in the problem, which means that the existing basis is no longer valid. This will be detected when re-optimizing a problem that has been altered in some way since it was last optimized (see below). A correct basis is generated automatically and no action needs to be taken. The basis can be lost in two ways: (1) if a row is deleted for which the slack is non-basic: the number of rows will decrease by one, but the number of basic rows + columns will be unchanged. (2) if a basic column is deleted: the number of basic rows + columns will decrease by one, but the number of rows will be unchanged. You can avoid losing the basis by only deleting rows for which the slack is basic, and columns which are non-basic. (The `XPRSgetbasis` function can be used to determine the basis status.) To delete a non-basic row without losing the basis, bring it into the basis first, and to delete a basic column without losing the basis, take it out of the basis first - the functions `XPRSgetpivots` and `XPRSpivot` may be useful here. However, remember that the message is only a warning and the Optimizer will generate a new basis automatically if necessary.

142 ***Type illegal <type>.***

An illegal priority type <type> has been found in the directives file `.DIR` and will be ignored. Please refer to Appendix A for a description of valid priority types.

143 ***No entity <ent>.***

Entity <ent> used in directives file `.DIR` cannot be found in the problem and its corresponding priority will be ignored. Please check spelling and that the column <ent> is actually declared as an entity in the `BOUNDS` section or is a set member.

151 ***Illegal MARKER.***

The line marking the start of a set of integer columns or a set of columns belonging to a Special Ordered Set in the `.MPS` file is incorrect.

152 ***Unexpected EOF.***

The Optimizer has found an unexpected EOF marker character. Please check that the input file is correct and input again.

153 ***Illegal card at line <line>.***

Line <line> of the `.MPS` file could not be interpreted. Please refer to the Reference Manual for information about the valid MPS format.

155 ***Cannot access control '<id>' via attribute routine <routine>.***

Controls cannot be accessed from attribute access routines.

156 ***Cannot access attribute '<id>' via control routine <routine>.***

Attributes cannot be accessed from control access routines.

157 ***Cannot access attribute <attribute_name> via control routine <routine>.***

Attributes cannot be accessed from control access routines.

158 ***Unrecognized callback name <callback> (<function>)***

The callback name provided to the API function is not recognized.

159 ***Failed to set default controls.***

Attempt failed to set controls to their defaults.

160 ***Cannot access <typename> type '<id>' via routine <routine>.***

Accessing an attribute or control requires using a routine with matching type.

161 ***Cannot access <typename> type '<name>' via routine <routine>.***

Accessing an attribute or control requires using a routine with matching type.

162 ***Recording and playback error : <info>.***

An error occurred in the recording and playback tool.

163 ***Failed to copy controls.***

Attempt failed to copy controls defined for one problem to another.

164 ***Problem is not presolved.***

Action requires problem to be presolved and the problem is not presolved.

167 ***Failed to allocate memory of size <bytes> bytes.***

The Optimizer failed to allocate required memory of size <bytes>.

168 ***Required resource not currently available : '<name>'.***

The resource <name> is required by an action but is unavailable.

169 ***Failed to create resource : '<name>'.***

The resource <name> failed to create.

170 ***Corrupt global file.***

Global file `.GLB` cannot be accessed. Please make sure that there is adequate disk space and that the file is not being used by another process.

171 ***Invalid row type for row <row>.***

`XPRSalter` (`ALTER`) cannot change the row type of <row> because the new type is invalid. Please correct and try again.

173 ***Name not recognized : <name>.***

The control name cannot be recognized.

178 ***Not enough spare rows to remove all violations.***

The Optimizer could not add more cuts to the matrix because there is not enough space. Please increase `EXTRAROWS` before input to improve performance.

179 ***Load MIP solution failed : '<status description>'.***

Attempt failed to load MIP solution into the Optimizer. See <status description» for details of the failure.

180 ***No change to this control allowed.***

The Optimizer does not allow changes to this control. If you have the student version, please contact your local sales office to discuss upgrading if you wish to change the value of controls. Otherwise check that the Optimizer was initialized properly and did not revert to student mode because of a security problem.

181 ***Cannot alter bound on BV, SC, UI, PI, or set member.***

`XPRSalter` (`ALTER`) cannot be used to change the upper or lower bound of a variable if its variable type is binary, semi-continuous, integer, partial integer, semi-continuous integer, or if it is a set member.

186 ***Inconsistent number of variables in problem.***

A compact format basis is being read into a problem with a different number of variables than the one for which the basis was created.

187 ***Unable to restore alternative system <system> save files.***

The svf file was created on a different operating system and cannot be restored on the current system.

191 ***Solution in file '<file>' (rows:<nrow>, cols:<ncol>) not compatible with problem.***

The size of the loaded problem is not compatible with problem size from the solution file.

192 ***Bad flags <flag string>.***

A flag string passed into a command line call is invalid.

193 ***Possible unexpected results from `XPRSreadbinsol` (`READBINSOL`) : <message>.***

A call to the `XPRSreadbinsol` (`READBINSOL`) may produce unexpected results. See <message> for details.

194 ***Failure writing to range file.***

Failure writing to range file.

195 ***Cannot read LP solution into presolved problem.***

An LP solution cannot be read into a problem in a presolved state.

197 ***Failed to register callback for event : '<event>'.***

Registering callback for an event failed.

199 ***Network simplex not authorized***

The Optimizer cannot use the network algorithm. Please contact your local sales office to upgrade your authorization if you wish to use it.

202 ***Control parser: <error>.***

A generic control parser error, for example a memory allocation failure.

243 ***The Optimizer requires a newer version of the XPRL library.***

You are using the XPRS library from one Xpress distribution and the XPRS library from a previous Xpress distribution. You should remove all other Xpress distributions from your system library path environment variable.

245 ***Not enough memory to presolve matrix.***

The Optimizer required more memory to presolve the matrix. Please increase your virtual paging space or physical memory. If this is not possible try setting `PRESOLVE` to `0` before optimizing, so that the presolve procedure is not performed.

247 ***Directive on non-global entity not allowed: <col>.***

Column <col> used in directives file `.DIR` is not a global entity and its corresponding priority will be ignored. A variable is a 'global entity' it is type is not continuous or if it is a set member. Please refer to Appendix A for details about valid entities and set types.

249 ***Insufficient improvement found.***

Insufficient improvement was found between barrier iterations which has caused the barrier algorithm to terminate.

250 ***Too many numerical errors.***

Too many numerical errors have been encountered by the barrier algorithm and this has caused the barrier algorithm to terminate.

251 ***Out of memory.***

There is not enough memory for the barrier algorithm to continue.

256 ***Simplex Optimizer only***

The Optimizer can only use the simplex algorithm. Please contact your local sales office to upgrade your authorization if you wish to use this command.

257 ***Simplex Optimizer only***

The Optimizer can only use the simplex algorithm. Please contact your local sales office to upgrade your authorization if you wish to use this command.

259 ***Warning: The Q matrix may not be semi-definite.***

The Q matrix must be positive (negative) semi-definite for a minimization (maximization) problem in order for the problem to be convex. The barrier algorithm has encountered numerical problems which indicate that the problem is not convex.

261 ***<ent> already declared as a global entity - old declaration ignored.***

Entity <ent> has already been declared as global entity. The new declaration prevails and the old declaration prevails and the old declaration will be disregarded.

262 ***Unable to remove shift infeasibilities of `&`.***

Perturbations to the right hand side of the constraints which have been applied to enable problem to be solved cannot be removed. It may be due to round off errors in the input data or to the problem being badly scaled.

263 ***The problem has been presolved.***

The problem in memory is the presolved one. An operation has been attempted on the presolved problem. Please optimize again calling `XPRSmaxim` (`MAXIM`), `XPRSminim` (`MINIM`) with the `l` flag or tuning presolve off by setting `PRESOLVE` to `0`. If the operation does not need to be performed on an optimized problem just load the problem again.

264 ***Not enough spare matrix elements to remove all violations.***

The Optimizer could not add more cuts to the matrix because there is not enough space. Please increase `EXTRAELEMS` before input to improve performance.

266 ***Cannot read basis for presolved problem. Re-input matrix.***

The basis cannot be read because the problem in memory is the presolved one. Please reload the problem with `XPRSreadprob` (`READPROB`) and try to read the basis again.

268 ***Cannot perform operation on presolved matrix. Please postsolve or re-input matrix.***

The problem in memory is the presolved one. Please postsolve or reload the problem and try the operation again.

279 ***The Optimizer has not been initialized.***

The Optimizer could not be initialized successfully. Please initialize it before attempting any operation and try again.

287 ***Cannot read in directives after the problem has been presolved.***

Directives cannot be read if the problem in memory is the presolved one. Please reload the problem and read the directives file `.DIR` before optimizing. Alternatively, re-optimize using the `-l` flag or set `PRESOLVE` to 0 and try again.

293 ***This license file does not specify the permitted problem size. Contact your vendor to obtain a valid license.***

The license file is invalid as it doesn't specify the permitted problem size. Please contact your local sales office.

302 ***Option must be C/c or O/o.***

The only valid options for the type of goals are C, c, O and o. Any other answer will be ignored.

305 ***Row <row> (number <num>) is an `N` row.***

Only restrictive rows, i.e. `G`, `L`, `R` or `E` type, can be used in this type of goal programming. Please choose goal programming for objective functions when using `N` rows as goals.

306 ***Option must be MAX/max or MIN/min.***

The only valid options for the optimization sense are MAX, max, MIN and min. Any other answer will be ignored.

307 ***Option must be P/p or D/d.***

The only valid options for the type of relaxation on a goal are P, p, D and d. Any other answer will be ignored.

308 ***Row <row> (number <num>) is an unbounded goal.***

Goal programming has found goal <row> to be unbounded and it will stop at this point. All goals with a lower priority than <row> will be ignored.

309 ***Row <row> (number <num>) is not an `N` row.***

Only `N` type rows can be selected as goals for this goal programming type. Please use goal programming for constraints when using rows whose type is not `N`.

310 ***Option must be A/a or P/p.***

The only valid options for the type of goal programming are A, a, P and p. Any other answer will be ignored.

314 ***Invalid number.***

The input is not a number. Please check spelling and try again.

316 ***Not enough space to add deviational variables.***

Increase `EXTRACOLS` before input. The Optimizer cannot find spare columns to spare deviational variables. Please try increasing `EXTRACOLS` before input to at least twice the number of constraint goals and try again.

318 ***Maximum number of allowed goals is 100.***

Goal programming does not support more than 100 goals and will be interrupted.

319 ***No Optimizer license found. Please contact your vendor to obtain a license.***

Your license does not authorize the direct use of the Optimizer. You probably have a license that authorizes other Xpress products, for example Mosel or BCL.

320 ***An internal error has occured. Please report to " SUPPORT_CONTACT_NAME " the circumstances under which this happened.***

An internal error has occured. Please report to " SUPPORT_CONTACT_NAME " the circumstances under which this happened.

324 ***Not enough extra matrix elements to complete elimination phase.***

Increase `EXTRAPRESOLVE` before input to improve performance. The elimination phase performed by the presolve procedure created extra matrix elements. If the number of such elements is larger than allowed by the `EXTRAPRESOLVE` parameter, the elimination phase will stop. Please increase `EXTRAPRESOLVE` before loading the problem to improve performance.

326 ***Linear Optimizer only***

You are not authorized to use the Quadratic Programming Optimizer. Please contact your local sales office to discuss upgrading to the QP Optimizer if you wish to use this command.

352 ***Command not authorized in this version.***

There has been an attempt to use a command for which your Optimizer is not authorized. Please contact your local sales office to upgrade your authorization if you wish to use this command.

361 ***`QMATRIX` or `QUADOBJ` section must be after `COLUMN` section.***

Error in matrix file. Please make sure that the `QMATRIX` or `QUADOBJ` sections are after the `COLUMNS` section and try again.

362 ***Duplicate elements not allowed in `QUADOBJ` section.***

The coefficient of a column appears more than once in the `QUADOBJ` section. Please make sure all columns have only one coefficient in this section.

363 ***Quadratic matrix must be symmetric in `QMATRIX` section.***

Only symmetric matrices can be input in the `QMATRIX` section of the `.MAT` or `.MPS` file. Please correct and try again.

368 ***`QSECTION` second element in line ignored: <line>.***

The second element in line <line> will be ignored.

381 ***Bug in lifting of cover inequalities.***

Internal error. Please contact you local support office.

386 ***This version is not authorized to run Goal Programming.***

The Optimizer you are using is not authorized to run Goal Programming. Please contact you local sales office to upgrade your authorization if you wish to use this command.

392 ***This version is not authorized to be called from BCL.***

This version of the Optimizer cannot be called from the subroutine library BCL. Please contact your local sales office to upgrade your authorization if you wish to run the Optimizer from BCL.

394 ***Fatal communications error.***

There has been a communication error between the master and the slave processes. Please check the network and try again.

395 ***This version is not authorized to be called from the Optimizer library.***

This version of the Optimizer cannot be called from the Optimizer library. Please contact your local sales office to upgrade your authorization if you wish to run the Optimizer using the libraries.

401 ***Invalid row type passed to <function>.***

Elements <num> of your array has invalid row type <type>. There has been an error in one of the arguments of function <function>. The row type corresponding to element <num> of the array is invalid. Please refer to the section corresponding to function <function> in 8 for further information about the row types that can be used.

402 ***Invalid row number passed to <function>.***

Row number <num> is invalid. There has been an error in one of the arguments of function <function>. The row number corresponding to element <num> of the array is invalid. Please make sure that the row numbers are not smaller than 0 and not larger than the total number of rows in the problem.

403 ***Invalid global entity passed to <function>.***

Element <num> of your array has invalid entity type <type>. There has been an error in one of the arguments of function <function>. The column type <type> corresponding to element <num> of the array is invalid for a global entity.

404 ***Invalid set type passed to <function>.***

Element <num> of your array has invalid set type <type>. There has been an error in one of the arguments of function <function>. The set type <type> corresponding to element <num> of the array is invalid for a set entity.

405 ***Invalid column number passed to <function>.***

Column number <num> is invalid. There has been an error in one of the arguments of function <function>. The column number corresponding to element <num> of the array is invalid. Please make sure that the column numbers are not smaller than `0` and not larger than the total number of columns in the problem, `COLS`, minus `1`. If the function being called is `XPRSgetobj` or `XPRSchgobj` a column number of `-1` is valid and refers to the constant in the objective function.

406 ***Invalid row range passed to <function>.***

Limit <lim> is out of range. There has been an error in one of the arguments of function <function>. The row numbers lie between 0 and the total number of rows of the problem. Limit <lim> is outside this range and therefore is not valid.

407 ***Invalid column range passed to <function>.***

Limit <lim> is out of range. There has been an error in one of the arguments of function <function>. The column numbers lie between 0 and the total number of columns of the problem. Limit <lim> is outside this range and therefore is not valid.

409 ***Invalid directive passed to <function>.***

Element <num> of your array has invalid directive <type>. There has been an error in one of the arguments of function <function>. The directive type <type> corresponding to element <num> of the array is invalid. Please refer to the Reference Manual for a list of valid directive types.

410 ***Invalid row basis type passed to <function>.***

Element <num> of your array has invalid row basis type <type>. There has been an error in one of the arguments of function <function>. The row basis type corresponding to element <num> of the array is invalid.

411 ***Invalid column basis type passed to <function>.***

Element <num> of your array has invalid column basis type <type>. There has been an error in one of the arguments of function <function>. The column basis type corresponding to element <num> of the array is invalid.

412 ***Invalid parameter number passed to <function>.***

Parameter number <num> is out of range. LP or MIP parameters and controls can be used in functions by passing the parameter or control name as the first argument or by passing an associated number. In this case number <num> is an invalid argument for function <function> because it does not correspond to an existing parameter or control. If you are passing a number as the first argument, please substitute it with the name of the parameter or control whose value you wish to set or get. If you are already passing the parameter or control name, please check 8 to make sure that is valid for function <function>.

413 ***Not enough spare rows in <function>.***

Increase `EXTRAROWS` before input. There are not enough spare rows to complete function <function> successfully. Please increase `EXTRAROWS` before `XPRSreadprob` (`READPROB`) and try again.

414 ***Not enough spare columns in <function>.***

Increase `EXTRACOLS` before input. There are not enough spare columns to complete function <function> successfully. Please increase `EXTRACOLS` before `XPRSreadprob` (`READPROB`) and try again.

415 ***Not enough spare matrix elements in <function>.***

Increase `EXTRAELEMS` before input. There are not enough spare matrix elements to complete function <function> successfully. Please increase `EXTRAELEMS` before `XPRSreadprob` (`READPROB`) and try again.

416 ***Invalid bound type passed to <function>.***

Element <elem> of your array has invalid bound type <type>. There has been an error in one of the arguments of function <function>. The bound type <type> of element number <num> of the array is invalid.

417 ***Invalid complement flag passed to <function>. Element <elem> of your array has invalid complement flag <flag>.***

Element <elem> of your array has an invalid complement flag <flag>. There has been an error in one of the arguments of function <function>. The complement flag corresponding to indicator constraint <num> of the array is invalid.

418 ***Invalid cut number passed to <function>.***

Element <num1> of your array has invalid cut number <num2>. Element number <num1> of your array contains a cut which is not stored in the cut pool. Please check that <num2> is a valid cut number.

