dash optimization

Xpress-Optimizer Reference manual

Release 18

Last update 23 April 2007

Published by Dash Optimization Ltd

How to Contact Dash

USA, Canada and all Americas

Dash Optimization Inc

Information and Sales: info@dashoptimization.com
Licensing: license-usa@dashoptimization.com
Product Support: support-usa@dashoptimization.com

Tel: +1 (201) 567 9445
Fax: +1 (201) 567 9443

Dash Optimization Inc.
560 Sylvan Avenue
Englewood Cliffs
NJ 07632
USA

Japan

Dash Optimization Japan

Information and Sales: info@jp.dashoptimization.com
Licensing: license@jp.dashoptimization.com
Product Support: support@jp.dashoptimization.com

Tel: +81 43 297 8836
Fax: +81 43 297 8827

WBG Marive-East 21F FASuC B2124
2-6 Nakase Mihama-ku
261-7121 Chiba
Japan

Worldwide

Dash Optimization Ltd

Information and Sales: info@dashoptimization.com
Licensing: license@dashoptimization.com
Product Support: support@dashoptimization.com

Tel: +44 1926 315862
Fax: +44 1926 315854

Leam House, 64 Trinity Street
Leamington Spa
Warwickshire CV32 5YN
UK

For the latest news and Xpress-MP software and documentation updates, please visit the Xpress-MP website at http://www.dashoptimization.com or subscribe to our mailing list.

Contents

Chapter 1
Introduction

1.1 Overview

Part of the Xpress-MP software suite, the Xpress-Optimizer is an extremely versatile and powerful tool which is well-suited to a broad range of optimization problems. Accessible both directly via a Console or graphical interface, or by means of library functions, the Optimizer combines ease of use with speed and flexibility for the best results. A broad range of access routines and optimization algorithms, employing the best techniques available allows the user to tackle problems previously too large to solve within a reasonable time span. Either as a standalone program or handling tasks within users' own programs, the Xpress-Optimizer opens a world of possibilities.

For users of Console Xpress, the 'Console Mode' encompasses a number of commands for using the Optimizer's core functionality. Models may be imported from either MPS or LP format files, optimized using any of the algorithms supported by the Optimizer and the solution viewed in a number of ways. Further control may be exerted over the optimization process by the setting of a number of controls which influence various aspects of the algorithms.

Users of the Xpress-MP Optimizer library enjoy all the functionality of the Console Mode, with additional access to an 'Advanced Mode'. This extension of the interface provides users with access to the internal data structures of the Optimizer and greater matrix management. The inclusion of a Cut Manager allowing the addition of cutting planes provides users with the possibility of 'tailor-made' strategies for efficiently solving mixed integer problems in individual models.

The main benefit of the library, however, is the ability to embed all Optimizer calls within a user's own program, automating much of the above and providing the possibility of harnessing the power of the Xpress-Optimizer within one's own applications. To this end the set of library functions are available for a number of common languages including C, Fortran, Java and Visual Basic. The convention employed throughout will be to give all descriptions in terms of the C structure, although differences and short examples for other languages may be found in the Xpress-MP Getting Started manual.

1.2 The Xpress-Optimizer

The Xpress-Optimizer is a Linear and Integer Programming optimizer which has been carefully programmed to take full advantage of matrix structure. The Integer Programming component has the ability to handle a variety of global entities, i.e. objects which must satisfy some nonlinear properties in an acceptable solution. We call such a solution an integer solution, although this nomenclature is not always strictly accurate.

The Optimizer can read in a problem in one of three ways:

- in MPS format from a matrix file;

- in a binary form (the Binary Interface Format created by the Xpress-Modeler);
- in LP format from a matrix file.

It will then perform linear optimization by the primal or dual simplex method, or the Newton barrier method if licensed. The Optimizer may be instructed to go on to search for integer solutions. Ranging (sensitivity analysis) may also be performed. For interfacing with spreadsheets or database programs, ASCII representations of the solution may be produced.

The Optimizer may also save intermediate solutions in the form of basis files. If an optimal linear solution has been found to one problem and the status of the variables has been stored in a basis file, then the simplex LP solution path for a similar model may be shortened by starting the process from the same point, reading in the basis file just obtained.

1.3 Integer Programming Considerations

Though many systems can be modeled accurately as Linear Programs, there are situations where discreteness is central to the decision-making problem. It appears that there are three major areas where such nonlinear facilities are required:

- when entities must inherently be selected from a discrete set;
- when modeling logical conditions;
- when finding the global optimum over functions.

Problems such as these can be modeled using any of Xpress-Mosel, Xpress-BCL or the Xpress-Modeler. If enabled by your license, the Optimizer can then be used to find the global optimum. Usually the underlying structure is that of a Linear Program, but optimization may be used successfully when the nonlinearities are separable into functions of just a few variables.

The Xpress-MP suite supports the following global entities:

Binary variables (BV) – decision variables that must take either the value `0` or the value `1`, sometimes called 0/1 variables;

Integer variables (UI) – decision variables that must take on integer values. Some upper limit must be specified;

Partial integer variables (PI) – decision variables that must take integer values below a specified limit but can take any value above that limit;

Semi-continuous variables (SC) – decision variables that must take on either the value `0`, or any value in a range whose lower an upper limits are specified. SCs help model situations where, if a variable is to be used at all, it has to be at some minimum level;

Semi-continuous integer variables (SI) – decision variables that must take either the value `0`, or any integer value in a range whose lower and upper limits are specified;

Special ordered sets of type one (SOS1) – an ordered set of non-negative variables of which at most one can take a nonzero value;

Special ordered sets of type two (SOS2) – an ordered set of non-negative variables of which at most two can be nonzero, and if two are nonzero, they must be consecutive in their ordering.

The Optimizer employs a *Branch and Bound* search to locate and ensure optimality of a solution. It has a set of default strategies which have been found to work well on most problems. However, the user should note that sophistication in modeling is important in large scale MIP

work. It has always been our experience that careful experimentation with realistic but small scale examples pays off when moving to large production versions of the models.

The special Integer Programming features of the Optimizer (except presolving) are used after solving the underlying Linear Programming problem to optimality. It is only following this that the nonlinearities and discreteness that have been specified in the input data are recognized. Examples of using the global search facilities of the Optimizer for the various interfaces may be found in Xpress-MP Getting Started manual and it is assumed that you have read the relevant sections of that guide before using the reference manuals.

1.4 Running the Xpress-Optimizer

1.4.1 Initialization

To be able to run, the Optimizer requires a valid licence file, `xpauth.ini`. This may be configured for a specific machine, ethernet address or an authorized dongle. If the license file or dongle is missing, a message will be displayed describing the problem and the Optimizer will exit. Under Windows the Optimizer looks for the license file in the directory of the executables; under Unix the directory pointed to by the `XPRESS` environment variable is searched. Note that the license file should always be kept in the same directory as the Xpress-MP DLL `xprl.dll`, which is installed by the default into the c:\XpressMP\bin folder.

Following this, the Optimizer will attempt to determine the problem name. If none is given, users of Console Xpress will be prompted for one and if this request is ignored a default problem name of `$$$$$$$$` will be assumed. For problem names specified by the user, only the (optional) drive and path and the (required) filename are noted, any extension being ignored.

The problem name, *problem_name*, is used as a base for all files generated by the Optimizer, with separate files distinguished by different extensions. Matrix input files are *problem_name*`.mat`, *problem_name*`.mps`, *problem_name*`.lp`, *problem_name*`.mat.gz`, *problem_name*`.mps.gz` and *problem_name*`.lp.gz` where files with the `.gz` extension are compressed in gzip format.

1.4.2 Console Xpress Options

For Console Xpress users, the syntax for calling the Optimizer is as follows:

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

The problem name may optionally be specified on the command line when the Optimizer is invoked, but if omitted the problem name will be prompted for as described above. The optional second argument allows you to specify a text file from which the optimizer console input will be read (as if they had been typed interactively).

The Optimizer console is an interactive command line interface to the library. The features of the Optimizer console include improved online help, auto- completion of command names, and command prompt integration. Type "help commands" to get a list of available commands, and then try, e.g., "help minim" for help on a specific command. Some other useful commands are "help attributes", "help controls", or just "help".

To use the auto-completion feature, type the first part of an optimizer command name and press the tab key, for example "min<tab>" or "mat<tab>". Once you have finished typing the command name portion of your input line, the optimizer console will auto-complete file names. For example, if you have a matrix named hpw15, try "readprob h<tab>".

The console also provides convenient access to the operating system's shell commands. For example, try typing "dir" (or "ls" under Unix). You can also use the "cd" command to change the working directory, which will be indicated in the prompt:

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

When started, the Optimizer console will attempt to read in an initialization file called `optimizer.ini` located in the current working directory. This is an ASCII file which may contain any statements or commands which you wish to use by default in any run of the Optimizer. If the initialization file does not exist or cannot be found, the Optimizer will be initialized with its own built-in defaults.

The Optimizer console for 2006A onwards is implemented as a TCL shell. Users should be aware that it is possible therefore to type incorrect input which is interpreted as valid TCL syntax rather than as an error. Users of releases prior to 2006A should consult the release notes for a list of incompatibilities between old and new consoles.

1.4.3 Interrupting an Optimization Run in Console Xpress

Console Xpress users of the Optimizer may interrupt the processing of any command in a graceful way by typing CTRL-C, following which the Optimizer will return to its > prompt. If optimization has been interrupted in this way, any solution process will stop at the first 'safe' place before returning to the prompt. Iterations may be resumed at a later point by re-typing the interrupted command.

Note that if you have typed ahead, the CTRL-C mechanism may fail under some operating systems.

1.4.4 Termination

The Optimizer can terminate in one of three possible ways:

(a) normally, with no errors;

(b) with an error status indicating the solution status;

(c) immediately upon detection of a fatal error.

A normal termination means that the Optimizer has been terminated successfully, possibly with the `QUIT` or `STOP` command. The problem has not necessarily been solved to optimality since this will depend on the commands issued and whether any non-fatal errors occurred.

If Console Xpress users terminate the Optimizer with the `STOP` command, then a nonzero return code indicates the solution status of the problem. A complete list of these return codes is provided in 9.

A fatal error stops the Optimizer immediately. Such errors usually arise from files being unavailable or disks being full, but may occur if available memory is exhausted.

1.5 Structure of this Manual

The main body of the manual is essentially organized into two parts. It begins in 2 with a brief overview of common Optimizer usage, introducing the various routines available and setting them in the context in which they are very often used. This is followed in 3 with a brief overview of the sorts of problems which may arise during the optimization process and the kind of solution that might be expected from a problem. 4 provides a description of some of the more-frequently used controls along with some ideas of how they may be used to enhance the solution process. Finally, 5 details some more advanced topics in Optimizer usage.

Following the first five chapters, the remainder forms the main reference section of the manual. 6 details all functions in both the Console and Advanced Modes alphabetically. 7 and 8 then provide a reference for the various controls and attributes, followed by a list of Optimizer error and return codes in 9. A description of several of the file formats is provided in A.

1.6 Conventions Used

Throughout the manual standard typographic conventions have been used, representing computer code fragments with a `fixed width font`, whilst equations and equation variables appear in *`italic type`*. Where several possibilities exist for the library functions, those with C syntax have been used, and C style conventions have been used for structures such as arrays etc. Console Mode routines which have both a Console and library equivalent are usually described in both forms, with the Console command bracketed. In sections which are irrelevant for Console users, only the library form is given. For differences in syntax between the Console and Library versions of commands and controls, see the introduction to 6. Where appropriate, the following have also been employed:

- square brackets [...] contain optional material;
- curly brackets {...} contain optional material, of which one must be chosen;
- entities in *italics* which appear in expressions stand for meta-variables. The description following the meta-variable describes how it is to be used;
- the symbol CTRL followed by a letter means 'hold down the CTRL key and type the letter'. The CTRL or Control key is usually at the bottom left hand corner of the keyboard;

Chapter 2
Problem-Solving with Xpress-MP

2.1 Overview

A wide range of routines are available for interacting with the Xpress-Optimizer, providing a handle on its core functionality through Console Mode commands, or more extensive access to its internal data structures through the Advanced Mode routines. Whilst the users of the Xpress-Optimizer are diverse and the range of problems studied by them is wide, most optimization tasks adhere to a fairly standard template, providing a "natural grouping" of the routines. A discussion of this forms the subject matter for this chapter.

The general layout of an optimization process is given in Figure 2.1. Generally, a user goes through five main processes in solving a problem. The Optimizer must first be initialized, preparing it to handle a problem. The problem matrix is then read into memory before one of the optimization routines is called to solve it. The solution to the problem may be viewed and finally the user may exit the Optimizer. This structure may be more complex if the Optimizer's capability for handling multiple problems is used, but it serves to structure its commands.

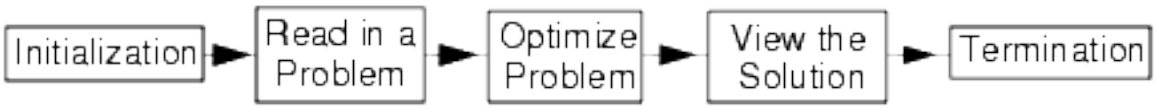


Figure 2.1: Optimizing a Problem

2.2 Initialization and Termination

2.2.1 The Optimizer

The first stage in running the Optimizer is the initialization process. In Console Xpress, this process is run automatically, but for library users, the system must be initialized manually by calling `XPRSinit` before any of the other library routines. This command checks for any necessary libraries, and runs security checks to determine licence information about your Xpress-MP installation. Only once this has completed can any of the other routines be run.

When all calls to the Optimizer routines have completed at the end of your session, the termination stage is entered. At this point, any memory used by the Optimizer is released for other processes and any files still open for reading or writing are closed. Again, this process is handled automatically for Console Xpress users when the `QUIT` or `STOP` command is given. Library users must call the routine `XPRSfree` to perform the equivalent tasks.

2.2.2 The Problem Environment

For library users, enjoying the benefits of a multiple problem environment, once the Optimizer has been initialized and before a problem matrix can be read in, a problem environment must

be created using the XPRScreateprob routine. The user may create any number of these (license permitting) and may optimize, store and work on several such at any given time, referring to them by passing the problem pointer as the first argument to each subsequent function call. When an environment is no longer needed, it can be removed using XPRSdestroyprob. It is good practice to remove all such environments before calling XPRSfree.

2.2.3 Optimizer Output

For Console Xpress users, output is sent both directly to the screen and to a log file. Library users have greater flexibility with this. Using the command XPRSsetlogfile, a log file can be specified to catch output associated to a particular problem. Alternatively, an output callback function may be employed, defined by the XPRSsetcbmessage command, to catch any output and process it directly within the application. See 5.3 for details.

2.3 Reading in a Problem

Once initialized, the Optimizer will accept a problem for optimization. Problems may be read into the Optimizer data structures in essentially two ways. The simplest of these is to read a matrix from an MPS or LP file using XPRSreadprob (READPROB), although library users can also load problems from their data structures using XPRSloadlp, XPRSloadqp, XPRSloadglobal, or XPRSloadqglobal. MPS names may then be added to the model using XPRSaddnames. Once the problem has been loaded, any subsequent call to one of these input routines will overwrite it.

It is important to note that for MIP problems presolve will take a copy of the matrix and modify it. The original matrix may be retrieved by calling XPRSpostsolve (POSTSOLVE). If the matrix is in the original state then XPRSpostsolve (POSTSOLVE)will return without doing anything.

2.4 Solving the Problem

Having loaded the problem, the user may (attempt to) find a solution to it.

The XPRSmaxim (MAXIM) and XPRSminim (MINIM) commands allow for maximization and minimization of LP, QP and mixed integer (MIP) problems, whilst the command XPRSglobal (GLOBAL) may be used to find the solution of MIP problems once the initial LP relaxation has been solved. Additionally, ranging may be performed using XPRSrange (RANGE).

A number of other possibilities are also available to users. The simplex LP optimization process may be accelerated by starting with an optimal basis from a previous, similar problem. Such bases may be saved and restored using the XPRSwritebasis (WRITEBASIS) and XPRSreadbasis (READBASIS) functions, or input from the user's own data structures using XPRSloadbasis. The user may additionally specify directives to influence the global search, which may be loaded from file using XPRSreaddirs (READDIRS). The format for the directives file is described in A. Other more advanced possibilities include the use of callback functions and customization of the cut manager, both of which are discussed in 5.

2.5 Viewing the Solution

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 may want to know why a problem has occurred and take steps to correct it. This is the subject matter for 3, following. In the former case, the user will probably want to consult the identified solution.

Xpress-MP provides a number of functions for accessing details of a solved problem. Using either of XPRSwritesol (WRITESOL) or XPRSwriteprtsol (WRITEPRTSOL) a full solution may

be obtained as an ASCII file. The first of these is suitable for interpretation by another application, whilst the second is primarily intended to be sent to a printer. Equivalent commands for obtaining range information are XPRSwriterange (WRITERANGE) and XPRSwriteprtrange (WRITEPRTRANGE).

Library users may also access the solution directly through their programs using XPRSgetlpsol, which will return the values of the decision variables, the slack variables, dual values and reduced costs for the current LP solution. The decision variables and slack variables for a MIP solution may be obtained with the XPRSgetmipsol function. In addition, the optimization algorithms provide extra details of a problem as it is solved by means of *problem attributes*. These are a set of objects which may be interrogated for particular information and transferred to user variables by type using the library functions XPRSgetintattrib, XPRSgetdblattrib and XPRSgetstrattrib. Examples of attributes include LPOBJVAL and MIPOBJVAL, which return the optimal values of the objective function for LP and (M)IP problems respectively. A full list of attributes may be found in 8.

2.6 Optimization by Example

The above sections provide a brief introduction to the most common features of the Xpress-Optimizer and its most general usage. Over the course of the following chapters, topics such as common problems and Optimizer performance will be considered, as well as some of the more advanced features of the Optimizer. The second part of the manual forms a reference for the various functions and their usage.

Examples of using the Optimizer are available from a number of sources, most notably from Xpress-MP Getting Started manual. This provides a straight forward, "hands on" approach to the Xpress-MP software suite and it is highly recommended that users read the relevant chapters before considering the reference manuals. Additional, more advanced, examples may be found on the Xpress-MP CD-ROM.

2.7 Quick Reference

2.7.1 Initialization and Termination

XPRSinit	Initialize the Optimizer.
XPRScreateprob	Create a problem environment.
XPRSsetlogfile	Direct all Optimizer output to a log file.
XPRSsetcbmessage	Define an output callback function.
XPRSdestroyprob	Destroy a problem environment.
XPRSfree	Release memory used by the Optimizer and close any open files.

2.7.2 Reading In a Problem

XPRSreadprob	Read in an MPS, LP, QP or binary interface format file.
XPRSloadlp	Load an LP problem into the Optimizer.
XPRSloadqp	Load a QP problem into the Optimizer.
XPRSloadglobal	Load a MIP problem into the Optimizer.
XPRSloadqglobal	Load a quadratic MIP problem into the Optimizer.
XPRSaddnames	Provide names for a range of rows or columns.

2.7.3 Solving the Problem

XPRSreadbasis	Read a basis from file.
XPRSloadbasis	Load a basis from a user's data structures.
XPRSreaddirs	Read a directives file.
XPRSmaxim	Maximize the problem's objective function.
XPRSminim	Minimize the problem's objective function.
XPRSglobal	Search for an integer solution.
XPRSrange	Calculate ranging information and save it to file.
XPRSgetbasis	Get the current basis into user arrays.
XPRSwritebasis	Write a basis to file.

