FICO® Xpress Optimization

Last update 20 March 2020

version 36.01

FICO® Xpress Nonlinear

Manual

FICO Decisions

Xpress Nonlinear

Deliverable Version: A

Last Revised: 20 March 2020

Version version 36.01

Contents

I. Overview

CHAPTER 1

Introduction

This part of the manual is intended to provide a general description of the facilities available for modeling with Xpress NonLinear. It is not an exhaustive list of possibilities, and it does not go into very great depth on some of the more advanced topics. All the functions and formats are given in more detail in the second part of this manual and the Xpress-Mosel Reference Manual (Module mmxnlp section).

Xpress Nonlinear consists of the Xpress Optimizer to solve linear, mixed integer linear, and convex quadratic problems, Xpress-SLP which uses Successive Linear Programming to solve non-linear models, and as an plugin Knitro.

The functionalities of Xpres NonLinear extend those of the Xpress Optimizer. Almost any problem that fits into the problem types supported by the Xpress Optimizer are automatically detected and converted into the appropriate format to take advantage of the power of the optimizer's purpose written algorithms.

Xpress-SLP is in essence, is a technique which involves making a linear approximation of the original problem at a chosen point, solving the linear approximation and seeing how "far away" the solution point is from the original chosen point. If it is "sufficiently close" then the solution is said to have converged and the process stops. Otherwise, a new point is chosen, based on the solution, and a new linear approximation is made. This process repeats (iterates) until the solution converges. Although this process will find a solution which is the optimum for the linear approximation, there is no guarantee that the solution will be the optimum for the original non-linear problem (that is to say: it may not be the best possible solution to the original problem). Such a solution is called a "local optimum", because it is a better solution than any others in the immediate neighbourhood, but may not be better than one a long way away.

The problem of local optima can be thought of as being like trying to find the deepest valley in a range of mountains. You can find a valley relatively easily (just keep going downhill). However, when you reach it, you have no idea whether there is a deeper valley somewhere else, because the mountains block your view. You have found a local optimum, but you do not know whether it is a global optimum. Indeed, in general, there is no way to find the global optimum except an exhaustive search (check every valley in the mountain range).

While Xpress-SLP is most powerful for large or integer nonlinear problems, Knitro which can take advantage of using second order partial derivative information can be more beneficial for hihgly nonlinear models.

1.1 Mathematical programs

There are many specialised forms of model in mathematical programming, and if such a form can be identified, there are usually much more efficient solution techniques available. This section describes some of the major types of problem that Xpress NonLinear can identify automatically.

1.1.1 Linear programs

Linear programming (LP) involves solving problems of the form

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax \leq b \end{array}$$

and in practice this encompasses, via transformations, any problem whose objective and constraints are linear functions.

Such problems were traditionally solved with the simplex method, although recently interior point methods have come to be favoured for larger instances. Linear programs can be solved quickly, and solution techniques scale to enormous sizes of the matrix A. However, few applications are genuinely linear. It was common in the past, however, to approximate general functions by linear counterparts when LPs were the only class of problem with efficient solution techniques.

1.1.2 Convex quadratic programs

Convex quadratic programming (QP) involves solving problems of the form

$$\begin{array}{ll} \text{minimize} & c^T x + x^T Q x \\ \text{subject to} & Ax \leq b \end{array}$$

for which the matrix Q is symmetric and positive semi-definite (that is, $x^T Q x \geq 0$ for all x). This encompasses, via transformations, all problems with a positive semi-definite Q and linear constraints. Such problems can be solved efficiently by interior point methods, and also by quadratic variants of the simplex method.

1.1.3 Convex quadratically constrained quadratic programs

Convex quadratically constrained quadratic programming (QCQP) involves solving problems of the form

$$\begin{array}{ll} \text{minimize} & c^T x + x^T Q x \\ \text{subject to} & Ax \leq b \\ & q_j^T x + x^T P_j x \leq d_j, \forall j \end{array}$$

for which the matrix Q and all matrices P_j are positive semi-definite. The most efficient solution techniques are based on interior point methods.

1.1.4 Second order conic problems

Second order conic problems is a special form of a convex quadratically constrained quadratic program, where although the quadratic matrix is not positive semi-definite, the feasible range of the problem is convex, and there are specialized algorithm to solve them.

$$\begin{array}{ll} \text{minimize} & c^T x + x^T Q x \\ \text{subject to} & Ax \leq b \\ & x \text{ is in } C_j, \forall j \end{array}$$

for which the matrix C_j is a convex second order cone and Q is positive semi-definite. The standard form of a second order cone is $x^T I x \leq y * y$ where y is non-negative, or (a rotated second order cone) $x^T I x \leq y * z$ where y and z are non-negative. Many quadratic problems can be formulated as a second order convex conic problem, including any convex quadratically constrained quadratic programs. Transformation happens automatically for most convertible problems.

1.1.5 General nonlinear optimization problems

Nonlinear programming (NLP) involves solving problems of the form

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & g_j(x) \leq b, \forall j \end{array}$$

where $f(x)$ is an arbitrary function, and $g(x)$ are a set of arbitrary functions. This is the most general type of problem, and any constrained model can be realised in this form via simple transformations.

Until recently, few practical techniques existed for tackling such problems, but it is now possible to solve even large instances using Successive Linear Programming solvers (SLP) or second-order methods.

1.1.6 Mixed integer programs

Mixed-integer programming (MIP), in the most general case, involves solving problems of the form

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & g_j(x) \leq b, \forall j \\ & x_k \text{ integral} \end{array}$$

It can be combined with any of the previous problem types, giving Mixed-Integer Linear Programming (MILP), Mixed-Integer Quadratic Programming (MIQP), Mixed-Integer Quadratically Constrained Quadratic Programming (MIQCQP), Mixed-Integer Second Order Conic Problems (MISOCP) and Mixed-Integer Nonlinear Programming (MINLP). Efficient solution techniques now exist for all of these classes of problem.

1.2 Technology Overview

In real-world applications, it is vital to match the right optimization technology to your problem. The FICO Xpress libraries provide dedicated, high performance implementations of optimization technologies for the many model classes commonly appearing in practical applications. This includes solvers for linear programming (LP), mixed integer programming (MIP), convex quadratic programming (QP), and convex quadratically constrained programming (QCQP), and general nonlinear programming (NLP).

1.2.1 The Simplex Method

The simplex method is one of the most well-developed and highly studied mathematical programming tools. The solvers in the FICO Xpress Optimizer are the product of over 30 years of research, and include high quality, competitive implementations of the primal and dual simplex methods for both linear and quadratic programs. A key advantage of the simplex method is that it can very quickly reoptimize a problem after it has been modified, which is an important step in solving mixed integer programs.

1.2.2 The Logarithmic Barrier Method

The interior point method of the FICO Xpress Optimizer is a state of the art implementation, with leading performance across a variety of large models. It is capable of solving not only the largest and most difficult linear and convex quadratic programs, but also convex quadratically constrained quadratic and second order conic programs. It includes optimized versions of both infeasible logarithmic barrier methods, and also homogeneous self-dual methods.

1.2.3 Outer approximation schemes

A drawback of the barrier methods is that they are not efficiently warms-tarted. This makes these methods unattractive for solving several related problems, like the ones arising from a branch and bound search. While for linear and convex quadratic problems the simplex methods can be used, there is no immediate such alternative for convex quadratic constrained and second order methods. To bridge the gap, outer approximation cutting schemes are used, which themselves may be warm started by a barrier solution.

1.2.4 Successive Linear Programming

For general nonlinear programs which are very large, highly structured, or contain a significant linear part, the FICO Xpress Sequential Linear Programming solver (XSLP) offers exceptional performance. Successive linear programming is a first order, iterative approach for solving nonlinear models. At each iteration, a linear approximation to the original problem is solved at the current point, and the distance of the result from the the selected point is examined. When the two points are sufficiently close, the solution is said to have converged and the result is returned. This technique is thus based upon solving a sequence of linear programming problems and benefits from the advanced algorithmic and presolving techniques available for linear problems. This makes XSLP scalable, as well as efficient for large problems. In addition, the relatively simple core concepts make understanding the solution process and subsequent tuning comparatively straightforward.

1.2.5 Second Order Methods

Also integrated into the Xpress suite is Knitro from Artelys, a second-order method which is particularly suited to large-scale continuous problems containing high levels of nonlinearity. Second order methods approximate a problem by examining quadratic programs fitted to a local region. This can provide information about the curvature of the solution space to the solver, which first-order methods do not have. Advanced implementations of such methods, like Knitro, may as a result be able to produce more resilient solutions. This can be especially noticeable when the initial point is close to a local optimum.

1.2.6 Mixed Integer Solvers

The FICO Xpress MIP Solver is a highly scalable parallel branch and bound framework for all classes of mixed integer programs. It is based on a branch and bound search utilizing continuous solvers, advanced cutting planes in-tree presolving and multiple heuristics, for discovering primal solutions and tightening best bounds. The search is guided by advanced methods for selecting branching variables and estimating sub-tree sizes/efforts. Mixed integer programming forms the basis of many important applications, and the implementation in the FICO Xpress Suite has proven itself in operation for some of the world’s largest organizations. Both XSLP and Knitro are also able to solve mixed integer nonlinear problems (MINLP).

1.3 API naming convention

Xpress Nonlinear has been developed as an extension to the XPRS library building on the SLP solver technology, which is reflected in the naming convention. All XPRS API functions are used the same way as normal to build the linear part of the problem, while the API functions prefixed with XSLP are used for all nonlinear aspects, independently of how the problem is solved afterwards (convex quadratic problems by a dedicated solver or Knitro instead of SLP). Some controls have both an XPRS and an XSLP counterpart, for example "XPRS_PRESOLVE" and "XSLP_PRESOLVE". In such cases, "XSLP_PRESOLVE" refers to the nonlinear presolver (even if another solver than SLP is used to solve the problem afterwards) and "XPRS_PRESOLVE" refers to problems that are not deemed as general

nonlinear (LP, MIP or convex quadratic); in such cases, if SLP solves one of such problems as part of its iterative process, the XPRS control is respected for such sub-solves.