419 ***Not enough space to store cuts in <function>.***

There is not enough space to complete function <function> successfully.

420 ***Too many saved matrices in savmat***

Version 12 compatibility interface only. There is a hard limit of at most 64 matrices that can be saved with `savmat` or `cpymat`

421 ***Matrix no. <mat> has not been saved. Cannot restore in resmat***

Version 12 compatibility interface only. No matrix with number <mat> has been saved with `savmat` or `cpymat`.

422 ***Solution is not available.***

There is no solution available. This could be because the problem in memory has been changed or optimization has not been performed. Please optimize and try again.

423 ***Duplicate rows/columns passed to <function>.***

Element <elem> of your array has duplicate row/col number <num>. There has been an error in one of the arguments of function <function>. The element number <elem> of the argument array is a row or column whose sequence number <num> is repeated.

424 ***Not enough space to store cuts in <function>.***

There is not enough space to complete function <function> successfully.

425 ***Column already basic.***

The column cannot be pivoted into the basis since it is already basic. Please make sure the variable is non-basic before pivoting it into the basis.

426 ***Column not eligible to leave basis.***

The column cannot be chosen to leave the basis since it is already non-basic. Please make sure the variable is basic before forcing it to leave the basis.

427 ***Invalid column type passed to <function>.***

Element <num> of your array has invalid column type <type>. There has been an error in one of the arguments of function <function>. The column type <type> corresponding to element <num> of the array is invalid.

429 ***No basis is available.***

No basis is available.

430 ***Column types cannot be changed during the global search.***

The Optimizer does not allow changes to the column type while the global search is in progress. Please call this function before starting the global search or after the global search has been completed. You can call `XPRSmaxim` (`MAXIM`) or `XPRSminim` (`MINIM`) with the `l` flag if you do not want to start the global search automatically after finding the LP solution of a problem with global entities.

434 ***Invalid name passed to `XPRSgetindex`.***

A name has been passed to `XPRSgetindex` which is not the name of a row or column in the matrix.

436 ***Cannot trace infeasibilities when integer presolve is turned on.***

Try `XPRSmaxim` (`XPRSmaxim`) / `XPRSminim` (`MINIM`) with the `l` flag. Integer presolve can set upper or lower bounds imposed by the column type as well as those created by the interaction of the problem constraints. The infeasibility tracing facility can only explain infeasibilities due to problem constraints.

459 ***Not enough memory for branch and bound tree***
Not enough resources for branch and bound tree (<type>)
Failure locking branch and bound tree (probably out of memory)
Failure in handling of branch and bound tree (<type>)

Functions to signal that an unexpected error happened during the management of the branch-and-bound tree for storing information from a global solve. The string <type> will provide more information about the particular failure. These errors are typical of running out of memory.

473 ***Row classification not available.***

474 ***Column passed to <routine> has inconsistent bounds. See column <index> of <count>.***

The bounds are inconsistent for column <index> of the <count> columns passed into routine <routine>.

475 ***Inconsistent bounds [<lb>,<ub>] for column <column name> in call to <routine>.***

The lower bound <lb> is greater than the upper bound <ub> in the bound pair given for column <column name> passed into routine <routine>.

476 ***Unable to round bounds [<lb>,<ub>] for integral column <column name> in call to <routine>.***

Either the lower bound <lb> is greater than the upper bound <ub> in the bound pair given for the integer column <column name> passed into routine <routine> or the interval defined by <lb> and <ub> does not contain an integer value.

501 ***Error at <line> Empty file.***

Read aborted. The Optimizer cannot read the problem because the file is empty.

502 ***Warning: 'min' or 'max' not found at <line.col>. No objective assumed.***

An objective function specifier has not been found at column <col>, line <line> of the LP file. If you wish to specify an objective function please make sure that 'max', 'maximize', 'maximum', 'min', 'minimize' or 'minimum' appear.

503 ***Objective not correctly formed at <line.col>. Aborting.***

The Optimizer has aborted the reading of the problem because the objective specified at line <line> of the LP file is incorrect.

504 ***No keyword or empty problem at <line.col>.***

There is an error in column <col> at line <line> of the LP file. Neither 'Subject to', 'subject to:', 'subject to', 'such that' 's.t.', or 'st' can be found. Please correct and try again.

505 ***A keyword was expected at <line.col>.***

A keyword was expected in column <col> at line <line> pf the LP file. Please correct and try again.

506 ***The constraint at <line.col> has no term.***

A variable name is expected at line <line> column <col>: either an invalid character (like '+' or a digit) was encountered or the identifier provided is unknown (new variable names are declared in constraint section only).

507 ***RHS at <line.col> is not a constant number.***

Line <line> of the LP file will be ignored since the right hand side is not a constant.

508 ***The constraint at <line> has no term.***

The LP file contains a constraint with no terms.

509 ***The type of the constraint at <line.col> has not been specified.***

The constraint defined in column <col> at line <line> of the LP file is not a constant and will be ignored.

510 ***Upper bound at <line.col> is not a numeric constant.***

The upper bound declared in column <col> at line <line> of the LP file is not a constant and will be ignored.

511 ***Bound at <line.col> is not a numeric constant.***

The bound declared in column <col> at line <line> of the LP file is not a constant and will be ignored.

512 ***Unknown word starting with an 'f' at <line.col>. Treated as 'free'.***

A word starting with an 'f' and not known to the Optimizer has been found in column <col> at line <line> of the LP file. The word will be read into the Optimizer as 'free'.

513 ***Wrong bound statement at <line.col>.***

The bound statement in column <col> at line <line> is invalid and will be ignored.

514 ***Lower bound at <line.col> is not a numeric constant. Treated as -inf.***

The lower bound declared in column <col> at line <line> of the LP file is not a constant. It will be translated into the Optimizer as the lowest possible bound.

515 ***Sign '<' expected at <line.col>.***

A character other than the expected sign '<' has been found in column <col> at line <line> of the LP file. This line will be ignored.

516 ***Problem has not been loaded.***

The problem could not be loaded into the Optimizer. Please check the other error messages appearing with this message for more information.

517 ***Row names have not been loaded.***

The name of the rows could not be loaded into the Optimizer. Please check the other error messages appearing with this message for more information.

518 ***Column names have not been loaded.***

The name of the columns could not be loaded into the Optimizer. Please check the other error messages appearing with this message for more information.

519 ***Not enough memory at <line.col>.***

The information in column <col> at line <line> of the LP file cannot be read because all the allocated memory has already been used. Please increase your virtual page space or physical memory and try again.

520 ***Unexpected EOF at <line.col>.***

An unexpected EOF marker character has been found at line <line> of the LP file and the loading of the problem into the Optimizer has been aborted. Please correct and try again.

521 ***Number expected for exponent at <line.col>.***

The entry in column <col> at line <line> of the LP file is not a properly expressed real number and will be ignored.

522 ***Line <line> too long (length>255).***

Line <line> of the LP file is too long and the loading of the problem into the Optimizer has been aborted. Please check that the length of the lines is less than 255 and try again.

523 ***The Optimizer cannot reach line <line.col>.***

The reading of the LP file has failed due to an internal problem. Please contact your local support office.

524 ***Constraints could not be read into the Optimizer. Error found at <line.col>.***

The reading of the LP constraints has failed due to an internal problem. Please contact your local support office.

525 ***Bounds could not be set into the Optimizer. Error found at <line.col>.***

The setting of the LP bounds has failed due to an internal problem. Please contact your local support office.

526 ***LP problem could not be loaded into the Optimizer. Error found at <line.col>.***

The reading of the LP file has failed due to an internal problem. Please contact your local support office.

527 ***Copying of rows unsuccessful.***

The copying of the LP rows has failed due to an internal problem. Please contact your local support office.

528 ***Copying of columns unsuccessful.***

The copying of the LP columns has failed due to an internal problem. Please contact your local support office.

529 ***Redefinition of constraint at <line.col>.***

A constraint is redefined in column <col> at line <line> of the LP file. This repeated definition is ignored.

530 ***Name too long. Truncating it.***

The LP file contains an identifier longer than 64 characters: it will be truncated to respect the maximum size.

531 ***Sign '>' expected here <line>.***

A greater than sign was expected in the LP file.

532 ***Quadratic term expected here <pos>***

The LP file reader expected to read a quadratic term at position <pos>: a variable name and '2' or the product of two variables. Please check the quadratic part of the objective in the LP file.

533 ***Wrong exponent value. Treated as 2 <pos>***

The LP file reader encountered an exponent different than 2 at position <pos>. Such exponents are automatically replaced by 2.

535 ***A constraint name was expected here.***

The LP file reader expected to read a row name in the ranges section.

536 ***Wrong range statement at <pos>.***

The LP file reader expected to read a range type in the ranges section.

538 ***Error when loading the SOS names***

The LP format file reader failed to create the SOS names. The previous error should explain why this failed.

539 ***Invalid indicator constraint condition at <line.col>***

The condition part in column <col> of the indicator constraint at line <line> is invalid.

545 ***A variable name was found but ignored at <pos> due to not appearing before.***

The LP file reader read a variable in bounds or integer type sections that does not appear in the matrix.

552 ***'S1|2:' expected here. Skipping <pos>***

Unknown set type read while reading the LP file at position <pos>. Please use set type 'S1' or 'S2'.

553 ***This set has no member. Ignoring it <pos>***

An empty set encountered while reading the LP file at position <pos>. The set has been ignored.

554 ***Weight expected here. Skipping <pos>***

A missing weight encountered while reading sets in the LP file at position <pos>. Please check definitions of the sets in the file.

555 ***Cannot presolve cut with `PRESOLVEOPS` bits 0, 5 or 8 set or bit 11 cleared.***

Can not presolve cut with `PRESOLVEOPS` bits 0, 5 or 8 set or bit 11 cleared.
No cuts can be presolved if the following presolve options are turned on:
bit 0: singleton column removal,
bit 5: duplicate column removal,
bit 8: variable eliminations
or if the option
bit 11: No advanced IP reductions is turned off. Please check the presolve settings.

557 ***Integer solution is not available***

Failed to retrieve an integer solution because no integer solution has been identified yet.

558 ***Column <col> duplicated in basis file - new entry ignored.***

Column <col> is defined in the basis file more than once. Any repeated definitions are ignored.

559 ***The old feature <feature> is no longer supported***

The feature <feature> is no longer supported and has been removed. Please contact Xpress support for help about replacement functionality.

602 ***Values must be specified for all columns when column indices are not provided.***

In a call to `XPRSaddmipsol` the column index array is optional. When this argument is omitted (given as `NULL`), the length of the solution value array must match `ORIGINALCOLS`.

604 ***String passed as parameter is too long***

The file name passed to `XPRSsetlogfile` can be at most 200 characters long.

606 ***Failed to parse list of diving heuristic strategies at position <pos>***

Invalid diving heuristic strategy number provided in position <pos> of the string controls `HEURDIVEUSE` or `HEURDIVETEST`. Please check control `HEURDIVESTRATEGY` for valid strategy numbers.

706 ***Not enough memory to add sets.***

Insufficient memory while allocating memory for the new sets. Please free up some memory, and try again.

707 ***Function can not be called during the global search***

The function being called cannot be used during the global search. Please call the function before starting the global search.

708 ***Invalid input passed to <function>***
Must specify mstart or mnel when creating matrix with columns

No column information is available when calling function <function>. If no columns were meant to be passed to the function, then please set the column number to zero. Note, that `mstart` and `mnel` should be set up for empty columns as well.

710 ***`MIPTOL` <val1> must not be less than `FEASTOL` <val2>***

The integer tolerance `MIPTOL` (`val1`) should not be set tighter than the feasibility tolerance `FEASTOL` (`val2`). Please increase `MIPTOL` or decrease `FEASTOL`.

711 ***`MIPTOL` <val1> must not be less than `FEASTOL` <val2>. Adjusting `MIPTOL`***

The integer tolerance `MIPTOL` (`val1`) must not be tighter than the feasibility tolerance `FEASTOL` (`val2`). The value of `MIPTOL` has been increased to (`val2`) for the global search.

712 ***Function not permitted when problem is presolved: <func>***

The problem is currently in a presolved state and the function <func> can only be called when the problem is in its original state. `XPRSpostsolve` can be called to return the problem to its original state.

713 ***<row/column> index out of bounds calling <function>. <index1> is '<' or '>' <bound>***

An index is out of its bounds when calling function <function>. Please check the indices.

715 ***Invalid objective sense passed to <function>. Must be `XPRS_OBJ_MINIMIZE` or `XPRS_OBJ_MAXIMIZE`.***

Invalid objective sense was passed to function <function>. Please use either `XPRS_OBJ_MINIMIZE` or `XPRS_OBJ_MAXIMIZE`.

716 ***Invalid names type passed to `XPRSgetnamelist`.***
Type code <num> is unrecognized.

An invalid name type was passed to `XPRSgetnamelist`.

717 ***Generic error.***

Used to promote license manager errors.

721 ***No IIS has been identified yet***

No irreducible infeasible set (IIS) has been found yet. Before running the function, please use `IIS -f`, `IIS -n` or `IIS -a` to identify an IIS.

722 ***IIS number <num> is not yet identified***

Irreducible infeasible set (IIS) with number <num> is not available. The number <num> stands for the ordinal number of the IIS. The value of <num> should not be larger than `NUMIIS`.

723 ***Unable to create an IIS sub-problem***

The irreducible infeasible set (IIS) procedure is unable to create the IIS approximation. Please check that there is enough free memory.

724 ***Error while optimizing the IIS sub-problem***

An error occurred while minimizing an irreducible infeasible set (IIS) sub-problem. Please check the return code set by the Optimizer.

725 ***Problems with variables for which shift infeasibilities cannot be removed are considered infeasible in the IIS***

The irreducible infeasible set (IIS) sub-problem being solved by the IIS procedure is on the boundary of being feasible or infeasible. For problems that are only very slightly infeasible, the Optimizer applies a technique called infeasibility shifting to produce a solution. Such solutions are considered feasible, although if solved as a separate problem, a warning message is given. For consistency reasons however, in the case of the IIS procedure such problems are treated as being infeasible.

726 ***This function is not valid for the IIS approximation. Please specify an IIS with count number > 0***

Irreducible infeasible set (IIS) number 0 (the ordinal number of the IIS) refers to the IIS approximation, but the functionality called is not available for the IIS approximation. Please use an IIS number between 1 and `NUMIIS`.

727 ***Bound conflict on column <col>; IIS will not continue***

There is a bound conflict on column <col>. Please check the bounds on the column, and remove any conflicts before running the irreducible infeasible set (IIS) procedure again (bound conflicts are trivial IISs by themselves).

728 ***Unknown file type specification <type>***

Unknown file type was passed to the irreducible infeasible set (IIS) sub-problem writer. Please refer to `XPRSiiswrite` for the valid file types.

729 ***Writing the IIS failed***

Failed to write the irreducible infeasible set (IIS) sub-problem or the comma separated file (.csv) containing the IIS information to disk. Please check access permissions.

730 ***Failed to retrieve data for IIS <num>***

The irreducible infeasible set (IIS) procedure failed to retrieve the internal description for IIS number <num>. This may be an internal error, please contact your local support office.

731 ***IIS stability error: reduced or modified problem appears feasible***

Some problems are on the boundary of being feasible or infeasible. For such problems, it may happen that the irreducible infeasible set (IIS) working problem becomes feasible unexpectedly. If the problem persists, please contact your local support office.

732 ***Unknown parameter or wrong parameter combination***

The wrong parameter or parameter combination was used when calling the irreducible infeasible set (IIS) console command. Please refer to the IIS command documentation for possible combinations.

733 ***Filename parameter missing***

No filename is provided for the `IIS -w` or `IIS -e` console command. Please provide a file name that should contain the irreducible infeasible set (IIS) information.

734 ***Problem data relevant to IISs is changed***

This failure is due to the problem being changed between iterative calls to IIS functions. Please start the IIS analysis from the beginning.

735 ***IIS function aborted***

The irreducible infeasible set (IIS) procedure was aborted by either hitting CTRL-C or by reaching a time limit.

736 ***Initial infeasible subproblem is not available. Run `IIS -f` to set it up***

The initial infeasible subproblem requested is not available. Please use the `IIS -f` function to generate it.

738 ***The approximation may be inaccurate. Please use `IIS` or `IIS -n` instead.***

The irreducible infeasible set (IIS) procedure was run with the option of generating the approximation of an IIS only. However, ambiguous duals or reduces costs are present in the initial infeasible subproblem. This message is always preceded by warning 737. Please continue with generating IISs to resolve the ambiguities.

739 ***Bound conflict on column <col>; Repairinfeas will not continue***

There is a bound conflict on column <col>. Please check the bounds on the column, and remove any conflicts before running the `XPRSrepairinfeas` procedure again (bound conflicts are trivial causes of infeasibility).

740 ***Unable to create relaxed problem***

The Optimizer is unable to create the relaxed problem. The relaxed problem may require significantly more memory than the base problem if many of the preferences are set to a positive value. Please check that there is enough free memory.

741 ***Relaxed problem is infeasible. Please increase freedom by introducing new nonzero preferences***

The relaxed problem remains infeasible. Zero preference values indicate constraints (or bounds) that will not be relaxed. Try introducing new nonzero preferences to allow the problem to become feasible.