2.7.4 Viewing the Solution

XPRSwritesol	Write the current solution to ASCII files.
XPRSwriteprtsol	Write the current solution in printable format to file.
XPRSwriterange	Write range information to ASCII files.
XPRSwriteprtrange	Write range information in printable format to file.
XPRSgetlpsol	Retrieve current LP solution values into users arrays.
XPRSgetmipsol	Retrieve last MIP solution values into users arrays.
XPRSgetintattrib	Recover the value of an integer problem attribute.
XPRSgetdblattrib	Recover the value of a double problem attribute.
XPRSgetstrattrib	Recover the value of a string problem attribute.

Chapter 3

Optimality, Infeasibility and Unboundedness

3.1 The Solution Process

Once a problem matrix has been constructed within the Optimizer, typically a user will call one of the optimization routines to attempt to solve it and, having done so, would be presented with an optimal solution which can then be used. Optimization is not always this simple, however, and many things can and do go wrong during the solution process. Very often in real situations problems are large and will take some time to solve. This is particularly true with complicated MIP problems. For problems that seem to be taking an unusually long time to solve, the optimization time can often be improved with some thought and careful setting of the control parameters. This is the subject of the following chapter.

Console users can safely terminate the solution process at any point using the CTRL-C keystroke. This allows the current point in the process to be saved before the Optimizer terminates. The process can subsequently be continued if desired by reissuing the command. Whilst library users do not have this facility, limits can be placed on the amount of time taken for an Optimizer run either by setting the control LPITERLIMIT, which provides an upper bound on the number of simplex iterations carried out, or BARITERLIMIT, which performs the same role for the Newton barrier method. Alternatively the control MAXTIME can be set to impose a "real time" limit on the process. If the solution process ever ends prematurely, it is worth checking that these three controls have been set to reasonable values and that it is not one of these that is interrupting the optimization.

Assuming none of these problems arise and the optimization run completes successfully, there are a number of possible outcomes for the resulting solution. Ideally, an optimal solution will have been found, confirming that the problem was well-posed and providing an answer that is usable. However, this is not the only outcome from the process. It is equally possible that no solution can be found, where the problem is said to be *infeasible*. Occasionally, potentially infinite solutions are also possible from certain models, resulting in *unboundedness*. In this chapter we briefly discuss some of these possibilities.

3.2 Infeasibility

A problem is said to be *infeasible* if no solution exists which satisfies all the constraints. The Xpress-Optimizer provides a number of means for diagnosing infeasibilities in models, depending on the point in the solution process at which they are detected.

3.2.1 Diagnosing Infeasibility During Presolve

The presolve facility, if used (see 5.2), makes a number of checks for infeasibility. If an infeasibility is detected, it is possible to trace this back and uncover the cause of it. This diagnosis is

carried out whenever the control parameter TRACE is set to 1 before the optimization routine XPRSmaxim (MAXIM) or XPRSminim (MINIM) is called. In such a situation, the cause of the infeasibility is then reported as part of the output from the optimization routine and the problem may be amended if necessary.

3.2.2 Irreducible Infeasible Sets

Another general technique to analyze infeasibility is to find a small portion 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 find an IIS for each of the infeasibilities in a model with the use of XPRSiis (NUMIIS). This search is controlled by the parameter MAXIIS, which controls the maximum number of IISs which will be found. In some cases an infeasibility can be represented by several different IISs and Xpress-MP will attempt to find the IIS with the smallest number of constraints in order to make the infeasibility easier to diagnose. The IISs may be retrieved by library users with XPRSgetiis.

Once an IIS has been found it is useful to know if dropping a single constraint or bound will completely remove the infeasibility represented by the IIS. An attempt is made to identify a subset of the IIS called a *sub-IIS isolation*. Removing any member of the sub-IIS isolation will remove all infeasibilities in the IIS without increasing the infeasibilities of other IISs. The sub-IIS isolations thus indicate the likely cause of each independent infeasibility and give an indication of which constraint or bound to drop. It is not always possible to find sub-IIS isolations, but if these subsets exist the members will be marked with a star. The sub-IIS isolations are searched for only if all independent IISs have been found.

3.2.3 Integer Infeasibility

In certain situations a MIP problem may turn out to be infeasible, while the equivalent LP problem (the LP relaxation) yields an optimal solution. In such circumstances the feasible region for the LP relaxation, while nontrivial, contains no solutions which satisfy the various integrality constraints and a solution may only be recovered either by dropping one of these, or one of the other problem constraints. These are perhaps the worst kind of infeasibilities as it is much harder to determine the cause.

3.3 Unboundedness

A problem is said to be *unbounded* if the objective function may be improved indefinitely whilst still satisfying the constraints of the model. When this occurs it usually signifies a problem in the formulation of the model being solved rather than a likely source of infinite profit. If the Optimizer detects unboundedness, the user should look again at the model being presented for missing, insufficient or inaccurate constraints, or data.

3.4 Scaling

When they are first formulated, problems sometimes contain numerical values which vary over many orders of magnitude. 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 RHS 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 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 Xpress-MP 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 RHS and non-infinite bound values do not exceed 10^8. 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 RHS/bound values that are close together.

In the above example, both the x-coefficient 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*.

Xpress-MP 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 employed by the Optimizer depends on the bits of the control parameter SCALING which are set. 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 problem is scaled during XPRSreadprob (READPROB), and may be rescaled later using XPRSscale (SCALE).

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).

Typically one may want to rescale the matrix following an alteration (using XPRSalter (ALTER)). By setting SCALING before calling XPRSscale (SCALE), a different scaling strategy can be employed from that used originally.

The default value of SCALING is 35, so row and column scaling are done by the maximum element method. If scaling is not required, SCALING must be set to 0 before any call to XPRSreadprob (READPROB).

The scaling is determined by the matrix elements only. The objective coefficients, right hand side values and bound values do not influence the scaling. Scaling of integer entities is not supported in XPRSglobal (GLOBAL), although a nonzero SCALING will scale the LP variables. This means that the scaling of integer entities should be considered carefully when formulating MIP problems.

3.5 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.

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}. And you are lucky if you can compute the solution values and reduced costs 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 RHS and cost coefficients. Under reasonable assumptions, an upper bound for the computed variable value accuracy is $4x\|K\|x \parallel \text{RHS} \parallel /10^{16}$, where $\parallel$ RHS $\parallel$ denotes the L-infinity norm of the RHS 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 4
Performance Issues

4.1 Choice of Algorithm

The Xpress-Optimizer is essentially "three solvers in one", offering users a choice of methods to be used for solving LP and QP problems: the *primal* and *dual simplex* algorithms and the *Newton barrier* interior point algorithm. The best algorithm to use for optimization in a given situation is problem-specific. As a general rule, the dual simplex is usually much faster than the primal simplex if the model is not infeasible or near-infeasibility. If the problem is likely to be infeasible, then the primal simplex is probably the best choice as it makes determining the cause of the infeasibility simpler. Interior point methods such as the Newton barrier algorithm perform better on certain classes of problems, although, for a problem matrix A, if A^TA is dense then the barrier will be slow.

As a default, the Xpress-Optimizer employs the dual simplex method for solving LP problems and the barrier method for solving QP problems. For most users this will be sufficient and they need not consider changing it. If a problem seems to be taking an unusually long time to solve, however, it may be worth experimenting with the different algorithms. These may be called by specifying flags to the optimization routines, `XPRSmaxim` (`MAXIM`) and `XPRSminim` (`MINIM`). The default algorithm used is determined by the value of the control parameter, `DEFAULTALG`.

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

4.2 Simplex Performance

4.2.1 The Simplex Method

The region defined by a set of linear constraints is a polyhedron, known as the *feasible region*. Points with the same linear objective function value, known as a *level set*, form a hyperplane. It follows that an optimal value of the objective function will occur only on the boundary of the feasible region and one will always occur at one of the vertices of the polyhedron. In some cases, when the level sets of the objective function are parallel to part of the polyhedron's boundary, the optimal solution will not be unique, consisting rather of a line, plane or hyperplane of solutions. However, in this case also, the optimal solution value may be found by considering only the vertices of the feasible region, although there will obviously be a continuum of decision variable values which will produce this optimum.

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, usually slightly faster than 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 either `XPRSmaxim` (`MAXIM`) or `XPRSminim` (`MINIM`).

4.2.2 Inversion

During optimization using the simplex method, every so often the Optimizer will go through a process known as inversion, with frequency determined by the controls `INVERTFREQ` and `INVERTMIN`. This will attempt to find a more compact representation of the current solution and check its accuracy. However, it is possible that the Optimizer may be unable to find a new representation of the current solution. This may be due to accuracy problems or an unstable initial condition produced by the crash or `XPRSreadbasis` (`READBASIS`). In such a situation, a number of the vectors will be rejected from the basis and replaced by unit vectors corresponding to slack or artificial variables. Following inversion, the Optimizer subsequently tries to adjust the current solution to find a more stable one before continuing with the algorithm.

4.2.3 Partial Pricing vs. Devex Pricing

The primal simplex Optimizer uses a mixture of partial pricing and Paul Harris" Devex pricing as determined by the control `PRICINGALG`. Typically, partial pricing results in a greater number of fast iterations whereas Devex pricing results in fewer, slow iterations. Which is better is highly problem-dependent. When set to `0`, the Optimizer will start by using partial pricing and automatically determine when to switch to Devex pricing. To prevent the Optimizer from switching to Devex pricing, the user can set `PRICINGALG` to `-1`. To force the Optimizer to switch to Devex pricing the `PRICINGALG` control can be set to `1`.

4.2.4 Output

Whilst searching for an optimal LP solution, the Console Optimizer writes an iteration log to the screen. The same information may be obtained by library users with the `XPRSsetlogfile` or `XPRSsetcblplog` functions. This log is produced every `LPLOG` iterations. If this is positive, a summary output is produced, whereas a more detailed log is produced if it is negative. When set to `0`, a log is displayed only when the solution terminates.

4.3 Barrier Performance

4.3.1 The Newton Barrier Method

In contrast to the simplex method, the Newton barrier is an interior point method for solving LP and QP problems. As the name suggests, such a method involves iteratively moving from one point to the next within the interior of the feasible region. Approaching the boundary of the region is penalized, so iterates of this method cannot leave the region. However, since optimal solutions of LP problems lie on the boundary of the feasible region, this penalty must be dynamically decreased as the algorithm proceeds in order to allow iterates to converge to the optimal solution.

Interior point methods typically yield a solution lying strictly in the interior of the feasible region and so can be only an approximation to the true optimal vertex solution. It is therefore the required proximity to the optimal solution which determines the number of iterations required, rather than the number of decision variables. 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 either `XPRSmaxim` (`MAXIM`) or `XPRSminim` (`MINIM`). This is used by default for QP problems, whose quadratic objective functions result in optimal solutions that generically lie on a face of the polyhedral feasible region, rather than at a vertex.

4.3.2 Controlling Barrier Performance

The Newton barrier method is influenced by a number of controls which can be altered if the solution search seems slow, or if numerical problems result. Ensuring that the CACHESIZE is set correctly can have a significant effect on the performance, and *must* be set manually on non-Intel and non-AMD platforms. Similarly, altering the ordering algorithm for the Cholesky factorization using BARORDER can affect performance. Setting this to 2 often produces better results, although the ordering itself can be significantly slower. Other controls such as DENSECOLLIMIT can be set manually to good effect. For example, numerical problems where dense columns are detected in the tree search can be eliminated by disabling the dense column handling. This is achieved by setting DENSECOLLIMIT to a larger number. In this case the optimization speed may be degraded, but numerical behavior is usually better.

4.3.3 Crossover

Typically the barrier algorithm terminates when it is within a given tolerance of the optimal solution. Since this solution will lie on the boundary of the feasible region, the Optimizer may perform a crossover at this stage, enabling the optimization to be completed using the simplex method and thus yielding a 'true' optimum. If a basic optimal solution is required, then this procedure must be activated before optimization starts. The CROSSOVER control governs this, set to 1 by default for LP problems. If CROSSOVER is set to 0, no crossover will be attempted and the solution provided will be that determined purely by the barrier method.

4.3.4 Convergence

The optimization is normally terminated when the primal and dual solutions are feasible and the relative duality gap is less than BARGAPSTOP, or in other words:

$$\frac{|primalobj - dualobj|}{1.0 + |dualobj|} \leq BARGAPSTOP$$

For example, BARGAPSTOP = 1.0 E-8 means that eight significant figures will be correct in the optimal objective value. The BARPRIMALSTOP and BARDUALSTOP parameters give the termination criteria for the primal and dual feasibilities. In general it should not be necessary to change the BARGAPSTOP, BARPRIMALSTOP and BARDUALSTOP controls, but if a crossover is employed and the simplex algorithm takes many iterations to get from the crossover basis to an optimal basis, then reducing these controls (e.g. by a factor of 10 to 100) may be worthwhile.

The Newton barrier algorithm computes a search direction at each iteration and takes a step in this direction. If this step size is less than BARSTEPSTOP then the algorithm terminates. If convergence is very slow then it may be better to terminate prematurely by setting a higher value for BARSTEPSTOP and hence invoking the simplex method at an earlier stage.

4.3.5 Output

As with the simplex method, the Console Optimizer can display an iteration log. Similarly, library users may obtain the same information by employing XPRSsetlogfile or XPRSsetcbbarlog. In both cases, this is dependent on the value of the BAROUTPUT control.

4.4 Integer Programming - The Global Search

4.4.1 The Branch and Bound Process

The Xpress-Optimizer uses the Branch and Bound technique to solve mixed integer programming (MIP) problems. A brief overview of this is given here to aid the user in guiding the search for integer solutions. The three major concepts involved are separation, relaxation and fathoming.

The relaxation applied to each integer programming problem is that of dropping the integrality constraints. The relaxed problem is a linear programming problem and can be solved, resulting in one of the following outcomes:

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

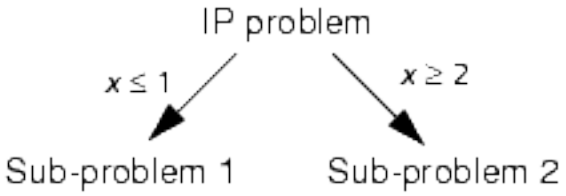
(b) The LP has a feasible solution, but some of the integrality constraints are not satisfied - the MIP has not yet been solved;

(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.

The final outcome (d) is a tricky case. It can only occur at the very first relaxation, in which case the model is not well posed. It will therefore be assumed 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. Suppose, for example, that the optimal LP value of an integer variable `x` is `1.34`, violating integrality. It follows that in any solution to the original problem either `x <= 1.0` or `x >= 2.0`. If the two resulting IP problems are solved, integer values of `x` are guaranteed not to be missed, so the problem is separated into two sub-problems.



If both of these sub-problems can be solved and the better of the two is chosen, then the MIP is solved. Exactly the same relaxation strategy is used to solve each of the sub-problems and consequently this is a recursive solution technique.

This can be depicted as a tree-searching algorithm with a certain degree of arbitrariness. Each *node* of the tree is a MIP sub-problem. That MIP is then relaxed and the LP relaxation can be solved. If the relaxation is not fathomed, then the MIP must be further separated into two more sub-problems, each having the same constraints as the node MIP, plus one further constraint each. Each node is therefore either fathomed or has two descendants.

At some point in exploring the tree an integer solution may be found, providing a bound on the solution to the problem. Clearly the LP relaxation of a MIP will have no worse an optimal objective function value than that of the MIP. The value of the best MIP node found can then act as a *cutoff* for outstanding nodes. If the value of the LP relaxation is no better than the cutoff, then any MIP descendant of the node cannot be better than the MIP solution value which has been found. Again the node can be fathomed and need be considered no further.

The concept of a cutoff value can be applied even when no integer solution has been found if it is known, or it 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 abandoned. There is a danger, however, that all integer feasible solutions, including the optimal one, may be missed if an overly optimistic cutoff value is chosen.

The cutoff concept can be more powerful if a solution within a certain tolerance of the integer optimum is sought. If an integer solution is found, this may be accepted if there is no other solution more than, say, 100 better than it. The cutoff can then be set to be 100 better than the solution that has just been found.

If the problem contains sets then the nodes of the Branch and Bound tree are separated by branching on the sets this is done by choosing a position in the set and setting all members of the set to 0 either above or below the chosen point. Each member of the set has a reference row entry and the sets are ordered by these reference row entries. 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 sets. This is especially the case if the sets are small.

4.4.2 Node and Variable Selection

The branch and bound technique leaves many choices open to the user. However, in practice the success of the technique is highly dependent upon two choices.

(a) At any given stage there will generally be several outstanding nodes which have not been fathomed. The choice of which to solve first is known as the *node selection problem*;

(b) Having chosen a node to tackle, deciding which variable to separate upon is known as the *variable selection problem*.

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. Since the Optimizer makes its variable selection when the LP relaxation has been solved, rather than when it has selected the node, the variable selection problem will be discussed first.

4.4.3 Variable Selection for Branching

Each global entry has a priority for branching, either the default value of 500 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. Up and down pseudo costs for each global entity can be specified, which are estimates of the per unit degradation of forcing the entity away from its LP value.

The Optimizer selects the branching entity from among those entities of the most important priority class which remain unsatisfied. Of these, it takes the one with the highest estimated cost of being satisfied (degradation).

A rather crude estimate of the best integer solution derivable from the node is made by summing the individual entities" estimates. If these estimates are consistently biased in some problem class, it may be worthwhile to specify pseudo costs different from the default of `0.1`. This can be achieved using the `XPRSreaddirs` (`READDIRS`) command.

4.4.4 Node Selection

Each active node has an LP relaxation value and an estimated degradation to an integer solution. The controls `NODESELECTION`, `BACKTRACK`, `VARSELECTION` and `BREADTHFIRST` determine the way the next node is selected.

The value of `NODESELECTION` defines the candidate set for node selection, i.e. the set of nodes from which one will be chosen, while the value of `BACKTRACK` defines the criterion used in selection of a node from the candidate set. If `NODESELECTION` is `1` (the usual default) then the two descendent nodes form the candidate set, but if both have been fathomed then all active nodes form the candidate set. If `NODESELECTION` is `2`, all nodes are always included in the candidate set resulting in a best, or breadth first, search. If `NODESELECTION` is `3`, a depth-first search is performed. If `NODESELECTION` is `4`, all nodes are considered for selection in priority order for the first `BREADTHFIRST` nodes, after which the usual default behavior is resumed.

For deciding between the nodes in the candidate set, the value of `BACKTRACK` determines the selection criterion. If `BACKTRACK` is `1` and `MIPTARGET` has not been set (either directly by the user or by the search previously finding an integer solution), then the node with the best

estimate is chosen. If BACKTRACK is 1 and MIPTARGET has been set, then the Forrest-Hirst-Tomlin Criterion is used. For minimization problems, this chooses the node with the highest value of:

(MIPTARGET − *objective* − *deg*) / *deg*

where *deg* is the estimated degradation. The value of VARSELECTION influences *deg*. If VARSELECTION is 1 (the default) then *deg* is assumed to come from the better of the two possible branching directions for each unsatisfied entity.

Various other ways of calculating *deg* can be actioned by setting VARSELECTION. The table below shows the possible values where up_j and $down_j$ are the estimated up and down degradations of branching on global entity j.

VARSELECTION	Estimated Degradation (deg)
1	$\sum_j min(up_j, down_j)$
2	$\sum_j (up_j + down_j)$
3	$\sum_j \left(2.0 \cdot min(up_j, down_j) + max(up_j, down_j)\right)$
4	$\sum_j max(up_j, down_j)$
5	$\sum_j down_j$
6	$\sum_j up_j$

If BACKTRACK is 2, the node with the smallest estimated solution is always chosen. If BACKTRACK is 3 the node with the smallest bound is always chosen.