CHAPTER 2

The Problem

2.1 Problem Definition

The diameter of a two-dimensional shape is the greatest distance between any two of its points. For a circle, this definition corresponds to the normal meaning of "diameter". For a polygon (with straight sides), it is equivalent to the greatest distance between any two vertices.

What is the greatest area of a polygon with N sides and a diameter of 1?

2.2 Problem Formulation

This formulation is one of two described by Prieto [1]. It is easy to visualize, and has advantages in later examples. The pentagon is about the smallest model which can reasonably be used – it is non-trivial but is still just about small enough to be written out in full.

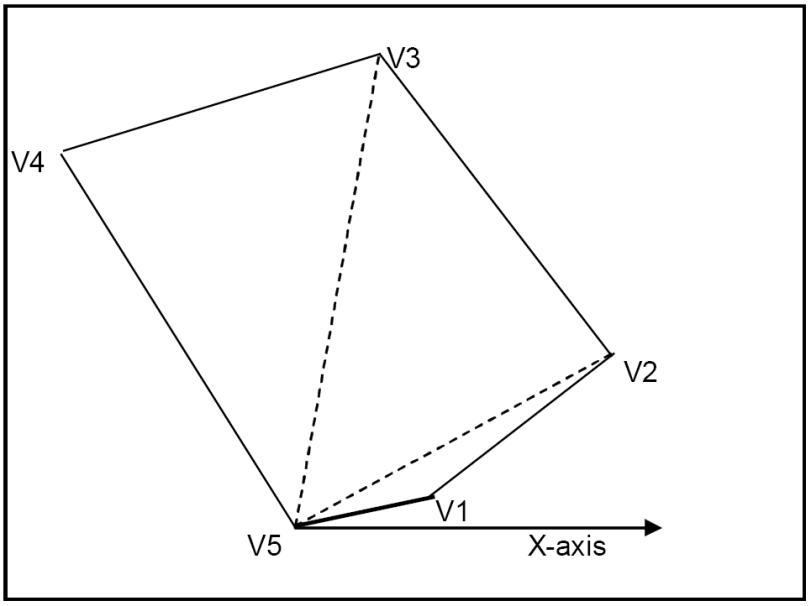

Figure 2.1: Polygon Example

One vertex (the highest-numbered, V_N) is chosen as the "base" point, and all the other vertices are measured from it, using (r, θ) coordinates – that is, the distance ("r") is measured from the vertex, and the angle or bearing of the vertex ("θ") is measured from the X-axis.

We shall use r_i and θ_i as the coordinates of vertex V_i. Then simple geometry and trigonometry gives:

- The area of the triangle $V_N V_i V_j$: area$(V_N V_i V_j) = \frac{1}{2} \cdot r_i \cdot r_j \cdot \sin(\theta_j - \theta_i)$
- The side $V_i V_j$ is given by: $(V_i V_j)^2 = r_i^2 + r_j^2 - 2 \cdot r_i \cdot r_j \cdot \cos(\theta_j - \theta_i)$
- The total area of the polygon is: $\sum_{i=2}^{N-1}$ area$(V_N V_i V_{i-1})$
- The maximum diameter of 1 requires that all the sides of all the triangles are ≤ 1 – that is:
  $r_i \leq 1$ for $i = 1, ..., N-1$
  and
  $V_i V_j \leq 1$ for $i = 1, ..., N-2, j = i+1, ..., N-1$

We have assumed in the diagram 2.1 and in the formulation that $\theta_i \leq \theta_{i+1}$ – in other words, the vertices are in order anti-clockwise. In fact, this is not just an assumption, and we need to include these constraints as well.

In the diagram, we have assumed that the first angle θ_1 is ≥ 0. This is not an additional restriction if we use the normal modeling convention that all variables are non-negative. We also assumed that the last vertex is still "above" the X-axis – that is, θ_{N-1} is $\leq 180^\circ$ (or π radians).

The requirement is therefore:

maximize	$\sum_{i=2}^{N-1}(r_i \cdot r_{i-1} \cdot \sin(\theta_i - \theta_{i-1})) * 0.5$	*(area of the polygon)*
subject to:	$r_i \leq 1$ for $i = 1, ..., N-1$	*(distances betweem V_N and other vertices)*
	$r_i^2 + r_j^2 - 2 \cdot r_i \cdot r_j \cdot \cos(\theta_j - \theta_i \leq 1$ for $i = 1, ..., N-2, j = i+1, ..., N-1$	*(distances between other pairs of vertices)*
	$\theta_1 \geq 0$	*(first bearing is non-negative)*
	$\theta_{i+1} - \theta_i \geq 0$ for $i = 1, ..., N-2$	*(bearings are in order)*
	$\theta_{N-1} \leq \pi$	*(last vertex is above X-axis)*

Reference:
(1) F.J. Prieto. *Maximum area for unit-diameter polygon of N sides, first model and second model* (Netlib AMPL programs *in* ftp://netlib.bell-labs.com/netlib/ampl/models).

CHAPTER 3

Modeling in Mosel

3.1 Basic formulation

Nonlinear capabilities in Mosel are provided by the `mmxnlp` module. Please refer to the module documentation for more details. This chapter provides a short introduction only.

The model uses the Mosel module `mmxnlp` which contains the extensions required for modeling general non-linear expressions. This automatically loads the *mmxprs* module, so there is no need to include this explicitly as well.

```
model "Polygon"
uses "mmxnlp"
```

We can design the model to work for any number of sides, so one way to do this is to set the number of sides of the polygon as a parameter.

```
parameters
  N=5
end-parameters
```

The meanings of most of these declarations will become apparent as the modeling progresses.

```
declarations
  area: nlctr
  rho: array(1..N) of mpvar
  theta: array(1..N) of mpvar
  objdef: mpvar
  D: array(1..N,1..N) of nlctr
end-declarations
```

- The distances are described as "`rho`", to distinguish them from the default names for the rows in the generated matrix (which are R1, R2, etc).
- The types `nlctr` (nonlinear constraint) are defined by the `mmxnlp` module.

```
area := sum(i in 2..N-1) (rho(i) * rho(i-1) * sin(theta(i)-theta(i-1)))*0.5
```

This uses the normal Mosel sum function to calculate the area. Notice that the formula is written in essentially the same way as normal, including the use of the `sin` function. Because the argument to the function is not a constant, Mosel will not try to evaluate the function yet; instead, it will be evaluated as part of the optimization process.

`area` is a Mosel object of type `nlctr`.

```
objdef = area
objdef is_free
```

What we really want to do is to maximize `area`. However, although Xpress NonLinear is happy in principle with a non-linear objective function, the Xpress Optimizer is not, unless it is handled in a special way. Xpress NonLinear therefore imposes the requirement that the objective function itself must be linear. This is not really a restriction, because – as in this case – it is easy to reformulate a non-linear objective function as an apparently linear one. Simply replace the function by a new `mpvar` and then maximize the value of the `mpvar`. In general, because the objective could have a positive or negative value, we make the variable free, so that it can take any value. In this example, we say:

`objdef = area`	defining the variable `objdef` to be equal to the non-linear expression `area`
`objdef is_free`	defining `objdef` to be a free variable
`maximize(objdef)`	maximizing the linear objective

This is firstly setting the standard bounds on the variables `rho` and `theta`. To reduce problems with sides of zero length, we impose a minimum of 0.1 on `rho(i)` instead of the default minimum of zero.

```
forall (i in 1..N-1) do
  rho(i) >= 0.1
  rho(i) <= 1
  setinitval(rho(i), 4*i*(N+1-i)/((N+1)^2))
  setinitval(theta(i), M_PI*i/N)
end-do
```

We also give Xpress NonLinear initial values by using the `setinitval` procedure. The first argument is the name of the variable, and the second is the initial value to be used. The initial values for `theta` are divided equally between 0 and π . The initial values for `rho` are designed to go from 0 (when *i* = 0 or *N*) to 1 (when *i* is about half way) and back.

```
forall (i in 1..N-2, j in i+1..N-1) do
  D(i,j) := rho(i)^2 + rho(j)^2 - rho(i)*rho(j)*2*cos(theta(j)-theta(i)) <lt/>= 1
end-do
```

This is creating the general constraints `D(i,j)` which constrain the other sides of the triangles to be $\leq$ 1.

These constraints could be made anonymous – that is, the assignment to an object of type `nlctr` could be omitted – but then it would not be possible to report the values.

```
forall (i in 2..N-1) do
  theta(i) >= theta(i-1) + 0.01
end-do
```

These anonymous constraints put the values of the `theta` variables in non-decreasing order. To avoid problems with triangles which have zero angles, we make each bearing at least 0.01 greater than its predecessor.

This is the boundary condition on the bearing of the final vertex.

```
theta(N-1) <= M_PI
```

3.2 Setting up and solving the problem

`loadprob(objdef)`

This procedure loads the currently-defined non-linear problem into the Xpress NonLinear optimization framework. This includes any purely linear part. Where a general constraint has a linear expression as

its left or right hand side, that linear expression will be retained as linear relationships (constant coefficients) in the matrix. Thus, for example, in the anonymous constraint defining `objdef`, the `objdef` coefficient will be identified as a linear term and will appear as a separate item in the problem.

```
maximise
```

Optimization is carried out with the `maximise` or `minimise` procedures. They can take a string parameter – for example `maxmimise("b")` – as described in the Xpress NonLinear and Xpress Optimizer reference manuals.