742 ***Repairinfeas stability error: relaxed problem is infeasible. You may want to increase the value of delta***

The relaxed problem is reported to be infeasible by the Optimizer in the second phase of the repairinfeas procedure. Try increasing the value of the parameter delta to improve stability.

743 ***Optimization aborted, repairinfeas unfinished***

The optimization was aborted by CTRL-C or by hitting a time limit. The relaxed solution is not available.

744 ***Optimization aborted, MIP solution may be sub-optimal***

The MIP optimization was aborted by either CTRL-C or by hitting a time limit. The relaxed solution may not be optimal.

745 ***Optimization of the relaxed problem is sub-optimal***

The relaxed solution may not be optimal due to early termination.

746 ***All preferences are zero, repairinfeas will not continue***
Use options `-a -b -r -lbp -ubp -lrp` or `-grp` to add nonzero preferences

Zero preference values indicate constraints (or bounds) that will not be relaxed. In case when all preferences are zero, the problem cannot be relaxed at all. Try introducing nonzero preferences and run `XPRSrepairinfeas` again.

748 ***Negative preference given for a <sense> bound on <row/column> <name>***

A negative preference value is set for constraint or bound <name>. Preference values should be non-negative. The preferences describe the modeler's willingness to relax a given constraint or bound, with zero preferences interpreted as the corresponding constraints or bounds not being allowed to be relaxed. Please provide a zero preference if the constraint or bound is not meant to be relaxed. Also note, that very small preferences lead to very large penalty values, and thus may increase the numerical difficulty of the problem.

749 ***Relaxed problem is infeasible due to `cutoff`***

A user defined `cutoff` value makes the relaxed problem infeasible. Please check the cutoff value `CURRMIPCUTOFF`.

750 ***Empty matrix file : <name>***

The MPS file <name> is empty. Please check the name of the file and the file itself.

751 ***Invalid column marker type found : <text>***

The marker type <text> is not supported by the MPS reader. Please refer to the Appendix A.2 for supported marker types.

752 ***Invalid floating point value : <text>***

The reader is unable to interpret the string <text> as a numerical value.

753 ***<num> lines ignored***

The MPS reader has ignored <num> number of lines. This may happen for example if an unidentified section was found (in which case warning 785 is also invoked).

754 ***Insufficient memory***

Insufficient memory was available while reading in an MPS file.

755 ***Column name is missing***

A column name field was expected while reading an mps file. Please add a column name to the row. If the `MPSFORMAT` control is set to 0 (fixed format) then please check that the name field contains a column name, and is positioned correctly.

756 ***Row name is missing in section `OBJNAME`***

No row name is provided in the `OBJNAME` section. If no user defined objective name is provided, the reader uses the first neutral row (if any) as the objective row. However, to avoid ambiguity, if no user defined objective row was meant to be supplied, then please exclude the `OBJNAME` section from the MPS file.

757 ***Missing objective sense in section `OBJSENSE`***

No objective sense is provided in section `OBJSENSE`. If no user defined objective sense is provided, the reader sets the objective sense to minimization by default. However, to avoid ambiguity, if no user defined objective sense was meant to be supplied, then please exclude the `OBJSENSE` section from the MPS file.

758 ***No `SETS` and `SOS` sections are allowed in the same file***

The Optimizer expects special order sets to be defined in the `SETS` section. However, for compatibility considerations, the Optimizer can also interpret the `SOS` section. The two formats differ only in syntax, and feature the same expressive power. Both a `SETS` and a `SOS` section are not expected to be present in the same matrix file.

759 ***File not in fixed format : <file>***

The Optimizer control `MPSFORMAT` was set to 0 to indicate that the mps file <file> being read is in fixed format, but it violates the MPS field specifications.

760 ***Objective row <row> defined in section `OBJNAME` or in `MPSOBJNAME` was not found***

The user supplied objective row <row> is not found in the MPS file. If the MPS file contains an `OBJNAME` section please check the row name provided, otherwise please check the value of the control `MPSOBJNAME`.

761 ***Problem name is not provided***

The `NAME` section is present in the MPS file, but contains no problem name (not even blanks), and the `MPSFORMAT` control is set to 0 (fixed format) preventing the reader to look for the problem name in the next line. Please make sure that a problem name is present, or if it's positioned in the next line (in which case the first column in the line should be a whitespace) then please set `MPSFORMAT` to 1 (free format) or -1 (autodetect format).

762 ***Missing problem name in section `NAME`***

Unexpected end of file while looking for the problem name in section `NAME`. The file is likely to be corrupted. Please check the file.

763 ***Ignoring range value for free row : <row>***

A range value is defined for free row <row>. Range values have no effect on free rows. Please make sure that the type of the row in the `ROWS` section and the row name in the `RANGE` section are both correct.

764 ***<sec> section is not yet supported in an MPS file, skipping section***

The section <sec> is not allowed in an MPS file. Sections like "`SOLUTION`" and "`BASIS`" must appear in separate "`.slx`" and "`.bas`" files.

765 ***Ignoring repeated specification for column : <col>***

Column <col> is defined more than once in the MPS file. Any repeated definitions are ignored. Please make sure to use unique column names. If the column names are unique, then please make sure that the `COLUMNS` section is organized in a contiguous order.

766 ***Ignoring repeated coefficients for row <row> found in `RANGE` <range>***

The range value for row <row> in range vector <range> in the `RANGE` section is defined more than once. Any repeated definitions are ignored. Please make sure that the row names in the `RANGE` section are correct.

767 ***Ignoring repeated coefficients for row <row> found in `RHS` <rhs>***

The value for row <row> in right hand side vector <rhs> is defined more than once in the `RHS` section. Any repeated definitions are ignored. Please make sure that the row names in the `RHS` section are correct.

768 ***Ignoring repeated specification for row : lt;rowgt;***

Row <row> is defined more than once in the MPS file. Any repeated definitions are ignored. Please make sure to use unique row names.

770 ***Missing prerequisite section <sec1> for section <sec2>***

Section <sec2> must be defined before section <sec1> in the MPS file being read. Please check the order of the sections.

771 ***Unable to open file : <file>***

Please make sure that file <file> exists and is not locked.

772 ***Unexpected column type : <type> : <column>***

The `COLUMNS` section contains the unknown column type <type>. If the `MPSFORMAT` control is set to 0 (fixed format) then please make sure that the type of the column is correct and positioned properly.

773 ***Unexpected number of fields in section : <sec>***

Unexpected number of fields was read by the reader in section <sec>. Please check the format of the line. If the `MPSFORMAT` control is set to 0 (fixed format) then please make sure that the fields are positioned correctly. This error is often caused by names containing spaces in free format, or by name containing spaces in fixed format but positioned incorrectly.

774 ***Unexpected row type : <type>***

The `ROWS` section contains the unknown row type <type>. If the `MPSFORMAT` control is set to 0 (fixed format) then please make sure that the type of the row is correct and positioned properly.

775 ***Unexpected set type : <type>***

The `SETS` or `SOS` section contains the unknown set type <type>. If the `MPSFORMAT` control is set to 0 (fixed format) then please make sure that the type of the row is correct and positioned properly.

776 ***Ignoring unknown column name <col> found in `BOUNDS`***

Column <col> found in the `BOUNDS` section is not defined in the `COLUMNS` section. Please check the name of the column.

777 ***Ignoring quadratic coefficient for unknown column : <col>***

Column <col> found in the `QUADOBJ` section is not defined in the `COLUMNS` section. Please check the name of the column.

778 ***Ignoring unknown column name <col> found in set <set>***

Column <col> found in the definition of set <set> in the `SETS` or `SOS` section is not defined in the `COLUMNS` section. Please check the name of the column.

779 ***Wrong objective sense: <sense>***

The reader is unable to interpret the string <sense> in the `OBJSENSE` section as a valid objective sense. The objective sense should be either `MAXIMIZE` or `MINIMIZE`. The reader accepts sub-strings of these if they uniquely define the objective sense and are at least 3 characters long. Note that if no `OBJSENSE` section is present, the sense of the objective is set to minimization by default. Please provide a valid objective sense.

780 ***Ignoring unknown row name <row> found in column <column>***

Row <row> found in the column <column> in the `COLUMNS` section is not defined in the `ROWS` section. Please check the name of the row.

781 ***Ignoring unknown row name <row> found in `RANGE`***

Row <row> found in the `RANGE` section is not defined in the `ROWS` section. Please check the name of the row.

782 ***Ignoring unknown row name <row> found in `RHS`***

Row <row> found in the RHS section is not defined in the `ROWS` section. Please check the name of the row.

783 ***Expecting numerical value***

A numerical value field was expected while reading an MPS file. Please add the missing numerical entry. If the `MPSFORMAT` control is set to 0 (fixed format) then please check that the value field contains a numerical value and is positioned correctly.

784 ***Null char in text file***

A null char ('\0') encountered in the MPS file. An MPS file is designed to be a text file and a null char indicates possible errors. Null chars are treated as spaces ' ' by the reader, but please check the origin of the null char.

785 ***Unrecognized section <sec> skipped***

The section <sec> is not recognized as an MPS section. Please check the section identifier string in the MPS file. In such cases, the reader stops reading to avoid unexpected results after reading.

787 ***Empty set: <set>***

No set members are defined for set <set> in the MPS file. Please check if the set is empty by intention.

788 ***Repeated definition of section <sec> ignored***

Section <sec> is defined more than once in the mps file. Any repeated definitions are ignored. Many sections may include various versions of the described part if the problem (like different `RHS` values, `BOUNDS` or `RANGES`), but please include those in the same section.

790 ***Wrong section in the basis file: <section>***

Unrecognized section <section> found in the basis file. Please check the format of the file.

791 ***`ENDATA` is missing. File is possibly corrupted***

The `ENDATA` section is missing from the end of the file. This possible indicates that part of the file is missing. Please check the file.

792 ***Ignoring `BS` field***

BS fields are not supported by the Optimizer, and are ignored. Basis files containing `BS` fields may be created by external software. Please convert `BS` fields to either `XU` or `XL` fields.

793 ***Superbasic variable outside bounds. Value moved to closest bound***

A superbasic variable in the basis file are outside its bounds. The value of the variable has been modified to satisfy its bounds. Please check that the value in the basis file is correct. In case the variable should be set to the value given by the basis file, please modify the bounds on the variable.

794 ***Value of fixed binary column <col> changed to <val>***

The lower and upper bound for binary variable <col> was to <val>. Binaries may only be fixed at level 0 or 1.

795 ***Xpress/Mosel extensions: number of opening and closing brackets mismatch***

The LP file appears to be created by Mosel, using the Xpress MPS extensions to include variable names with whitespaces, however the file seems to be broken due to a mismatch in opening and closing brackets.

796 ***Char <c> is not supported in a name by file format. It may not be possible to read such files back correctly. Please set FORCEOUTPUT to 1 to write the file anyway, or use scrambled names.***

Certain names in the problem object may be incompatible with different file formats (like names containing spaces for LP files). If the Optimizer might be unable to read back a problem because of non-standard names, it will give an error message and won't create the file. However, you may force output using control `FORCEOUTPUT` or change the names by using scrambled names (`-s` option for `XPRSwriteprob`).

797 ***Wrong section in the solution file: <sec>***

Section <sec> is not supported in `.slx` MPS solution files.

798 ***Empty <type> file : <file>***

File <file> of type <type> is empty.

799 ***Ignoring quadratic coefficients for unknown row name <row>***

No row with name <row> was defined in the `ROWS` sections. All rows having a `QCMATRIX` section must be defined as a row with type '`L`' or '`G`' in the `ROWS` section.

835 ***Given solution column count does not match given problem***

The given solution contains a different column count compared to the loaded problem.

843 ***Delayed row (lazy constraint) <row> is not allowed to be of type 'N'. Row ignored***

Delayed rows cannot be neutral. Please define all neutral rows as ordinary ones in the `ROWS` section.

847 ***Model cut (user cut) <row> is not allowed to be of type 'N'. Row ignored***

Model cuts cannot be neutral. Please define all neutral rows as ordinary ones in the `ROWS` section.

862 ***Quadratic constraint rows must be of type 'L' or 'G'. Wrong row type for row <row>***

All quadratic rows must be of type 'L' or 'G' in the ROWS section of the MPS file (and the corresponding quadratic matrix be positive semi-definite).

863 ***The current version of the Optimizer does not yet support MIQCQP problems***

The current version of the Optimizer does not yet support mixed integer quadratically constrained problems.

864 ***Quadratic constraint rows must be of type 'L' or 'G'. Wrong row type for row <row>***

A library function was trying to define (or change to) a row with type 'L' having quadratic coefficients. All quadratic rows are required to be of type 'L' (and the corresponding quadratic matrix be positive semi-definite).

865 ***Row <row> is already quadratic***

Cannot add quadratic constraint matrices together. To change an already existing matrix, either use the XPRSchgqrowcoeff library function, or delete the old matrix first.

866 ***The divider of the quadratic objective at <pos> must be 2 or omitted***

The LP file format expects, tough may be omitted, an "/2" after the each quadratic objective term defined between square brackets. No other divider is accepted. The role of the "/2" is to notify the user of the implied division in the quadratic objective (that does not apply to quadratic constraints).

867 ***Not enough memory for tree search***

There is not enough memory for one of the nodes in the tree search.

884 ***Fatal user error detected in callback***

An error occurred during a user callback.

898 ***Cannot define range for quadratic rows. Range for row <row> ignored***

Quadratic constraints are required to be convex, and thus it is not allowed to set a range on quadratic rows. Each quadratic row should have a type of 'L' or 'G'.

899 ***The quadratic objective is not convex. Set IFCHECKCONVEXITY=0 to disable check***

The quadratic objective is not convex. Please check that the proper sense of optimization (minimization or maximization) is used.

900 ***The quadratic part of row <row> defines a non-convex region. Set IFCHECKCONVEXITY=0 to disable check.***

The quadratic in <row> is not convex. Please check that the proper sense of constraint is defined (less or equal or greater or equal constraint).

901 ***901 Duplicated QCMATRIX section for row <row> ignored.***

The MPS file may contain one Q matrix for each row. In case of duplicates, only the first is loaded into the matrix

902 ***Calling function <func> is not supported from the current context.***

This XPRS function cannot be called from this callback.

903 ***Row <row> with right hand side value larger than infinity ignored.***

The matrix file being read contains a right hand side that is larger than the predefined infinity constant XPRS_PLUSINFINITY. Row is made neutral.

904 ***Function is not allowed outside optnode callback.***

The used function of the branching manager is not allowed to call outside optnode.

905 ***Bad index passed to function.***

The index passed to function is not in range of the attribute.

906 ***Global entity cannot be branched further.***

The selected global entity is fixed and cannot be branched further.

907 ***Column is continuous and cannot be branched.***

The given column is of continuous type. The used function does not support branching on continuous columns.

909 ***Limit exceeded.***

The limit of a certain object is exceeded.

910 ***Empty branch or branching object.***

The given branch or branching object is empty.

911 ***Invalid information provided for branching object.***

The given branching object contains invalid information.

912 ***Branching object(s) cannot be changed/used at this time.***

The branching object is not fix yet. Hence, it cannot be changed/used.

913 ***Required data missing in function call for branching object.***

Data is missing in function call for branching object.

914 ***Unexpected error triggered for branching object.***

Unexpected error happened on a branching object.

915 ***Branching object (ID=<id>) rejected because it is empty or contains empty branches.***

Improper branching object.

918 ***Module error.***

Model can not be modified.

919 ***Column must be of type semi-continuous, semi-integer or partial integer to change its global bound.***

The global bound can be modified only for semi-continuous, semi-integer or partial integer column.

920 ***Semi-continuous lower bound for column <column> must be non-negative.***

Only non-negative lower bounds can be specified for semi-continuos columns.

921 ***Partial integer limit for column <column> is outside the allowed range of 0 to*** $2^{2}8 - 1$**.**

The give limit for the column is out of the allowed range.

932 ***Unknown column name <column> found in piecewise linear term.***

One of the columns given in the PWLOBJ or PWLCON section of the MPS file was not defined in the COLUMNS section.

933 ***Breakpoints for piecewise linear constraint <index> not sorted and contradicting points (<x>,<y1>) and (<x>,<y2>) given.***

The breakpoints for this piecewise linear function were not given as a sorted list, and due to contradicting points this cannot be fixed by sorting (since it is unclear what the left- and right-limit of the discontinuity would be).

934 ***Breakpoints for piecewise linear constraint <index> not sorted, will be sorted internally.***

The breakpoints for the piecewise linear function were not given as a sorted list, these will be reordered internally.

935 ***Piecewise linear breakpoints not given consecutively for variable <column>, will reorder them.***

The breakpoints for this column were not given consecutively but with breakpoints for another variable in between, will be fixed internally.

936 ***Ignoring duplicate piecewise linear breakpoint (<x>,<y>) for piecewise linear constraint <index>.***

The same breakpoint was given twice for this piecewise linear constrant, the second one will be dropped.

937 ***Piecewise linear section for column <column> contained at most one breakpoint.***

The piecewise linear function over this column only contained a single breakpoint or no breakpoints at all, which is not enough to define the function.