4.4.5 Adjusting the Cutoff Value

Since the parameters MIPRELCUTOFF and MIPADDCUTOFF have nonzero default values we must consider carefully the effect of setting MIPADDCUTOFF at different places in the set of commands to the Optimizer. If MIPADDCUTOFF is set prior to XPRSmaxim (MAXIM) or XPRSminim (MINIM) then its value may be altered by the optimization process. At the end of the LP optimization step, MIPADDCUTOFF is set to:

max (MIPADDCUTOFF, 0.01 · MIPRELCUTOFF · *LP_value*)

where *LP_value* is the optimal value found by the LP Optimizer. When this formula is not required and a known value is to be specified for MIPADDCUTOFF, then MIPADDCUTOFF must be set after the LP Optimizer has been run. If a value is specified for MIPRELCUTOFF it must be specified before the LP Optimizer is run.

4.4.6 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 top node). This incorporates reduced cost fixing, binary variable fixing and probing at the top node. 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). The integer preprocessing is not influenced by the linear (l) flag in XPRSmaxim (MAXIM) and XPRSminim (MINIM).

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

Bit	Value	Action
0	1	reduced cost fixing;
1	2	variable fixing;
2	4	probing at root node.

So a value of 1+2=3 for `MIPPRESOLVE` causes reduced cost fixing and variable fixing.

Chapter 5
Implementing Algorithms

5.1 Viewing and Modifying the Matrix

Whilst the simple procedures laid out in the previous chapters will be sufficient for many users of the Xpress-Optimizer, it is sometimes necessary after a model has been loaded to view and change certain properties of a problem's matrix, prior to re-optimizing the altered problem. This may be particularly important if the original problem was found to be infeasible and constraints (matrix rows) needed to be removed to make the feasible region nontrivial. For library users, Xpress-MP provides several functions specifically dedicated to this purpose, a few of which we mention below.

5.1.1 Viewing the Matrix

The Optimizer supports functions which provide access to the objective function, constraint right hand sides, bounds and the matrix elements both prior to and following optimization. In the former case, all information about the problem is available, although clearly the solution will not be. In the latter case, the structures available are dependent on whether or not the matrix is in a presolved state. This is described fully in the following section, so we do not elaborate on it here, other than to remark that full matrix information is only available if the matrix is *not* presolved. If it *is* still presolved, then only partial information will be available. If you are unsure of the matrix status, you may consult the problem attribute `PRESOLVESTATE` to determine its state.

For this section we will be concerned only with matrices which are not in a presolved state. For such matrices, the rows represent the constraints of the problem and may be obtained using `XPRSgetrows`. Their type and range are accessed via the functions `XPRSgetrowtype` and `XPRSgetrowrange`, whilst the names for each constraint may be returned by the `XPRSgetnames` command. The right hand side values and their ranges are available by means of the `XPRSgetrhs` and `XPRSgetrhsrange` routines. Related to the matrix rows, the coefficients of the objective function are similarly obtainable with use of the `XPRSgetobj` routine, whilst the related `XPRSgetqobj` function returns coefficients for quadratic objective functions.

The matrix columns represent the decision variables for the problem and as such the user may also be interested in information about these. The columns will typically have names, which the user may want to access, again possible using the `XPRSgetnames` function. Upper and lower bounds for the columns may be accessed with the commands `XPRSgetub` and `XPRSgetlb`, whilst their type and range are obtainable by means of the functions `XPRSgetcoltype` and `XPRSgetcolrange`, much as for the matrix rows. The `XPRSgetcols` function obtains the matrix columns themselves.

The reference section of this manual gives the precise form and usage for each of these functions in 6 and the user is advised to consult those pages for details and examples before employing them in their own programs.

5.1.2 Modifying the Matrix

In some instances, a model which has previously been solved must be changed before the modified matrix is re-presented for optimization, and a set of routines is provided for this task. Rows and columns can be added (using XPRSaddrows, XPRSaddcols) or deleted from the model (using XPRSdelrows, XPRSdelcols). If rows or columns are to be added, for maximum efficiency, space should be reserved for them before the matrix is read by setting the EXTRAROWS, EXTRACOLS, EXTRAELEMS and EXTRAMIPENTS controls. If this is not done, resizing will be carried out automatically, but more space may be allocated than the user actually requires, potentially resulting in slower solution times.

In the same way, existing row and column types may also be altered (using XPRSchgrowtype, XPRSchgcoltype), as may the matrix coefficients (using XPRSchgcoef, or XPRSchgmcoef if several are to be changed). Right hand sides and their ranges may be changed with XPRSchgrhs and XPRSchgrhsrange, whilst the coefficients of the objective function may be changed with XPRSchgobj. Those for quadratic objective functions may be similarly altered using XPRSchgqobj or XPRSchgmqobj if several such are to be changed.

As mentioned above, a matrix may not be modified if it has been presolved and has not been postsolved, except that the variable bounds may be altered (using XPRSchgbounds). In the following section the Presolve facility will be discussed, along with some suggestions for getting around the difficulties of working with a presolved matrix. Examples of all the above functions and their precise syntax may be found within the reference pages of this manual in 6, to which the user is referred for details of how they might be employed in an Optimizer library program.

5.2 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

5.2.1 Linear Programming Problems

For a linear problem, presolve is called as a default by the XPRSmaxim (MAXIM) and XPRSminim (MINIM) routines, tidying the matrix before the main optimization algorithm is invoked. Following optimization, the whole matrix is automatically *postsolved* to recover a solution to the original problem and restoring the original matrix. Consequently, either before optimization or immediately following solution the full matrix may be viewed and altered as described above, being in its original form.

If for some reason the optimization is interrupted before it has completed, either using the CTRL-C key combination, or due to insufficient LPITERLIMIT or MAXTIME settings, then the problem will remain in its presolved form. The problem may be returned to its original state by calling XPRSpostsolve (POSTSOLVE).

5.2.2 (Mixed) Integer Programming Problems

If a model contains global entities, integer presolve methods such as bound tightening and coefficient tightening are also applied to tighten the LP relaxation. 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 specifying the `l` flag with the `XPRSmaxim` (`MAXIM`) or `XPRSminim` (`MINIM`) command. This removes the global constraints from the variables, preventing the LP relaxation 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`. If you are not interested in the LP relaxation, then it is slightly more efficient to solve the LP relaxation and do the global search in one go, which can be done by specifying the `g` flag for the `XPRSmaxim` (`MAXIM`) or `XPRSminim` (`MINIM`) command.

When `XPRSglobal` (`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`). This can be retrieved later by calling `XPRSreadbinsol` (`READBINSOL`).

After calling `XPRSglobal` (`GLOBAL`), 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.

5.2.3 Common Causes of Confusion

It should be noted that most of the library routines described above and in 6, which modify the matrix will not work on a presolved matrix. The only exceptions are that cuts may be added using the cut pool manager and that the variable bounds may be changed (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 `XPRSgetpresolvebasis`, `XPRSgetscaledinfeas`, `XPRSloadpresolvebasis` and `XPRSloadpresolvedirs`.

5.3 Using the Callbacks

5.3.1 Optimizer Output

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, by far the vast majority of situations in which such routines might be called are associated with

the global search, and will be addressed below.

5.3.2 LP Search Callbacks

In place of catching the standard output from the Optimizer and saving it to a log file, the callback XPRSsetcbmessage 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.

Alternatively, the pair of functions XPRSsetcblplog and XPRSsetcbbarlog 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 this routine should be called.

5.3.3 Global Search Callbacks

When a problem with global entities is to be optimized, a large number of LP 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. Such a routine may be called whenever a new node is selected using XPRSsetcbprenode, and could be used to change the choice of node. Routines could be specified using either of the XPRSsetcboptnode or XPRSsetcbintsol callbacks, to be invoked whenever an optimal solution or an integer solution is found at a particular node, or using XPRSsetcbinfnode for when a node is found to be infeasible. Whenever a node is cut off as a result of an improved integer solution being found, a routine may be called if specified using XPRSsetcbnodecutoff. Using XPRSsetcbchgbranch or XPRSsetcbchgnode, routines can be specified to be invoked whenever a new branching variable is set or whenever the code backtracks to select a new node. Perhaps more technically, XPRSsetcbsepnode and XPRSsetcbestimate may specify routines determining how to separate on a node or obtaining the estimated degradation at each node from branching on the user's global entities.

The final global callback, XPRSsetcbgloballog, 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.

5.4 Working with the Cut Manager

5.4.1 Cuts and the Cut Pool

The global search for a solution of a (mixed) integer problem involves optimization of a large number of LP problems, known as *nodes*. This process 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 any 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 are 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 you keep track of 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.

5.4.2 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 usually best to delete only those cuts with basic slacks, or else 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 may be returned using XPRSgetcutlist.

5.4.3 User Cut Manager Routines

Users may also write their own cut manager routines to be called at various points during the Branch and Bound search. Such routines must be defined in advance using library function calls, similar to callbacks and are defined according to the frequency at which they should be called. At the beginning of the process a cut manager initialization routine may be called and should be specified using XPRSsetcbinitcutmgr. In the same way, at the end of the process, XPRSsetcbfreecutmgr allows the specification of a termination routine. The command XPRSsetcbcutmgr allows the definition of a routine which may be called at each node in the tree.

Further details of these functions may be found in 6 within the functional reference which follows.

5.5 Goal Programming

5.5.1 Overview

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 *Archimedian* 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 Archimedian 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 Archimedian 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 Archimedian 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.

5.5.2 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.

5.5.3 Archimedian 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.

5.5.4 Pre-emptive Goal Programming Using Objective Functions

Suppose that we now have a set of objective functions of which we know 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	$\leq$	100			

For each N row 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	$\geq$	-4.615385 - 0.14.615385

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	$\leq$	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	$\geq$	141.943995 - 0.2141.943995

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

5.5.5 Archimedian 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	$\leq$	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 6

Console and Library Functions

A large number of routines are available for both Console and Library users of the 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.

6.1 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 Xpress, only the following functions will be of relevance:

Command	Description	Page
EXIT	Terminate the Console Optimizer.	p. 64
HELP	Quick reference help for the optimizer console	p. 120
PRINTRANGE	Writes the ranging information to screen.	p. 150
PRINTSOL	Write the current solution to screen.	p. 151
QUIT	Terminate the Console Optimizer.	p. 152
STOP	Terminate the Console Optimizer.	p. 196
ALTER	Alters or changes matrix elements, right hand sides and constraint senses in the current problem.	p. 40
BASISCONDITION	Calculates the condition number of the current basis after solving the LP relaxation.	p. 41
FIXGLOBAL	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. 65
GETMESSAGESTATUS	Manages suppression of messages.	p. 92
GLOBAL	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.	p. 116
GOAL	Perform goal programming.	p. 118
IIS	Initiates the search for Irreducible Infeasible Sets (IIS) amongst problems which are linear infeasible.	p. 121
MAXIM, MINIM	Begins a search for the optimal LP solution.	p. 143
POSTSOLVE	Postsolve the current matrix when it is in a presolved state.	p. 147
RANGE	Calculates the ranging information for a problem and saves it to the binary ranging file problem_name.rng.	p. 153
READBASIS	Instructs the Optimizer to read in a previously saved basis from a file.	p. 154
READBINSOL	Reads a solution from a binary solution file.	p. 155

READDIRS	Reads a directives file to help direct the global search.	p. 156
READPROB	Reads an (X)MPS or LP format matrix from file.	p. 158
RESTORE	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.	p. 160
SAVE	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.	p. 162
SCALE	Re-scales the current matrix.	p. 163
SETMESSAGESTATUS	Manages suppression of messages.	p. 193
SETPROBNAME	Sets the current default problem name. This command is rarely used.	p. 194
WRITEBASIS	Writes the current basis to a file for later input into the Optimizer.	p. 200
WRITEBINSOL	Writes the current MIP or LP solution to a binary solution file for later input into the Optimizer.	p. 201
WRITEOMNI	Writes the current solution to the binary OMNI format file SOLFILE, as recorded in the solution file problem_name .sol. Optionally the current matrix may also be written. All information is appended to this file.	p. 202
WRITEPROB	Writes the current problem to an MPS or LP file.	p. 204
WRITEPRTRANGE	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).	p. 205
WRITEPRTSOL	Writes the current solution to a fixed format ASCII file, problem_name .prt.	p. 206
WRITERANGE	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.	p. 207
WRITESOL	Writes the current solution to a CSV format ASCII file, problem_name.asc (and .hdr).	p. 209

For a list of functions by task, refer to 2.7.

6.2 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 for the sake of clarity. The library name for a function is on the left and the Console Xpress name, where relevant, 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. Where the function forms part of the Console Mode, the library syntax is described first, followed by the Console Xpress 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 9. 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 Xpress-MP 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.

XPRSaddcols

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[]);
```

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+1` 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	Zero 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	≤	24		subject to:	x + 4y + 2z	≤	24
		y	≤	5			y + z	≤	5
		3x + y	≤	20			3x + y	≤	20
		x + y	≤	9			x + y + 3	≤	9
							z	≤	12

Using `XPRSaddcols`, the following transforms (a) into (b) and then names the new variable using XPRSaddnames:

```
obj[0] = 3;
mstart[] = {0, 3};
mrwind[] = {0, 1, 3};
matval[] = {2.0, 1.0, 3.0};
bdl[0] = 0.0; bdu[0] = 12.0;
...
XPRSaddcols(prob,1,3,obj,mstart,mrwind,matval,bdl,bdu);
XPRSaddnames(prob,2,"z",2,2);
```

Further information

1. For maximum efficiency, space for the extra rows and elements should be reserved by setting the EXTRACOLS, EXTRAELEMS and EXTRAMIPENTS controls before loading the problem.
2. 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.
3. If the columns are added to a MIP problem then they will be continuous variables.

Related topics

XPRSaddnames, XPRSaddrows, XPRSalter, XPRSdelcols.

XPRSaddcuts

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[]);
```

Arguments

`prob` The current problem.

`ncuts` Number of cuts to add.

`mtype` Integer array of length `ncuts` containing the cut types. The cut types can be any positive 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 $\geq$ 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.

Related controls

Double

`MATRIXTOL` Zero tolerance on matrix elements.

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.

Related topics

`XPRSaddrows`, `XPRSdelcpcuts`, `XPRSdelcuts`, `XPRSgetcpcutlist`, `XPRSgetcutlist`, `XPRSloadcuts`, `XPRSstorecuts`, 5.4.

XPRSaddnames

Purpose

When a model is loaded, the rows, columns and sets of the model may not have names associated with them. This may not be important as the rows, columns and sets can be referred to by their sequence numbers. However, if you wish row, column and set names to appear in the ASCII solutions files, the names for a range of rows or 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.
cnames	Character buffer containing the null-terminated string names - each name may be at most MPSNAMELENGTH+1 characters including the compulsory null terminator. If this control is to be changed, this must be done before loading the problem.
first	Start of the range of rows, columns or sets.
last	End of the range of rows, columns or sets.

Related controls

Integer

MPSNAMELENGTH	Maximum name length in characters.

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.

XPRSaddrows

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[]);
```

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` Integer array of length `newrow` containing the offsets in the `mclind` and `dmatval` arrays of the start of the elements for each row. 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+1` 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` Zero tolerance on matrix elements.

Example

Suppose the current problem was:

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, 3};
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

For maximum efficiency, space for the extra rows and elements should be reserved by setting the EXTRAROWS and EXTRAELEMS controls before loading the problem.

Related topics

XPRSaddcols, XPRSaddcuts, XPRSaddnames, XPRSdelrows.

XPRSaddsets

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, char
      qrtype[], int msstart[], int mclind[], double dref[]);
```

Arguments

`prob` The current problem.

`newsetsx` 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+1` 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 - each name may be at most `MPSNAMELENGTH+1` characters including the compulsory null terminator. If this control is to be changed, this must be done before loading the problem.
`first`	Start of the range of sets.
`last`	End of the range of sets.

Related controls

Integer

`MPSNAMELENGTH`	Maximum name length in characters.

Example

Add set names (`set1` and `set2`) to a problem:

```
char snames[] = "set1\0set2"
...
XPRSaddsetnames(prob,snames,0,1);
```

Related topics

`XPRSaddnames`, `XPRSloadglobal`, `XPRSloadqglobal`.

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 200 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` Zero 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 A.7. The `MODIFY` format of the MPS `REVISE` data is also supported.
2. The command `XPRSalter` (`ALTER`) and the control `EXTRAELEMS` work together to enable the user to change values and constraint senses in the problem held in memory. For maximum efficiency, it should be set to reserve space for additional matrix elements. Defining the maximum number of extra elements that can be added, it must be set before `XPRSreadprob` (`READPROB`).
3. It is not possible to alter an integer model which has been presolved. If it is required to alter such a model after optimization, either turn the presolve off by setting `PRESOLVE` to `0` prior to optimization, or reread the model with `XPRSreadprob` (`READPROB`).

Related topics

A.7.

Purpose

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.

```
XPRSminim(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
MINIM
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. The condition number is shown both for the scaled problem and in parenthesis for the original problem.

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 appear in the array.

Related controls

Double

ETATOL	Zero 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 will work with the basis for the presolved problem.

Related topics

XPRSftran.

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`	Zero 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.

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 MIP search is started. If `XPRSchgcoltype` is called after a problem has been presolved, the presolved column numbers must be supplied. It is not possible to change a column into a partial integer, semi-continuous or semi-continuous integer variable.
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.

Related topics

`XPRSaddcols`, `XPRSchgrowtype`, `XPRSdelcols`, `XPRSgetcoltype`.

XPRSchgmcoef

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[]);
```

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`	Zero 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

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[]);
```

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`.

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 $[6x_1^2 + 3x_1x_2 + 3x_2x_1] / 2$ to the objective function:

```
icol = jcol = 0; dval = 6.0;
XPRSchgqobj(prob,icol,jcol,dval);
icol = 0; jcol = 1; dval = 3.0;
XPRSchgqobj(prob,icol,jcol,dval);
```

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` .

XPRSchgrhs

Purpose

Used to change right hand side elements of the matrix.

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 elements to change.
`mindex`	Integer array of length `nels` containing the indices of the rows on which the right hand side elements 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 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);
XPRSsetcbmessage(prob1,callback,NULL);
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 200 characters to contain the name for the copied problem. 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.

Related topics

`XPRSdestroyprob`, `XPRScopyprob`, `XPRSinit`.

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, XPRScut mcutind[]);
```

Arguments

prob	The current problem.
itype	Cut type.
interp	Way in which the cut type is interpreted: −1 drop all cuts; 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, 5.4.

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, 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	Type of the cut to be deleted.
interp	Way in which the cut itype is interpreted: −1 delete 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.

Related topics

XPRSaddcuts, XPRSdelcpcuts, XPRSgetcutlist, XPRSloadcuts, 5.4.

XPRSdelnode

Purpose

Deletes the specified node from the list of outstanding nodes in the Branch and Bound tree search.

Synopsis

```
int XPRS_CC XPRSdelnode(XPRSprob prob, int inode, int ifboth);
```

Arguments

`prob` The current problem.

`inode` Number of the node to delete.

`ifboth` Flag which must be one of:

- `0` meaning that the next descendant is to be deleted;
- `1` meaning that both descendants are to be deleted.

Example

```
XPRSdelnode(prob,10,0);
```

This deletes node number `10` in the tree search and its next descendent.

Further information

This routine might most effectively be called from a callback within the Branch and Bound search.

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`.

XPRSdelsets

Purpose

Delete sets from a problem.