With the default settings of the parameters, you will see usually nothing from the optimizer. The following parameters affect what is produced:

`xnlp_verbose`	Normally set to false. If set to true, it produces standard Xpress NonLinear iteration logging.
`xprs_verbose`	Normally set to false. If set to true, then information from the optimizer will also be output.
`xslp_log`	Normally set to -1. If set to 0, limited information is output from the SLP iterations. Settings of 1 or greater produce progressively more information for each SLP iteration.
`xslp_slplog`	If xslp_log is set to 0, this determines the frequency with which SLP progress is reported. The default is 10, which means that it prints every 10 SLP iterations.

3.3 Looking at the results

Within Mosel, the values of the variables and named constraints can be obtained using the `getsol`, `getslack` and similar functions. A simple report lists just the area and the positions of the vertices:

```
writeln("Area = ", getobjval)
forall (i in 1..N-1) do
  writeln("V", i, ": r=", getsol(rho(i)), " theta=", getsol(theta(i)))
end-do
```

This produces the following result for the case N=5:

```
Area = 0.657166
V1: r=0.616416 theta=0.703301
V2: r=1 theta=1.33111
V3: r=1 theta=1.96079
V4: r=0.620439 theta=2.58648
```

3.4 Parallel evaluation of Mosel user functions

It is possible to use parallel evaluations of simple Mosel functions that return a single real value. These functions may take an arbitrary array of nlctr expressions as input. It is the modeler's responsibility to ensure that the user functions to be called in parallel are thread-safe (i.e., they do not depend upon shared resources). Assuming the name of the user function is `MyFunc`, the user function before enabling the parallel version is expected to be declared as `usefuncMosel('MyFunc')`.

In order for mmxnlp to be able to utilize parallel user function evaluations, the user function must be implemented as a public function in a Mosel package. Any initialization necessary to enable the

evaluation of the user function should be performed as part of the package initialization (which is the code in in the main body of the package).

To enable parallel evaluations, a parallel enabled version of the user function needs to be generated using the mmxnlp procedure `generateUFparallel`, which takes two arguments: the compiled package .bim name implementing the user function and the name of the user function within the package. It is good practice to use a separate Mosel model to perform this generation, keeping it separate from the main model. Multiple generated parallel user functions may be used within a single model.

The generator will produce a single Mosel file, the Mosel package `MyFunc_master`. This package also includes the worker model which will be responsible for the user function evaluations and will be resident in memory during the execution. The package also implements the parallel version of the user function, called `MyFunc_parallel`.

After compiling and including the master package into your model, it is this function that should be used in the actual model as `userfuncMosel('MyFunc_parallel',XSLP_DELTAS)`. In most cases, no other modifications are necessary, as the parallel function will detect the number of threads in the system and will start that many worker threads automatically. These will be shut down when your model finishes. Each worker's initialization code is performed only once, at the time of its first execution.

It may be necessary to explicitly start the worker threads, either to control the number of threads used, or to pass specific parameter settings to the user function package. This can be done by the procedure `MyFunc_StartWorkers( ThreadCount : integer, UfPackageParameters : string )`. In case it is necessary to stop the workers, the procedure `MyFunc_StopWorkers` may be used.

In case the user functions are computationally very expensive, by modifying the connection string in the generated module it is possible to utilize distributed/cloud-based computation of the user functions.

The worker model will only be compiled into memory during execution, but may be modified as necessary within the master model. For debugging purposes, it may be practical to redirect the worker to a file.

CHAPTER 4

Modeling in Extended MPS Format

4.1 Basic formulation

Standard MPS format uses a fixed format text file to hold the problem information. Extended MPS format has two main differences from the standard form:

- The records in the file are free-format – that is, the fields are not necessarily in fixed columns or of fixed size, and each field is delimited by one or more spaces.
- The standard MPS format allows only numbers to be used in the "coefficient" fields – extended MPS format allows the use of formulae.
- There is an optional extra section in extended MPS format, holding additional data and structures for Xpress NonLinear.

We shall tend to use a fairly fixed format, to aid readability.

```
NAME POLYGON
```

The first record of any MPS file is the `NAME` record, which has the name which may be used to create file names where no other name is specified, and is also written into the matrix and solution files.

```
ROWS
```

The `ROWS` record introduces the list of rows of the problem – this includes the objective function as well as all the constraints.

```
 N OBJ
 E OBJEQ
 G T2T1
 G T3T2
 G T4T3
 L V1V2
 L V1V3
 L V2V3
 L V2V4
 L V3V4
```

The first character denotes the type of constraint. The possible values are:

N	not constraining (always used for the objective function, but may be used elsewhere).
E	equality: the left hand side (LHS) is equal to the right hand side (RHS).
L	less than or equal to: the LHS is less than or equal to the RHS.
G	greater than or equal to: the LHS is greater than or equal to the RHS.

The second field is the name used for the constraint. In MPS file format, everything has a name. Therefore, within each type of entity (rows, columns, etc) each name must be unique. In general, you should try to ensure that names are unique across all entities, to avoid possible confusion.

You should also try to make the names meaningful, so that you can understand what they mean.

In the example:

OBJ	is the objective function.
OBJEQ	is the "equality" version of the objective function which, as explained below, is required because we are trying to optimize a non-linear objective.
TiTj	is the constraint that will ensure $\theta_i \geq \theta_j (j = i - 1)$.
ViVj	is the constraint that will ensure that the distance between V_i and V_j is ≤ 1.

COLUMNS

The COLUMNS record introduces the list of columns and coefficients in the matrix. In a normal linear problem, all the variables will appear explicitly as columns in this section. However, in non-linear problems, it is possible for variables to appear only in formulae and so they may not appear explicitly. In the example, the variables THETA1 to THETA4 appear explicitly, the variables RHO1 to RHO4 appear only in formulae. Constraints which involve only one variable in a linear way (that is, they limit the value of a variable to a minimum value, a maximum value or both – possibly equal – values) are usually put in a separate "BOUNDS" section which appears later.

```
OBJX OBJ   1.0
OBJX OBJEQ -1.0
```

The first field is the name of the column. All "COLUMNS" records for a column must be together. The second field is the name of the row (which was defined in the ROWS section). The third field is the value. It is not necessary to include zero values – only the non-zeros are required

If the coefficients are constant, then it is possible to put two on each record, by putting a second row name and value after the first (as in the example for THETA2 and THETA3 below).

The constraints putting θ_i in order are all linear – that is, the coefficients are all constant.

```
THETA1  T2T1   -1
THETA2  T2T1   1       T3T2   -1
THETA3  T3T2   1       T4T3   -1
THETA4  T4T3   1
```

The RHS of any constraint must be constant. Therefore, to write THETA2 $\geq$ THETA1, we must actually write THETA2 - THETA1 $\geq$ 0. The constraint T2T1 has coefficient -1 in THETA1 and +1 in THETA2.

We want to maximise the area of the polygon. The formula for this is the sum of the areas of the triangles with one vertex at V5 – i.e.:

```
0.5 * RHO1 * RHO2 * SIN ( THETA2 - THETA1 ) +
0.5 * RHO2 * RHO3 * SIN ( THETA3 - THETA2 ) +
0.5 * RHO3 * RHO4 * SIN ( THETA4 - THETA3 )
```

– which is a non-linear function. Xpress NonLinear does not itself have a problem with non-linear objective functions, but Xpress distinguishes between the original N-type row which contains the objective function coefficients when the matrix is read in, and the objective function which is actually optimized. To avoid any confusion between these two "objectives", Xpress NonLinear also requires that the objective function as passed to Xpress Optimizer is linear. What we want to do is:

maximize `AREA`, where `AREA` is a non-linear function.

We create a new variable – called in this example `OBJX` – and write:

`OBJX` = `AREA` (or, because the RHS must be constant, `AREA` – `OBJX` = `0`)

and then: maximize `OBJX`, where `OBJX` is just a variable.

The constraint linking `OBJX` and `AREA` was defined as the equality constraint `OBJEQ` in the `ROWS` section, and `AREA` is the formula given above. This is where the coefficient of -1 in column `OBJX` comes from.

Every item in the matrix has to be in a coefficient – that is, it is the multiplier of a variable. However, the formula for area, as written, is not a coefficient of anything. There are several ways of dealing with this situation. We shall start by breaking the formula up into coefficient form – that is, to write it as X1*formula1 + X2*formula2 + ... Our formula could then be:

```
RHO1 * ( 0.5 * RHO2 * SIN ( THETA2 - THETA1 ) ) +
RHO2 * ( 0.5 * RHO3 * SIN ( THETA3 - THETA2 ) ) +
RHO3 * ( 0.5 * RHO4 * SIN ( THETA4 - THETA3 ) )
```

which is of the right form and can be written in the `COLUMNS` section as follows:

```
RHO1 OBJEQ = 0.5 * RHO2 * SIN ( THETA2 - THETA1 )
RHO2 OBJEQ = 0.5 * RHO3 * SIN ( THETA3 - THETA2 )
RHO3 OBJEQ = 0.5 * RHO4 * SIN ( THETA4 - THETA3 )
```

Notice that the formula begins with an equals sign. When this is used in the coefficient field, it always means that a formula is being used rather than a constant. The formula must be written on one line – it does not matter how long it is – and each token (variable, constant, operator, bracket or function name) must be delimited by spaces.

When a formula is used, you can only write one coefficient on the record – the option of a second coefficient only applies when both coefficients are constants.

The constraints for the distances between pairs of vertices are relationships of the form:

```
RHO1 * RHO1 + RHO2 * RHO2 - 2 * RHO1 * RHO2 * COS ( THETA2 - THETA1 ) <= 1
```

These can again be split into coefficients, for example:

```
RHO1 * ( RHO1 - 2 * RHO2 * COS ( THETA2 - THETA1 ) ) + RHO2 * ( RHO2 )
```

This looks a little strange, because `RHO2` appears as a coefficient of itself, but that is perfectly all right. This section of the matrix contains a set of records (one for each of the `ViVj` constraints) like this:

```
RHO1   V1V2 = RHO1 - 2 * RHO2 * COS ( THETA2 - THETA1 )
RHO2   V1V2 = RHO2
```

Note that because the records for each column must all appear together, the coefficients for – for example – RHO1 in this segment must be merged in with those in the previous (OBJEQ) segment.