938 ***Discontinuity at the <beginning/end> of piecewise linear constraint <index> without a matching bound.***

The first or last two breakpoints of the piecewise linear function shared the same value for the input variable. While this is allowed to model discontinuous functions, it is only allowed for the first or last points if they match the corresponding bound and do not leave the function undefined before or after them.

939 ***Non-convex piecewise linear function with unbounded domain for column <column>, may lead to unbounded LP relaxation for reformulation even if MIP is bounded.***

The piecewise linear function over the unbounded variable was non-convex and the sum of slopes for the lower and upper limit is negative, potentially leading to an unbounded LP relaxation for the reformulation even though the MIP might be bounded. Consider giving explicit bounds for this variable if possible.

940 ***General constraint type <type> for constraint <name> not supported, should be one of MAX, MIN, AND, OR, ABS, PWL***

The general constraint <name> was assigned the type <type>, which is not supported. The type should be "MAX", "MIN", "AND", "OR", "ABS", or "PWL".

941 ***Entry <entry> of general constraint <name> neither a column nor a constant***

One of the lines in general constraints section for constraint <name> consisted of <entry>, which is neither a column name nor a valid constant (and in case of free-format MPS-format also not a GENCONS-type).

942 ***<type> constraint <index> includes non-binary variable <name>. Only binaries allowed in AND/OR general constraints.***

The <index>-th general constraint, which is either an "AND" or an "OR" constraint, includes variable <name> which is not binary.

943 ***The absolute value constraint <index> includes <number> variables. ABS-constraints should consist of exactly 2 variables.***

The <index>-th general constraint, which is an "ABS" constraint, includes more than two variables.

944 ***The general constraint <index> consists of <number> elements. General constraints need to have at least two elements.***

The <index>-th general constraint consists of less than two elements. All general constraints must include at least two elements ("abs" exactly two).

945 ***Error when loading the piecewise linear constraint names.***

The LP format file reader failed to create the piecewise linear constraint names. The previous error should explain why this failed.

946 ***Error when loading the general constraint names.***

The LP format file reader failed to create the general constraint names. The previous error should explain why this failed.

947 ***PWLCON section refers to piecewise linear constraint <name>, which was not defined in the PWLNAM section.***

In the MPS format, all piecewise linear constraints must first be defined in the PWLNAM section before adding extreme points in the PWLCON section.

948 ***Missing bracket or comma in piecewise linear function or general constraint***

Piecewise linear functions should be given as (<val1>, <val2>), general constraints as <keyword> (<col1>, <col2>, ..., <colk>). In this case one of the commas or brackets was missing.

949 ***Invalid piecewise linear constraint number passed to <function>.Piecewise linear constraint number <val1> is invalid <val2>***

There has been an error in one of the arguments of function <function>. The <val1>th given piecewise linear constraint number <val2> is invalid. Please make sure that the piecewise linear constraint numbers are not smaller than 0 and not larger than the total number of piecewise linear constraints in the current problem, given by attribute `PWLCONS`.

950 ***Invalid general constraint number passed to <function>. <val1>-th General constraint number <val2> is invalid***

There has been an error in one of the arguments of function <function>. The <val1>-th given general constraint number <val2> is invalid. Please make sure that the general constraint numbers are not smaller than 0 and not larger than the total number of general constraints in the current problem, given by attribute `GENCONS`.

951 ***Input column or resultant of <name> deleted, piecewise linear constraint is also dropped***

An <XPRSdelcols> call left the piecewise linear constraint <name> without input column or without resultant. Thus the piecewise linear constraint is dropped.

952 ***Column deletion left <name> with no resultant or no inputs, general constraint is also dropped***

An <XPRSdelcols> call left the general constraint <name> without resultant or without any inputs (either columns or constant values in case of <MIN>/<MAX>). Thus the general constraint is dropped.

953 ***Column <index> appears as both resultant and input variable in piecewise linear/general constraint <index>.***

The resultant of the piecewise linear or general constraint is equal to one of the input columns. These need to be distinct.

954 ***__pwlobj() appearing outside of objective.***

The keyword __pwlobj(<col>) may be used inside the linear objective function to declare that the objective contribution of this variable will later be defined in the pwlobj section, but it should never appear in any other section.

955 ***Adjusting extremal breakpoints of piecewise linear constraint <index> to match bounds of input.***

The first/last breakpoint of piecewise linear constraint <index> (and potentially further ones afterwards) did not match the bound of the input variable and needed to be adjusted, even with presolve disabled. Note that this change will persist after the solve.

956 ***Given breakpoint (<val1>, <val2>) for piecewise linear constraint <index> with infinite values and non-constant slope.***

The piecewise linear constraint contained a breakpoint with infinite values. All breakpoints should have finite value, for piecewise linear constraints with unbounded rays a finite point along that ray should be given as the first/last breakpoint and the solver will extend that ray in a way that ensures numerical stability of the formulation.

957 ***Non-numerical value in problem definition as <name> of <name>.***

A non-numerical value (inf/nan) was given as a matrix or objective coefficient, right-hand side or column bound. The only allowed exceptions are lower bounds or right-hand sides of 'G' constraints with value -inf and upper bounds or right-hand sides of 'L' constraints at +inf.

1001 ***Solution value redefined for column: <col>: <val1> -> <val2>***

Multiple definition of variable <col> is not allowed. Please use separate `SOLUTION` sections to define multiple solutions.

1002 ***Missing solution values in section <sec>. Only <val1> of <val2> defined***

Not all values were defined in the `SOLUTIONS` section. Variables with undefined values are set to 0.

1003 ***Please postsolve the problem first with `XPRSpostsolve` (postsolve).***

Not all values were defined in the `SOLUTIONS` section. Variables with undefined values are set to 0.

1004 ***Negative semi-continuous lower bound (<val>) for column <col> replaced with zero***

Wrong input parameter for the lower bound of a semi-continuous variable was modified to 0.

1005 ***Unrecognized column name : <col>***

No column with name <col> is present in the problem object while loading solution.

1006 ***Failed to capture solution information.***

Solution information is not available.

1020 ***Function <function> cannot be called here.***

The specified function can not be called.

1022 ***Error (<error>) while trying to run branching script.***

Branching script error.

1028 ***Unable to keep branch and bound tree memory usage below <val>Gb - currently using <val>Gb;***

The Optimizer was unable to keep the tree memory usage below the limit defined by the `TREEMEMORYLIMIT` control - the solve will continue but performance will be impaired.

1030 ***Duplicate names are not allowed - row/column/set/constraint <row1> would have same name as row/column/set/constraint <row2>.***

Each row, column, special ordered set, piecewise linear constraint and general constraint should have unique name.

1034 ***Unknown column name <col> found in indicators***

Columns <col> found in the `INDICATORS` section is not defined in the `COLUMNS` section. Please check the name of the column.

1035 ***Unknown row name <row> found in indicators***

Row <row> found in the `INDICATORS` section is not defined in the `ROWS` section. Please check the name of the row.

1036 ***Unexpected indicator type : <type>***

Indicator type <type> found in the `INDICATORS` section is invalid. The type should be '`IF`'.

1037 ***Unexpected indicator active value : <value> for row <row>***

The value <value> found in the `INDICATORS` section is invalid. Values in this section should be either 0 and 1.

1038 ***Unsupported row type for indicator constraint <row>***

Rows configured as indicator constraints should have a type of '`L`' or '`G`'.

1039 ***Non-binary variable <col> used as an indicator binary***

The variable used in the condition part of an indicator constraint should be of type binary.

1054 ***Please use the FICO-SLP solver for general nonlinear problems. Contact " LICENSING_CONTACT_NAME " for a license.***

To solve general nonlinear problems the FICO-SLP solver can be used.

1055 ***Can not resume global search - not currently solving a MIP.***

The global search can not be resumed, the current problem is not mixed integer optimization.

1059 ***Unrecognized string identifier <id> passed to function <func>***

The string <id> given as input to the function <func> does not match any expected identifiers. Double-check spelling of <id> and consult documentation of function <func>, if available.

1071 ***Unable to dualize problems with quadratic coefficients***

The current version of the Optimizer only supports dualization of linear problems. Remove quadratic terms.

1074 ***Could not write tree to global file.***

Branch-and-bound tree memory saving is disabled; re-enable this feature to allow the tree to be compressed and saved to the global file.

1075 ***Message too long: message <mes> must be shorter than <maxlen> characters***

Could not write an error/info/warning message, due to a problem, row, column or set name being too long. Shorten all names s.t. they consist of at most <maxlen> characters.

1082 ***Tree heap create failed.***

Failed to create private heap for branch-and-bound tree; probably due to insufficient memory. If possible, try to free up memory on your system, reduce the problem size or set appropriate working limits.

1090 ***No license capacity.***

The FICO Xpress license file does not specify an Optimizer capacity; license has been incorrectly generated, please contact support@fico.com .

1091 ***User cuts not allowed***

User cuts are not accessible in the tree when in-tree presolving is turned on. Deactivate in-tree presolving by setting `TREEPRESOLVE` to `0`.

1092 ***Invalid scale factor***

An invalid scale factor was given for a row or a column. Scale factors are provided as the exponents of powers of two, and must be between 0 and `MAXSCALEFACTOR`

1093 ***Column scaling not allowed.***

Scaling of binary, integer and partial integer columns is not allowed.

1094 ***Invalid SOCP constraint detected.***

The Optimizer has detected an invalid SOCP constraint: a quadratic row has incorrectly been identified as a second order cone. Please contact support.

1097 ***The dependent variable of row (e.g. variable z in $z^2 \geq x^2 + y^2$) must be defined non-negative, otherwise the constraint is non-convex.***

The problem is not formulated in a standard second order conic formulation.

1098 ***Both dependent variables of row (e.g. variables u and v in $u * v \geq x^2 + y^2$) must be defined non-negative.***

The problem is not formulated in a standard second order conic formulation.

1100 ***Cut limit reached***

Failed to create cut, since the limit of storable cuts was reached. Please restart your solve with a less aggressive cutting strategy.

1101 ***Cannot call optimization routine recursively***

You cannot call `XPRSglobal`, `XPRSminim` or `XPRSmaxim` within a call of `XPRSglobal`, `XPRSminim` or `XPRSmaxim` on the same problem pointer. Either terminate the running optimization process or use a separate problem pointer. See also error code 1101.

1102 ***Presolve detected infeasibility on non-convex quadratic row***

All quadratic matrices in the quadratic constraints must be positive semi-definite or a second-order cone.

1103 ***Insufficient name buffer***

The supplied name buffer is too small. Allocate more memory for the buffer or use shorter names.

1104 ***No row/column/set names***

Tried to access a group of names that do not exist. Provide names to columns/rows/sets before doing so.

1105 ***Cannot create remote server session***

Creating the remote compute server session failed. This may have multiple reasons, the error message will specify the cause.

1106 ***Corrupt Xpress compute server interface***

There appears to be a mismatch between the libraries of Xpress. Please reinstall Xpress.

1107 ***Connection to remote server lost***

Connection to the compute Insight server was lost.

1108 ***Unable to create remote server job data package (disk full?)***

Solves using the compute server rely on creating temporary files.

1109 ***Remote server returned an error.***

...

1110 ***The compute server returned an error.***

This error forwards an error reported by the compute server.

1111 ***Remote server returned a warning***

This warning forwards a warningreported by the compute server.

1112 ***Remote server temporary directory provided creates a path too long.***

Names used during a compute server solve is directed from the probname and a unique identifier. Please shorten the probname.

1113 ***Limiting parallel MIP tasks***

When control `RESOURCESTRATEGY` is enabled, this warning notifies that the number of maximum MIP tasks have been limited to preserve memory.

9999 ***Generic error message***

Please contact support@fico.com.

Appendix

APPENDIX A

Log and File Formats

A.1 File Types

The Optimizer generates or inputs a number of files of various types as part of the solution process. By default these all take file names governed by the problem name (*problem_name*), but distinguished by their three letter extension. The file types associated with the Optimizer are as follows:

Extension	Description	File Type
.alt	Matrix alteration file, input by `XPRSalter` (`ALTER`).	ASCII
.asc	CSV format solution file, output by `XPRSwritesol` (`WRITESOL`).	ASCII
.bss	Basis file, output by `XPRSwritebasis` (`WRITEBASIS`), input by `XPRSreadbasis` (`READBASIS`).	ASCII
.csv	Output file, output by `XPRSiiswrite`.	ASCII
.dir	Directives file (MIP only), input by `XPRSreaddirs` (`READDIRS`).	ASCII
.glb	Global file (MIP only), used by `XPRSglobal` (`GLOBAL`).	Binary
.gol	Goal programming input file, input by`XPRSgoal` (`GOAL`).	ASCII
.grp	Goal programming output file, output by`XPRSgoal` (`GOAL`).	ASCII
.hdr	Solution header file, output by `XPRSwritesol` (`WRITESOL`) and `XPRSwriterange` (`WRITERANGE`).	ASCII
.lp	LP format matrix file, input by `XPRSreadprob` (`READPROB`).	ASCII
.mps	MPS / XMPS format matrix file, input by `XPRSreadprob` (`READPROB`).	ASCII
.prt	Fixed format solution file, output by `XPRSwriteprtsol` (`WRITEPRTSOL`).	ASCII
.rng	Range file, output by `XPRSrange` (`RANGE`).	Binary
.rrt	Fixed format range file, output by `XPRSwriteprtrange` (`WRITEPRTRANGE`).	ASCII
.rsc	CSV format range file, output by `XPRSwriterange` (`WRITERANGE`).	ASCII
.sol	Solution file created by `XPRSwritebinsol` (`WRITEBINSOL`).	Binary
.slx	Solution file created by `XPRSwriteslxsol` (`WRITESLXSOL`).	ASCII
.xtm	Tuner method created by `XPRStunerwritemethod`.	ASCII
.xtr	Tuner result created by `XPRStune`.	XML
.json	Remote Solving Configuration file	JSON

In the following sections we describe the formats for a number of these.

Note that CSV stands for comma-separated-values text file format.

A.2 XMPS Matrix Files

The FICO Xpress Optimizer accepts matrix files in LP or MPS format, and an extension of this, XMPS format. In that the latter represents a slight modification of the industry-standard, we provide details of it here.

XMPS format defines the following fields:

Field	1	2	3	4	5	6
Columns	2-3	5-12	15-22	25-36	40-47	50-61

The following sections are defined:

NAME	the matrix name;
ROWS	introduces the rows;
COLUMNS	introduces the columns;
QUADOBJ / QMATRIX	introduces a quadratic objective function;
QCMATRIX	introduces the quadratic constraints;
DELAYEDROWS	introduces the delayed rows;
MODELCUTS	introduces the model cuts;
INDICATORS	introduces the indicator contraints;
SETS	introduces SOS definitions;
RHS	introduces the right hand side(s);
RANGES	introduces the row ranges;
BOUNDS	introduces the bounds;
GENCONS	introduces general constraints;
ENDATA	signals the end of the matrix.
compatibility	additional sections for extensions of the MPS format that can be read but not written.

All section definitions start in column `1`.

A.2.1 NAME section

Format:	Cols 1-4	Field 3
	`NAME`	*model_name*

A.2.2 ROWS section

Format:	Cols 1-4
	`ROWS`

followed by row definitions in the format:

Field 1	Field 2
type	*row_name*

The row types (Field 1) are:

N	unconstrained (for objective functions);
L	less than or equal to;
G	greater than or equal to;
E	equality.

A.2.3 COLUMNS section

Format:	Cols 1-7
	`COLUMNS`

followed by columns in the matrix in column order, i.e. all entries for one column must finish before those for another column start, where:

Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
blank	*col*	*row1*	*value1*	*row2*	*value2*

specifies an entry of *value1* in column *col* and row *row1* (and *value2* in *col* and row *row2*). The Field 5/Field 6 pair is optional.

A.2.4 QUADOBJ / QMATRIX section (Quadratic Programming only)

A quadratic objective function can be specified in an MPS file by including a `QUADOBJ` or `QMATRIX` section. For fixed format XMPS files, the section format is as follows:

Format:	Cols 1-7
	`QUADOBJ`

or

Format:	Cols 1-7
	`QMATRIX`

followed by a description of the quadratic terms. For each quadratic term, we have:

Field 1	Field 2	Field 3	Field 4
blank	*col1*	*col2*	*value*

where *col1* is the first variable in the quadratic term, *col2* is the second variable and *value* is the associated coefficient from the `Q` matrix. In the `QMATRIX` section all nonzero `Q` elements must be specified. In the `QUADOBJ` section only the nonzero elements in the upper (or lower) triangular part of `Q` should be specified. In the `QMATRIX` section the user must ensure that the `Q` matrix is symmetric, whereas in the `QUADOBJ` section the symmetry of `Q` is assumed and the missing part is generated automatically.

Note that the Q matrix has an implicit factors of 0.5 when included in the objective function.
This means, for instance that an objective function of the form

$$5x^2 + 7xy + 9y^2$$

is represented in a `QUADOBJ` section as:

```
QUADOBJ
x      x      10
x      y       7
y      y      18
```

(The additional term '`y x 7`' is assumed which is why the coefficient is not doubled); and in a `QMATRIX` section as:

```
QMATRIX
x      x      10
x      y       7
y      x       7
y      y      18
```

The `QUADOBJ` and `QMATRIX` sections must appear somewhere after the `COLUMNS` section and must only contain columns previously defined in the columns section. Columns with no elements in the problem matrix must be defined in the `COLUMNS` section by specifying a (possibly zero) cost coefficient.