Synopsis

```
int XPRS_CC XPRSdelsets(XPRSprob prob, int ndelsets, const int mindex[]);
```

Arguments

`prob` The current problem.

`ndelsets` Number of sets to delete.

`mindex` An integer array of length `ndelsets` 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","");
XPRSmaxim(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`.

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 9.
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).

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 XPRSfixglobal(XPRSprob prob);
FIXGLOBAL
```

Argument

`prob` The current problem.

Example 1 (Library)

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

```
XPRSreadprob(prob,"myprob","");
XPRSminim(prob,"g");
XPRSfixglobal(prob);
XPRSminim(prob,"l");
XPRSwriteprtsol(prob);
```

Example 2 (Console)

A similar set of commands at the console would be as follows:

```
READPROB
MINIM -g
FIXGLOBAL
MINIM -l
PRINTSOL
```

Further information

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 `XPRSrange` (`RANGE`) after calling `XPRSfixglobal` (`FIXGLOBAL`).

Related topics

`XPRSglobal` (`GLOBAL`), `XPRSrange` (`RANGE`).

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	Zero 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.

XPRSgetbanner

Purpose

Returns the banner and copyright message.

Synopsis

```
int XPRS_CC XPRSgetbanner(char *banner);
```

Argument

`banner` Buffer long enough to hold the banner (plus a null terminator). This can be at most 256 characters.

Example

The following calls `XPRSgetbanner` to return banner information at the start of the program:

```
char banner[256];
...
if(XPRSinit(NULL))
{
  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 areas.

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);
XPRSminim(prob,"");
XPRSgetbasis(prob,rstatus,cstatus);
```

Related topics

`XPRSgetpresolvebasis`, `XPRSloadbasis`, `XPRSloadpresolvebasis`.

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

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);
```

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 elements for each column. 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 `mrwind` and `dmatval` arrays 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.

Example

```
int nels, cols, first = 0, last;
...
XPRSgetintattrib(prob,XPRS_COLS,&cols);
last = cols-1;
XPRSgetcols(prob,NULL,NULL,NULL,0,&nels,first,last);
```

This returns in `nels` the number of nonzero matrix elements in all columns of the matrix.

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 variables;
- `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.

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` Cut 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 an absolute slack value 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 slack variables for the cuts will be returned.

Further information

1. The violated cuts can be obtained by setting the `delta` parameter to the size of the 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`.

Related topics

`XPRSdelcpcuts`, `XPRSgetcpcuts`, `XPRSgetcutlist`, `XPRSloadcuts`, 5.4.

XPRSgetcpcuts

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, XPRScut mindex[], int ncuts,
       int size, int mtype[], char qrtype[], int 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.
`qrtype`	Character array of length at least `ncuts` where the sense of the cuts (`L`, `G`, or `E`) will be returned.
`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.
`mcols`	Integer array of length `size` where the column indices of the cuts will be returned.
`dmatval`	Double array of length `size` where the matrix values will be returned.
`drhs`	Double array of length at least `ncuts` where the right hand side elements for the cuts will be returned.

Related topics

`XPRSgetcpcutlist`, `XPRSgetcutlist`, 5.4.

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` Cut 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`, 5.4.

XPRSgetdaysleft

Purpose

Returns the number of days left until an evaluation license expires.

Synopsis

```
int XPRS_CC XPRSgetdaysleft(int *days);
```

Argument

`days` Pointer to an integer where the number of days is to be returned.

Example

The following calls `XPRSgetdaysleft` to print information about the license:

```
int days;
...
XPRSinit(NULL);
if(XPRSgetdaysleft(&days) == 0) {
  printf("Evaluation license expires in %d days\n", days);
} else {
  printf("Not an evaluation license\n");
}
```

Further information

This function can only be used with evaluation licenses, and if called when a normal license is in use returns an error code of 32. The expiry information for evaluation licenses is also included in the Optimizer banner message.

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 8, 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;
...
XPRSmaxim(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 7, 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`,...).

`mpri` Integer array of length `ndir` containing the priorities for the columns and sets.

`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.

`ddpc` Double array of length `ndir` containing the down pseudo costs for the columns and sets.

Further information

1. The value `ndir` denotes the number of directives, at most `MIPENTS`, obtainable with `XPRSgetintattrib(prob,XPRS_MIPENTS,& mipents);`.
2. Any of the arguments except `prob` and `ndir` may be `NULL` if not required.

Related topics

`XPRSloaddirs`, `XPRSloadpresolvedirs`.

XPRSgetglobal

Purpose

Retrieves global information about a problem. It must be called before XPRSmaxim or XPRSminim 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[]);
```

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.

XPRSgetiis

Purpose

Retrieves the final Irreducible Infeasible Set (IIS) found in an IIS search.

Synopsis

```
int XPRS_CC XPRSgetiis(XPRSprob prob, int *colnumber, int *rownumber, int
      miiscol[], int miisrow[]);
```

Arguments

`prob` The current problem.

`colnumber` Number of columns in the IIS.

`rownumber` Number of rows in the IIS.

`miiscol` Integer array of length `colnumber` containing the column indices for the IIS set. May be `NULL` if not required.

`miisrow` Integer array of length `rownumber` containing the row indices for the IIS set. May be `NULL` if not required.

Related controls

Integer

`MAXIIS`	Number of Irreducible Infeasible Sets to be found

Example

The following finds and retrieves an IIS in the problem `prob` :

```
int ncols, nrows, *miiscol, *miisrow;
...
XPRSiis(prob,"");
XPRSgetiis(prob,&ncols,&nrows,NULL,NULL);
miiscol = malloc(ncols*sizeof(int));
miisrow = malloc(nrows*sizeof(int));
XPRSgetiis(prob,&ncols,&nrows,miiscol,miisrow);
```

Further information

1. `XPRSgetiis` can only be called after the function `XPRSiis`, which computes the IISs in a problem.
2. To retrieve all of the IIS in a problem, you need to call `XPRSiis` and `XPRSgetiis` repeatedly, setting `MAXIIS` to 1, then 2, then 3, etc. For example:

```
XPRSiis(prob, "");
XPRSgetintattrib(prob, XPRS_NUMIIS, &niis);
for (i=1; i<=niis; i++) {
  XPRSsetintcontrol(prob, XPRS_MAXIIS, i);
  XPRSiis(prob, "");
  XPRSgetiis(prob, &ncols, &nrows, miiscol, miisrow);
  /* display or use IIS */
}
```

Related topics

`XPRSiis`.

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` String of length `MPSNAMELENGTH` (plus a null terminator) holding the name of the row or column.

`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.

Related controls

Integer

`MPSNAMELENGTH` Maximum name length in characters.

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`.

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	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 values

91	A current problem is not available.
422	A solution is not available.

Related controls

Double

FEASTOL	Zero 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

XPRSgetiis, XPRSgetscaledinfeas, XPRSiis.

XPRSgetintattrib

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);
```

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 8, 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

Purpose

Enables users to recover the values of various integer control parameters

Synopsis

```
int XPRS_CC XPRSgetintcontrol(XPRSprob prob, int ipar, int *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 7, 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;
...
XPRSmaxim(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`.

XPRSgetlasterror

Purpose

Returns the last error encountered during an optimization run.

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(XPRScreateprob(&prob))
  error(prob,"XPRScreateprob");
```

Related topics

9, ERRORCODE, XPRSsetcbmessage, XPRSsetlogfile.

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 (plus a null terminator).

`length` Length of the buffer. This should be 512 or more since messages can be quite long.

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

The error message included 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.

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 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 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;
...
XPRSmaxim(prob,"");
XPRSgetintattrib(prob,XPRS_ORIGINALCOLS,&cols);
x = malloc(cols*sizeof(double));
XPRSgetlpsol(prob,x,NULL,NULL,NULL);
XPRSglobal(prob);
y = malloc(cols*sizeof(double));
XPRSgetmipsol(prob,y,NULL);
```

Further information

1. If called during an XPRSglobal callback the solution of the current node will be returned.
2. If the matrix is modified after calling XPRSmaxim or XPRSminim, then the solution will no longer be available.
3. 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

XPRSgetpresolvesol, XPRSgetmipsol, XPRSwriteprtsol, XPRSwritesol.

Purpose

Manages suppression of messages.

Synopsis

```
int XPRS_CC XPRSgetmessagestatus(XPRSprob prob, int errcode, int
      *status);
GETMESSAGESTATUS
```

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 9 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.

Further information

1. Use the `SETMESSAGESTATUS` console function to print the value of the suppression status to screen.
2. 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;
...
XPRSmaxim(prob,"g");
XPRSgetintattrib(prob,XPRS_ORIGINALCOLS,&cols);
x = malloc(cols*sizeof(double));
XPRSgetmipsol(prob,x,NULL);
```

Further information

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.

Related topics

`XPRSgetpresolvesol`, `XPRSwriteprtsol`, `XPRSwritesol`.

XPRSgetmqobj

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);
```

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 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`	Double array of length `size` containing the objective function coefficients.
`nels`	Pointer to the integer where the number of nonzero elements in the `mclind` and `dobjval` arrays 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.

Further information

1. The objective function is of the form `c'x+0.5x'Qx` where `Q` is positive semi-definite for minimization problems and negative semi-definite for maximization problems. If this is not the case the optimization algorithms may converge to a local optimum or may not converge at all. Note that only the upper or lower triangular part of the `Q` matrix is returned.
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

`XPRSchgmqobj`, `XPRSchgqobj`, `XPRSgetqobj`.

XPRSgetnames

Purpose

Returns the names for the rows, columns or set 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.

`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 or set in the range.

`last` Last row, column or set in the range.

Related controls

Integer

`MPSNAMELENGTH` Maximum name length in characters.

Example

The following example retrieves the row and column names of the current problem:

```
int cols, rows, nl;
...
XPRSgetintattrib(prob,XPRS_ORIGINALCOLS,&cols);
XPRSgetintattrib(prob,XPRS_ORIGINALROWS,&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]` in C, use

```
int namelength;
...

XPRSgetintattrib(prob,XPRS_NAMELENGTH,&namelength);
printf("%s",names + i*(8*namelength+1));
```

Related topics

`XPRSaddnames`.

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.

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","");
XPRSminim(prob,"");
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]);
XPRSglobal(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);
XPRSmaxim(prob,"");
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

In order to get a one-to-one mappng between the presolve problem and the original problem the elimination operations of the presolve must be turned off using;

```
XPRSsetintcontrol(prob,XPRS_PRESOLVEOPS,255);
```

Related topics

5.2.

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","");
XPRSmaxim(prob,"");
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]);
XPRSglobal(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`.

Related topics

`XPRSgetlpsol`, 5.2.

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 200 characters to contain the current problem name.

Example

The following returns the problem name into `probname`:

```
char probname[200];
...
XPRSgetprobname(prob,probname);
```

Related topics

XPRSsetprobname.

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

`XPRSchgqobj`, `XPRSchgmqobj`.

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

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);
```

Arguments

`prob` The current problem.

`mstart` Integer array which will be filled with the indices indicating the starting offsets in the `mclind` and `dmatval` arrays for each requested row. It must be of length at least `last-first+2`. Column `i` starts at position `mstart[i]` in the `mrwind` and `dmatval` arrays, and has `mstart[i+1]-mstart[i]` elements in it. May be `NULL` if not required.

`mclind` 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 `mclind` and `dmatval` arrays will be returned. If the number of nonzero elements is greater that `size`, then only `size` elements will be returned. If `nels` is smaller that `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, mstart[3], mclind[6];
double dmatval[6];
...
XPRSgetrows(prob,mstart,mclind,dmatval,size,&nels,0,1);
for(i=0;i<nels;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`, `mclind` 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.

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	Zero 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

XPRSgetiis, XPRSgetinfeas, XPRSiis.

XPRSgetsol

Purpose

This function is deprecated and will be removed in version 18. Users should use the XPRSgetlpsol or XPRSgetmipsol functions instead. It is included for compatibility with version 16 and is used to obtain the solution values following optimization.

Synopsis

```
int XPRS_CC XPRSgetsol(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.

Related controls

Integer

SOLUTIONFILE	Enable/disable the binary solution file.

Example

The following sequence of commands will get the solution (x) at the top node and the optimal MIP solution (y) for a problem:

```
int cols;
double *x, *y;
...
XPRSmaxim(prob,"");
XPRSgetintattrib(prob,XPRS_ORIGINALCOLS,&cols);
x = malloc(cols*sizeof(double));
XPRSgetsol(prob,x,NULL,NULL,NULL);
XPRSglobal(prob);
y = malloc(cols*sizeof(double));
XPRSgetsol(prob,y,NULL,NULL,NULL);
```

Further information

1. If the matrix is modified after calling XPRSmaxim or XPRSminim, then the solution will no longer be available.
2. If the problem has been presolved, then XPRSgetsol returns the solution to the original problem. The only way to obtain the presolved solution is to call the related function, XPRSgetpresolvesol.

Related topics

XPRSgetlpsol, XPRSgetmipsol.

XPRSgetstrattrib

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);
```

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 8, 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.

Example

The following retrieves the name of the matrix just loaded:

```
char matrixname[256];
...
XPRSreadprob(prob,"myprob","");
XPRSgetstrattrib(prob, XPRS_MATRIXNAME, matrixname);
```

Related topics

XPRSgetdblattrib, XPRSgetintattrib.

XPRSgetstrcontrol

Purpose

Returns the value of a given string control parameters.

Synopsis

```
int XPRS_CC XPRSgetstrcontrol(XPRSprob prob, int ipar, char *cgval);
```

Arguments

`prob`	The current problem.
`ipar`	Control parameter whose value is to be returned. A full list of all controls may be found in 7, 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.

Example

In the following, the value of `MPSBOUNDNAME` is retrieved and displayed:

```
char mpsboundname[256];
...
XPRSgetstrcontrol(prob, XPRS_MPSBOUNDNAME, mpsboundname);
printf("Name = %s\n", mpsboundname);
```

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`, `XPRSmaxim` and `XPRSminim`.

XPRSgetversion

Purpose

Returns the full Optimizer version number in the form 15.10.03, where 15 is the majorrelease, 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 supercedes the `VERSION` control, which only returns the first two parts of the version number. Relase 2004 versions of the Optimizer have a three-part version number.

Related topics

`XPRSinit`.

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.

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.
KEEPMIPSOL	Number of integer solutions to store.
MAXMIPSOL	Maximum number of MIP solutions to find.
MAXNODE	Maximum number of nodes in Branch and Bound search.
MAXSLAVE	Number of slave processors used for parallel MIP search.
MAXTIME	Maximum time allowed.
MIPLOG	Global print flag.
MIPPRESOLVE	Type of integer preprocessing to be performed.
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

DEGRADEFACTOR	Factor to multiply estimated degradations by.
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.
MIPTARGET	Target object function for global.

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 XPRSpostolve (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.9).
6. Optimizer library users can check the final status of the global search using the MIPSTATUS problem attribute.
7. The Optimizer supports global (i.e. active node list) files in excess of 2 GigaBytes by spreading the data over multiple files. The initial global file is given the name *probname*.glb and subsequent files are named *probname*.glb.1, *probname*.glb.2,.... No individual file will be bigger than 2GB.

Related topics

XPRSfixglobal (FIXGLOBAL), XPRSinitglobal, XPRSmaxim (MAXIM)/XPRSminim (MINIM), A.9.

Purpose

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 200 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`.

Related controls

Integer

`KEEPMIPSOL` Number of partial solutions to store when using pre-emptive goal programming.

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 Archimedian 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 Archimedian/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

5.5.

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.

Purpose

Initiates the search for Irreducible Infeasible Sets (IIS) amongst problems which are linear infeasible.

Synopsis

```
int XPRS_CC XPRSiis(XPRSprob prob, const char *flags);
IIS [-flags]
```

Arguments

`prob` The current problem.

`flags` Flags to pass to `XPRSiis` (`IIS`). Can be set to the following:

- `o` display IIS information and optimization process logs on screen;
- `f` write IIS information into a file *problem_name*`.iis`.

If no flags are set, only the IIS information will be displayed to the screen.

Related controls

Integer

`MAXIIS` Number of Irreducible Infeasible Sets to be found.

Example 1 (Library)

This example searches for IISs and then questions the problem attribute `NUMIIS` to determine how many were found:

```
int iis;
...
XPRSiis(prob,"f");
XPRSgetintattrib(prob, XPRS_NUMIIS, &iis);
printf("number of IISs = %d\n", iis);
```

Example 2 (Console)

Calling the command on its own begins a search for as many IISs as have been specified by the control `MAXIIS`:

```
IIS
```

Further information

1. A model may have several infeasibilities. Repairing a single IIS may not make the model feasible. For this reason the Optimizer 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 `XPRSiis` (`IIS`) 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 independent infeasibilities in the model.

2. The problem attribute, `NUMIIS`, allows the user to recover the number of IISs found in a particular search. See Example 1 for details of this.

Related topics

`XPRSgetiis`, 3.2.

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 Xpress-MP 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.

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`.

XPRSinitglobal

Purpose

Reinitialises the global tree search. By default if XPRSglobal is interrupted and called again the global search will continue from where it left off. If XPRSinitglobal is called after the first call to XPRSglobal, the global search will start from the top node when XPRSglobal is called again.

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);
XPRSmaxim(prob,"g");
```

Related topics

XPRSglobal, XPRSmaxim (MAXIM)/XPRSminim (MINIM).

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);
XPRSminim(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`.

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,
       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. A value of `-1` indicates load all cuts of type `itype`.

`mcutind` Array containing pointers to the cuts to be loaded into the matrix. This array may be `NULL` if `ncuts` is `-1`, otherwise it has length `ncuts`. Any indices of `-1` will be ignored so that the array `mindex` returned from `XPRSstorecuts` can be passed directly to `XPRSloadcuts`.

Related topics

`XPRSaddcuts`, `XPRSdelcpcuts`, `XPRSdelcuts`, `XPRSgetcplist`, 5.4.

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. 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

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[]);
```

Arguments

`prob` The current problem.

`probname` A string of up to 200 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 length `ngents` containing the column indices of the global entities.

`dlim` Double array 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`	Zero tolerance on matrix elements.
`SOSREFTOL`	Minimum gap between reference row entries.

Example

The following specifies an integer problem, `globalEx`, corresponding to:

maximize:	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

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[]);
```

Arguments

`prob` The current problem.

`probname` A string of up to 200 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. 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 cound 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	Zero tolerance on matrix elements.

Example

Given an LP problem:

maximize:	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.

XPRSloadmipsol

Purpose

Loads a 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;
`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);
XPRSminim(prob,"g");
```

Further information

The values for the continuous variables in the `dsol` array are ignored and are calculated by fixing the integer variables and reoptimizing.

Related topics

`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);
XPRSminim(prob,"g");
```

Further information

1. 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 re-optimized. This continues until no violated cuts remain.
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

5.4.

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","");
XPRSminim(prob,"");
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","");
XPRSminim(prob2,"");
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};
...
XPRSminim(prob,"");
XPRSloadpresolvedirs(prob,1,mcols,mpri,NULL,NULL,NULL);
XPRSminim(prob,"g");
```

Related topics

XPRSgetdirs, XPRSloaddirs.

XPRSloadqglobal

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[], const 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[]);
```

Arguments

`prob` The current problem.

`probname` A string of up to 200 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 length `ngents` containing the column indices of the global entities.

`dlim` Double array 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`	Zero 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+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

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[]);
```

Arguments

`prob` The current problem.

`probname` A string of up to 200 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	Zero 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+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, int mrow[],
       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);
XPRSminim(prob,"");
```

Related topics

5.2.

Purpose

Begins a search for the optimal LP solution.