RHS

The RHS record introduces the right hand side section.

The RHS section is formatted very much like a COLUMNS section with constant coefficients. There is a column name – it is actually the name of the right hand side – and then one or two entries per record. Again, only the non-zero entries are actually required.

```
RHS1 T2T1 .001  T3T2 .001
RHS1 T4T3 .001  V1V2 1
RHS1 V1V3 1     V1V4 1
RHS1 V2V3 1     V2V4 1
RHS1 V3V4 1
```

RHS1 is the name we have chosen for the right hand side. It is possible – although beyond the scope of this guide – to have more than one right hand side, and to select the one you want. Note that, in order to ensure we do have a polygon with N sides, we have made the relationship between theta(i) and theta(i-1) a strict inequality by adding 0.001 as the right hand side. If we did not, then two of the vertices could coincide and so the polygon would effectively lose one of its sides.

BOUNDS

The BOUNDS record introduces the BOUNDS section which typically holds the values of constraints which involve single variables.

Like the RHS section, it is possible to have more than one set of BOUNDS, and to select the one you want to use. There is therefore in each record a bound name which identifies the set of bounds to which it belongs. We shall be using only ones set of bounds, called BOUND1.

Bounds constrain a variable by providing a lower limit or an upper limit to its value. By providing a limit of - ∞ for the lower bound, it is possible to create a variable which can take on any value – a "free" variable. The following bound types are provided:

- LO a lower bound.
- UP an upper bound.
- FX a fixed bound (the upper and lower limits are equal).
- FR a free variable (no lower or upper limit).
- MI a "minus infinity" variable – it can take on any non-positive value.

There are other types of bound which are used with integer programming, which is beyond the scope of this guide.

```
FR BOUND1 OBJX
LO BOUND1 RHO1    0.01
UP BOUND1 RHO1    1
LO BOUND1 RHO2    0.01
UP BOUND1 RHO2    1
LO BOUND1 RHO3    0.01
UP BOUND1 RHO3    1
LO BOUND1 RHO4    0.01
UP BOUND1 RHO4    1
UP BOUND1 THETA4  3.1415926
```

A record in a BOUNDS section can contain up to four fields. The first one is the bound type (from the list above). The second is the name of the BOUNDS set being used (ours is always BOUND1). The third is the name of the variable or column being bounded. Unless the bound type is FR or MI, there is a fourth field which contains the value of the bound.

Although we know that the area is always positive (or at least non-negative), a more complicated problem might have an objective function which could be positive or negative – you could make a profit or a loss – and so OBJX needs to be able to take on po sitive and negative values. The fact that it is marked as "free" here does not mean that it can actually take on any value, because it is still constrained by the rest of the problem.

The upper bounds on RHO1 to RHO4 provide the rest of the restrictions which ensure that the distances between any two vertices are = 1, and the limit on THETA4 ensures that the whole polygon is above the X-axis. Just to make sure that we do not "lose" a side because the value of RHOi becomes zero, we set a lower bound of 0.01 on all the rhos, performing a similar function to the RHS values of .001 for TiTj.

```
ENDATA
```

The last record in the file is the ENDATA record.

Although this is sufficient to define the model, it is usually better to give Xpress NonLinear some idea of where to start – that is, to provide a set of initial values for the variables. You do not have to provide values for everything, but you should try to provide them for every variable which appears in a non-linear coefficient, or which has a non-linear coefficient. In our current example, that means everything except OBJX.

```
SLPDATA
```

The SLPDATA record introduces a variety of different special items for Xpress NonLinear. It comes as the last section in the model (before the ENDATA record). We are using it at this stage for defining initial values. These are done with an IV record.

```
IV IVSET1 RHO1 0.555
IV IVSET1 RHO2 0.888
IV IVSET1 RHO3 1
IV IVSET1 RHO4 0.888
```

Just as with the RHS and BOUNDS sections, it is possible to have more than one set of initial values – perhaps because the same structure is used to solve a whole range of problems where the answers are so different that it does not make much sense to start always from the same place. In this example, we are using only one set – IVSET1.

The IV record contains four fields. The first one is IV, which indicates the type of SLPDATA being provided. The second is the name of the set of initial values. The third is the name of the variable and the fourth is the value being provided.

In the case of IV records, it is possible – and indeed perhaps necessary – to provide initial values which are zero. The default value (which is used if no value is provided) is not zero, so if you want to start with a zero value you must say so.

4.2 Using the nonlinear optimizer console-based interface

The nonlinear console is a data-driven console-based interface for operating Xpress NonLinear, an extension of the Xpress Optimizer console. The optimizer console switches to nonlinear is a valid nonlinear license is detected.

The example will use screen-based input and output. You can also put the commands into a file and execute it in batch mode, or use the embedded TCL scripting language.

Commands are not case-sensitive except where the case is important (for example, the name of the objective function). We shall use upper case for commands and lower case for the arguments which would change for other models. Each parameter in a command must be separated by at least one space from the preceding parameter or command.

```
optimizer
```

This starts the optimizer program. This checks for the existence of the Xpress Optimizer DLLs. If you are using an OEM version of the Xpress DLL, you may need a special password or license file from your usual supplier.

```
READPROB polygon
```

This reads a non-linear problem from the file polygon.mat.

```
MAXIM
```

This form of the maximize command does a non-linear optimization with the default settings of all the parameters (it will recognise the problem as an SLP one automatically).

```
WRITEPRTSOL
```

This will use the normal Xpress function to write to solution in a text form to a file with the same name as the input, but with a ".prt" suffix.

```
Q
```

This (the abbreviation for the `QUIT` command) terminates the optimizer console program.

4.3 Coefficients and terms

So far we have managed to express the formulae as coefficients. However, there are constraints – for example `SIN(A) ≤ 0.5` – which cannot be expressed directly using coefficients. The extended MPS format has a special reserved column name – the equals sign – which is effectively a variable with a fixed value of 1.0, and which can be used to hold formulae of any type, whether they can be expressed as coefficients or not. The area formula and distance constraints could all be written in a more readable form by using the "equals column". The area formula is rather long to write in this guide, but the distance constraints look like this:

```
= V1V2 RHO1 * RHO1 + RHO2 * RHO2 – 2 * RHO1 * RHO2 * COS ( THETA2 – THETA1 )
= V1V3 RHO1 * RHO1 + RHO3 * RHO3 – 2 * RHO1 * RHO3 * COS ( THETA3 – THETA1 )
```

CHAPTER 5

The Xpress NonLinear API Functions

Instead of writing an extended MPS file and reading in the model from the file, it is possible to embed Xpress NonLinear directly into your application, and to create the problem, solve it and analyze the solution entirely by using the Xpress NonLinear API functions. This example uses the C header files and API calls. We shall assume you have some familiarity with the Xpress Optimizer API functions in XPRS.DLL.

The structure of the model and the naming system will follow that used in the previous section, so you should read the chapter 4 first.

5.1 Header files

The header file containing the Xpress NonLinear definitions is `xslp.h`. This must be included together with the Xpress Optimizer header `xprs.h`. `xprs.h` must come first.

```
#include "xprs.h"
#include "xslp.h"
```

5.2 Initialization

Xpress NonLinear and Xpress Optimizer both need to be initialized, and an empty problem created. All Xpress NonLinear functions return a code indicating whether the function completed successfully. A non-zero value indicates an error. For ease of reading, we have for the most part omitted the tests on the return codes, but a well-written program should always test the values.

```
XPRSprob mprob;
XSLPprob sprob;

if (ReturnValue=XPRSinit(NULL)) goto ErrorReturn;
if (ReturnValue=XSLPinit()) goto ErrorReturn;
if (ReturnValue=XPRScreateprob(&mprob)) goto ErrorReturn;
if (ReturnValue=XSLPcreateprob(&sprob, &mprob)) goto ErrorReturn;
```

5.3 Callbacks

It is good practice to set up at least a message callback, so that any messages produced by the system appear on the screen or in a file. The `XSLPsetcbmessage` function sets both the Xpress NonLinear and Xpress Optimizer callbacks, so that all messages appear in the same place.

```
XSLPsetcbmessage(sprob, XSLPMessage, NULL);
```

```
void XPRS_CC XSLPMessage(XSLPprob my_prob, void *my_object, char *msg, int len,
       int msg_type)
{
  switch (msg_type) {
  case 4: /* error */
  case 3: /* warning */
  case 2: /* dialogue */
  case 1: /* information */
    printf("%s\n", msg);
    break;
  default: /* exiting */
    fflush(stdout);
    break;
  }
}
```

This is a simple callback routine, which prints any message to standard output.

5.4 Creating the linear part of the problem

The linear part of the problem, and the definitions of the rows and columns of the problem are carried out using the normal Xpress Optimizer functions.

```
#define MAXROW 20
#define MAXCOL 20
#define MAXELT 50
  int nRow, nCol, nSide, nRowName, nColName;
  int Sin, Cos;
  char RowType[MAXROW];
  double RHS[MAXROW], OBJ[MAXCOL], Element[MAXELT];
  double Lower[MAXCOL], Upper[MAXCOL];
  int ColStart[MAXCOL+1], RowIndex[MAXELT];
  char RowNames[500], ColNames[500];
```

In this example, we have set the dimensions by using `#define` statements, rather than working out the actual sizes required from the number of sides and then allocating the space dynamically.

```
  nSide = 5;
  nRowName = 0;
  nColName = 0;
```

By making the number of sides a variable (`nSide`) we can create other polygons by changing its value.