A.2.5 QCMATRIX section (Quadratic Constraint Programming only)

Quadratic constraints may be added using `QCMATRIX` sections.

Format:	Cols 1-8	Field 3
	`QCMATRIX`	`row_name`

Each constraint having quadratic terms should have it's own `QCMATRIX` section. The `QCMATRIX` section exactly follows the description of the `QMATRIX` section, i.e. for each quadratic term, we have:

Field 1	Field 2	Field 3	Field 4
blank	*col1*	*col2*	*value*

where `col1` is the first variable in the quadratic term, `col2` is the second variable and value is the associated coefficient from the Q matrix. All nonzero Q elements must be specified. The user must ensure that the Q matrix is symmetric. For instance a constraint of the form

$$qc1 : x + 5x^2 + 7xy + 9y^2 <= 2$$

is represented as:

```
NAME example
ROWS
 L qc1
COLUMNS
   x      qc1     1
   y      qc1     0
QCMATRIX qc1
   x      x       5
   x      y       3.5
   y      x       3.5
   y      y       9
RHS
 RHS1     qc1     2
END
```

The `QCMATRIX` sections must appear somewhere after the `COLUMNS` section and must only contain columns previously defined in the columns section. Columns with no elements in the problem matrix must be defined in the COLUMNS section by specifying a (possibly zero) cost coefficient. QCMATRICES must be defined only for rows of type L or G and must have no range value defined in the RANGE section.

Note that the FICO Xpress Optimizer can only solve convex (MI)QPs. Thus the quadratic matrix should be positive semi-definite for <= rows and negative semi-definite for >= rows so that the defined region is convex. Otherwise the problem will need to be solved by the nonlinear solver.

NOTE: technically, there is one exception for the restriction on the row type being L or G. If the row is the first nonbinding row (type N) then the section is treated as a QMATRIX section instead. Please be aware, that this also means that the objective specific implied divider of 2 will be assumed (Q matrix has an implicit factors of 0.5 when included in the objective function, see the QMATRIX section). It's probably much better to use the QMATRIX or QUADOBJ sections to define quadratic objectives.

NOTE:

A.2.6 DELAYEDROWS section

This specifies a set of rows in the matrix that will be treated as delayed rows during a global search. These are rows that must be satisfied for any integer solution, but will not be loaded into the active set of constraints until required.

This section should be placed between the ROWS and COLUMNS sections. A delayed row may be of type L, G or E. Each row should appear either in the ROWS or the DELAYEDROWS section, not in both. Otherwise, the format used is the same as of the ROWS section.

Format:	Cols 1-11
	DELAYEDROWS

followed by row definitions in the format:

Field 1	Field 2
type	*row_name*

NOTE: For compatibility reasons, section names DELAYEDROWS and LAZYCONS are treated as synonyms.

A.2.7 MODELCUTS section

This specifies a set of rows in the matrix that will be treated as model cuts during a global search. During presolve the model cuts are removed from the matrix. Following optimization, the violated model cuts are added back into the matrix and the matrix is re-optimized. This continues until no violated cuts remain. This section should be placed between the ROWS and COLUMNS sections. Model cuts may be of type L, G or E. The model cuts must be "true" model cuts, in the sense that they are redundant at the optimal MIP solution. The Optimizer does not guarantee to add all violated model cuts, so they must not be required to define the optimal MIP solution.

Each row should appear either in the ROWS, DELAYEDROWS or in the MODELCUTS section, not in any two or three of them. Otherwise, the format used is the same as of the ROWS section.

Format:	Cols 1-9
	MODELCUTS

followed by row definitions in the format:

Field 1	Field 2
type	*row_name*

NOTE: A problem is not allowed to consists solely from model cuts. For compatibility reasons, section names `MODELCUTS` and `USERCUTS` are treated as synonyms.

A.2.8 INDICATORS section

This specifies that a set of rows in the matrix will be treated as indicator constraints during a global search. These are constraints that must be satisfied only when their associated controlling binary variables have specified values (either 0 or 1).

This section should be placed after any `QUADOBJ`, `QMATRIX` or `QCMATRIX` sections. The section format is as follows:

Format:	Cols 1-10
	`INDICATORS`

Subsequent records give the associations between rows and the controlling binary columns, with the following form:

Field 1	Field 2	Field 3	Field 4
type	*row_name*	*col_name*	*value*

which specifies that the row `row_name` must be satisfied only when column `col_name` has value `value`. Here type must always be `IF` and value can be either 0 or 1. Also referenced rows must be of type `L` or `G` only, and referenced columns must be binary.

A.2.9 SETS section (Integer Programming only)

Format:	Cols 1-4
	`SETS`

This record introduces the section which specifies any Special Ordered Sets. If present it must appear after the `COLUMNS` section and before the `RHS` section. It is followed by a record which specifies the type and name of each set, as defined below.

Field 1	Field 2
type	*set*

Where *type* is `S1` for a Special Ordered Set of type 1 or `S2` for a Special Ordered Set of type 2 and *set* is the name of the set.

Subsequent records give the set members for the set and are of the form:

Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
blank	*set*	*col1*	*value1*	*col2*	*value2*

which specifies a set member *col1* with reference value *value1* (and *col2* with reference value *value2*). The Field 5/Field 6 pair is optional.

A.2.10 RHS section

Format:	Col 1-3
	`RHS`

followed by the right hand side as defined below:

Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
blank	*rhs*	*row1*	*value1*	*row2*	*value2*

specifies that the right hand side column is called *rhs* and has a value of *value1* in row *row1* (and a value of *value2* in row *row2*). The Field 5/Field 6 pair is optional.

A.2.11 RANGES section

Format:	Cols 1-6
	`RANGES`

followed by the right hand side ranges defined as follows:

Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
blank	*rng*	*row1*	*value1*	*row2*	*value2*

specifies that the right hand side range column is called *rng* and has a value of *value1* in row *row1* (and a value of *value2* in row *row2*). The Field 5/Field 6 pair is optional.

For any row, if b is the value given in the `RHS` section and r the value given in the `RANGES` section, then the activity limits below are applied:

Row Type	**Sign of r**	**Upper Limit**	**Lower Limit**
G	+	b+r	b
L	+	b	b-r
E	+	b+r	b
E	-	b	b+r

A.2.12 BOUNDS section

Format:	Cols 1-6
	`BOUNDS`

followed by the bounds acting on the variables:

Field 1	Field 2	Field 3	Field 4
type	*blank*	*col*	*value*

The Linear Programming bound types are:

UP	for an upper bound;
LO	for a lower bound;
FX	for a fixed value of the variable;
FR	for a free variable;
MI	for a non-positive ('minus') variable;
PL	for a non-negative ('plus') variable (the default).

There are six additional bound types specific to Integer Programming:

UI	for an upper bounded general integer variable;
LI	for a lower bounded general integer variable;
BV	for a binary variable;
SC	for a semi-continuous variable;
SI	for a semi-continuous integer variable;
PI	for a partial integer variable.

The value specified is an upper bound on the largest value the variable can take for types `UP`, `FR`, `UI`, `SC` and `SI`; a lower bound for types `LO` and `LI`; a fixed value for type `FX`; and ignored for types `BV`, `MI` and `PL`. For type `PI` it is the switching value: below which the variable must be integer, and above which the variable is continuous. If a non-integer value is given with a `UI` or `LI` type, only the integer part of the value is used.

- **Integer variables** may only take integer values between `0` and the upper bound. Integer variables with an upper bound of unity are treated as binary variables.
- **Binary variables** may only take the values `0` and `1`. Sometimes called 0/1 variables.
- **Partial integer variables** must be integral when they lie below the stated value, above that value they are treated as continuous variables.
- **Semi-continuous variables** may take the value zero or any value between a lower bound and some finite upper bound. By default, this lower bound is `1.0`. Other positive values can be specified as an explicit lower bound. For example

  ```
  BOUNDS
  LO x 0.8
  SC x 12.3
  ```

  means that `x` can take the value zero or any value between `0.8` and `12.3`.
- **Semi-continuous integer variables** may take the value zero or any integer value between a lower bound and some finite upper bound.

A.2.13 GENCONS section

Format:	Cols 1-7
	`GENCONS`

This record introduces the section which specifies any general constraints, namely min, max, and, or, abs, pwl constraints. If present it must appear after the `COLUMNS` section. It is followed by a record which specifies the type and name of each general constraint, as defined below.

Field 1	Field 2
type	*name*

Where *type* is `MAX` for a maximum-constraint, `MIN` for a minimum-constraint, `AND` for an and-constraint, `OR` for an or-constraint, `ABS` for an absolute-value-constraint or `PWL` for a piecewise linear constraint and *name* is the name of the general constraint.

Subsequent records for min/max/and/or/abs give the elements for the constraint and are of the form:

Field 1	Field 2
blank	*col/val*

For all general constraints, the first given element (which needs to be the name of a column) will be the so-called "resultant". For the max- and min-constraints, the resultant is followed by an arbitrary number of further column names or values, and the resultant should be the maximum/minimum of the remaining columns and values. For the and- and or-constraints the resultant is followed by an arbitrary number of further column names, where all the columns (including the resultant) need to be binary, and the resultant will be one if and only if all (and) or at least one (or) of the remaining variables are one. For the abs-constraint, the resultant should be followed by exactly one futher column name, and the resultant will take the absolute value of the other column.

As an example, the constraint `z = max x, y, 5.0` could be written as

```
 GENCONS
 MAX m1
    z
x
y
5.0
```

For piecewise linear constraints the format is slightly different, consisting of exactly one line of the form:

Field 1	Field 2
col1	*col2*

and at least three lines of the form:

Field 1	Field 2
val1	*val2*

The first line defines the two variables that should be restricted by a piecewise linear relationship and the points given in the remaining lines will define the piecewise linear function, which is extended beyond the first and last point according to the slope of the previous ones. For instance the piecewise linear constraint

```
y = 0, if x < 0
y = x, if 0 <= x <= 5
y = 2x - 5, if 5 < x <= 10
y = 15, if x > 10
```

could be represented as:

```
 GENCONS
 PWL p1
    x  y
-1 0
    0  0
5  5
10 15
11 15
```

A.2.14 ENDATA section

Format:	Cols 1-6
	`ENDATA`

is the last record of the file.

A.2.15 Compatibility

The optimizer is also able to read in some further sections defined by extensions of the LP format. This includes the SOS section, which is a different way of writing down special ordered sets, and the following sections that offer different ways of formulating piecewise linear constraints and objectives:

A.2.16 PWLOBJ section

Piecewise linear objective functions may be added using `PWLOBJ` sections.

Format:	Cols 1-6
	`PWLOBJ`

The piecewise linear objective function is defined via its extreme points, i.e., the function itself is given by all convex combinations of neighboring extreme points as well as the infinite rays defined by the first two and last two points. Each row of the `PWLOBJ` section defines one extreme point for one column.

Field 1	Field 2	Field 3	Field 4
blank	*col*	*value1*	*value2*

where `col` is the variable whose objective contribution is defined through the piecewise linear function and `value2` is the objective contribution if the variable takes `value1`. For instance the piecewise linear objective function

```
0, if x < 0
x, if 0 <= x <= 5
2x - 5, if 5 < x <= 10
15, if x > 10
```

could be represented as:

```
PWLOBJ
   x         -1 0
   x         0  0
   x         5  5
   x         10 15
   x         11 15
```

If there are piecewise linear objective functions for multiple variables, these should be given consecutively (i.e., first all extreme points for x, then all for y). Furthermore, for each variable, the extreme points should be sorted according to non-decreasing `value1`. The piecewise linear functions do not necessarily need to be continuous, in this case two extreme points with identical `value1` and different `value2` can be given and the first one will be used as the lefthand-limit and the second one as the righthand-limit. Note that for `value1` itself, both `value2` can appear in the solution due to tolerances.

A.2.17 PWLNAM section

`PWLNAM` is the first of the two sections defining piecewise linear constraints.

Format:	Cols 1-6
	`PWLNAM`

Similar to the piecewise linear objective, the piecewise linear constraints will mainly be defined through its extreme points, which happens in the `PWLCON` section. In addition to that, however, the two variables involved in the restriction y = f(x), with piecewise linear function f, need to be specified, and additionally a pre- and postslope are given, defining the slope of the piecewise linear function before the first and after the last extreme point. Each piecewise linear function needs to be named, to later refer to it in the `PWLCON` section when specifying the extreme points.

Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
blank	*name*	*col1*	*col2*	*value1*	*value2*

where `col1` is the resulting variable (y above), `col2` is the input variable (x above), `value1` is the preslope defining the piecewise linear function up to the first extreme point and `value1` is the postslope defining it after the last extreme point.

A.2.18 PWLCON section

`PWLCON` is the second of the two sections defining piecewise linear constraints.

Format:	Cols 1-6
	`PWLCON`

Each piecewise linear constraint introduced in the `PWLNAM` section needs to be further specified through its extreme points, defining the behaviour between the pre- and postlope. Each line consists of the name of a piecewise linear constraint introduced in the `PWLNAM` followed by a list of extreme points. Similar to the PWLOBJ section, the functions can be discontinuous, in which case the extreme points have to be given in the correct order.

Field 1	Field 2	Field 3	Field 4
blank	*name*	*value1*	*value2*

where `name` is the name of a piecewise linear function introduced in the `PWLNAM` section and `value1` and `value2` define an extreme point, where `value1` is the value of the input variable and `value1` is the corresponding output value. For instance the piecewise linear constraint

```
y = 0, if x < 0
y = x, if 0 <= x <= 5
y = 2x - 5, if 5 < x <= 10
y = 15, if x > 10
```

could be represented as:

```
PWLNAM
 pwc1  y    x1   0 0
PWLCON
 pwc1   0 0
 pwc1   5 5
 pwc1   10 15
```

A.3 LP File Format

Matrices can be represented in text files using either the MPS file format (`.mps` or `.mat` files) or the LP file format (`.lp` files). The LP file format represents matrices more intuitively than the MPS format in that it expresses the constraints in a row-oriented, algebraic way. For this reason, matrices are often written to LP files to be examined and edited manually in a text editor. Note that because the variables are 'declared' as they appear in the constraints during file parsing the variables may not be stored in the FICO Xpress Optimizer memory in the way you would expect from your enumeration of the variable names. For example, the following file:

```
Minimize
obj: - 2 x3

Subject To
c1: x2 - x1 <= 10
c2: x1 + x2 + x3 <= 20

Bounds
x1 <= 30

End
```

after being read and rewritten to file would be:

```
\Problem name:
Minimize
 - 2 x3

Subject To
c1: x2 - x1 <= 10
c2: x3 + x2 + x1 <= 20

Bounds
x1 <= 30

End
```

Note that the last constraint in the output `.lp` file has the variables in reverse order to those in the input `.lp` file. The ordering of variables in the last constraint of the rewritten file is the order that the variables were encountered during file reading. Also note that although the optimal solution is unique for this particular problem in other problems with many equal optimal solutions the path taken by the solver may depend on the variable ordering and therefore by changing the ordering of your constraints in the .lp file may lead to different solution values for the variables.

A.3.1 Rules for the LP file format

The following rules can be used when you are writing your own `.lp` files to be read by the FICO Xpress Optimizer.

A.3.2 Comments and blank lines

Text following a backslash (\) and up to the subsequent carriage return is treated as a comment. Blank lines are ignored. Blank lines and comments may be inserted anywhere in an `.lp` file. For example, a common comment to put in LP files is the name of the problem:

```
\Problem name:  prob01
```

A.3.3 File lines, white space and identifiers

White space and carriage returns delimit variable names and keywords from other identifiers. Keywords are case insensitive. Variable names are case sensitive. Although it is not strictly necessary, for clarity of your LP files it is perhaps best to put your section keywords on their own lines starting at the first character position on the line. There is no maximum on the length names of on the length of input lines. Lines may be broken for continuation wherever you may use white space.

A.3.4 Sections

The LP file is broken up into sections separated by section keywords. The following are a list of section keywords you can use in your LP files. A section started by a keyword is terminated with another section keyword indicating the start of the subsequent section.

Section keywords	Synonyms	Section contents
`maximize` or `minimize`	`maximum max minimum min`	One linear expression describing the objective function.
`subject to`	`subject to:` `such that` `st` `s.t.` `st.` `subjectto` `suchthat` `subject` `such`	A list of constraint expressions.
`bounds`	`bound`	A list of bounds expressions for variables.
`integers`	`integer` `ints` `int`	A list of variable names of integer variables. Unless otherwise specified in the bounds section, the default relaxation interval of the variables is [0, 1].
`generals`	`general` `gens` `gen`	A list of variable names of integer variables. Unless otherwise specified in the bounds section, the default relaxation interval of the variables is [0, `XPRS_PLUSINFINITY`].
`binaries`	`binary` `bins` `bin`	A list of variable names of binary variables.
`semi-continuous`	`semi continuous` `semis` `semi` `s.c.`	A list of variable names of semi-continuous variables.
`semi integers`	`s.i.`	A list of semi-integer bound expressions for variables.
`partial integer`	`p.i.`	A list of variable names of partial integer variables.
`general constraints`	`general constraint` `gencons` `g.c.`	A list of min/max/and/or/abs and piecewise linear constraints.