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.
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	Zero tolerance in the Cholesky decomposition.
ELIMTOL	Markowitz tolerance for elimination phase of presolve.
ETATOL	Zero tolerance on eta elements.
FEASTOL	Zero 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.

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 are 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.

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

`XPRSminim`, `XPRSmaxim`

XPRSpresolvecut

Purpose

Presolves a cut formulated in terms of the original variables such that it can be added to a presolved matrix.

Synopsis

```
int XPRS_CC XPRSpresolvecut(XPRSprob prob, char qrtype, double drhso, int
      nzo, const int mcolso[], const double dvalo[], double *drhsp, int
      *nzp, int mcolsp[], double dvalp[], int *status);
```

Arguments

`prob` The current problem.

`qrtype` The row type of the cut:
- `L` indicates a $\leq$ cut;
- `E` indicates a = cut;
- `G` indicates a $\geq$ cut.

`drhso` The right-hand side constant of the cut to presolve.

`nzo` Number of elements in the `mcolso` and `dvalo` arrays.

`mcolso` Integer array of length `nzo` containing the column indices of the cut to presolve.

`dvalo` Double array of length `nzo` containing the non-zero coefficients of the cut to presolve.

`drhsp` Pointer to the double where the presolved right-hand side will be returned.

`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 cut. It must be allocated to hold at least `COLS` elements.

`dvalp` Double array which will be filled with the coefficients of the presolved cut. It must be allocated to hold at least `COLS` elements.

`status` Status of the presolved cut:
- `0` The cut was successfully presolved;
- `1` Presolving the cut failed.

Related controls

Integer

PRESOLVE	Turns presolve on or off.
PRESOLVEOPS	Selects the presolve operations.

Example

Suppose we want to add the cut $2x_1 + x_2 \leq 1$ 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;
...
XPRSpresolvecut(prob, qrtype, drhso, 2, mindo, dvalo, &drhsp,
                &nzp, mindp, dvalp, &status);
if (!status) {
  mtype = 0;
  mstart[0] = 0; mstart[1] = nzp;
  XPRSaddcuts(prob, 1, &mtype, &qrtype, &drhsp, mstart, mindp,
              dvalp);
}
```

Further information

The are certain presolve operations that can prevent presolving a cut successfully. These are singleton column removal, duplicate column removal and variable eliminations. Thus, bits 0, 5 and 8 (total value 289) of PRESOLVEOPS should be cleared before calling XPRSminim or XPRSmaxim on a matrix. Furthermore, bit 11 (value 2048) should be set to prevent some further reductions that conflict with presolving cuts. A safe settting of PRESOLVEOPS is therefore 2270.

Related topics

XPRSaddcuts, XPRSstorecuts.

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`	Zero tolerance on print values.

Further information

See `WRITEPRTRANGE` for more information.

Related topics

`XPRSgetcolrange`, `XPRSgetrowrange`, `XPRSrange` (`RANGE`), `XPRSwriteprtsol`, `XPRSwriterange`, A.5.

Purpose

Writes the current solution to the screen.

Synopsis

PRINTSOL

Related controls

Integer

MAXPAGELINES Number of lines between page breaks.

Double

OUTPUTTOL Zero tolerance on print values.

Further information

See WRITEPRTSOL for more information.

Related topics

XPRSgetlpsol, XPRSgetmipsol, XPRSwriteprtsol.

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 9.
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`).

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","");
XPRSmaxim(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
MAXIM
RANGE
PRINTRANGE
```

Further information

1. A basic optimal solution to the problem must be available, i.e. `XPRSmaxim` (`MAXIM`) or `XPRSminim` (`MINIM`) 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 `XPRSfixglobal` (`FIXGLOBAL`) first and the remaining LP resolved - see `XPRSfixglobal` (`FIXGLOBAL`).

Related topics

`XPRSgetcolrange`, `XPRSgetrowrange`, `XPRSwriteprtrange` (`WRITEPRTRANGE`), `XPRSwriterange` (`WRITERANGE`).

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 200 characters containing the file name from which the basis is to be read. If omitted, the default *problem_name* is used with a `.bss` extension.

`flags` 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,"","");
XPRSmaxim(prob,"g");
```

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
MAXIM -g
```

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`).

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 200 characters containing the file name from which the solution is to be read. If omitted, the default *problem_name* is used with a `.sol` extension.

`flags` 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`).

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 200 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");
XPRSminim(prob,"g");
```

Example 2 (Console)

```
READPROB
READDIRS
MINIM -g
```

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 XPRSmaxim (MAXIM) or XPRSminim (MINIM).

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, XPRSglobal (GLOBAL) 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 XPRSglobal (GLOBAL) 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 XPRSglobal (GLOBAL) 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.

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 file name, a string of up to 200 characters from which the problem is to be read. 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;
- `b` read in a problem in the mp-model binary interface file format. The files `probname.bif` and `probname.sol`, generated by mp-model's `BIFGENERATE` command, are required.
- `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.
MPSERRIGNORE	Number of minor errors to ignore.
MPSFORMAT	Specifies format of MPS files.
MPSNAMELENGTH	Maximum name length in characters.
SCALING	Type of scaling.

Double

MATRIXTOL	Zero 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)

If `child[1]=child::file[1]`

```
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, SETS, RHS, RANGES and BOUNDS must 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.

7. Long MPS vector names are supported in MPS files, LP files, directives files and basis files. The `MPSNAMELENGTH` control specifies the maximum number of characters in MPS vector names and must be set before the file is read in. Internally it is rounded up to the smallest multiple of `8`, and must not exceed `64`.

Related topics

`XPRSloadglobal`, `XPRSloadlp`, `XPRSloadqglobal`, `XPRSloadqp`.

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);
RESTORE [probname]
```

Arguments

`prob` The current problem.

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

Example 1 (Library)

```
XPRSrestore(prob,"")
```

Example 2 (Console)

```
2: child[1]=child::file[1]

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`). The file *problem_name*`.sol` is also required and, if recommencing optimization in a global search, the files *problem_name*`.glb` and *problem_name*`.ctp` are required too. Note that `.svf` files are particular to the release of the Optimizer used to create them. They can only be read using the same release Optimizer as used to create them.

2. (*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.

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`	Integer and number of RHS coefficients whose sensitivity ranges are sought.
`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`.

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);
SAVE
```

Argument

`prob` The current problem.

Example 1 (Library)

```
XPRSsave(prob);
```

Example 2 (Console)

```
SAVE
```

Further information

The data structures are written to the file *problem_name*`.svf`. Optimization may recommence from the same point when the data structures are restored by a call to `XPRSrestore` (`RESTORE`). Under such circumstances, the file *problem_name*`.sol` and, if a branch and bound search is in progress, the global files *problem_name*`.glb` and *problem_name*`.ctp` are also required. These files will be present after execution of `XPRSsave` (`SAVE`), but will be modified by subsequent optimization, so no optimization calls may be made after the call to `XPRSsave` (`SAVE`). Note that the `.svf` files created are particular to the release of the Optimizer used to create them. They can only be read using the same release Optimizer as used to create them.

Related topics

`XPRSrestore` (`RESTORE`).

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);
XPRSminim(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
MINIM
```

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, XPRSsetcbsepnode.

Synopsis

```
int XPRS_CC XPRSsetbranchbounds(XPRSprob prob, const 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:

```
XPRSsetcbsepnode(prob,nodeSep,void);
```

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, XPRSsetcbestimate, XPRSsetcbsepnode, XPRSstorebounds, 5.4.

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, XPRSsetcbsepnode.

Synopsis

```
int XPRS_CC XPRSsetbranchcuts(XPRSprob prob, int ncuts, 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.

Related topics

XPRSgetcpcutlist, XPRSloadcuts, XPRSsetcbestimate, XPRSsetcbsepnode, XPRSstorecuts, 5.4.

XPRSsetcbbarlog

Purpose

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

Synopsis

```
int XPRS_CC XPRSsetcbbarlog (XPRSprob prob, int (XPRS_CC *fubl)(XPRSprob
      my_prob, void *my_object), void *object);
```

Arguments

`prob` The current problem.

`fubl` The callback function itself. This takes two arguments, `my_prob` and `my_object`, and has an integer return value. If the value returned by `fubl` 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, `fubl`.

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

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

Example

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

```
XPRSsetcbbarlog(prob,barLog,NULL);
XPRSmaxim(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

XPRSsetcbgloballog, XPRSsetcblplog, XPRSsetcbmessage.

XPRSsetcbchgbranch

Purpose

Declares a branching variable callback function, called every time a new branching variable is set or selected during the MIP search.

Synopsis

```
int XPRS_CC XPRSsetcbchgbranch(XPRSprob prob, void (XPRS_CC
      *fucb)(XPRSprob my_prob, void *my_object, int *entity, int *up,
      double *estdeg), void *object);
```

Arguments

`prob` The current problem.

`fucb` 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, `fucb`.

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

`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` The estimated degradation at the node.

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

Example

The following example demonstrates use of the branching rule to branch on the most violated integer of binary during the global search:

```
typedef struct {
  double* soln;
  char* type;
  double tol;
  int cols;
} solutionData;
...
solutionData nodeData;
...
XPRSminim(prob,"");
XPRSsetintcontrol(prob, XPRS_MIPLOG, 3);
XPRSsetintcontrol(prob, XPRS_CUTSTRATEGY, 0);

/* setup data */
XPRSgetintattrib(prob, XPRS_COLS, &(nodeData.cols));
XPRSgetdblcontrol(prob, XPRS_MATRIXTOL, &(nodeData.tol));
nodeData.soln =
  (double*) malloc(sizeof(double)*nodeData.cols);
nodeData.type =
  (char*) malloc(sizeof(char)*nodeData.cols);
XPRSgetcoltype(prob, nodeData.type, 0, nodeData.cols-1);

XPRSsetcbchgbranch(prob, varSelection, &nodeData);
XPRSglobal(prob);
```

The callback function might resemble:

```
void XPRS_CC varSelection(XPRSprob prob, void* vdata,
  int *iColumn, int *iUp, double *dEstimate)
{
  double dDist, dUpDist, dDownDist, dGreatestDist=0;
  int iCol;

  solutionData *nodeData = (solutionData*) vdata;
  XPRSgetpresolvesol(prob, (*nodeData).soln, NULL, NULL,
                     NULL);
  for(iCol=0;iCol<(*nodeData).cols;iCol++)
  if((*nodeData).type[iCol]=='I' ||
     (*nodeData).type[iCol]=='B')
  {
    dUpDist=ceil((*nodeData).soln[iCol]) -
              (*nodeData).soln[iCol];
    dDownDist = (*nodeData).soln[iCol] -
                  floor((*nodeData).soln[iCol]);
    dDist = (dUpDist>dDownDist)?dUpDist:dDownDist;
    if(dDownDist > (*nodeData).tol &&
       dUpDist > (*nodeData).tol)
      if( dDist > dGreatestDist )
      {
        *iColumn = iCol;
        dGreatestDist = dDist;
      }
  }
}
```

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

XPRSsetcbchgnode, XPRSsetcboptnode, XPRSsetcbinfnode, XPRSsetcbintsol, XPRSsetcbnodecutoff, XPRSsetcbprenode.

XPRSsetcbchgnode

Purpose

Declares a node selection callback function. This is called every time the code backtracks to select a new node during the MIP search.

Synopsis

```
int XPRS_CC XPRSsetcbchgnode(XPRSprob prob, void (XPRS_CC *fusn)(XPRSprob
       my_prob, void *my_object, int *nodnum), void *object);
```

Arguments

`prob` The current problem.

`fusn` 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, `fusn`.

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

`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, `fusn`.

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:

```
XPRSminim(prob,"");
XPRSsetintcontrol(prob,XPRS_MIPLOG,3);
XPRSsetintcontrol(prob,XPRS_NODESELECTION,2);
XPRSsetcbchgnode(prob,nodeSelection,NULL);
XPRSglobal(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` on the Xpress-MP CD-ROM.

Related topics

`XPRSsetcboptnode`, `XPRSsetcbinfnode`, `XPRSsetcbintsol`, `XPRSsetcbnodecutoff`, `XPRSsetcbchgbranch`, `XPRSsetcbprenode`.

XPRSsetcbcutlog

Purpose

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

Synopsis

```
int XPRS_CC XPRSsetcbcutlog(XPRSprob prob, int (XPRS_CC *fucl)(XPRSprob
      my_prob, void *my_object), void *object);
```

Arguments

`prob` The current problem.

`fucl` 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, `fucl`.

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

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

Related topics

`XPRSsetcbcutmgr`, `XPRSsetcbfreecutmgr`, `XPRSsetcbinitcutmgr`.

XPRSsetcbcutmgr

Purpose

Declares a user-defined cut manager routine, called at each node of the Branch and Bound search.

Synopsis

```
int XPRS_CC XPRSsetcbcutmgr(XPRSprob prob, int (XPRS_CC *fcme)(XPRSprob
      my_prob, void *my_object), void *object);
```

Arguments

`prob` The current problem

`fcme` 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, `fcme`.

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

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

Related controls

Integer

EXTRAELEMS	Number of extra matrix elements to be allowed for.
EXTRAROWS	Number of extra rows to be allowed for.

Further information

1. For maximum efficiency, the space-allocating controls EXTRAROWS, EXTRAELEMS should be specified by the user if their values are known. If this is not done, resizing will occur automatically, but more space may be allocated than the user requires.
2. The cut manager routine will be called repeatedly at each node until it returns a value of `0`. The sub-problem is automatically optimized if any cuts are added or deleted.
3. The Xpress-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

XPRSsetcbcutlog, XPRSsetcbfreecutmgr, XPRSsetcbinitcutmgr.

XPRSsetcbdestroymt

Purpose

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

Synopsis

```
int XPRS_CC XPRSsetcbdestroymt(XPRSprob prob, void (XPRS_CC
      *fmt)(XPRSprob my_prob, void *my_object), void *object);
```

Arguments

`prob`	The current thread problem.
`fmt`	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 to the callback function.
`object`	A user-defined object to be passed to the callback function.

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

`XPRSsetcbmipthread`.

XPRSsetcbestimate

Purpose

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.

Synopsis

```
int XPRS_CC XPRSsetcbestimate(XPRSprob prob, int (XPRS_CC *fbe)(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);
```

Arguments

`prob` The current problem.

`fbe` 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, `fbe`.

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

`iglsel` Selected user global entity (must be non-negative).

`iprio` Priority of selected user global entity.

`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.

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

Related topics

`XPRSsetbranchcuts`, `XPRSsetcbsepnode`, `XPRSstorecuts`.

XPRSsetcbfreecutmgr

Purpose

Declares a user termination routine to be called at the end of the Branch and Bound search.

Synopsis

```
int XPRS_CC XPRSsetcbfreecutmgr(XPRSprob prob, int (XPRS_CC
      *fcms)(XPRSprob my_prob, void *my_object), void *object);
```

Arguments

`prob` The current problem.

`fcms` 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, `fcms`.

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

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

Related topics

`XPRSsetcbcutlog`, `XPRSsetcbcutmgr`, `XPRSsetcbinitcutmgr`.

XPRSsetcbgloballog

Purpose

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

Synopsis

```
int XPRS_CC XPRSsetcbgloballog(XPRSprob prob, int (XPRS_CC
      *fugl)(XPRSprob my_prob, void *my_object), void *object);
```

Arguments

`prob` The current problem.

`fugl` 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, `fugl`.

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

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

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);
XPRSsetcbgloballog(prob, globalLog, NULL);
XPRSminim(prob,"g");
```

The callback function may resemble:

```
XPRS_CC int globalLog(XPRSprob prob, void *data)
{
  int nodes, nodedepth;

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

  return 0;
}
```

See the example `depthfirst.c` on the Xpress-MP CD-ROM.

Further information

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

Related topics

`XPRSsetcbbarlog`, `XPRSsetcblplog`, `XPRSsetcbmessage`.

XPRSsetcbinfnode

Purpose

Declares a user optimal node callback function, called after the current node has been found to be infeasible during the Branch and Bound search.

Synopsis

```
int XPRS_CC XPRSsetcbinfnode(XPRSprob prob, void (XPRS_CC *fuin)(XPRSprob
      my_prob, void *my_object), void *object);
```

Arguments

`prob` The current problem

`fuin` 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, `fuin`.

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

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

Related controls

Integer

`NODESELECTION` Node selection control.

Example

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

```
XPRSsetintcontrol(prob,XPRS_NODESELECTION,2);
XPRSsetcbinfnode(prob,nodeInfeasible,NULL);
XPRSmaxim(prob,"g");
```

The callback function may resemble:

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

See the example `depthfirst.c` on the Xpress-MP CD-ROM.

Related topics

`XPRSsetcbchgnode`, `XPRSsetcboptnode`, `XPRSsetcbintsol`, `XPRSsetcbnodecutoff`, `XPRSsetcbchgbranch`, `XPRSsetcbprenode`.

XPRSsetcbinitcutmgr

Purpose

Declares a user callback routine, called to initialize the cut manager.

Synopsis

```
int XPRS_CC XPRSsetcbinitcutmgr(XPRSprob prob, int (XPRS_CC
      *fcmi)(XPRSprob my_prob, void *my_object), void *object);
```

Arguments

`prob` The current problem.

`fcmi` 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, `fcmi`.

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

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

Related topics

`XPRSsetcbcutlog`, `XPRSsetcbcutmgr`, `XPRSsetcbfreecutmgr`.

XPRSsetcbintsol

Purpose

Declares a user integer solution callback function, called every time an integer solution is found by heuristics or during the Branch and Bound search.

Synopsis

```
int XPRS_CC XPRSsetcbintsol(XPRSprob prob, void (XPRS_CC *fuis)(XPRSprob
      my_prob, void *my_object), void *object);
```

Arguments

`prob` The current problem.

`fuis` 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, `fuis`.

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

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

Example

The following example prints integer solutions as they are discovered in the global search, without using the solution file:

```
XPRSsetcbintsol(prob,printsol,NULL);
XPRSmaxim(prob,"g");
```

The callback function might resemble:

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

  XPRSgetintattrib(my_prob, XPRS_COLS, &cols);
  XPRSgetdblattrib(my_prob, XPRS_LPOBJVAL, &objval);
  x = malloc(cols * sizeof(int));
  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]);
}
```

Further information

This callback is useful if the user wants to retrieve the integer solution when it is found.

Related topics

XPRSsetcbchgnode, XPRSsetcboptnode, XPRSsetcbinfnode, XPRSsetcbnodecutoff, XPRSsetcbchgbranch, XPRSsetcbprenode.

XPRSsetcblplog

Purpose

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

Synopsis

```
int XPRS_CC XPRSsetcblplog(XPRSprob prob, int (XPRS_CC *fuil)(XPRSprob
      my_prob, void *my_object), void *object);
```

Arguments

prob The current problem.

fuil 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, fuil.

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

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

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);
XPRSsetcblplog(prob,lpLog,NULL);
XPRSreadprob(prob,"problem","");
XPRSminim(prob,"");
```

The callback function may resemble:

```
int XPRS_CC lpLog(XPRSprob my_prob, void *my_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

XPRSsetcbbarlog, XPRSsetcbgloballog, XPRSsetcbmessage.

XPRSsetcbmessage

Purpose

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

Synopsis

```
int XPRS_CC XPRSsetcbmessage(XPRSprob prob, void (XPRS_CC *fop)(XPRSprob
      my_prob, void *my_object, const char *msg, int len, int msgtype),
      void *object);
```

Arguments

`prob` The current problem.

`fop` 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 to the callback function.

`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.