It is useful – at least while building a model – to be able to see what has been created. We will therefore create meaningful names for the rows and columns. `nRowName` and `nColName` count along the character buffers `RowNames` and `ColNames`.

```
  nRow = nSide-2 + (nSide-1)*(nSide-2)/2 + 1;
  nCol = (nSide-1)*2 + 2;
  for (i=0; i<nRow; i++) RHS[i] = 0;
```

The number of constraints is:

`nSide-2`	for the relationships between adjacent thetas.
`(nSide-1)*(nSide-2)/2`	for the distances between pairs of vertices.
`1`	for the `OBJEQ` non-linear "objective function".

The number of columns is:

nSide−1	for the thetas.
nSide−1	for the rhos.
1	for the OBJX objective function column.
1	for the "equals column".

We are using "C"-style numbering for rows and columns, so the counting starts from zero.

```
nRow = 0;
RowType[nRow++] = 'E'; /* OBJEQ */
nRowName = nRowName + 1 + sprintf(&RowNames[nRowName], "OBJEQ");
for (i=1; i<nSide-1; i++) {
  RowType[nRow++] = 'G'; /* T2T1 .. T4T3 */
  RHS[i] = 0.001;
  nRowName = nRowName + 1 + sprintf(&RowNames[nRowName], "T%dT%d", i+1, i);
}
```

This sets the row type indicator for OBJEQ and the theta relationships, with a right hand side of 0.001. We also create row names in the RowNames buffer. Each name is terminated by a NULL character (automatically placed there by the sprintf function). sprintf returns the length of the string written, excluding the terminating NULL character.

```
for (i=1; i<nSide-1; i++) {
  for (j=i+1; j<nSide; j++) {
    RowType[nRow] = 'L';
    RHS[nRow++] = 1.0;
    nRowName = nRowName + 1 + sprintf(&RowNames[nRowName], "V%dV%d", i, j);
  }
}
```

This defines the L-type rows which constrain the distances between pairs of vertices. The right hand side is 1.0 (the maximum value) and the names are of the form ViVj.

```
for (i=0; i<nCol; i++) {
  OBJ[i] = 0;        /* objective function */
  Lower[i] = 0;      /* lower bound normally zero */
  Upper[i] = XPRS_PLUSINFINITY; /* upper bound = infinity */
}
```

This sets up the standard column data, with objective function entries of zero, and default bounds of zero to plus infinity. We shall change these for the individual items as required.

```
nCol = 0;
nElement = 0;
ColStart[nCol] = nElement;
OBJ[nCol] = 1.0;
Lower[nCol++] = XPRS_MINUSINFINITY; /* free column */
Element[nElement] = -1.0;
RowIndex[nElement++] = 0;
nColName = nColName + 1 + sprintf(&ColNames[nColName], "OBJX");
```

This starts the construction of the matrix elements. nElement counts through the Element and RowIndex arrays, nCol counts through the ColStart, OBJ, Lower and Upper arrays. The first column, OBJX, has the objective function value of +1 and a value of -1 in the OBJEQ row. It is also defined to be "free", by making its lower bound equal to minus infinity.

```
iRow = 0
for (i=1; i<nSide; i++) {
  nColName = nColName + 1 + sprintf(&ColNames[nColName], "THETA%d", i);
  ColStart[nCol++] = nElement;
```

```
  if (i < nSide-1) {
    Element[nElement] = -1;
    RowIndex[nElement++] = iRow+1;
  }
  if (i > 1) {
    Element[nElement] = 1;
    RowIndex[nElement++] = iRow;
  }
  iRow++;
}
```

This creates the relationships between adjacent thetas. The tests on `i` are to deal with the first and last thetas which do not have relationships with both their predecessor and successor.

```
Upper[nCol-1] = 3.1415926;
```

This sets the bound on the final theta to be π . The column index is `nCol-1` because `nCol` has already been incremented.

```
nColName = nColName + 1 + sprintf(&ColNames[nColName], "=");
ColStart[nCol] = nElement;
Lower[nCol] = Upper[nCol] = 1.0; /* fixed at 1.0 */
nCol++;
```

This creates the "equals column" – its name is "=" and it is fixed at a value of 1.0.

```
for (i=1; i<nSide; i++) {
  Lower[nCol] = 0.01;         /* lower bound */
  Upper[nCol] = 1;
  ColStart[nCol++] = nElement;
  nColName = nColName + 1 + sprintf(&ColNames[nColName], "RHO%d", i);
}
ColStart[nCol] = nElement;
```

The remaining columns – the rho variables – have only non-linear coefficients and so they do not appear in the linear section except as empty columns. They are bounded between 0.01 and 1.0 but have no entries. The final entry in `ColStart` is one after the end of the last column.

```
XPRSsetintcontrol(mprob, XPRS_MPSNAMELENGTH, 16);
```

If you are creating your own names – as we are here – then you need to make sure that Xpress Optimizer can handle both the names you have created and the names that will be created by Xpress NonLinear. Typically, Xpress NonLinear will create names which are three characters longer than the names you have used. If the longest name would be more than 8 characters, you should set the Xpress Optimizer name length to be larger – it comes in multiples of 8, so we have used 16 here. If you do not make the name length sufficiently large, then the `XPRSaddnames` function will return an error either here or during the Xpress NonLinear "construct" phase.

```
XPRSloadlp(mprob, "Polygon", nCol, nRow, RowType, RHS, NULL,
  OBJ, ColStart, NULL, RowIndex, Element, Lower, Upper);
```

This actually loads the model into Xpress Optimizer. We are not using ranges or column element counts, which is why the two arguments are `NULL`.

```
XPRSaddnames(mprob, 1, RowNames, 0, nRow-1);
XPRSaddnames(mprob, 2, ColNames, 0, nCol-1);
```

The row and column names can now be added.

5.5 Adding the non-linear part of the problem

Be warned – this section is complicated, but it is the most efficient way – from SLP's point of view – to input formulae. See the next section for a much easier (but less efficient) way of inputting the formulae directly.

```
#define MAXTOKEN 200
#define MAXCOEF 20
...
  int Sin, Cos;
  ColIndex[MAXCOL];
  FormulaStart[MAXCOEF];
  Type[MAXTOKEN];
  double Value[MAXTOKEN], Factor[MAXCOEF];
```

The arrays for the non-linear part can often be re-used from the linear part. The new arrays are `ColIndex` (for the column index of the coefficients), `FormulaStart` and `Factor` for the coefficients, and `Type` and `Value` to hold the internal forms of the formulae.

```
  XSLPgetindex(sprob, XSLP_INTERNALFUNCNAMES, "SIN", &Sin);
  XSLPgetindex(sprob, XSLP_INTERNALFUNCNAMES, "COS", &Cos);
```

We will be using the Xpress NonLinear internal functions `SIN` and `COS`. The `XSLPgetindex` function finds the index of an Xpress NonLinear entity (character variable, internal or user function).

```
  nToken = 0;
  nCoef = 0;
  RowIndex[nCoef] = 0;
  ColIndex[nCoef] = nSide;
  Factor[nCoef] = 0.5;
  FormulaStart[nCoef++] = nToken;
```

For each coefficient, the following information is required:

`RowIndex`	the index of the row.
`ColIndex`	the index of the column.
`FormulaStart`	the beginning of the internal formula array for the coefficient.
`Factor`	this is optional. If used, it holds a constant multiplier for the formula. This is particularly useful where the same formula appears in several coefficients, but with different signs or scaling. The formula can be used once, with different factors.

```
for (i=1; i<nSide-1; i++) {
  Type[nToken] = XSLP_COL;
  Value[nToken++] = nSide+i+1;
  Type[nToken] = XSLP_COL;
  Value[nToken++] = nSide+i;
  Type[nToken] = XSLP_OP;
  Value[nToken++] = XSLP_MULTIPLY;
  Type[nToken] = XSLP_RB;
  Value[nToken++] = 0;
  Type[nToken] = XSLP_COL;
  Value[nToken++] = i+1;
  Type[nToken] = XSLP_COL;
  Value[nToken++] = i;
  Type[nToken] = XSLP_OP;
  Value[nToken++] = XSLP_MINUS;
  Type[nToken] = XSLP_IFUN;
  Value[nToken++] = Sin;
```

```
    Type[nToken] = XSLP_OP
    Value[nToken++] = XSLP_MULTIPLY;
    if (i>1) {
      Type[nToken] = XSLP_OP;
      Value[nToken++] = XSLP_PLUS;
    }
  }
```

This looks very complicated, but it is really just rather large. We are using the "reverse Polish" or "parsed" form of the formula for area. The original formula, written in the normal way, would look like this:
`RHO2 * RHO1 * SIN ( THETA2 – THETA1 ) + .......`
In reverse Polish notation, tokens are pushed onto the stack or popped from it. Typically, this means that a binary operation A x B is written as A B x (push A, push B, pop A and B and push the result). The first term of our area formula then becomes:
`RHO2 RHO1 * ) THETA2 THETA1 – SIN *`
Notice that the right hand bracket appears as an explicit token. This allows the `SIN` function to identify where its argument list starts – and incidentally allows functions to have varying numbers of arguments.

Each token of the formula is written as two items – `Type` and `Value`.
`Type` is an integer and is one of the defined types of token, as given in the `xslp.h` header file. `XSLP_CON`, for example, is a constant; `XSLP_COL` is a column.
`Value` is a double precision value, and its meaning depends on the corresponding `Type`. For a `Type` of `XSLP_CON`, `Value` is the constant value; for `XSLP_COL`, `Value` is the column number; for `XSLP_OP` (arithmetic operation), `Value` is the operand number as defined in `xslp.h`; for a function (type `XSLP_IFUN` for internal functions, `XSLP_FUN` for user functions), `Value` is the function number.
A list of tokens for a formula is always terminated by a token of type `XSLP_EOF`.