Variables that do not appear in any of the variable type registration sections (i.e., `integers`, `generals`, `binaries`, `semi-continuous`, `semi integer`, `partial integer`) are defined to be continuous variables by default. That is, there is no section defining variables to be continuous variables.

With the exception of the objective function section (`maximize` or `minimize`) and the constraints section (`subject to`), which must appear as the first and second sections respectively, the sections may appear in any order in the file. The only mandatory section is the objective function section. Note that you can define the objective function to be a constant in which case the problem is a so-called constraint satisfaction problem. The following two examples of LP file contents express empty problems with constant objective functions and no variables or constraints.

Empty problem 1:

```
Minimize

End
```

Empty problem 2:

```
Minimize

0

End
```

The end of a matrix description in an LP file can be indicated with the keyword `end` entered on a line by itself. This can be useful for allowing the remainder of the file for storage of comments, unused matrix definition information or other data that may be of interest to be kept together with the LP file.

A.3.5 Variable names

Variable names can use any of the alphanumeric characters (a-z, A-Z, 0-9) and any of the following symbols:

```
!"#$%&/,.;?@_`'{}()|~'
```

A variable name can not begin with a number or a period. Care should be taken using the characters E or e since these may be interpreted as exponential notation for numbers.

A.3.6 Linear expressions

Linear expressions are used to define the objective function and constraints. Terms in a linear expression must be separated by either a + or a − indicating addition or subtraction of the following term in the expression. A term in a linear expression is either a variable name or a numerical coefficient followed by a variable name. It is not necessary to separate the coefficient and its variable with white space or a carriage return although it is advisable to do so since this can lead to confusion. For example, the string " 2e3x" in an LP file is interpreted using exponential notation as 2000 multiplied by variable x rather than 2 multiplied by variable e3x. Coefficients must precede their associated variable names. If a coefficient is omitted it is assumed to be 1.

A.3.7 Objective function

The objective function section can be written in a similar way to the following examples using either of the keywords `maximize` or `minimize`. Note that the keywords `maximize` and `minimize` are not used for anything other than to indicate the following linear expression to be the objective function. Note the following two examples of an LP file objective definition:

```
Maximize
```

```
 - 1 x1 + 2 x2 + 3x + 4y
```

or

```
Minimize
- 1 x1 + 2 x2 + 3x + 4y
```

No line continuation character is required to break the objective function across multiple lines and it can be broken wherever you may use white space.

The objective function can be named in the same way as for constraints (see later) although this name is ignored internally by the FICO Xpress Optimizer. Internally the objective function is always named `__OBJ___`.

A.3.8 Constraints

The section of the LP file defining the constraints is preceded by the keyword `subject to`. Each constraint definition must begin on a new line. A constraint may be named with an identifier followed by a colon before the constraint expression. Constraint names must follow the same rules as variable names. If no constraint name is specified for a constraint then a default name is assigned of the form `C0000001`, `C0000002`, `C0000003`, etc. Constraint names are trimmed of white space before being stored.

The constraints are defined as a linear expression in the variables followed by an indicator of the constraint's sense and a numerical right-hand side coefficient. The constraint sense is indicated intuitively using one of the tokens: >=, <=, or =. For example, here is a named constraint:

```
depot01: - x1 + 1.6 x2 - 1.7 x3 <= 40
```

Note that tokens > and < can be used, respectively, in place of the tokens >= and <=.

No line continuation character is required when breaking a constraint across multiple lines, and lines may be broken for continuation wherever you may use white space.

A.3.9 Delayed rows

Delayed rows are defined in the same way as general constraints, but after the "delayed rows" keyword. Note that delayed rows shall not include quadratic terms. The definition of constraints, delayed rows and model cuts should be sequentially after each other.

For example:

```
Minimize
obj: x1 + x2
subject to
x1 <= 10
x1 + x2 >= 1
delayed rows
x1 >= 2
end
```

For compatibility reasons, the term "lazy constraints" is used as a synonym to "delayed rows".

A.3.10 Model cuts

Model cuts are defined in the same way as general constraints, but after the "model cuts" keyword. Note that model cuts shall not include quadratic terms. The definition of constraints, delayed rows and model cuts should be sequentially after each other.

For example:

```
Minimize
obj: x1 + x2
subject to
x1 <= 10
x1 + x2 >= 1
model cuts
x1 >= 2
end
```

For compatibility reasons, the term "user cuts" is used as a synonym to "model cuts".

A.3.11 Indicator contraints

Indicator constraints are defined in the constraints section together with general constraints (that is, under the keyword "subject to"). The syntax is as follows:

```
constraint_name: col_name = value -> linear_inequality
```

which means that the constraint `linear_inequality` should be enforced only when the variable `col_name` has value `value`.

As for general constraints, the `constraint_name:` part is optional; `col_name` is the name of the controlling binary variable (it must be declared as binary in the `binaries` section); and `value` may be either 0 or 1. Finally the `linear_inequality` is defined in the same way as for general constraints.

For example:

```
Minimize
obj: x1 + x2
subject to
x1 + 2 x2 >= 2
x1 = 0 -> x2 >= 2
binary
x1
end
```

A.3.12 Bounds

The list of bounds in the bounds section are preceded by the keyword `bounds`. Each bound definition must begin on a new line. Single or double bounds can be defined for variables. Double bounds can be defined on the same line as `10 <= x <= 15` or on separate lines in the following ways:

```
10 <= x
15 >= x
```

or

```
x >= 10
x <= 15
```

If no bounds are defined for a variable the FICO Xpress Optimizer uses default lower and upper bounds. An important point to note is that the default bounds are different for different types of variables. For continuous variables and variables declared in the `generals` section, the interval defined by the default bounds is [0, `XPRS_PLUSINFINITY`], while for variables declared in the `integers` section (see later) the relaxation interval defined by the default bounds is [0, 1]. Note that the constant `XPRS_PLUSINFINITY` is defined in the FICO Xpress Optimizer header files in your FICO Xpress Optimizer libraries package.

If a single bound is defined for a variable the FICO Xpress Optimizer uses the appropriate default bound as the second bound. Note that negative upper bounds on variables must be declared together with an explicit definition of the lower bound for the variable. Also note that variables can not be declared in the bounds section. That is, a variable appearing in a bounds section that does not appear in the objective section or in the constraint section is ignored.

Bounds that fix a variable can be entered as simple equalities. For example, x6 = 7.8 is equivalent to 7.8 <= x6 <= 7.8. The bounds +∞ (positive infinity) and $-\infty$ (negative infinity) must be entered as strings (case insensitive):

```
+infinity, -infinity, +inf, -inf.
```

Note that the keywords `infinity` and `inf` may not be used as a right-hand side coefficient of a constraint.

A variable with a negative infinity lower bound and positive infinity upper bound may be entered as `free` (case insensitive). For example, `x9 free` in an LP file bounds section is equivalent to:

```
- infinity <= x9 <= + infinity
```

or

```
- infinity <= x9
```

In the last example here, which uses a single bound is used for `x9` (which is positive infinity for continuous example variable `x9`).

A.3.13 Generals, Integers and binaries

The `generals`, `integers` and `binaries` sections of an LP file is used to indicate the variables that must have integer values in a feasible solution. The difference between the variables registered in each of these sections is in the definition of the default bounds that the variables will have. For variables registered in the `generals` section the default bounds are 0 and `XPRS_PLUSINFINITY`. For variables registered in the `integers` section the default bounds are 0 and 1. The bounds for variables registered in the `binaries` section are 0 and 1.

The lines in the `generals`, `integers` and `binaries` sections are a list of white space or carriage return delimited variable names. Note that variables can not be declared in these sections. That is, a variable appearing in one of these sections that does not appear in the objective section or in a constraint in the constraint section is ignored.

It is important to note that you will only be able to use these sections if your FICO Xpress Optimizer is licensed for Mixed Integer Programming.

A.3.14 Semi-continuous and semi-integer

The `semi-continuous` and `semi integers` sections of an LP file relate to two similar classes of variables and so their details are documented here simultaneously.

The `semi-continuous` (or `semi integers`) section of an LP file are used to specify variables as semi-continuous (or semi-integer) variables, that is, as variables that may take either (a) value 0 or (b) real (or integer) values from specified thresholds and up to the variables' upper bounds.

The lines in a `semi-continuous` (or `semi integers`) section are a list of white space or carriage return delimited entries that are variable name-number pair. For the `semi-continuous` secion it is also possible to provide a variable name only. The following example shows the format of entries in the `semi-continuous` section.

```
Semi-continuous
x7 >= 2.3
x8
x9 > 4.5
```

The following example shows the format of entries in the `semi integer` section.

```
Semi integers
x7 >= 3
x9 > 5
```

Note that it is possible to use either the >= token or the > token. The resulting threshold will be identical for both cases. It is not possible to use the <= token.

The threshold of the interval within which a variable may have real (or integer) values is defined in two ways depending on whether the entry for the variable is (i) a variable name or (ii) a variable name-number pair. If the entry is just a variable name, then the variable's threshold is the variable's lower bound, defined in the `bounds` section (see earlier). If the entry for a variable is a variable name-number pair, then the variable's threshold is the number value in the pair.

It is important to note that if (a) the threshold of a variable is defined by a variable name-number pair and (b) a lower bound on the variable is defined in the `bounds` section, then:

Case 1) If the lower bound is less then zero, then the lower bound is zero.

Case 2) If the lower bound is greater than zero but less than the threshold, then the value of zero is essentially cut off the domain of the semi-continuous (or semi-integer) variable and the variable becomes a simple bounded continuous (or integer) variable.

Case 3) If the lower bound is greater than the threshold, then the variable becomes a simple lower bounded continuous (or integer) variable.

If no upper bound is defined in the `bounds` section for a semi-continuous (or semi-integer) variable, then the default upper bound that is used is the same as for continuous variables, for semi-continuous variables, and `generals` section variables, for semi-integer variables.

It is important to note that you will only be able to use this section if your FICO Xpress Optimizer is licensed for Mixed Integer Programming.

A.3.15 Partial integers

The `partial integers` section of an LP file is used to specify variables as partial integer variables, that is, as variables that can only take integer values from their lower bounds up to specified thresholds and then take continuous values from the specified thresholds up to the variables' upper bounds.

The lines in a `partial integers` section are a list of white space or carriage return delimited variable name-integer pairs. The integer value in the pair is the threshold below which the variable must have integer values and above which the variable can have real values. Note that lower bounds and upper bounds can be defined in the `bounds` section (see earlier). If only one bound is defined in the `bounds` section for a variable or no bounds are defined then the default bounds that are used are the same as for continuous variables.

The following example shows the format of the variable name-integer pairs in the `partial integers` section.

```
Partial integers
x11 >= 8
x12 >= 9
```

Note that you can not use the <= token in place of the >= token.

It is important to note that you will only be able to use this section if your FICO Xpress Optimizer is licensed for Mixed Integer Programming.

A.3.16 Special ordered sets

Special ordered sets are defined as part of the `constraints` section of the LP file. The definition of each special ordered set looks the same as a constraint except that the sense is always = and the right hand side is either `S1` or `S2` (case sensitive) depending on whether the set is to be of type 1 or 2, respectively. Special ordered sets of type 1 require that, of the non-negative variables in the set, one at most may be non-zero. Special ordered sets of type 2 require that at most two variables in the set may be non-zero, and if there are two non-zeros, they must be adjacent. Adjacency is defined by the weights, which must be unique within a set given to the variables. The weights are defined as the coefficients on the variables in the set constraint. The sorted weights define the order of the special ordered set. It is perhaps best practice to keep the special order sets definitions together in the LP file to indicate (for your benefit) the start of the special ordered sets definition with the comment line `\Special Ordered Sets` as is done when a problem is written to an LP file by the FICO Xpress Optimizer. The following example shows the definition of a type 1 and type 2 special ordered set.

```
Sos101: 1.2 x1 + 1.3 x2 + 1.4 x4 = S1
Sos201: 1.2 x5 + 1.3 x6 + 1.4 x7 = S2
```

It is important to note that you will only be able to use special ordered sets if your FICO Xpress Optimizer is licensed for Mixed Integer Programming.

A.3.17 Quadratic programming problems

Quadratic programming problems (QPs) with quadratic objective functions are defined using a special format within the objective function description. The algebraic coefficients of the function `x'Qx` appearing in the objective for QP problems are specified inside square brackets `[]`. All quadratic coefficients must appear inside square brackets. Multiple square bracket sections may be used and quadratic terms in the same variable(s) may appear more than once in quadratic expressions.

Division by two of the QP objective is either implicit, or expressed by a `/2` after the square brackets, thus `[...]` and `[...]/2` are equivalent.

Within a square bracket pair, a quadratic term in two different variables is indicated by the two variable names separated by an asterisk (*). A squared quadratic term is indicated with the variable name followed by a carat () and then a 2.

For example, the LP file objective function section:

```
Minimize
obj: x1 + x2 + [ x12 + 4 x1 * x2 + 3 x22 ] /2
```

Note that if in a solution the variables `x1` and `x2` both have value 1 then value of the objective function is `1 + 1 + (1*1 + 4*1*1 + 3*1*1) / 2 = 2 + (8) / 2 = 6`.

It is important to note that you will only be able to use quadratic objective function components if your FICO Xpress Optimizer is licensed for Quadratic Programming.

A.3.18 Quadratic Constraints

Quadratic terms in constraints are introduced using the same format and rules as for the quadratic objective, but without the implied or explicit `/2` after the square brackets.

For example:

```
Minimize
obj: x1 + x2
s.t.
x1 + [ x1^2 + 4 x1 * x2 + 3 x2^2 ] <= 10
x1 >= 1
end
```

Please be aware of the differences of the default behaviour of the square brackets in the objective compared to the constraints. For example problem:

```
min y + [ x^2 ]
st.
 x >= 1
 y >= 1
end
```

Has an optimal objective function value of 1.5, while problem:

```
min t
 s.t.
 -t + y + [ x^2 ] <= 0
 x >= 1
 y >= 1
end
```

has an optimum of 2. The user is suggested to use the explicit `/2` in the objective function like:

```
min y + [ x^2 ] / 2
st.
 x >= 1
 y >= 1
end
```

to make sure that the model represents what the modeller meant.

Note that the FICO Xpress Optimizer can only solve convex (MI)QPs. Thus quadratic rows must be of type <= or >=, and the quadratic matrix should be positive semi-definite for <= rows and negative semi-definite for >= rows so that the defined region is convex. Otherwise the problem will need to be solved by the nonlinear solver.

A.3.19 General Constraints

The general constraints section started by the record `General Constraints` specifies min, max, and, or, abs and piecewise linear constraints. Each line defines one such constraint, beginning with a name, followed by a colon, a resultant variable, a sign, a keyword and further variables (or breakpoints for the piecewise linear constraints) in brackets with spaces and commas. The keywords are `MAX` for a maximum-constraint, `MIN` for a minimum-constraint, `AND` for an and-constraint, `OR` for an or-constraint, `ABS` for an absolute-value-constraint and `PWL` for a piecewise linear constraint. For the max- and min-constraints, the resultant is followed by an arbitrary number of further column names or values, and the resultant should be the maximum/minimum of the remaining columns and values. For the and- and or-constraints the resultant is followed by an arbitrary number of further column names, where all the columns (including the resultant) need to be binary, and the resultant will be one if and only if all (and) or at least one (or) of the remaining variables are one. For the abs-constraint, the resultant should be followed by exactly one further column name, and the resultant will take the absolute value of the other column. For the piecewise linear constraints, there needs to be exactly one input variable, followed by a colon and a list of breakpoints. Note that general constraints may only introduce new variables if they are placed immediately after the subject to (or delayed rows, model cuts or pwl) sections. An example for a max-constraint would be

```
myCons: m = MAX ( x , y , 0.0 )
```

An example for a piecewise linear constraint would be

```
myPwl: y = PWL ( x ): (-1,-1) (0,0) (10,20) (10,0) (11,0)
```

defining that `y = f(x)`, where f is a piecewise linear function with value x if x is negative, `y = 2x` if `0 <= x <= 10` and `y = 0` if x is larger than 10.

A.3.20 Extended naming convention

If the names in the problem do not comply with the LP file format, the optimizer will automatically check if uniqueness and reproducibility of the names could be preserved by prepending "x(" and appending ")" to all names, i.e. the parenthesis inside the original names are always presented in pairs. In these cases, the optimizer will create an LP file with the extended naming convention format. Use control `FORCEOUTPUT` to force the optimizer to write the names in the problem out as they are.

A.3.21 Compatibility to other extensions

The FICO Xpress Optimizer is also able to read (but not write) further sections defined by extensions of the LP format. These include the SOS section, as a different way of defining special orderred sets, and the PWLObj and PWL sections for piecewise linear objective and constraints.