Related controls

Integer

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

Example

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

```
XPRSsetcbmessage(prob,Message,NULL);
```

The callback function might resemble:

```
void XPRS_CC Message(XPRSprob my_prob, void* my_object,
                     const char *msg, int len, int msgtype)
{
  switch(msgtype)
  {
    case 4:  /* error */
    case 3:  /* warning */
    case 2:  /* dialogue */
    case 1:  /* information */
             printf("%s\n", msg);
             break;
    default: /* exiting - buffers need flushing */
             fflush(stdout);
             break;
  }
}
```

Further information

1. Any screen output is disabled automatically whenever a user output callback is set.
2. Screen output is never produced by the Optimizer DLL running under Windows. The only way to enable screen output from the Optimizer DLL is to define this callback function and use it to print the messages to the screen (`stdout`).
3. 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`.
4. Visual Basic, users must use the alternative function `XPRSetcbmessageVB` to define the callback; this is required because of the different way VB handles strings.

Related topics

`XPRSsetcbbarlog`, `XPRSsetcbgloballog`, `XPRSsetcblplog`, `XPRSsetlogfile`.

XPRSsetcbmipthread

Purpose

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

Synopsis

```
int XPRS_CC XPRSsetcbmipthread(XPRSprob prob, void (XPRS_CC
      *fmt)(XPRSprob my_prob, void *my_object, XPRSprob thread_prob),
      void *object);
```

Arguments

`prob` The current problem.

`fmt` 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.

Related controls

Integer

`MIPTHREADS` Number of MIP threads to create.

Example

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

```
XPRSsetcbmipthread(prob,mipthread,NULL);

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

}
```

Related topics

`XPRSsetcbdestroymt`.

XPRSsetcbnodecutoff

Purpose

Declares a user node cutoff callback function, called every time a node is cut off as a result of an improved integer solution being found during the Branch and Bound search.

Synopsis

```
int XPRS_CC XPRSsetcbnodecutoff(XPRSprob prob, void (XPRS_CC
      *fucn)(XPRSprob my_prob, void *my_object, int nodnum), void
      *object);
```

Arguments

`prob` The current problem.

`fucn` 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 node is cut off as the result of an improved integer solution being found.

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

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

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

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

Example

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

```
XPRSsetcbnodecutoff(prob,Cutoff,NULL);
XPRSmaxim(prob,"g");
```

The callback function might resemble:

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

See the example `depthfirst.c` on the Xpress-MP CD-ROM.

Further information

This function allows the user to keep track of the eligible nodes.

Related topics

XPRSsetcbchgnode, XPRSsetcboptnode, XPRSsetcbinfnode, XPRSsetcbintsol, XPRSsetcbchgbranch, XPRSsetcbprenode.

XPRSsetcboptnode

Purpose

Declares an optimal node callback function, called after an optimal solution for the current node has been found during the Branch and Bound search.

Synopsis

```
int XPRS_CC XPRSsetcboptnode(XPRSprob prob, void (XPRS_CC *fuon)(XPRSprob
      my_prob, void *my_object, int *feas), void *object);
```

Arguments

`prob` The current problem.

`fuon` 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, `fuon`.

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

`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, `fuon`.

Example

The following prints an optimal solution once found:

```
XPRSsetcboptnode(prob,nodeOptimal,NULL);
XPRSmaxim(prob,"g");
```

The callback function might resemble:

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

  XPRSgetintattrib(prob, XPRS_NODES, &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` on the Xpress-MP CD-ROM.

Further information

The cost of optimizing the node will be avoided if the node is declared to be infeasible from this callback function.

Related topics

`XPRSsetcbchgnode`, `XPRSsetcbinfnode`, `XPRSsetcbintsol`, `XPRSsetcbnodecutoff`, `XPRSsetcbchgbranch`, `XPRSsetcbprenode`.

XPRSsetcbprenode

Purpose

Declares a preprocess node callback function, called before the node has been optimized, so the solution at the node will not be available.

Synopsis

```
int XPRS_CC XPRSsetcbprenode(XPRSprob prob, void (XPRS_CC *fupn)(XPRSprob
      my_prob, void *my_object, int *nodinfeas), void *object);
```

Arguments

`prob` The current problem.

`fupn` 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 re-optimized and the node may be made infeasible by setting `*nodinfeas` to `1`.

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

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

`nodinfeas` Whether or not the node is infeasible. If the node is found to be infeasible during preprocessing, this is set to `1` indicating that the node should not be re-optimized.

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

Example

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

```
XPRSsetcbprenode(prob, preNode, NULL);
XPRSminim(prob,"g");
```

The callback function might resemble:

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

Related topics

XPRSsetcbchgnode, XPRSsetcbinfnode, XPRSsetcbintsol, XPRSsetcbnodecutoff, XPRSsetcboptnode.

XPRSsetcbsepnode

Purpose

Declares a separate callback function to specify how to separate a node in the Branch and Bound tree using a global object. A node can be separated by applying either cuts or bounds to each node. These are stored in the cut pool.

Synopsis

```
int XPRS_CC XPRSsetcbsepnode(XPRSprob prob, int (XPRS_CC *fse)(XPRSprob
      my_prob, void *my_object, int ibr, int iglsel, int ifup, double
      curval), void *object);
```

Arguments

`prob` The current problem.

`fse` 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, `fse`.

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

`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, `fse` .

Example

This example minimizes a problem, before defining a user separate callback function for the global search:

```
XPRSminim(prob,"");
XPRSsetcbsepnode(prob,nodeSep,NULL);
XPRSglobal(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);
  }
  XPRSsetbranchcuts(my_prob, 1, &index);
  return 0;
}
```

Further information

1. The user separate routine is called `nbr` times where `nbr` is returned by the estimate callback function, `XPRSsetcbestimate`. This allows multi-way branching to be performed.
2. The bounds and/or cuts to be applied at a node must be specified in the user separate routine by calling `XPRSsetbranchcuts`.

Related topics

`XPRSsetbranchcuts`, `XPRSsetcbestimate`, `XPRSstorecuts`.

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 7, or from the list in the `xprs.h` header file.
`dsval`	Value to which the control parameter is to be set.

Example

The following sets the double control `DEGRADEFACTOR` to `1.0`:

```
XPRSsetdblcontrol(prob, XPRS_DEGRADEFACTOR, 1.0);
```

Related topics

`XPRSgetdblcontrol`, `XPRSsetintcontrol`, `XPRSsetstrcontrol`.

XPRSsetdefaultcontrol

Purpose

Sets a single control to its default value.

Synopsis

```
int XPRS_CC XPRSsetdefaultcontrol(XPRSprob prob, int ipar);
```

Arguments

`prob` The current problem.

`ipar` Integer, double or string control parameter whose default value is to be set. A full list of all controls may be found in 7, or from the list in the `xprs.h` header file.

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);
XPRSmaxim(prob, "g");
XPRSwriteprtsol(prob);
XPRSsetdefaultcontrol(prob, XPRS_PRESOLVE);
XPRSmaxim(prob, "g");
```

Related topics

XPRSsetdefaults, XPRSsetintcontrol, XPRSsetdblcontrol, XPRSsetstrcontrol.

XPRSsetdefaults

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);
```

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);
XPRSmaxim(prob, "g");
XPRSwriteprtsol(prob);
XPRSsetdefaults(prob);
XPRSreadprob(prob);
XPRSmaxim(prob, "g");
```

Related topics

XPRSsetdefaultcontrol, XPRSsetintcontrol, XPRSsetdblcontrol, XPRSsetstrcontrol.

XPRSsetintcontrol

Purpose

Sets the value of a given integer control parameter.

Synopsis

```
int XPRS_CC XPRSsetintcontrol(XPRSprob prob, int ipar, int 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 7, 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);
XPRSmaxim(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

Purpose

This directs all Optimizer output to a log file.

Synopsis

```
int XPRS_CC XPRSsetlogfile(XPRSprob prob, const char *filename);
```

Arguments

`prob` The current problem.

`filename` The name of the file to which all output will be directed. If set to `NULL`, redirection of the output will stop and all screen output will be turned back on (except for DLL users where screen output is always turned off).

Example

The following directs output to the file `logfile.log`:

```
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 `XPRSsetcbmessage`, which will be called every time a line of text is output. Defining a user output callback will turn all screen output off. To discard all output messages the `OUTPUTLOG` integer control can be set to `0`.

Related topics

`XPRSsetcbmessage`.

Purpose

Manages suppression of messages.

Synopsis

```
int XPRS_CC XPRSsetmessagestatus(XPRSprob prob, int errcode, int status);
SETMESSAGESTATUS errcode [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 9 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 1 (Library)

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);
XPRSminim(prob,"");
```

Example 2 (Console)

An equivalent set of commands for the Console user may look like:

```
SETMESSAGESTATUS 91 0
MINIM
```

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`.

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 200 characters containing the problem name.

Example 1 (Library)

The following sets the current problem name to `jo`:

```
char sProblem[]="jo";
...
XPRSsetprobname(prob,sProblem);
```

Example 2 (Console)

```
READPROB bob
MINIM
SETPROBNAME jim
READPROB
```

The above will read the problem `bob` and then read the problem `jim`.

Related topics

`XPRSreadprob` (`READPROB`).

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 7, 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`.

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
MAXIM -g
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:

```
XPRSsetcbsepnode(prob,nodeSep,void);
```

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`, `XPRSsetcbestimate`, `XPRSsetcbsepnode`.

XPRSstorecuts

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[]);
```

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 positive 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]-1` containing the column indices in the cuts.

`dmatval` Double array of length `mstart[ncuts]-1` containing the matrix values for the cuts.

Related controls

Double

`MATRIXTOL` Zero 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, XPRSsetcbestimate, XPRSsetcbsepnode, 5.4.

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 200 characters containing the file name from which the basis is to be written. If omitted, the default *problem_name* is used with a `.bss` extension.

`flags` Flags to pass to `XPRSwritebasis` (`WRITEBASIS`):

- `i` output the internal presolved basis.
- `t` output a compact advanced form of the basis.

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", "");
XPRSmaxim(prob, "");
XPRSwritebasis(prob, "", "");
XPRSglobal(prob);
```

This reads in a matrix file, maximizes the LP, saves the basis and performs a global search. Saving an IP basis is generally not very useful, so in the above example only the LP basis is saved.

Example 2 (Console)

An equivalent set of commands to the above for console users would be:

```
READPROB
MAXIM
WRITEBASIS
GLOBAL
```

Further information

1. The `c` 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`).

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 200 characters containing the file name to which the solution is to be written. If omitted, the default *problem_name* is used with a `.sol` extension.

`flags` 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", "");
XPRSmaxim(prob, "g");
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
MAXIM -g
WRITEBINSOL
```

Related topics

`XPRSgetlpsol`, `XPRSgetmipsol`, `XPRSreadbinsol` (`READBINSOL`), `XPRSwritesol` (`WRITESOL`), `XPRSwriteprtsol` (`WRITEPRTSOL`).

Purpose

Writes the current solution to the binary OMNI format file SOLFILE, as recorded in the solution file *problem_name* .sol. Optionally the current matrix may also be written. All information is appended to this file.

Synopsis

```
int XPRS_CC XPRSwriteomni(XPRSprob prob);
WRITEOMNI
```

Argument

prob The current problem.

Related controls

Integer

OMNIFORMAT Whether to include matrix coefficients in OMNI output.

SOLUTIONFILE The binary solution file must be enabled.

String

OMNIDATANAME Data for OMNI data name field.

Example 1 (Library)

```
XPRSreadprob(prob, "bob", "");
XPRSminim(prob, "");
XPRSsetintcontrol(prob, XPRS_OMNIFORMAT, 1);
XPRSwriteomni(prob);
XPRSreadprob(prob, "jo", "");
XPRSsetstrcontrol(prob, XPRS_OMNIDATANAME, "Jo");
XPRSminim(prob, "");
XPRSwriteomni(prob);
```

Will put two solutions on SOLFILE, the first taking the contents of the data name field from the problem name in bob.mat, the second having its data name field "Jo".

Example 2 (Console)

An equivalent set of commands at the Console would be the following:

```
READPROB bob
MINIM
OMNIFORMAT = 1
WRITEOMNI
READPROB jo
OMNIDATANAME = "Jo"
MINIM
WRITEOMNI
```

Further information

1. To conform with OMNI standards the output is appended to the file called SOLFILE. If this file already exists the output of XPRSwriteomni (WRITEOMNI) will be appended to SOLFILE, otherwise SOLFILE is created in the current working directory. The file is written in OMNI Standard Format, as defined in the document "OMNI Standard Interface to Various Optimizers", available from Haverly Systems. Note that the status codes generated are "OPTM", "INFE", "FEAS" and "UNBD". Some early versions of the OMNI format specification used other codes.

2. The MHDR, MROW and MCOL data are only generated if the control OMNIFORMAT is set nonzero before the call to XPRSwriteomni (WRITEOMNI), for example:

```
READPROB
MINIM
WRITEBASIS
OMNIFORMAT = 1
WRITEOMNI
QUIT
```

3. The data name field in the OMNI format (bytes 8 to 15, starting at 0) is filled from the control OMNIDATANAME, which should be set prior to a call of XPRSwriteomni (WRITEOMNI).

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 200 characters to contain the file name to which the problem is to be written. If omitted, the default *problem_name* is used with a `.mat` 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:

- `p` full precision of numerical values;
- `o` one element per line;
- `n` scaled;
- `s` scrambled vector names;
- `l` output in LP format;
- `x` output MPS file in hexadecimal format.

Example 1 (Library)

The following example outputs the current problem in full precision, LP format with scrambled vector names to the file *problem_name*`.lp`.

```
XPRSwriteprob(prob, "", "lps");
```

Example 2 (Console)

```
WRITEPROB -p C:myprob
```

This instructs the Optimizer to write an MPS matrix to the file `myprob.mat` on the `C:` drive in full precision.

Further information

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`).

The hexadecimal format is useful for saving the exact internal precision of the matrix.

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`).

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` Zero 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", "");
XPRSmaxim(prob, "");
XPRSrange(prob);
XPRSwriteprttange(prob);
```

Example 2 (Console)

An equivalent set of commands for the Console user would be:

```
READPROB
MAXIM
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.5.

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 200 characters containing the file name to which the solution is to be written. If omitted, the default *problem_name* will be used. The extension `.prt` will be appended.

`flags` 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` Zero 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", "");
XPRSmaxim(prob, "");
XPRSwriteprtsol(prob, "", "");
```

Example 2 (Console)

```
READPROB
MAXIM
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.

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 200 characters containing the file name to which the solution is to be written. If omitted, the default *problem_name* will be used. The extensions .hdr and .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 Zero 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", "");
XPRSminim(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.5.

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 200 characters containing the file name to which the solution is to be written. If omitted, the default *problem_name* will be used. The extensions `.hdr` and `.asc` will be appended.

`flags` Flags to control which optional fields are output:

- `s` sequence number;
- `n` name;
- `t` type;
- `b` basis status;
- `a` activity;
- `c` cost (columns), slack (rows);
- `l` lower bound;
- `u` upper bound;
- `d` dj (column; reduced costs), dual value (rows; shadow prices);
- `r` right hand side (rows).

If no flags are specified, all fields are output.
Additional flags:

- `e` outputs every MIP or goal programming solution saved;
- `p` outputs in full precision;
- `q` only outputs vectors with nonzero optimum value;
- `x` output the current LP solution instead of the MIP solution.

Related controls

Double

`OUTPUTTOL` Zero 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", "");
XPRSminim(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.

3. If `KEEPMIPSOL` has been used to store a number of MIP or goal programming solutions, the `e` flag can be used to output solution information for every solution kept. The best solution found is still output to *problem_name*`.hdr` and *problem_name*`.asc`. Any other solutions are output to the header files *problem_name*`.hd0`, *problem_name*`.hd1`,... and ASCII solution files *problem_name*`.as0`, *problem_name*`.as1`,....

Related topics

`XPRSgetlpsol`, `XPRSgetmipsol`, `XPRSwriterange` (`WRITERANGE`), `XPRSwriteprtsol` (`WRITEPRTSOL`).

Chapter 7

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.