The loop writes each term in order, and adds terms (using the `XSLP_PLUS` operator) after the first pass through the loop.

```
for (i=1; i<nSide-1; i++) {
  for (j=i+1; j<nSide; j++) {
    RowIndex[nCoef] = iRow++;
    ColIndex[nCoef] = nSide;
    Factor[nCoef] = 1.0;
    FormulaStart[nCoef++] = nToken;

    Type[nToken] = XSLP_COL;
    Value[nToken++] = nSide+i;
    Type[nToken] = XSLP_CON;
    Value[nToken++] = 2;
    Type[nToken] = XSLP_OP;
    Value[nToken++] = XSLP_EXPONENT;
    Type[nToken] = XSLP_COL;
    Value[nToken++] = nSide+j;
    Type[nToken] = XSLP_CON;
    Value[nToken++] = 2;
    Type[nToken] = XSLP_OP;
    Value[nToken++] = XSLP_PLUS;
    Type[nToken] = XSLP_CON;
    Value[nToken++] = 2;
    Type[nToken] = XSLP_COL;
    Value[nToken++] = nSide+i;
    Type[nToken] = XSLP_OP;
    Value[nToken++] = XSLP_MULTIPLY;
    Type[nToken] = XSLP_COL;
    Value[nToken++] = nSide+j;
    Type[nToken] = XSLP_OP;
    Value[nToken++] = XSLP_MULTIPLY;
    Type[nToken] = XSLP_RB;
    Value[nToken++] = 0;
    Type[nToken] = XSLP_COL;
    Value[nToken++] = j;
```

```
      Type[nToken] = XSLP_COL;
      Value[nToken++] = i;
      Type[nToken] = XSLP_OP;
      Value[nToken++] = XSLP_MINUS;
      Type[nToken] = XSLP_IFUN;
      Value[nToken++] = Cos;
      Type[nToken] = XSLP_OP
      Value[nToken++] = XSLP_MULTIPLY;
      Type[nToken] = XSLP_OP;
      Value[nToken++] = XSLP_MINUS;
      Type[nToken] = XSLP_EOF;
      Value[nToken++] = 0;
    }
  }
```

This writes the formula for the distances between pairs of vertices. It follows the same principle as the previous formula, writing the formula in parsed form as:
`RHOi 2 RHOj 2 + 2 RHOi * RHOj * ) THETAj THETAi - COS * -`

```
XSLPloadcoefs(sprob, nCoef, RowIndex, ColIndex, Factor,
FormulaStart, 1, Type, Value);
```

The `XSLPloadcoefs` is the most efficient way of loading non-linear coefficients into a problem. There is an `XSLPaddcoefs` function which is identical except that it does not delete any existing coefficients first. There is also an `XSLPchgcoef` function, which can be used to change individual coefficients one at a time. Because we are using internal parsed format, the "Parsed" flag in the argument list is set to 1.

5.6 Adding the non-linear part of the problem using character formulae

Provided that all entities – in particular columns and user functions – have explicit and unique names, the non-linear part can be input by writing the formulae as character strings. This is not as efficient as using the `XSLPloadcoefs()` function but is generally easier to understand.

```
/* Build up nonlinear coefficients */
/* Allow space for largest formula - approx 50 characters per side for area */
  CoefBuffer = (char *) malloc(50*nSide);
```

We shall be using large formulae, so we need a character buffer large enough to hold the largest formula we are using. The estimate here is 50 characters per side of the polygon for the area formula, which is the largest we are using.

```
/* Area */
  Factor = 0.5;

  BufferPos = 0;
  for (i=1; i<nSide-1; i++) {
    if (i > 1) {
      BufferPos = BufferPos + sprintf(&CoefBuffer[BufferPos], " + ");
    }
    BufferPos = BufferPos + sprintf(&CoefBuffer[BufferPos], "RHO%d * RHO%d *
                SIN ( THETA%d - THETA%d )", i+1, i, i+1, i);
  }
  XSLPchgccoef(sprob, 0, nSide, &Factor, CoefBuffer);
```

The area formula is of the form:
`(RHO2*RHO1*SIN(THETA2-THETA1) + RHO3*RHO2*SIN(THETA3-THETA2) + ... ) / 2`
The loop writes the product for each consecutive pair of vertices and also puts in the "+" sign after the first one.

The XSLPchgccoef function is a variation of XSLPchgcoef but uses a character string for the formula instead of passing it as arrays of tokens. The arguments to the function are:

RowIndex	the index of the row.
ColIndex	the index of the column.
Factor	this is optional. If used, it holds the address of a constant multiplier for the formula. This is particularly useful where the same formula appears in several coefficients, but with different signs or scaling. The formula can be used once, but with different factors. To omit it, use a NULL argument.
CoefBuffer	the formula, written in character form.

In this case, RowIndex is zero and ColIndex is nSide (the "equals" column).

```
/* Distances */
  Factor = 1.0;
  for (i=1; i<nSide-1; i++) {
    for (j=i+1; j<nSide; j++) {
      sprintf(CoefBuffer, "RHO%d ^ 2 + RHO%d ^ 2 - 2 * RHO%d * RHO%d *
              COS ( THETA%d - THETA%d )", j, i, j, i, j, i);
      XSLPchgccoef(sprob, iRow, nSide, &Factor, CoefBuffer);
      iRow++;
    }
```

This creates the formula for the distance between pairs of vertices and writes each into a new row in the "equals" column.

Provided you have given names to any user functions in your program, you can use them in a formula in exactly the same way as SIN and COS have been used above.

5.7 Checking the data

Xpress NonLinear includes the function XSLPwriteprob which writes out a non-linear problem in text form which can then be checked manually. Indeed, the problem can then be run using the XSLP console program, provided there are no user functions which refer back into your compiled program. In particular, this facility does allow small versions of a problem to be checked before moving on to the full size ones.

```
XSLPwriteprob(sprob, "testmat", "");
```

The first argument is the Xpress NonLinear problem pointer; the second is the name of the matrix to be produced (the suffix ".mat" will be added automatically). The last argument allows various different types of output including "scrambled" names – that is, internally-generated names will be used rather than those you have provided. For checking purposes, this is obviously not a good idea.

5.8 Solving and printing the solution

```
XSLPmaxim(sprob, "");
```

The XSLPmaxim and XSLPminim functions perform a non-linear maximization or minimization on the current problem. The second argument can be used to pass flags as defined in the Xpress NonLinear Reference Manual.

```
XPRSwriteprtsol(mprob);
```

The standard Xpress Optimizer solution print can be obtained by using the `XPRSwriteprtsol` function. The row and column activities and dual values can be obtained using the `XPRSgetsol` function.

In addition, you can use the `XSLPgetvar` function to obtain the values of SLP variables – that is, of variables which are in non-linear coefficients, or which have non-linear coefficients. If you are using cascading (see the Xpress NonLinear reference manual for more details) so that Xpress NonLinear recalculates the values of the dependent SLP variables at each SLP iteration, then the value from `XSLPgetvar` will be the recalculated value, whereas the value from `XPRSgetsol` will be the value from the LP solution (before recalculation).

5.9 Closing the program

The `XSLPdestroyprob` function frees any system resources allocated by Xpress NonLinear for the specific problem. The problem pointer is then no longer valid. `XPRSdestroyprob` performs a similar function for the underlying linear problem `mprob`. The `XSLPfree` function frees any system resources allocated by Xpress NonLinear. You must then call `XPRSfree` to perform a similar operation for the optimizer.

```
XSLPdestroyprob(sprob);
XPRSdestroyprob(mprob);
XSLPfree();
XPRSfree();
```

If these functions are not called, the program may appear to have worked and terminated correctly. However, in such a case there may be areas of memory which are not returned to the system when the program terminates and so repeated executions of the program will result in progressive loss of available memory to the system, which will manifest iself in poorer performance and could ultimately produce a system crash.

5.10 Adding initial values

So far, Xpress NonLinear has started by using values which it estimates for itself. Because most of the variables are bounded, these initial values are fairly reasonable, and the model will solve. However, in general, you will need to provide initial values for at least some of the variables. Initial values, and other information for SLP variables, are provided using the `XSLPloadvars` function.

```
int VarType[MAXCOL];
double InitialValue[MAXCOL];
```

To load initial values using `XSLPloadvars`, we need an array (`InitialValue`) to hold the initial values, and a `VarType` array which is a bitmap to describe what information is being set for each variable.

```
for(i=1; i<nSide; i++) {
  ...
  InitialValue[nCol] = 3.14159*((double)i) / ((double)nSide);
  VarType[nCol] = 4;
  ...
}
...
for(i=1; i<nSide; i++) {
```

```
  InitialValue[nCol] = 1;
  VarType[nCol] = 4;
}
```

These sections extend the loops for the columns in the earlier example. We set initial values for the thetas so that the vertices are spaced at equal angles; the rhos are all started at 1. We do not need to set a value for the equals column, because it is fixed at one. However, it is good practice to do so. In each case we set `VarType` to 4 because (as described in the Xpress NonLinear Reference Manual) Bit 2 of the type indicates that the initial value is being set.

```
for(i=0; i<nCol; i++) ColIndex[i] = i
XSLPloadvars(sprob, nCol-1, &ColIndex[1], &VarType[1], NULL, NULL, NULL,
             &InitialValue[1], NULL);
```

`XSLPloadvars` can take several other arguments apart from the initial value. It is a general principle in Xpress NonLinear that using `NULL` for an argument means that there is no information being provided, and the current or default value will not be changed.

Because we built up the initial values as we went, the `VarType` and `InitialValue` arrays include column 0, which is `OBJX` and is not an SLP variable. As all the rest are SLP variables, we can simply start these arrays at the second item, and reduce the variable count by 1.