The piecewise linear objective section is started by the `PWLObj` line. It is followed by lines consisting of one variable name and a list of extreme points defining the piecewise linear objective function for this variable. For example the line

```
x: (-1,-1) (0,0) (10,20) (10,0) (11,0)
```

defines that if x is negative, the objective contribution is x. If x is between 0 and 10, then the objective contribution is 2x and if x is larger than 10, then the objective contribution is 0. For each variable, the extreme points should be sorted according to non-decreasing variable value. The piecewise linear functions do not necessarily need to be continuous, in this case two extreme points with identical variable values and different function values can be given and the first one will be used as the lefthand-limit and the second one as the righthand-limit. Note that for the point where the discontinuity appears, both function values can appear in the solution due to tolerances.

The piecewise linear constraint section is started by the `PWL` keyword. Each piecewise linear constraint defines a restriction y = f(x), where f is a piecewise linear function. The lines in the input format consist of a name, the input variable, a preslope, the extreme points and a postslope. The preslope defines the function before the first extreme point and the postslope defines it after the last one. Discontinuities are possible as for the objective function. Note that pwl sections may only introduce new variables if they are placed immediately after the subject to (or delayed rows, model cuts or general constraints) sections. Above example would look as follows, assuming that instead of the objective it now defines the value of a variable y :

```
pwlc1: y = x 1 (0,0) (10,20) (10,0) 0
```

A.4 ASCII Solution Files

Solution information is available from the Optimizer in a number of different file formats depending on the intended use. The `XPRSwritesol` (`WRITESOL`) command produces two files, *problem_name*`.hdr` and *problem_name*`.asc`, whose output has comma separated fields and is primarily intended for input into another program. By contrast, the command

XPRSwriteprtsol (WRITEPRTSOL) produces fixed format output intended to be sent directly to a printer, the file *problem_name*.prt. All three of these files are described below.

A.4.1 Solution Header .hdr Files

This file only contains one line of characters comprising header information which may be used for controlling the reading of the .asc file (which contains data on each row and column in the problem). The single line is divided into fourteen fields, separated by commas, as follows:

Field	Type	Width	Description
1	string	10	matrix name;
2	integer	4	number of rows in problem;
3	integer	6	number of structural columns in problem;
4	integer	4	sequence number of the objective row;
5	string	3	problem status (see notes below);
6	integer	4	direction of optimization (0=none, 1=min, 2=max);
7	integer	6	number of iterations taken;
8	integer	4	final number of infeasibilities;
9	real	12	final object function value;
10	real	12	final sum of infeasibilities;
11	string	10	objective row name;
12	string	10	right hand side row name;
13	integer	1	flag: integer solution found (1), otherwise 0;
14	integer	4	matrix version number.

- Character fields contain character strings enclosed in double quotes.
- Integer fields contain right justified decimal digits.
- Fields of type real contain a decimal character representation of a real number, right justified, with six digits to the right of the decimal point.
- The status of the problem (field 5) is a single character as follows:

C	optimization interrupted (like ctrl-c);
O	optimal;
N	infeasible;
S	stability problems;
U	unbounded;
Z	unfinished.

A.4.2 CSV Format Solution .asc Files

The bulk of the solution information is contained in this file. One line of characters is used for each row and column in the problem, starting with the rows, ordered according to input sequence number. Each line contains ten fields, separated by commas, as follows:

Field	Type	Width	Description
1	integer	6	input sequence number of variable;
2	string	10	variable (row or column vector) name;
3	string	3	variable type (C=column; N, L, G, E for rows);
4	string	4	variable status (LL, BS, UL, EQ or **);
5	real	12	value of activity;
6	real	12	slack activity (rows) or input cost (columns;)
7	real	12	lower bound (−1000000000 if none);
8	real	12	upper bound (1000000000 if none);
9	real	12	dual activity (rows) or reduced cost (columns);
10	real	12	right hand side value (rows) or blank (columns).

- The field Type is as for the `.hdr` file.
- The variable type (field 3) is defined by:
  C structural column;
  N `N` type row;
  L `L` type row;
  G `G` type row;
  E `E` type row;
- The variable status (field 4) is defined by:
  `LL` non-basic at lower bound;
  `**` basic and infeasible;
  BS basic and feasible;
  UL non-basic at upper bound;
  EQ equality row;
  SB variable is super-basic;
  ?? unknown.

A.4.3 Fixed Format Solution (.prt) Files

This file is the output of the `XPRSwriteprtsol` (`WRITEPRTSOL`) command and has the same format as is displayed to the console by `PRINTSOL`. The format of the display is described below by way of an example, for which the simple example of the FICO Xpress Getting Started manual will be used.

The first section contains summary statistics about the solution process and the optimal solution that has been found. It gives the matrix (problem) name (`simple`) and the names of the objective function and right hand sides that have been used. Then follows the number of rows and columns, the fact that it was a maximization problem, that it took two iterations (simplex pivots) to solve and that the best solution has a value of `171.428571`.

```
Problem Statistics
Matrix simple
Objective *OBJ*
RHS *RHS*
Problem has      3 rows and      2 structural columns

Solution Statistics
Maximization performed
Optimal solution found after      3 iterations
Objective function value is     171.428571
```

Next, the *Rows Section* presents the solution for the rows, or constraints, of the problem.

```
Rows Section
Number Row    At    Value      Slack Value Dual Value  RHS
N   1    *OBJ*  BS  171.428571  -171.428571  .000000      .000000
L   2    second UL  200.000000      .000000  .571429   200.000000
L   3    first  UL  400.000000      .000000  .142857   400.000000
```

The first column shows the constraint type: `L` means a ’less than or equal to’ constrain; `E` indicates an ’equality’ constraint; `G` refers to a ’greater than or equal to’ constraint; `N` means a ’nonbinding’ constraint – this is the objective function.

The sequence numbers are in the next column, followed by the name of the constraint. The `At` column displays the status of the constraint. A `UL` indicator shows that the row is at its upper limit. In this case a $\leq$ row is hard up against the right hand side that is constraining it. `BS` means that the constraint is not active and could be removed from the problem without changing the optimal value. If there were $\geq$ constraints then we might see `LL` indicators, meaning that the constraint was at its lower limit. Other possible values include:

**	basic and infeasible;
EQ	equality row;
??	unknown.

The `RHS` column is the right hand side of the original constraint and the `Slack Value` is the amount by which the constraint is away from its right hand side. If we are tight up against a constraint (the status is `UL` or `LL`) then the slack will be `0`.

The `Dual Value` is a measure of how tightly a constraint is acting. If a row is hard up against a $\leq$ constraint then it might be expected that a greater profit would result if the constraint were relaxed a little. The dual value gives a precise numerical measure to this intuitive feeling. In general terms, if the right hand side of a $\leq$ row is increased by 1 then the profit will increase by the dual value of the row. More specifically, if the right hand side increases by a sufficiently small δ then the profit will increase by δ `x` dual value, since the dual value is a marginal concept. Dual values are sometimes known as *shadow prices*.

Finally, the *Columns Section* gives the solution for the columns, or variables.

```
Columns Section
Number Column At    Value    Input Cost   Reduced Cost
C   4     a     BS  114.285714    1.000000  .000000
C   5     b     BS   28.571429    2.000000  .000000
```

The first column contains a `C` meaning column (compare with the rows section above). The number is a sequence number. The name of the decision variable is given under the `Column` heading. Under `At` is the status of the column: `BS` means it is away from its lower or upper bound, `LL` means that it is at its lower bound and `UL` means that the column is limited by its upper bound. Other possible values include:

**	basic and infeasible;
EQ	equality row;
SB	variable is super-basic;
??	unknown.

The `Value` column gives the optimal value of the variable. For instance, the best value for the variable `a` is `114.285714` and for variable `b` it is `28.571429`. The `Input Cost` column tells you the coefficient of the variable in the objective function.

The final column in the solution print gives the `Reduced Cost` of the variable, which is always zero for variables that are away from their bounds – in this case, away from zero. For variables which are zero, it may be assumed that the per unit contribution is not high enough to make production viable. The reduced cost shows how much the per unit profitability of a variable would have to increase before it would become worthwhile to produce this product. Alternatively, and this is where the name *reduced cost* comes from, the cost of production would have to fall by this amount before any production could include this without reducing the best profit.

A.4.4 ASCII Solution (.slx) Files

These files provide an easy to read format for storing solutions. An `.slx` file has a header `NAME` containing the name of the matrix the solution belongs to. Each line contains three fields as follows:

Field	Type	Width	Description
1	char	1	variable type;
2	string	variable	name of variable;
3	real	variable	value of activity.

The variable type (field 1) is defined by:
`C` structural column;
`S` LP solution only: slack variables;
`D` LP solution only: dual variables;
`R` LP solution only: reduced costs.

The file is closed by `ENDATA`.

It is possible to store multiple solutions in the same `.slx` file by repeating the `NAME` field following by the additional solution information.

Example

```
NAME solution 1
 C x1 0
 C x2 1
NAME solution 2
 C x1 1
 C x2 0
ENDATA
```

A.5 ASCII Range Files

Users can display range (sensitivity analysis) information produced by `XPRSrange` (`RANGE`) either directly, or by printing it to a file for use. Two functions exist for this purpose, namely `XPRSwriteprtrange` (`WRITEPRTRANGE`) and `XPRSwriterange` (`WRITERANGE`). The first of these, `XPRSwriterange` (`WRITERANGE`) produces two files, *problem_name*`.hdr` and *problem_name*`.rsc`, both of which have fixed fields and are intended for use as input to another program. By way of contrast, command `XPRSwriteprtrange` (`WRITEPRTRANGE`) outputs information in a format intended for sending directly to a printer (*problem_name*`.rrt`). The information provided by both functions is essentially the same and the difference lies purely in the intended purpose for the output. The formats of these files are described below.

A.5.1 Solution Header (.hdr) Files

This file contains only one line of characters comprising header information which may be used for controlling the reading of the `.rsc` file. Its format is identical to that produced by `XPRSwritesol` (`WRITESOL`) and is described in *Solution Header (.hdr) Files* above.

A.5.2 CSV Format Range (.rsc) Files

The bulk of the range information is contained in this file. One line of characters is used for each row and column in the problem, starting with the rows, ordered according to input sequence number. Each line contains 16 fields, separated by commas, as follows:

Field	Type	Width	Description
1	integer	6	input sequence number of variable;
2	string	*	variable (row or column vector) name;
3	string	3	variable type (`C`=column; `N`, `L`, `G`, `E` for rows);
4	string	4	variable status (`LL`, `BS`, `UL`, `EQ` or `**`);
5	real	12	value of activity;
6	real	12	slack activity (rows) or input cost (columns);
7	real	12	lower activity;
8	real	12	unit cost down;
9	real	12	lower profit;
10	string	*	limiting process;
11	string	4	status of limiting process at limit (`LL`, `UL`);
12	real	12	upper activity;
13	real	12	unit cost up;
14	real	12	upper profit;
15	string	*	limiting process;
16	string	4	status of limiting process at limit (`LL`, `UL`).

* these fields are variable length depending on the maximum name length

- The field Type is as for the `.hdr` file.
- The variable type (field 3) is defined by:
  `C` structural column;
  `N` `N` type row;
  `L` `L` type row;
  `G` `G` type row;
  `E` `E` type row;
- The variable status (field 4) is defined by:
  `LL` non-basic at lower bound;
  `**` basic and infeasible;
  `BS` basic and feasible;
  `UL` non-basic at upper bound;
  `EQ` equality row;
  `??` unknown.
- The status of limiting process at limit (fields 11 and 16) is defined by:
  LL non-basic at lower bound;
  UL non-basic at upper bound;
- A full description of all fields can be found below.

A.5.3 Fixed Format Range (.rrt) Files

This file is the output of the `XPRSwriteprtrange` (`WRITEPRTRANGE`) command and has the same format as is displayed to the console by PRINTRANGE. This format is described below by way of an example.

Output is displayed in three sections, variously showing summary data, row data and column data. The first of these is the same information as displayed by the XPRSwriteprtsol (WRITEPRTSOL) command (see above), resembling the following:

```
Problem Statistics
Matrix PLAN
Objective C0______
RHS R0______
Problem has 7 rows and 5 structural columns

Solution Statistics
Minimization performed
Optimal solution found after 6 iterations
Objective function value is 15.000000
```

The next section presents data for the rows, or constraints, of the problem. For each constraint, data are displayed in two lines. In this example the data for just one row is shown:

```
Rows Section
Vector Activity Lower actvty Unit cost DN Upper cost Limiting AT
Number Slack    Upper actvty Unit cost UP            Process
G C1 10.000000    9.000000  -1.000000                x4       LL
LL 2 .000000     12.000000   1.000000                C6       UL
```

In the first of the two lines, the row type (N, G, L or E) appears before the row name. The value of the activity follows. Then comes Lower actvty, the level to which the activity may be decreased at a cost per unit of decrease given by the Unit cost DN column. At this level the unit cost changes. The Limiting Process is the name of the row or column that would change its status if the activity of this row were decreased beyond its lower activity. The AT column displays the status of the limiting process when the limit is reached. It is either LL, meaning that it leaves or enters the basis at its lower limit, or UL, meaning that it leaves or enters the basis at its upper limit. In calculating Lower actvty, the lower bound on the row as specified in the RHS section of the matrix is ignored.

The second line starts with the current status of the row and the sequence number. The value of the slack on the row is then shown. The next four pieces of data are exactly analogous to the data above them. Again, in calculating Upper actvty, the upper bound on that activity is ignored.

The columns, or variables, are similarly displayed in two lines. Here we show just two columns:

```
Columns Section
Vector Activity   Lower actvty Unit costDN Upper cost Limiting AT
Number Input cost Upper actvty Unit costUP Lower cost Process
C x4   1.000000   -2.000000    5.000000    6.000000   C5        LL
BS 8   1.000000    3.000000    1.000000     .000000   C1        LL

C x5   2.000000   -1.000000    2.000000    6.000000   X3        LL
UL 9   4.000000    3.000000   -2.000000  -very large  X2        LL
```

The vector type is always C, denoting a column. The Activity is the optimal value. The Lower/Upper actvty is the activity level that would result from a cost coefficient increase/decrease from the Input cost to the Upper/Lower cost (assuming a minimization problem). The lower/upper bound on the column is ignored in this calculation. The Unit cost DN/UP is the change in the objective function per unit of change in the activity down/up to the Lower/Upper activity. The interpretation of the Limiting Processes and AT statuses is as for rows. The second line contains the column's status and sequence number.

Note that for non-basic columns, the Unit costs are always the (absolute) values of the reduced costs.

A.6 The Directives (.dir) File

This consists of an unordered sequence of records which specify branching priorities, forced branching directions and pseudo costs, read into the Optimizer using the `XPRSreaddirs` (`READDIRS`) command. By default its name is of the form *problem_name*`.dir`.

Directive file records have the format:

Col 2-3	Col 5-12	Col 25-36
type	*entity*	*value*

type is one of:

PR	implying a priority entry (the value gives the priority, which must be an integer between 0 and 1000. Values greater than 1000 are rejected, and real values are rounded down to the next integer. A low value means that the entity is more likely to be selected for branching.)
UP	the entity is to be forced up (value is not used)
DN	the entity is to be forced down (value is not used)
PU	an up pseudo cost entry (the value gives the cost)
PD	a down pseudo cost entry (the value gives the cost)
MC	a model cut entry (value is not used)
DR	a delayed row entry (value is not used)
BR	force the optimizer to branch on the entity even if it is satisfied. If a node solution is global feasible, the optimizer will first branch on any branchable entity flagged with BR before returning the solution.

entity is the name of a global entity (vector or special ordered set), or a mask. A mask may comprise ordinary characters which match the given character: a `?` which matches any single character, or a `*`, which matches any string or characters. A `*` can only appear at the end of a mask.

value is the value to accompany the type.

For example:

```
PR x1* 2
```

gives global entities (integer variables etc.) whose names start with `x1` a priority of `2`. Note that the use of a mask: a `*` matches all possible strings after the initial `x1`.

A.7 IIS description file in CSV format

This file contains information on a single IIS of an infeasible LP.

Field	Description
Name	Name of a row or column in conflict.
Type	Type of conflicting variable (row or column vector).
Sense	Sense of conflicting variable: (LE or GE) to indicate or rows. (LO or UP) to indicate lower or upper bounds for columns.
Bound	Value associated with the variable, i.e. RHS for rows and bound values for columns.
Dual value	The dual multipliers corresponding to the contradiction deducible from the IIS. Summing up all the rows and columns in the IIS multiplied by these values yields a contradicting constraint. This value is negative for <= rows and upper bounds, and positive for >= rows and lower bounds.
In iso	Indicates if the row or column is in isolation.

Note that each IIS may contain a row or column with only on one of its possible senses. This also means that equality rows and columns with both lower and upper bounds, only one side of the restriction may be present. Range constraints in an IIS are converted to greater than or equal constraints.

An IIS often contains other columns than those listed in the IIS. Such columns are free, and have no associated conflicting bounds.

The information contained in these files is the same as returned by the XPRSgetiisdata function, or displayed by (IIS -p).

A.8 The Matrix Alteration (.alt) File

The Alter File is an ASCII file containing matrix revision statements, read in by use of the XPRSalter (ALTER) command, and should be named *problem_name*.alt by default. Each statement occupies a separate line of the file and the final line is always empty. The statements consist of *identifiers* specifying the object to be altered and *actions* to be applied to the specified object. Typically the identifier may specify just a row, for example R2, specifying the second row if that name has been assigned to row 2. If a coefficient is to be altered, the associated variable must also be specified. For example:

```
RRRRRRRR
CCRider
2.087
```

changes the coefficient of CCRider in row RRRRRRRR to 2.087. The *action* may be one of the following possibilities.