7.1 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 Xpress-MP 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 Xpress-MP 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);
```

AUTOPERTURB

Description Simplex: This indicates whether automatic perturbation is performed. If this is set to 1, the problem will be perturbed by the amount PERTURB 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 XPRSmaxim (MAXIM), XPRSminim (MINIM).

BACKTRACK

Description Branch and Bound: This determines how the next node in the tree search is selected for processing.

Type Integer

Values

1	If MIPTARGET is not set, choose the node with the best estimate. If MIPTARGET is set (by the user or by the global search previously finding an integer solution), the choice is based on the Forrest-Hirst-Tomlin Criterion, which takes into account the best current integer solution and seeks a new node which represents a large potential improvement.
2	Always choose the node with the best estimated solution.
3	Always choose the node with the best bound on the solution.

Default value 3

Affects routines 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 XPRSmaxim (MAXIM), XPRSminim (MINIM).

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 1.0E-08

Affects routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

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 1.0E-08

Affects routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

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 XPRSmaxim (MAXIM), XPRSminim (MINIM).

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 200

Affects routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

BARORDER

Description Newton barrier: This specifies the ordering algorithm for the Cholesky factorization, used to preserve the sparsity of the factorized matrix.

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 XPRSmaxim (MAXIM), XPRSminim (MINIM).

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 XPRSmaxim (MAXIM), XPRSminim (MINIM).

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 1.0E-08

Affects routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

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-10

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 XPRSmaxim (MAXIM), XPRSminim (MINIM).

BARTHREADS

Description Newton barrier: The number of threads implemented to run the algorithm. This is usually set to the number of processors when running Parallel Xpress-MP on a single multi-processor machine.

Type Integer

Default value 1

Note The value of BARTHREADS depends on the user's authorization. If it is set to a value higher than that specified by the licence, then it will be reset by the Optimizer immediately prior to optimization. Obtaining its value after the optimization will give an indication of how many processors were actually used.

Affects routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

BIGM

Description The infeasibility penalty used if the "Big M" method is implemented.

Type Double

Default value Dependent on the matrix characteristics.

Affects routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

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 `XPRSmaxim` (`MAXIM`), `XPRSminim` (`MINIM`).

BRANCHCHOICE

Description Once a global entity has been selected for branching, this control determines whether the branch with the minimum or the maximum estimate is solved first.

Type Integer

Values
- `0` Minimum estimate branch first.
- `1` Maximum estimate branch first.

Default value `0`

Affects routines `XPRSglobal` (`GLOBAL`).

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 `10`

Affects routines `XPRSglobal` (`GLOBAL`).

CACHESIZE

Description Newton barrier: L2 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 -1

Note Specifying the correct L2 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 512 kB is assumed.

For multi-processor machines, use the cache size of a single CPU.

Specify the size in kB: for example, 0.5 MB means 512 kB and a value of 512 should be used when setting `CACHESIZE`.

Affects routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

CHOLESKYALG

Description Newton barrier: type of Cholesky factorization used.

Type Integer

Values

0	Pull Cholesky;
1	Push Cholesky.

Default value 0

Affects routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

CHOLESKYTOL

Description Newton barrier: The zero 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 XPRSmaxim (MAXIM), XPRSminim (MINIM).

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 XPRSglobal (GLOBAL).

CPUTIME

Description Which time to be used in reporting solution times.

Type Integer

Values	0	If elapsed time is to be used.
	1	If CPU time is to be used.
Default value	1	
Affects routines	XPRSmaxim (MAXIM), XPRSminim (MINIM), XPRSglobal (GLOBAL).	

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	XPRSmaxim (MAXIM), XPRSminim (MINIM).	

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	Crossover to a basic solution.
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	XPRSmaxim (MAXIM), XPRSminim (MINIM).	

CSTYLE

Description Convention 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.

Affects routines XPRSglobal (GLOBAL).

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 XPRSglobal (GLOBAL).

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 XPRSglobal (GLOBAL).

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

Affects routines XPRSmaxim (MAXIM), XPRSminim (MINIM), XPRSglobal (GLOBAL).

DEGRADEFACTOR

Description Branch and Bound: Factor to multiply estimated degradations associated with an unexplored node in the tree. The estimated degradation is the amount by which the objective function is expected to worsen in an integer solution that may be obtained through exploring a given node.

Type Double

Default value 1.0

Affects routines XPRSglobal (GLOBAL).

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 XPRSmaxim (MAXIM), XPRSminim (MINIM).

DUALGRADIENT

Description Simplex: This specifies the dual simplex pricing method.

Type Integer

Values

`-1`	Determine automatically.
`0`	Devex.
`1`	Steepest edge.

Default value `-1`

Affects routines `XPRSmaxim` (`MAXIM`), `XPRSminim` (`MINIM`).

See also `PRICINGALG`.

DUALIZE

Description Newton Barrier: This specifies whether the Newton Barrier method should solve the dual problems.

Type Integer

Values

`-1`	Determine automatically.
`0`	Solve the primal problem.
`1`	Solve the dual problem.

Default value `-1`

Affects routines `XPRSmaxim` (`MAXIM`), `XPRSminim` (`MINIM`).

ELIMTOL

Description The Markowitz tolerance for the elimination phase of the presolve.

Type Double

Default value `1.0E-03`

Affects routines `XPRSmaxim` (`MAXIM`), `XPRSminim` (`MINIM`).

ETATOL

Description Zero 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 `XPRSmaxim` (`MAXIM`), `XPRSminim` (`MINIM`), `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. The space allowed for cut coefficients is equal to the number of extra matrix elements remaining after rows and columns have been added but before the global optimization starts. EXTRAELEMS is set automatically by the optimizer when the matrix is first input to allow space for cuts, but if you add rows or columns, this automatic setting will not be updated. So if you wish cuts, either automatic cuts or user cuts, to be added to the matrix and you are adding rows or columns, EXTRAELEMS must be set before the matrix is first input, to allow space both for the cuts and any extra rows or columns that you wish to add.

Type Integer

Default value Hardware/platform dependent.

Affects routines XPRSreadprob (READPROB), XPRSloadglobal, XPRSloadlp, XPRSloadqglobal, XPRSloadqp, XPRSsetcbcutmgr.

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 The initial number of extra elements to allow for in the presolve.

Type Integer

Default value Hardware/platform dependent.

Note The space required to store extra presolve elements is allocated dynamically, so it is not necessary to set this control.

Affects routines XPRSreadprob (READPROB), XPRSloadglobal, XPRSloadlp, XPRSloadqglobal, XPRSloadqp.

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. The space allowed for cuts is equal to the number of extra rows remaining after rows have been added but before the global optimization starts. EXTRAROWS is set automatically by the optimizer when the matrix is first input to allow space for cuts, but if you add rows, this automatic setting will not be updated. So if you wish cuts, either automatic cuts or user cuts, to be added to the matrix and you are adding rows, EXTRAROWS must be set before the matrix is first input, to allow space both for the cuts and any extra rows that you wish to add.

Type Integer

Default value Dependent on the matrix characteristics.

Affects routines XPRSreadprob (READPROB), XPRSloadglobal, XPRSloadlp, XPRSloadqglobal, XPRSloadqp, XPRSsetcbcutmgr.

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

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 0

Affects routines XPRSglobal (GLOBAL).

FEASTOL

Description This is the zero tolerance on right hand side values, bounds and range values, i.e. the bounds of basic variables. If one of these is less than or equal to FEASTOL in absolute value, it is treated as zero.

Type Double

Default value 1.0E-06

Affects routines XPRSmaxim (MAXIM), XPRSminim (MINIM), XPRSgetinfeas.

GOMCUTS

Description Branch and Bound: The number of rounds of Gomory cuts at the top node. These can always be generated if the current node does not yield an integral solution. However, Gomory cuts are not usually as effective as lifted cover inequalities in reducing the size of the feasible region.

Type Integer

Default value -1 — determined automatically.

Affects routines XPRSglobal (GLOBAL).

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.

Type Integer

Default value -1

Affects routines XPRSglobal (GLOBAL).

HEURDIVESPEEDUP

Description Branch and Bound: Changes the emphasis of the diving heuristic from solution quality to diving speed.

Type Integer

Values

-1	Automatic selection.
0-3	Emphasis bias from emphasis on quality (0) to emphasis on speed (3).

Default value -1

Affects routines XPRSglobal (GLOBAL).

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.
1-10	Available pre-set strategies for rounding infeasible global entities and reoptimizing during the heuristic dive.

Default value -1

Affects routines XPRSglobal (GLOBAL).

See also HEURSTRATEGY.

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 XPRSglobal (GLOBAL).

HEURMAXSOL

Description Branch and Bound: This specifies the maximum number of heuristic solutions that will be found in the tree search.

Type Integer

Default value −1

Affects routines XPRSglobal (GLOBAL).

HEURNODES

Description Branch and Bound: This specifies the maximum number of nodes at which heuristics are used in the tree search.

Type Integer

Default value −1

Affects routines XPRSglobal (GLOBAL).

HEURSEARCHFREQ

Description Branch and Bound: This specifies how often the local search heuristic should be run in the tree.

Type Integer

Values

0	Disabled in the tree.
n>0	Number of nodes between each run.

Default value 500

Affects routines XPRSglobal (GLOBAL).

See also HEURSTRATEGY.

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 XPRSglobal (GLOBAL).

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 XPRSmaxim (MAXIM), XPRSminim (MINIM).

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 XPRSmaxim (MAXIM), XPRSminim (MINIM).

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.

Default value 1

Note This gets reset to the default value after optimization has started.

Affects routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

KEEPMIPSOL

Description Branch and Bound: The number of integer solutions to store. During a global search, any number of integer solutions may be found, which may or may not represent optimal solutions. See XPRSglobal (GLOBAL). Goal Programming: The number of partial solutions to store in the pre-emptive goal programming. Pre-emptive goal programming solves a sequence of problems giving a sequence of partial solutions. See XPRSgoal (GOAL). The stored solutions can only be accessed in a limited way - see the notes below. An alternative method of storing multiple integer solutions from the Optimizer library (or Mosel) is to use an integer solution callback function to retrieve and store them - see XPRSsetcbintsol for details.

Type Integer

Values

1	store the best/final solution only.
n=2-11	store the *n* best/most recent solutions.

Default value 1

Note Multiple solutions are kept by storing them on separate binary solution files. The best/final solution is stored on the default solution file, *probname*.sol, as usual. The next best solution (if found) is stored on a solution file named *probname*.so0, the next best on *probname*.so1, and so on up to *probname*.so9, or until there are no further solutions. The only function able to access the multiple solution files is XPRSwritesol (WRITESOL) - refer to its "e" flag. It is also possible to use other functions that access the solution from the solution file by renaming a particular stored solution file, e.g., *probname*.so3, to the default solution file *probname*.sol before using the function. A list of functions that may be used to access the solution from the solution file may be found under the SOLUTIONFILE control.

Affects routines XPRSglobal (GLOBAL), XPRSgoal (GOAL).

See also XPRSwritesol (WRITESOL) with its e flag; XPRSsetcbintsol.

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 XPRSmaxim (MAXIM), XPRSminim (MINIM).

LINELENGTH

Description Maximum line length for LP files.

Type Integer

Default value 512

Affects routines XPRSreadprob (READPROB)

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 XPRSglobal.

LNPITERLIMIT

Description Number of iterations to perform in improving each lift-and-project cut.

Type Integer

Default value 10

Note By setting the number to zero a Gomory cut will be created instead.

Affects routines XPRSglobal (GLOBAL).

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 XPRSmaxim (MAXIM), XPRSminim (MINIM).

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

Affects routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

See also A.8.

MARKOWITZTOL

Description The Markowitz tolerance used for the factorization of the basis matrix.

Type Double

Default value 0.01

Affects routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

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.

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 `XPRSmaxim` (`MAXIM`), `XPRSminim` (`MINIM`), `XPRSglobal` (`GLOBAL`).

MAXIIS

Description This controls the number of Irreducible Infeasible Sets to be found using the `XPRSiis` (`IIS`) function.

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`

Affects routines `XPRSiis` (`IIS`), `XPRSgetiis`.

MAXMIPSOL

Description Branch and Bound: This specifies a limit on the number of integer solutions to be found by the Optimizer before it pauses and asks whether or not to continue. 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`

Affects routines `XPRSglobal` (`GLOBAL`).

MAXNODE

Description Branch and Bound: The maximum number of nodes that will be explored before the Optimizer pauses and asks whether or not to continue.

Type Integer

Default value 100000000

Affects routines XPRSglobal (GLOBAL).

MAXPAGELINES

Description Number of lines between page breaks in printable output.

Type Integer

Default value 23

Affects routines XPRSwriteprtsol (WRITEPRTSOL), XPRSwriteprtrange (WRITEPRTRANGE).

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 XPRSglobal (GLOBAL), XPRSmaxim (MAXIM), XPRSminim (MINIM).

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. It is set automatically after an LP Optimizer command, unless it was previously set by the user. The cutoff may be updated automatically whenever a MIP solution is found 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 XPRSglobal (GLOBAL), XPRSmaxim (MAXIM), XPRSminim (MINIM).

See also MIPRELCUTOFF, MIPADDCUTOFF.

MIPABSSTOP

Description Branch and Bound: The absolute tolerance determining whether the global search will continue or not. It will terminate if
$|$MIPOBJVAL − BESTBOUND$| \leq$ 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 XPRSglobal (GLOBAL).

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 cutoff. Once an integer solution has been found whose objective function is equal to or better than MIPABSCUTOFF, 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 MIPABSCUTOFF 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 XPRSglobal (GLOBAL).

See also MIPRELCUTOFF, MIPABSSTOP, MIPABSCUTOFF.

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 XPRSglobal (GLOBAL).

See also A.9.

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	Logical preprocessing will be performed at each node. This is performed on binary variables, often resulting in fixing their values based on the constraints. This greatly simplifies the problem and may even determine optimality or infeasibility of the node before the simplex method commences.
2	Probing of binary variables is performed at the top node. This sets certain binary variables and then deduces effects on other binary variables occurring in the same constraints.

Default value Dependent on the matrix characteristics.

Note If the user has not set MIPPRESOLVE then its value is determined automatically after presolve (in the XPRSmaxim (MAXIM), XPRSminim (MINIM) call) according to the properties of the matrix.

Affects routines XPRSglobal (GLOBAL).

See also 5.2, PRESOLVE, PRESOLVEOPS.

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 MIPABSCUTOFF. 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 XPRSglobal (GLOBAL).

See also MIPABSCUTOFF, MIPADDCUTOFF, MIPRELSTOP.

MIPRELSTOP

Description Branch and Bound: This determines whether or not the global search will terminate. Essentially it will stop if:
$|\text{MIPOBJVAL} - \text{BESTBOUND}| \leq \text{MIPRELSTOP} \times \text{BESTBOUND}$
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

Affects routines XPRSglobal (GLOBAL).

See also MIPABSSTOP, MIPRELCUTOFF.

MIPTARGET

Description Branch and Bound: The target object function for the global search (only used by certain node selection criteria). This is set automatically after an LP optimization routine, unless it was previously set by the user.

Type Double

Default value 1.0E+40

Affects routines XPRSglobal (GLOBAL).

See also BACKTRACK.

MIPTHREADS

Description Branch and Bound: The number of threads implemented to run the parallel MIP code. This is usually set to the number of processors when running Parallel Xpress-MP on a single multi-processor machine.

Type Integer

Default value 0

Note The value of MIPTHREADS depends on the user's authorization. If it is set to a value higher than that specified by the licence, then it will be reset by the Optimizer immediately prior to optimization. Obtaining its value after the optimization will give an indication of how many processors were actually used.

Affects routines XPRSmaxim (MAXIM), XPRSminim (MINIM), XPRSglobal (GLOBAL).

See also SHAREMATRIX.

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 XPRSglobal (GLOBAL).

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 `1`

Affects routines `XPRSreadprob` (`READPROB`).

MPSERRIGNORE

Description Number of errors to ignore whilst reading an MPS file.

Type Integer

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`

Affects routines `XPRSalter` (`ALTER`), `XPRSreadbasis` (`READBASIS`), `XPRSreadprob` (`READPROB`).

MPSNAMELENGTH

Description Maximum length of MPS names in characters. If reset, this must be before any problem is input. Internally it is rounded up to the smallest multiple of 8. MPS names are right padded with blanks.

Type Integer

Default value 8

Note MPSNAMELENGTH must not be set to more than 64 characters.

Affects routines XPRSaddnames, 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 `XPRSsetcbchgbranchXPRSsetcbchgnode`, `XPRSsetcboptnode`, `XPRSsetcbinfnode`, `XPRSsetcbintsol`, `XPRSsetcbnodecutoff`, `XPRSsetcbprenode`.

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 `XPRSglobal` (`GLOBAL`).

OMNIDATANAME

Description Data for OMNI data name field. As with all string controls, this is of length 64 characters plus a null terminator, `\0`.

Type String

Default value 64 blanks

Affects routines `XPRSwriteomni` (`WRITEOMNI`).

OMNIFORMAT

Description Whether to include matrix coefficients in XPRSwriteomni (WRITEOMNI output.

Type Integer

Values

0	Matrix coefficients not to be included in output.
1	Include coefficients in output (new style).
2	Include coefficients in output (old style - prior to 1999).

Default value 0

Affects routines XPRSwriteomni (WRITEOMNI).

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, XPRSmaxim (MAXIM), XPRSminim (MINIM).

OUTPUTLOG

Description This controls the level of output produced by the Optimizer during optimization. The possible options are to print all messages or to disable printing altogether. Output is sent to the screen (stdout) by default, but may be intercepted by a user function using the user output callback; see XPRSsetcbmessage. However, under Windows, no output from the Optimizer DLL is sent to the screen. The user must define a callback function and print messages to the screen them self if they wish output to be displayed.

Type Integer

Values

0	Turn all output off.
1	Print messages.

Default value 1

Affects routines XPRSsetcbmessage, 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 XPRSmaxim (MAXIM), XPRSminim (MINIM).

Type Double

Default value Dependent on the matrix characteristics.

Affects routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

PERTURB

Description The factor by which the problem will be perturbed prior to optimization if the control AUTOPERTURB has been set to 1. A value of 0.0 results in an automatically determined perturbation value.

Type Double

Default value 0.0 — perturbation value is determined automatically by default.

Affects routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

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 XPRSmaxim (MAXIM), XPRSminim (MINIM), XPRSpivot.

PPFACTOR

Description The partial pricing candidate list sizing parameter.

Type Double

Default value `1.0`

Affects routines `XPRSmaxim` (`MAXIM`), `XPRSminim` (`MINIM`).

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.

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.

Affects routines `XPRSmaxim` (`MAXIM`), `XPRSminim` (`MINIM`).

See also 5.2, `PRESOLVEOPS`.

PRESOLVEOPS

Description This 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.
14	Linearly dependant row removal.
15	No integer variable and SOS detection.

Default value 511 (bits 0 — 8 incl. are set)

Affects routines XPRSmaxim (MAXIM), XPRSminim (MINIM), XPRSpresolvecut.

See also 5.2, 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.

Default value 0

Affects routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

See also DUALGRADIENT.

PROBNAME

Description The current problem name

Type String

Affects routines XPRSgetprobname, XPRSsetprobname

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 XPRSglobal (GLOBAL), XPRSreaddirs (READDIRS).

REFACTOR

Description Indicates whether the optimization should restart using the current representation of the factorization in memory.

Type Integer

Values

0	Do not refactor on reoptimizing.
1	Refactor on reoptimizing.

Default value 0 — for the global search. 1 — for reoptimizing.

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 XPRSglobal (GLOBAL), XPRSmaxim (MAXIM), XPRSminim (MINIM).

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 XPRSmaxim (MAXIM), XPRSminim (MINIM), XPRSpivot.

SBBEST

Description Number of infeasible global entities on which to perform strong branching.

Type Integer

Values

`-1`	determined automatically.
`0`	disable strong branching.
`n>0`	perform strong branching on *n* entities at each node.

Default value `-1`

Note The two recommended settings for this control are `-1` (automatic) and `0` (disabled). Positive values normally result in an excessive amount of strong branching (and thus time) at each node. Setting `SBBEST=0` will turn strong branch off.

Affects routines `XPRSglobal` (`GLOBAL`).

See also `SBITERLIMIT`, `SBSELECT`.

SBEFFORT

Description Adjusts the overall amount of effort when using strong branching to select an infeasible global entitiy 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 `XPRSglobal` (`GLOBAL`).

See also `SBBEST`, `SBITERLIMIT`, `SBSELECT`.

SBESTIMATE

Description Choose the estimate to be used to select candiates for strong branching.

Type Integer

Values

`-1`	Automatically determined.
`1-4`	Different variants of local pseudo costs.

Default value `-1`

Affects routines `XPRSglobal` (`GLOBAL`).

See also `SBBEST`, `SBITERLIMIT`, `SBSELECT`.

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 XPRSglobal (GLOBAL).

See also SBBEST, SBSELECT.

SBSELECT

Description The size of the candidate list of global entities for strong branching.

Type Integer

Default value −1 — determined automatically.

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 XPRSglobal (GLOBAL).

SBTHREADS

Description The number of parallel threads to use for strong branching.

Type Integer

Values

0	Parallel strong branching disabled.
>1	Number of parallel threads to start.

Default value 0

Affects routines XPRSglobal (GLOBAL).

See also SBBEST.

SCALING

Description This 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	Objective function scaling.

Default value 163

Affects routines XPRSloadglobal, XPRSloadlp, XPRSloadqglobal, XPRSloadqp, XPRSreadprob (READPROB), XPRSscale (SCALE).

See also 3.4.

SHAREMATRIX

Description This determines whether the matrix is shared by the parallel MIP.

Type Integer

Values

0	Matrix is not shared.
1	Matrix is shared.

Default value 0

Note If the matrix is shared then cuts will not be generated in the tree. However, sharing the matrix will save memory.

Affects routines XPRSglobal (GLOBAL).

See also MIPTHREADS

SOLUTIONFILE

Description The SOLUTIONFILE control is deprecated and will be removed in version 18. Binary solution files are no longer created automatically and it is now necessary to explicitly create a binary solution file with the XPRSwritebinsol (WRITEBINSOL) command. The XPRSwriteprtsol (WRITEPRTSOL), XPRSwritesol (WRITESOL) and PRINTSOL commands will write or print reports for the solution in memory. To write a report on a solution in binary solution file, the solution must first be loaded with the XPRSreadbinsol (READBINSOL) command. The XPRSgetbasis, XPRSgetinfeas and XPRSgetlpsol commands all obtain information from the solution in memory. To obtain information on a solution in binary solution file, the solution must first be loaded with the XPRSreadbinsol (READBINSOL) command.

The XPRSgetsol function is deprecated and will be removed in version 18. Users should use the XPRSgetlpsol function to get the current LP solution and XPRSgetmipsol function to get the last found MIP solution. The XPRSgetlpsol will read the current LP solution from memory and the XPRSgetmipsol will read the current MIP solution from memory.

Type Integer

Values

−1	The binary file is not created. The XPRSgetsol function will return the LP solution until the MIP search starts at which point the last found MIP solution will be returned.
0	The binary file is not created. The XPRSgetsol function will return the current solution in memory.
1	The binary solution file will be created and used to store the final LP solution, or, if a MIP solution has been found, the best known MIP solution. The solution is written to the file by the XPRSmaxim (MAXIM), XPRSminim (MINIM) and XPRSglobal (GLOBAL) functions. The binary solution file will remain after the Optimizer has finished.

Default value −1

Note The solution stored in memory is overwritten by certain operations, including infeasibility analysis (XPRSiis). If the solution is required from the solution file, it should be obtained before any call to XPRSgetiis.

Affects routines XPRSfixglobal (FIXGLOBAL), XPRSgetbasis, XPRSgetiis, XPRSgetinfeas, XPRSgetsol, XPRSglobal (GLOBAL), XPRSmaxim (MAXIM), XPRSminim (MINIM), XPRSwriteomni (WRITEOMNI), XPRSwriteprtsol (WRITEPRTSOL), XPRSwritesol (WRITESOL).

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 less than 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).

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 XPRSmaxim (MAXIM), XPRSminim (MINIM).

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.

Type Integer

Default value 1

Affects routines XPRSglobal (GLOBAL).

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.

Type Integer

Default value 1

Affects routines XPRSglobal (GLOBAL).

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 minimum estimate. The variable estimates are also combined to calculate the overall estimate of the node, which, depending on the BACKTRACK setting, may be used to choose between outstanding nodes.

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.

Default value −1

Affects routines XPRSglobal (GLOBAL).

VERSION

Description The Optimizer version number, e.g. `1301` meaning release 13.01.

Type Integer

Default value Software version dependent

Chapter 8
Problem Attributes

During the optimization process, various properties of the problem being solved are stored and made available to users of the Xpress-MP 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.

8.1 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 Xpress-MP 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, &amp;lpobjval);

printf(&quot;The objective value is %2.1f\n&quot;, lpobjval);
```

ACTIVENODES

Description Number of outstanding nodes.