CHAPTER 6

The Nonlinear Console Program

6.1 The Console Nonlinear

The nonlinear `optimizer` is an extension to the FICO Xpress Optimizer interactive console.

The console for nonlinear is started from the command line using the following syntax:

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

6.1.1 The nonlinear console extensions

The nonlinear console is an extension of the Xpress optimizer console. The optimizer automatically switches to nonlinear mode if a nonlinear license is detected. All the optimizer console commands work the same way as in the normal optimizer console. The active working problem for those commands is the actual linearization after augmentation, and the linear part of the problem before augmentation.

Optimizer console commands with an extended effect:

`readprob`	Read in an MPS/MAT or LP file
`minim`	Minimize an LP, a MIP or an SLP problem
`maxim`	Maximize an LP, a MIP or an SLP problem
`lpoptimize`	Minimize or maximize a problem
`mipoptimize`	Solve the problem to MIP optimality
`writeprob`	Export the problem into file
`dumpcontrols`	Display controls which are at a non default value

The MPS file can be an extended MPS file containing an NLP model. The `minim` and `maxim` commands will call `XPRSminim` or `XPRSmaxim` for LP and MIP problems, and `XSLPminim` and `XSLPmaxim` for SLP problems respectively; with the same applying to `lpoptimize` and `mipoptimize`. All these commands accept the same flags as the corresponding library function

New commands:

cascade	Perform cascading
cascadeorder	Recalculate the cascading order
construct	Construct the augmented problem
dumpattributes	Display problem attributes
reinitialize	Reinitialize an augmented problem
setcurrentiv	Copy the current solution as initial value
slp_save	XSLPsave
slp_scaling	Display scaling statistics
unconstruct	Remove the augmentation
validate	Validate the current solution
validatekkt	Validate the kkt conditions for the current solution

In order to separate XSLP controls and attributes for the XPRS ones, all XSLP controls and attributes are pretagged as XSLP_ or SLP_, for example XSLP_ALGORITHM.

6.1.2 Common features of the Xpress Optimizer and the Xpress Nonlinear Optimizer console

All features of the Xpress optimizer console program is supported. For a full description, please refer to the Xpress optimizer reference manual.

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

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

The Console XSLP provides a quick and convenient interface for operating on a single problem loaded into XSLP. The Console XSLP problem contains the problem data as well as (i) control variables for handling and solving the problem and (ii) attributes of the problem and its solution information.

The Console SLP auto–completion feature is a useful way of reducing key strokes when issuing commands. To use the auto–completion feature, type the first part of an optimizer command name followed by the Tab key. For example, by typing "CONST" followed by the Tab key Console Xpress will complete to the "CONSTRUCT". Note that once you have finished inputting the command name portion of your command line, Console Xpress can also auto–complete on file names. Note that the auto–completion of file names is case–sensitive.

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

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

Finally, note that when the Console XSLP is first started it will attempt to read in an initialization file named optimizer.ini from the current working directory. This is an ASCII "script" file that may contain commands to be run at start up, which are intended to setup a customized default Console Xpress environment for the user (e.g., defining custom controls settings on the Console Xpress problem).

The Console XSLP interactive command line hosts a TCL script parser (http://www.tcl.tk). With TCL scripting the user can program flow control into their optimizer scripts. Also TCL scripting provides the user with programmatic access to a powerful suite of functionality in the TCL library. With scripting support the Console Xpress provides a high level of control and flexibility well beyond that which can be achieved by combining operating system batch files with simple piped script files. Indeed, with

scripting support the Console XSLP is ideal for (i) early application development, (ii) tuning of model formulations and solving performance and (iii) analyzing difficulties and bugs in models.

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

```
[xpress C:\] xslp_algorithm=166
[xpress C:\] xslp_algorithm
166
```

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

```
[xpress C:\] set xslp_algorithm 166
166
[xpress C:\] $miplog
166
```

For examples on how TCL can be used for scripting, tuning and testing models, please refer to the Xpress Optimizer reference manual.

Console XSLP users may interrupt the running of the commands (e.g., `minim`) by typing Ctrl–C. Once interrupted Console Xpress will return to its command prompt. If an optimization algorithm has been interrupted in this way, any solution process will stop at the first 'safe' place before returning to the prompt.

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

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

The XSLP console program can be used as a direct substitute for the Xpress Optimizer console program. The one exception is the fixed format MPS files, which is not supported by XSLP and thus neither by the XSLP console.

II. Advanced

CHAPTER 7

Nonlinear Problems

Xpress NonLinear will solve nonlinear problems. In this context, a nonlinear problem is one in which there are nonlinear relationships between variables or where there are nonlinear terms in the objective function. There is no such thing as a nonlinear variable — all variables are effectively the same — but there are nonlinear constraints and formulae. A nonlinear *constraint* contains terms which are not linear. A nonlinear *term* is one which is not a constant and is not a variable with a constant coefficient. A nonlinear constraint can contain any number of nonlinear terms.

Xpress NonLinear will also solve linear problems — that is, if the problem presented to Xpress NonLinear does not contain any nonlinear terms, then Xpress NonLinear will still solve it, using the normal optimizer library.

The solution mechanism used by Xpress-SLP is *Successive* (or *Sequential*) *Linear Programming*. This involves building a linear approximation to the original nonlinear problem, solving this approximation (to an optimal solution) and attempting to validate the result against the original problem. If the linear optimal solution is sufficiently close to a solution to the original problem, then the SLP is said to have *converged*, and the procedure stops. Otherwise, a new approximation is created and the process is repeated. Xpress-SLP has a number of features which help to create good approximations to the original problem and therefore help to produce a rapid solution.

When license, Xpress NonLinear may also utilize Knitro to solve nonlinear problems.

Note that although the solution is the result of an optimization of the linear approximation, there is no guarantee that it will be an optimal solution to the original nonlinear problem. It may be a local optimum — that is, it is a better solution than any points in its immediate neighborhood, but there is a better solution rather further away. However, a converged SLP solution will always be (to within defined tolerances) a self-consistent — and therefore practical — solution to the original problem.

7.1 Coefficients and terms

Later in this manual, it will be helpful to distinguish between formulae written as coefficients and those written as terms.

If X is a variable, then in the formula $X * f(Y)$, $f(Y)$ is the *coefficient* of X.

If $f(X)$ appears in a nonlinear constraint, then $f(X)$ is a *term* in the nonlinear constraint.

If $X * f(Y)$ appears in a nonlinear constraint, then the entity $X * f(Y)$ is a *term* in the nonlinear constraint.

As this implies, a formula written as a variable multiplied by a coefficient can always be viewed as a term, but there are terms which cannot be viewed as variables multiplied by coefficients. For example, in the constraint
$X - SIN(Y) = 0$,
$SIN(Y)$ is a *term* and cannot be written as a coefficient.

7.2 SLP variables

A variable which appears in a nonlinear coefficient or term is described as an *SLP variable*.

Normally, any variable which has a nonlinear coefficient will also be treated as an SLP variable. However, it is possible to set options so that variables which do not appear in nonlinear coefficients or terms are not treated as SLP variables.

Any variable, whether it is related to a nonlinear term or not, can be defined by the user as an SLP variable. This is most easily achieved by setting an initial value for the variable.

7.3 Local and global optimality

A globally optimal solution is a feasible solution with the best possible objective value. In general, the global optimum for a problem is not unique. By contrast, a locally optimal solution has the best possible objective value within an open neighbourhood around it. For a convex problem, every local optimum is a global optimum, but for general nonlinear problems, this is not the case.

For convex problems, which include linear, convex quadratic and convex quadratically constrained programs, solvers in the FICO Xpress library will always provide a globally optimal solution when one exists. This also holds true for mixed integer problems whose continuous relaxation is convex.

When a problem is of a more general nonlinear type, there will typically be many local optima, which are potentially widely spaced, or even in parts of the feasible region which are not connected. For these problems, both XSLP and Knitro guarantee only that they will return a locally optimal solution. That is, the result of optimization will be a solution which is better than any others in its immediate neighborhood, but there might exist other solutions which are far distant which have a better objective value.

Finding a guaranteed global optimum for an arbitrary nonlinear function requires an exhaustive search, which may be orders of magnitude more expensive. To use an analogy, it is the difference between finding a valley in a range of mountains, and finding the deepest valley. When standing in a particular valley, there is no way to know whether there is a deeper valley somewhere else.

Neither local nor global optima are typically unique. The solution returned by a solver will depend on the control settings used and, particularly for non-convex problems, on the initial values provided. A connected set of initial points yielding the same locally optimal solutions is sometimes referred to as a region of attraction for the solution. These regions are typically both algorithm and setting dependent.

7.4 Convexity

Convex problems have many desirable characteristics from the perspective of mathematical optimization. Perhaps the most significant of these is that should both the objective and the feasible region be convex, any local optimally solutions found are also known immediately to be globally optimal.

A constraint $f(x) \leq 0$ is convex if the matrix of second derivatives of f, that is to say its Hessian, is positive semi-definite at every point at which it exists. This requirement can be understood geometrically as requiring every point on every line segment which connects two points satisfying the constraint to also satisfy the constraint. It follows trivially that linear functions always lead to convex constraints, and that a nonlinear equality constraint is never convex.

For regions, a similar property must hold. If any two points of the region can be connected by a line segment which lies fully in the region itself, the region is convex. This extension is straightforward

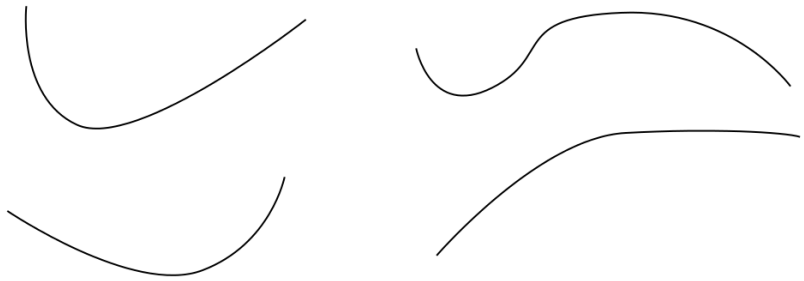

Figure 7.1: Two convex functions on the left, and two non-convex functions on the right.

when the the properties of convex functions are considered.