A.8.1 Changing Upper or Lower Bounds

An upper or lower bound of a column may be altered by specifying the special 'rows' **LO and **UP for lower and upper bounds respectively.To change the objective coefficient of a column use the string **OBJ. For example, to change the lower bound (to 1.234), upper bound (to 5.678) and the objective (to 1234.0) of column x____0305 would look like:-

```
**LO
x____0305
1.234
**UP
x____0305
5.678
**OBJ
```

```
x___0305
1234.0
```

A.8.2 Changing Right Hand Side Coefficients

Right hand side coefficients of a row may be altered by changing values in the 'column' with the name of the right hand side.

A.8.3 Changing Constraint Types

The direction of a constraint may be altered. The row name is given first, followed by an action of `**NTx`, where `x` is one of:

N	for the new row type to be constrained;
L	for the new row type to be 'less than or equal to';
G	for the new row type to be 'greater than or equal to';
E	for the new row type to be an equality.
R	for the new row type to be a range row.

Note that `N` type rows will not be present in the matrix in memory if the control `KEEPNROWS` has been set to zero before `XPRSreadprob` (`READPROB`).

When turning a row to ranged row, a third entry, the range, is expected to be defined following `**NTR`. The rules for changing a row to a ranged row follow that of `XPRSchgrhsrange`.

A.9 The Tuner Method (.xtm) File

The tuner method file is in a straightforward plain text format. For example, when the two controls `MAXTIME` and `THREADS` are set by the user on the current problem and then `XPRStunerwritemethod` is called, the generated xtm file will look similar to the following:

```
FIXED-CONTROLS
  MAXTIME               = 100
  THREADS               = 1
TUNABLE-CONTROLS
  SBEFFORT              = 0.25, 4
  HEURSEARCHEFFORT      = 0.5, 2
  CUTFACTOR             = 0.5, 1, 5
  SCALING               = 0
  PRESOLVE              = 0
  VARSELECTION          = 2, 7
  CUTFREQ               = 2
  SYMMETRY              = 0, 1, 2
  COVERCUTS             = 0, 2
  GOMCUTS               = 0, 2, 10
  TREECOVERCUTS         = 0
  TREEGOMCUTS           = 0
  HEURSTRATEGY          = 0
  SBESTIMATE            = 1, 2, 3, 4, 5, 6
  HEURSEARCHROOTCUTFREQ= 2, 5
  HEURSEARCHROOTSELECT = 0, 3, 5
  HEURSEARCHTREESELECT = 0, 3, 5
  ROOTPRESOLVE          = 1
  PREPROBING            = 3
  BRANCHDISJ            = 0
```

The tuner method file consists of a section of fixed controls and a section of tunable controls.

A.9.1 The fixed controls

The fixed controls section starts with FIXED-CONTROLS, followed by control setting lines in assignment form. Each control in this section can only be assigned to one value. If the same control appears several times in this section, its first appearance will be used.

When writting out the tuner method file, all the controls set for the current problem will be included in the fixed control section. When reading in the tuner method file using XPRStunerreadmethod, these controls won't be applied to the current problem immediately, they will only be applied to the worker problem used in the tuner.

This section can be empty.

A.9.2 The tunable controls

The tunable controls section starts with TUNABLE-CONTROLS, followed by control setting lines in assignment form. Each control in this section can be assigned to one value, or multiple values seperated by commas. A control may appear multiple times in this section.

When reading in a tuner method file and writing it out again, the tunable controls may appear in a different order. If there is a control appearing multiple times in the original tuner method file, when written out, it will be combined into a single line with multiple values.

For bit vector controls, such as PRESOLVEOPS, SCALING, or HEURSEARCHROOTSELECT, it is possible to either specify concrete assignments to the control or to specify individual bits that should be used for tuning. In the latter case, one has to use a colon instead of an equals sign. For example, "SCALING = 24, 675" will tune using the two given concrete values 24 and 675, while "SCALING : 3, 4, 9" will individually toggle bits 3, 4, and 9 of the default value for SCALING and potentially also try combinations of those later during tuning.

This section can be empty. When both the fixed and the tunable secitons are empty, the tuner will then use a pre-defined factory tuner method.

A.10 The Simplex Log

During the simplex optimization, a summary log is displayed every n iterations, where n is the value of LPLOG. This summary log has the form:

Its	The number of iterations or steps taken so far.
Obj Value	The objective function value.
S	The current solution method (p primal; d dual).
Ninf	The number of infeasibilities.
Nneg	The number of variables which may improve the current solution if assigned a value away from their current bounds.
Sum inf	The scaled sum of infeasibilities. For the dual algorithm this is the scaled sum of dual infeasibilities when the number of negative dj's is non-zero.
Time	The number of seconds spent iterating.

A more detailed log can be displayed every n iterations by setting LPLOG to -n. The detailed log has the form:

Its	The number of iterations or steps taken so far.
S	The current solution method (p primal; d dual).
Ninf	The number of infeasibilities.
Obj Value	If the solution is infeasible, the scaled sum of infeasibilities, otherwise: the objective value.
In	The sequence number of the variable entering the basis (negative if from upper bound).
Out	The sequence number of the variable leaving the basis (negative if to upper bound).
Nneg	The number of variables which may prove the current solution if assigned a value away from their current bounds.
Dj	The scaled rate at which the most promising variable would improve the solution if assigned a value away from its current bound (reduced cost).
Neta	A measure of the size of the inverse.
Nelem	Another measure of the size of the inverse.
Time	The number of seconds spent iterating.

If `LPLOG` is set to 0, no log is displayed until the optimization finishes.

A.11 The Barrier Log

The first line of the barrier log displays statistics about the Cholesky decomposition needed by the barrier algorithm. This line contains the following values:

Dense cols	The number of dense columns identified in the factorization.
NZ(L)	The number of nonzero elements in the Cholesky factorization.
Flops	The number of floating point operations needed to perform one factorization.

During the barrier optimization, a summary log is displayed in every iteration. This summary log has the form:

Its	The number of iterations taken so far.
P.inf	Maximal violation of primal constraints.
D.inf	Maximal violation of dual constraints.
U.inf	Maximal violation of upper bounds.
Primal obj	Value of the primal objective function.
Dual obj	Value of the dual objective function.
Compl	Value of the average complementarity.

After the barrier algorithm a crossover procedure may be applied. This process prints at most 3 log lines about the different phases of the crossover procedure. The structure of these lines follows The Simplex Log described in the section above.

If `BAROUTPUT` is set to 0, no log is displayed until the barrier algorithm finishes.

A.12 The Global Log

During the branch and bound tree search (see `XPRSglobal` (`GLOBAL`)), a summary log of nine columns of information is frequently printed. By default, the printing frequency increases over time. If `MIPLOG` is explicitly set to a negative value `-n`, a log line will be printed every n nodes. The nine columns consist of:

Node	A sequential node number.
BestSoln	The value of the best integer solution found so far.
BestBound	A bound on the value of the optimal integer solution.
Sols	The number of integer solutions that have been found.
Active	The number of active (unsolved) nodes in the branch and bound tree search.
Depth	The depth of the current node in the branch and bound tree.
Gap	The percentage gap between the best solution and the best bound.
GInf	The number of global infeasibilities at the current node.
Time	The time taken.

This log is also printed when an integer feasible solution is found. An asterisk (`*`) printed in front of the node number indicates that a solution has been found by an integral LP relaxation. Single characters indicate that a heuristic solution has been found. Lower case characters stand for different strategies of the Optimizer's diving heuristic: the letter `a` corresponds to strategy 1, the letter `b` to strategy 2, and so forth. Compare control `HEURDIVESTRATEGY`. By default, several strategies are applied. Upper case letters stand for special search heuristics. More precisely, `R`, `L`, `M`, `C`, `U`, and `Z` stand for the different modes of local search that can be selected by controls `HEURSEARCHROOTSELECT` and `HEURSEARCHTREESELECT`. For technical reasons, a `U` might also appear after a restart. The letter `F` represents the feasibility pump, `T` stands for a trivial heuristic. `S`, `G`, and `B` are reserved for special purpose heuristics for problems with set packing/partitioning constraints, GUBs, and branching on constraints, respectively. An `E` indicates that a solution has been found during the calculation of branching estimates.

During root node cutting, the column Node is replaced by two columns Its and Type, columns Active and Depth are replaced by Add and Del, respectively. These have the following meaning:

Its	A counter for the number of cutting plane separation loops.
Type	The type of cuts that have been generated this round: `G` – Gomory cuts, `M` – model cuts, `O` – outer approximation cuts (only for nonlinear problems), `N` – network-based cuts, `K` – any other type of cuts
Add	Number of cuts added to the LP relaxation in this iteration
Del	Number of cuts deleted from the LP relaxation in this iteration

If `MIPLOG` is set to `3`, a detailed log of eight columns of search information is printed for each node:

Branch	A sequential node number.
Parent	The node number of the parent of the current node. A `D` or a `U` marks whether the current node is the down child or the up child, respectively, of its parent.
Solution	The optimal value of the LP relaxation at the current node.
Entity	The global entity on which the Optimizer will branch after this node.
Value	The current value of the entity chosen for branching.
Active	The number of active nodes in the tree search.
GInf	The number of global infeasibilities.
Time	The time taken.

Not all the information described above is present for all nodes. If the LP relaxation is cut off, only Branch and Parent (and possibly Solution) are displayed. If the LP relaxation is infeasible, only Branch and Parent appear. The rest of the line will consist of a text message `relaxation exceeds cutoff` or `relaxation infeasible`. If an integer solution is discovered, this is highlighted before the log line is printed.

If `MIPLOG` is set to `2`, the detailed log is printed at integer feasible solutions only. When `MIPLOG` is set to `1`, the tree node logs are surpressed, but cutting loop logs will still be displayed. If `MIPLOG` is set to `0`, neither cut nor node log wil be pritned. In any case, LP logs and intermediate status messages might still be printed.

A.13 The Tuner Log

While the tuner evaluates various control settings, it prints a summary log for each finished run. When tuning a MIP problem, the summary log consists of eight columns of information:

RunID	A sequential tuner run number.
Stat	Status of a finished run: `S` - Solved, `T` - Timeout, `U` - Unsolved and `C` - Cancelled.
Solution	The best integer solution.
Bound	The best bound.
Integral	The primal dual integral.
Gap	The relative MIP gap.
RunTime	The time spent for solving with this control setting.
TotTime	The total time spent for the tuner.

When tuning an LP problem, the summary log consists of five columns of information:

RunID	A sequential tuner run number.
Stat	Status of a finished run: `S` - Solved, `T` - Timeout, `U` - Unsolved and `C` - Cancelled.
Solution	The LP objective.
RunTime	The time spent for solving with this control setting.
TotTime	The total time spent for the tuner.

When the tuner finds an improving control setting, it will highlight the run with an asterisk (*) at the beginning of the log line. The tuner will also specify the control parameters and the log file name for the improving run.

If a control setting has been evaluated in previous tuner runs, its result can be reused. In this case, the tuner will print an extra `H` in the `Stat` column.

A.14 The Remote Solving Configuration file

This configuration file allows the user to control some of the ways the Xpress solver interacts with the remote Insight Compute Interface. It contains advanced configuration settings; it is expected that most users will not need to use these configuration options.

To use the configuration file, set the environment variable `XPRESS_COMPUTE_CONFIG` to the full path of the file including the file name itself; you must also set the `XPRESS_COMPUTE` and `XPRESS_COMPUTE_URL` environment variables to activate remote solving in the usual way. Changes to the configuration are only read when the Optimizer is first initialized with XPRSinit.

The configuration file must be a valid JSON document, containing a single object with key-value pairs. All keys are optional and Xpress will use sensible defaults for anything you do not specify. For example:

```
{
  "logLevel": 101,
  "caCertsPath": "C:/xpressmp/ssl/ca-bundle.crt"
}
```

The remainder of this section details the individual keys that can be set.

A.14.1 caCertsPath

This field can be set to the absolute path of the certificate bundle file to use for authenticating SSL certificates when communicating with a remote server using HTTPS. If unspecified, Xpress will look for a file `ca-bundle.crt` in the path specified by the `MOSEL_SSL` environment variable (if set), or the `.mmssl` folder of the user's home directory (if not). If this file does not exist, it can be created in the default location by executing the command `mmssl setup`

For example:

```
{
  "caCertsPath": "C:/xpressmp/ssl/ca-bundle.crt"
}
```

A.14.2 cleanupJobs

This field controls whether Xpress should delete a compute job from the remote server when it successfully completes or is interrupted. The default is `false`. Set to `true` to automatically delete successfully completed jobs from the Insight Compute Interface.

For example:

```
{
  "cleanupJobs": true
}
```

A.14.3 executionService

This field contains the name of the Insight execution service that will be used for jobs. If unset, the Insight server's default execution service is used. Where the execution service name is set in the configuration file and the `COMPUTEEXECSERVICE` control, the value from the control will be used.

For example:

```
{
  "executionService": "SECONDARY"
}
```

A.14.4 logLevel

This field controls additional lines written to the problem's log that describe communication between the local Xpress application and the remote Insight Compute Interface. (Note it doesn't affect lines written by the remote Insight Compute Interface itself.) Supported levels are:

- `0` - write no extra log lines
- `1` - write error and warning messages only
- `2` - write error, warning and notification messages (the default)
- `101` - as `2`, but also write lines for every HTTP request made and event message received from the server

- `102` - as `101`, but include extra debugging output. Should be set on the advice of FICO product support only.

For example:

```
{
  "logLevel": 0
}
```

A.14.5 maxRetries

This field controls how many times a failed request to the remote server will be retried before we show an error to the user. There will be an exponentially increasing delay before each retry (200ms, 400ms, 800ms, etc) - the default setting of `8` means we try each request for about 51 seconds before the user is informed of an error. Set to a lower value if you want to see errors quicker, or `0` to disable retries entirely.

For example:

```
{
  "maxRetries": 2
}
```

A.14.6 trustSrv

This field controls whether Xpress should trust the remote server without checking its certificate, when an HTTPS URL is used. If set to `false`, the authenticity of the remote server is checked using the list of trusted certification authorities and the operation will be aborted if the verification fails. Set to `true` if you want to use a server that has a known invalid or self-signed certificate, and acknowledge the security risks this brings. Default is `false`.

For example:

```
{
  "trustSrv": true
}
```

Appendix B

Contacting FICO

FICO provides clients with support and services for all our products. Refer to the following sections for more information.

Product support

FICO offers technical support and services ranging from self-help tools to direct assistance with a FICO technical support engineer. Support is available to all clients who have purchased a FICO product and have an active support or maintenance contract. You can find support contact information and a link to the Customer Self Service Portal (online support) on the Product Support home page (www.fico.com/en/product-support).

The FICO Customer Self Service Portal is a secure web portal that is available 24 hours a day, 7 days a week from the Product Support home page. The portal allows you to open, review, update, and close cases, as well as find solutions to common problems in the FICO Knowledge Base.

Please include *'Xpress'* in the subject line of your support queries.

Product education

FICO Product Education is the principal provider of product training for our clients and partners. Product Education offers instructor-led classroom courses, web-based training, seminars, and training tools for both new user enablement and ongoing performance support. For additional information, visit the Product Education homepage at www.fico.com/en/product-training or email producteducation@fico.com.

Product documentation

FICO continually looks for new ways to improve and enhance the value of the products and services we provide. If you have comments or suggestions regarding how we can improve this documentation, let us know by sending your suggestions to techpubs@fico.com.

Please include your contact information (name, company, email address, and optionally, your phone number) so we may reach you if we have questions.

Sales and maintenance

If you need information on other Xpress Optimization products, or you need to discuss maintenance contracts or other sales-related items, contact FICO by:

- Phone: +1 (408) 535-1500 or +44 207 940 8718
- Web: www.fico.com/optimization and use the available contact forms

Related services

Strategy Consulting: Included in your contract with FICO may be a specified amount of consulting time to assist you in using FICO Optimization Modeler to meet your business needs. Additional consulting time can be arranged by contract.

Conferences and Seminars: FICO offers conferences and seminars on our products and services. For announcements concerning these events, go to www.fico.com or contact your FICO account representative.

FICO Community

The FICO Community is a great resource to find the experts and information you need to collaborate, support your business, and solve common business challenges. You can get informal technical support, build relationships with local and remote professionals, and improve your business practices. For additional information, visit the FICO Community (community.fico.com/welcome).

About FICO

FICO (NYSE:FICO) powers decisions that help people and businesses around the world prosper. Founded in 1956 and based in Silicon Valley, the company is a pioneer in the use of predictive analytics and data science to improve operational decisions. FICO holds more than 165 US and foreign patents on technologies that increase profitability, customer satisfaction, and growth for businesses in financial services, telecommunications, health care, retail, and many other industries. Using FICO solutions, businesses in more than 100 countries do everything from protecting 2.6 billion payment cards from fraud, to helping people get credit, to ensuring that millions of airplanes and rental cars are in the right place at the right time. Learn more at www.fico.com.

Index

Numbers