Type Integer

Set by routines XPRSdelnode, XPRSglobal, XPRSinitglobal.

BARAASIZE

Description Number of nonzeros in AA^T.

Type Integer

Set by routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

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 XPRSmaxim (MAXIM), XPRSminim (MINIM).

BARDENSECOL

Description Number of dense columns found in the matrix.

Type Integer

Set by routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

BARDUALINF

Description Sum of the dual infeasibilities for the Newton barrier algorithm.

Type Double

Set by routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

BARDUALOBJ

Description Dual objective value calculated by the Newton barrier algorithm.

Type Double

Set by routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

BARITER

Description Number of Newton barrier iterations.

Type Integer

Set by routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

BARLSIZE

Description Number of nonzeros in L resulting from the Cholesky factorization.

Type Integer

Set by routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

BARPRIMALINF

Description Sum of the primal infeasibilities for the Newton barrier algorithm.

Type Double

Set by routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

BARPRIMALOBJ

Description Primal objective value calculated by the Newton barrier algorithm.

Type Double

Set by routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

BARSTOP

Description Convergence criterion for the Newton barrier algorithm.

Type Double

Set by routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

BESTBOUND

Description Value of the best bound determined so far by the global search.

Type Double

Set by routines XPRSglobal.

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 XPRSglobal.

BRANCHVAR

Description The branching variable at a node of the Branch and Bound tree.

Type Integer

Set by routines XPRSglobal.

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 5.2.

Set by routines XPRSloadglobal, XPRSloadlp, XPRSloadqglobal, XPRSloadqp, XPRSmaxim (MAXIM), XPRSminim (MINIM), XPRSreadprob.

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 5.2.

Set by routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

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 5.2.

Set by routines XPRSloadglobal, XPRSloadlp, XPRSloadqglobal, XPRSloadqp, XPRSmaxim (MAXIM), XPRSminim (MINIM), 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 9.2 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 XPRSsetcbmessage.

Type Integer

Set by routines Any.

NUMIIS

Description Number of IISs found.

Type Integer

Set by routines XPRSiis.

LPOBJVAL

Description Value of the objective function of the last LP solved.

Type Double

Set by routines XPRSmaxim (MAXIM), XPRSminim (MINIM), XPRSglobal.

See also MIPOBJVAL, OBJRHS.

LPSTATUS

Description LP solution status.

Type Integer

Values

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).

Note The possible return values are defined as constants in the Optimizer C header file and VB .bas file.

Set by routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

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.

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 5.2.

Set by routines XPRSaddcols, XPRSchgcoltype, XPRSdelcols, XPRSloadglobal, XPRSloadqglobal, XPRSreadprob.

See also SETS.

MIPINFEAS

Description Number of integer infeasibilities at the current node.

Type Integer

Set by routines XPRSglobal.

See also PRIMALINFEAS.

MIPOBJVAL

Description Objective function value of the best integer solution found.

Type Double

Set by routines XPRSglobal.

See also LPOBJVAL.

MIPSOLNODE

Description Node at which the last integer feasible solution was found.

Type Integer

Set by routines XPRSglobal.

MIPSOLS

Description Number of integer solutions that have been found.

Type Integer

Set by routines XPRSglobal.

MIPSTATUS

Description Global (MIP) solution status.

Type Integer

Values

XPRS_MIP_LP_OPTIMAL LP has been optimized. Once the MIP optimization proper has begun, only the following four status codes will be returned.

XPRS_MIP_INFEAS Global search complete - no integer solution found.

XPRS_MIP_SOLUTION Global search incomplete - an integer solution has been found.

XPRS_MIP_OPTIMAL Global search complete - integer solution found.

XPRS_MIP_NO_SOL_FOUND Global search incomplete - no integer solution found.

XPRS_MIP_LP_NOT_OPTIMAL LP has not been optimized.

XPRS_MIP_NOT_LOADED Problem has not been loaded.

Note If the XPRS_MIP_LP_OPTIMAL status code is returned, it implies that the optimization halted during or directly after the LP optimization - for instance, if the LP relaxation is infeasible or unbounded. In this case please check the value of LP solution status using LPSTATUS.
The possible return values are defined as constants in the Optimizer C header file and VB .bas file. Refer to one of those files for the value of the return codes listed above.

Set by routines XPRSglobal, XPRSloadglobal, XPRSloadqglobal, XPRSmaxim (MAXIM), XPRSminim (MINIM), XPRSreadprob.

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 XPRSglobal.

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 XPRSglobal, XPRSinitglobal.

NODES

Description Number of nodes solved so far in the global search. The node numbers start at `1` for the first (top) node in the Branch and Bound tree. Nodes are numbered consecutively.

Type Integer

Set by routines XPRSglobal, XPRSinitglobal.

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 5.2. 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.

Set by routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

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.

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 XPRSglobal, XPRSinitglobal.

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 XPRSmaxim (MAXIM), XPRSminim (MINIM).

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 5.2.

Set by routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

See also SUMPRIMALINF, DUALINFEAS, MIPINFEAS.

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 5.2.

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 5.2.

Set by routines XPRSaddrows, XPRSdelrows, XPRSloadglobal, XPRSloadlp, XPRSloadqglobal, XPRSloadlp, XPRSmaxim (MAXIM), XPRSminim (MINIM), XPRSreadprob.

SIMPLEXITER

Description Number of simplex iterations performed.

Type Integer

Set by routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

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 5.2.

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 5.2.

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.

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 5.2.

Set by routines XPRSmaxim (MAXIM), XPRSminim (MINIM).

See also PRIMALINFEAS.

Chapter 9

Return Codes and Error Messages

9.1 Optimizer Return Codes

The following table shows the possible return codes from the subroutine library functions:

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.

9.2 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.

16 ***Null column <col>.***
Column <col> has a zero coefficient in all the rows in the COLUMN section. Please remove empty columns to avoid this warning message.

17 ***More than one RHS not permitted - subsequent ones ignored.***
Only one RHS is allowed for a constraint. The first RHS value that has been read will be kept and any subsequent ones ignored.

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: buy IP Optimizer from Dash Associates.***

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 6.

41 ***Global error - contact Dash.***

Internal error. Please contact your local support office.

45 ***Failure to open global file - aborting. (Perhaps disk is full).***

Xpress-MP 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.

60 ***Out of memory - program aborted.***

The Optimizer cannot allocate any more memory. Please increase your virtual page space or physical memory.

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>.***

Xpress-MP 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>.***

Xpress-MP 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>.***

Xpress-MP has failed to read the file <file>. Please make sure that the file exists and that it has not been corrupted.

68 ***<num> errors in sizing parameter <par> - fatal.***

During initialization Xpress-MP has encountered <num> errors. Please contact your local support office.

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>.***

Xpress-MP 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.

79 ***File error.***

The Optimizer has encountered a file error. Please make sure that there is adequate disk space and that the volume is not corrupt.

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 Xpress-MP 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.

90 ***Not enough memory for Devex pricing: PRICINGALG has been set to -1.***

The Optimizer required more memory to perform Devex pricing. Please increase your virtual paging space, physical memory or do not use the Devex pricing algorithm.

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>.***

Xpress-MP 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.

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>***

Xpress-MP 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.

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.

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.

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 ***Too many files open for reading: <file>.***
The Optimizer cannot read from file <file> because there are too many files already open. Please close some files and try again.

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.

172 ***Scaling too bad to continue.***
Scaling is too bad for recursion entries. Please try scaling your matrix.

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.

180 ***No change to this SSV 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.

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.

246 ***Wrong release of binary files. Release <rel1> detected.***
Release <rel2> required. The binary file *.BIF created with release <rel1> cannot be read. Please try using release <rel2> or the previous one <rel2>-1 to create the binary file and try again.

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.

255 ***Not enough space to presolve matrix. Increase <par> before XPRSreadprob (READPROB).***

The is not enough space to presolve the matrix. Please increase parameter <par> before XPRSreadprob (READPROB) or turn the presolve procedure off by setting PRESOLVE to 0.

256 ***Simplex Optimizer only: buy barrier Optimizer from Dash Associates.***
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.

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. Re-input matrix.***
The problem in memory is the presolved one. Please reload the problem and try the operation again.

277 ***This version is not authorized to run as a DLL.***
The Optimizer subroutine library is not authorized to run as a DLL. Please contact your local sales office to upgrade your authorization if you wish to tun the Optimizer as a DLL.

278 ***No purchase authorization found.***
Please check that the xpress.pwd file can be found. The Optimizer will try to use the xpress.pwd file located in its directory and if this fails it will look for it in the directory pointed to by the environment variable XPRESS. If you needed a dongle to run the Optimizer please check that it has been inserted. Contact you local support office if the problem persists.

279 ***Xpress-MP has not been initialized.***
The Optimizer could not be initialized successfully. Please initialize it before attempting any operation and try again.

285 ***Cut pool is full.***
The Optimizer has run out of space to store cuts.

286 ***Cut pool is full.***
The Optimizer has run our of space to store cuts.

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.

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.

320 ***This version is not authorized to run under Windows NT.***
The Optimizer is not authorized to run under Windows NT. Please contact your local sales office to upgrade your authorization if you wish to run it on this platform.

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: buy QP Optimizer from Dash.***
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.

349 ***Release <rel1> of binary files used with version <rel2>.***
The binary file created with release <rel1> of the software is being used with release <rel2>.

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.

364 ***Problem has too many QP matrix elements. Please increase M_Q.***
Problem cannot be read because there are too many quadratic elements. Please increase M_Q and try again.

366 ***Problems with Quadratic terms can only be solved with the barrier.***
An attempt has been made to solve a quadratic problem using an algorithm other than the barrier. Please use XPRSmaxim (MAXIM), XPRSminim (MINIM) with the b flag to invoke the barrier solver.

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.

387 ***Parallel code not initialized - continuing in serial mode.***
Parallel mode cannot be used. Please check that the number of slaves is greater than 0 and that HPPVM is installed correctly.

388 ***Slave number <num> has failed.***
Slave number <num> has failed and its tasks will be reallocated to the remaining slaves.

389 ***Incorrect type of dongle, or security violation on slave <num>.***
Xpress-MP could not be initialized on slave <num> and its tasks will be reallocated to the remaining slaves. See the section on initialization in Chapter 1 for details. Contact your local support office if the problem persists.

390 ***Slave number <num> has failed - insufficient memory.***
Process on slave <num> has been aborted because there is not enough memory. Please increase your virtual page space or physical memory and try again. The tasks of the failing slaves will be reallocated to the remaining slaves.

391 ***All slaves have failed - continuing in serial mode.***
All slaves have failed and the optimization will continue in serial mode.

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.

396 ***Insufficient memory on slave <num>.***
Process on slave <num> cannot be started because there is insufficient memory on this slave. Please increase your virtual page space or physical memory and try again. The tasks of the failing slave will be reallocated to the remaining slaves.

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 6 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.

408 ***Too long a row or column name passed to <function>.***

Name number <num> is too long. There has been a error in one the arguments of function <function>. The row or column name corresponding to element <num> of the array is too long.

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 6 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.

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.

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.

428 ***Increase EXTRAMIPENTS before input.***
There are not enough spare global entities to complete function <function> successfully. Please increase EXTRAMIPENTS before input and try again.

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.

473 ***Row classification not available.***

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.

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 staring with an 'f' and not know to Xpress-MP has been found in column <col> at line <line> of the LP file. The word will be read into Xpress-MP 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 Xpress-MP 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 Xpress-MP. 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 Xpress-MP. 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 Xpress-MP. 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 ***Xpress-MP cannot reach line <line.col>.***
The reading of the LP file has failed due to an internal problem. Please contact your local support office immediately.

524 ***Constraints could not be read into Xpress-MP. Error found at <line.col>.***
The reading of the LP constraints has failed due to an internal problem. Please contact your local support office immediately.

525 ***Bounds could not be set into Xpress-MP. Error found at <line.col>.***
The setting of the LP bounds has failed due to an internal problem. Please contact your local support office immediately.

526 ***LP problem could not be loaded into Xpress-MP. Error found at <line.col>.***
The reading of the LP file has failed due to an internal problem. Please contact your local support office immediately.

527 ***Copying of rows unsuccessful.***
The copying of the LP rows has failed due to an internal problem. Please contact your local support office immediately.

528 ***Copying of columns unsuccessful.***
The copying of the LP columns has failed due to an internal problem. Please contact your local support office immediately.

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.

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
.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 byXPRSgoal (GOAL).	ASCII
.grp	Goal programming output file, output byXPRSgoal (GOAL).	ASCII
.hdr	Solution header file, output by XPRSwritesol (WRITESOL) and XPRSwriterange (WRITERANGE).	ASCII
.iis	IIS output file, output by XPRSiis (IIS).	ASCII
.lp	LP format matrix file, input by XPRSreadprob (READPROB).	ASCII
.mat	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
.svf	Optimizer state file, output by XPRSsave (SAVE), input by XPRSrestore (RESTORE).	Binary

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 Xpress-MP 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	introduces a quadratic objective function;
SETS	introduces SOS definitions;
RHS	introduces the right hand side(s);
RANGES	introduces the row ranges;
BOUNDS	introduces the bounds;
ENDATA	signals the end of the matrix.

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 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.2.6 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.2.7 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.2.8 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.2.9 ENDATA section

Format:	Cols 1-6
	`ENDATA`

is the last record of the file.

A.3 LP File Format

Matrices can be represented in text files using either the MPS file format (`.mat` or `.mps` 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 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 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. The maximum length for any name is 64. The maximum length of any line of input is 512. Lines can be continued if required. No line continuation character is needed when expressions are required to span multiple 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_MAXINT`].
`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 integer`	`s.i.`	A list of variable names of semi-integer variables.
`partial integer`	`p.i.`	A list of variable names of partial integer variables.

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
```

Generally objective functions are defined using many terms and since the maximum length of any line of file input is 512 characters the objective function definitions are typically always broken with line continuations. No line continuation character is required and lines may be broken for continuation wherever you may use white space.

Note that the sense of objective is defined only after the problem is loaded and when it is optimized by the Xpress-Optimizer when the user calls either the `minim` or `maxim` operations. The objective function can be named in the same way as for constraints (see later) although this name is ignored internally by the 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 <=.

Generally, constraints are defined using many terms and since the maximum length of any line of file input is 512 characters the constraint definitions are typically always broken with line continuations. No line continuation character is required and lines may be broken for continuation wherever you may use white space.

A.3.9 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 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 the interval defined by the default bounds is [0, `XPRS_PLUSINFINITY`] while for variables declared in the `integers` and `generals` section (see later) the relaxation interval defined by the default bounds is [0, 1] and [0, `XPRS_MAXINT`], respectively. Note that the constants `XPRS_PLUSINFINITY` and `XPRS_MAXINT` are defined in the Xpress-Optimizer header files in your Xpress-Optimizer libraries package.

If a single bound is defined for a variable the 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 a constraint 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 $+\infty$ (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.3.10 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_MAXINT. 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 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 Xpress-Optimizer is licensed for Mix Integer Programming.

A.3.11 Semi-continuous and semi-integer

The semi-continuous and semi integer 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 integer) 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 integer) section are a list of white space or carriage return delimited entries that are either (i) a variable name or (ii) a variable name-number pair. 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 integer
x7 >= 3
x8
x9 >= 5
```

Note that you can not use the <= token in place of 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 Xpress-Optimizer is licensed for Mix Integer Programming.

A.3.12 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 Xpress-Optimizer is licensed for Mix Integer Programming.

A.3.13 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 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 Xpress-Optimizer is licensed for Mix Integer Programming.

A.3.14 Quadratic programming problems

Quadratic programming problems (QPs) with quadratic objective functions are defined using a special format within the objective function description. Note that quadratic terms may appear

only in the objective function and not in any constraints. The algebraic coefficients of the function x'Qx appearing in the objective for QP problems are specified inside square brackets []. Division by two of the QP objective component in the square bracket is implicit. 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.

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 ]
```

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 Xpress-Optimizer is licensed for Quadratic Programming.

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:

O	optimal;
N	infeasible;
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 Xpress-MP 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.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)

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 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.7.1 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.

A.7.2 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.7.3 Changing Constraint Types

The direction of a constraint may be altered. The row name is given first, followed by an action of `**NT`*x*, 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.

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).

A.8 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.9 The Global Log

During the Branch and Bound tree search (see XPRSglobal (GLOBAL)), a summary log of nine columns of information is printed every n nodes, where $-n$ is the value of MIPLOG. These columns consist of:

Node	A sequential node number.
BestSoln	The value of the best integer feasible solution found.
BestBound	A bound on the value of the best integer feasible solution that can be found.
Sols	The number of integer feasible solutions that have been found.
Active	The number of active 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 infeasabilities at the current node.
Time	The time taken.

This log is also printed when an integer feasible solution is found. Stars (`*`) printed on both sides of the log indicate a solution has been found. Pluses (`+`) printed on both sides of the log indicate a heuristic solution has been found.

If `MIPLOG` is set to `3`, a more detailed log of eight columns of information is printed for each node in the tree search:

Branch	A sequential node number.
Parent	The node number of the parent of this node.
Solution	The optimum value of the LP relaxation at the node.
Entity	If it is necessary to continue the search from this node, then this global entity will be separated upon.
Value / Bound	The current value of the entity chosen above for separation. A `U` or an `L` follows: If the letter is `U` (resp. `L`) then a new upper (lower) bound will first be applied to the entity. Thus the entity will be forced down (up) on the first branch from this node.
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 the Branch and Parent (and possibly Solution) are displayed. If the LP relaxation is infeasible, only the Branch and Parent appear. 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 `0` or `1`, no log is displayed and status messages only are displayed at the end of the search. The LP iteration log is suppressed, but messages from the LP Optimizer may be seen if major numerical difficulties are encountered.

Index

A

B

C