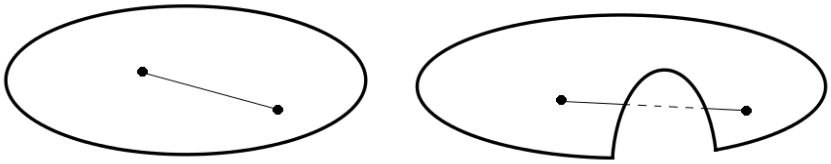

Figure 7.2: A convex region on the left and a non-convex region on the right.

It is important to note that convexity is necessary for some solution techniques and not for others. In particular, some solvers require convexity of the constraints and objective function to hold only in the feasible region, whilst others may require convexity to hold across the entire space, including infeasible points. In the special case of quadratic and quadratically constrained programs, Xpress NonLinear seamlessly migrates problems to solvers whose convexity requirements match the convexity of the problem.

7.5 Converged and practical solutions

In a strict mathematical sense, an algorithm is said to have converged if repeated iterations do not alter the coordinates of its solution significantly. A more practical view of convergence, as used in the nonlinear solvers of the Xpress suite, is to also consider the algorithm to have converged if repeated iterations have no significant effect on either the objective value or upon feasibility. This will be called extended convergence to distinguish it from the strict sense.

For some problems, a solver may visit points at which the local neighborhood is very complex, or even malformed due to numerical issues. In this situation, the best results may be obtained when convergence of some of the variables is forced. This leads to practical solutions, which are feasible and converged in most variables, but the remaining variables have had their convergence forced by the solver, for example by means of a trust region. Although these solutions are not locally optimal in a strict sense, they provide meaningful, useful results for difficult problems in practice.

7.6 The duals of general, nonlinear program

The dual of a mathematical program plays a fundamental role in the theory of continuous optimization. Each variable in a problem has a corresponding partner in that problem's dual, and the values of those

variables are called the reduced costs and dual multipliers (shadow prices). Xpress NonLinear makes estimates of these values available. These are normally defined in a similar way to the usual linear programming case, so that each value represents the rate of change of the objective when either increasing the corresponding primal variable or relaxing the corresponding primal constraint.

From an algorithmic perspective, one of the most important roles of the dual variables is to characterize local optimality. In this context, the dual multipliers and reduced costs are called Lagrange multipliers, and a solution with both primal and dual feasible variables satisfies the Karush-Kuhn-Tucker conditions. However, it is important to note that for general nonlinear problems, there exist situations in which there are no such multipliers. Geometrically, this means that the slope of the objective function is orthogonal to the linearization of the active constraints, but that their curvature still prevents any movement in the improving direction.

As a simple example, consider:

$$\begin{array}{ll} \text{minimize} & y \\ \text{subject to} & x^2 + y^2 \leq 1 \\ & (x-2)^2 + y^2 \leq 1 \end{array}$$

which is shown graphically in figure 7.3.

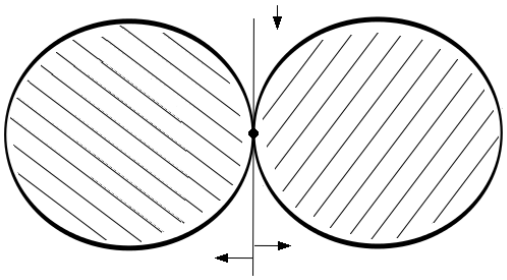

Figure 7.3: A problem admitting no dual values

This problem has a single feasible solution at (1,0). Reduced costs and dual multipliers could never be meaningful indicators of optimality, and indeed are not well-defined for this problem. Intuitively, this arises because the feasible region lacks an interior, and the existence of an interior (also referred to as the Slater condition) is one of several alternative conditions which can be enforced to ensure that such situations do not occur. The other common condition for well-defined duals is that the gradients of the active constraints are linearly independent.

Problems without valid duals do not often arise in practice, but it is important to be aware of the possibility. Analytic detection of such issues is difficult, and they manifest instead in the form of unexpectedly large or otherwise implausible dual values.

CHAPTER 8

Extended MPS file format

One method of inputting a problem to Xpress NonLinear is from a text file which is similar to the normal MPS format matrix file. The Xpress NonLinear file uses *free format* MPS-style data. All the features of normal free-format MPS are supported. There are no changes to the sections except as indicated below.

Note: the use of free-format requires that no name in the matrix contains any leading or embedded spaces and that no name could be interpreted as a number. Therefore, the following names are invalid:

`B 02` because it contains an embedded space;

`1E02` because it could be interpreted as 100 (the scientific or floating-point format number, 1.0E02).

It is possible to use column and row names inlcuding mathematical operators. A variable name **a+b** is valid. However, as an expression **a + b** would be interpreted as the addition of variables **a** and **b** - note the spaces between the variable names - it is best practice to avoid such names when possible. SLP will produce a warning if such names are encountered in the MPS file.

8.1 Formulae

One new feature of the Extended MPS format is the *formula*. A formula is written in much the same way as it would be in any programming language or spreadsheet. It is made up of (for example) constants, functions, the names of variables, and mathematical operators. The formula always starts with an equals sign, and each item (or *token*) is separated from its neighbors by one or more spaces.

Tokens may be one of the following:

- A constant;
- The name of a variable;
- An arithmetic operator "+", "-", "*", "/";
- The exponentiation operator "**" or "^";
- An opening or closing bracket "(" or ")";
- A comma "," separating a list of function arguments;
- The name of a supported internal function such as LOG, SIN, EXP;
- The name of a user-supplied function;

- A colon ":" preceding the return argument indicator of a multi-valued function;
- The name of a return argument from a multi-valued function.

The following are valid formulae:

= *SIN* (*A* / *B*) `SIN` is a recognized internal function which takes one argument and returns one result (the sin of its argument).

= *A* ^ *B* ^ is the exponentiation symbol. Note that the *formula* may have valid syntax but it still may not be possible to evaluate it (for example if *A* = –1 and *B* = 0.5).

= *MyFunc*1 (*C*1 , – *C*2 , *C*3 : 1) `MyFunc1` must be a function which can take three arguments and which returns an array of results. This formula is asking for the first item in the array.

= *MyFunc*2 (*C*1 , – *C*2 , *C*3 : *RVP*) `MyFunc1` must be a function which can take three arguments and which returns an array of results. This formula is asking for the item in the array which is named `RVP`.

The following are not valid formulae:

SIN (*A*) Missing the equals sign at the start

= *SIN*(*A*) No spaces between adjacent tokens

= *A* * * *B* "**" is exponentiation, "* *" (with an embedded space) is not a recognized operation.

= *MyFunc*1 (*C*1 , – *C*2 , *C*3 , 1) If `MyFunc1` is as shown in the previous set of examples, it returns an array of results. The last argument to the function must be delimited by a colon, not a comma, and is the name or number of the item to be returned as the value of the function.

There is no limit in principle to the length of a formula. However, there is a limit on the length of a record read by `XSLPreadprob`, which is 31000 characters. Parsing very long records can be slow, and consideration should be given to pre-parsing them and passing the parsed formula to Xpress NonLinear rather than asking it to parse the formula itself.

8.2 COLUMNS

Normal MPS-style records of the form

column row1 value1 [*row2 value2*]

are supported. Non-linear relationships are modeled by using a formula instead of a constant in the *value1* field. If a formula is used, then only one coefficient can be described in the record (that is, there can be no *row2 value2*). The formula begins with an equals sign ("=") and is as described in the previous section.

A formula must be contained entirely on one record. The maximum record length for files read by `XSLPreadprob` is 31000. Note that there are limits applied by the Optimizer to the lengths of the names of rows and columns.

Variables used in formulae may be included in the `COLUMNS` section as variables, or may exist only as items within formulae. A variable which exists only within formulae is called an *implicit variable*.

Sometimes the non-linearity cannot be written as a coefficient. For example, in the constraint
$Y - LOG(X) = 0$,
`LOG(X)` cannot be written in the form of a coefficient. In such a case, the reserved column name "=" may be used in the first field of the record as shown:

Y	*MyRow*	1
=	*MyRow*	= – *LOG* (*X*)

Effectively, "=" is a column with a fixed activity of 1.0 .

When a file is read by `XSLPreadprob`, more than one coefficient can be defined for the same column/row intersection. As long as there is at most one constant coefficient (one not written as a formula), the coefficients will be added together. If there are two or more constant coefficients for the same intersection, they will be handled by the Optimizer according to its own rules (normally additive, but the objective function retains only the last coefficient).

8.3 BOUNDS

Bounds can be included for variables which are not defined explicitly in the `COLUMNS` section of the matrix. If they are not in the `COLUMNS` section, they must appear as variables within formulae (*implicit variables*). A `BOUNDS` entry for an item which is not a column or a variable will produce a warning message and will be ignored.

Global entities (such as integer variables and members of Special Ordered Sets) must be defined explicitly in the `COLUMNS` section of the matrix. If a variable would otherwise appear only in formulae in coefficients, then it should be included in the `COLUMNS` section with a zero entry in a row (for example, the objective function) which will not affect the result.

8.4 SLPDATA

`SLPDATA` is a new section which holds additional information for solving the non-linear problem using SLP.

Many of the data items have a *setname*. This works in the same way as the `BOUND`, `RANGE` or `RHS` name, in that a number of different values can be given, each with a different set name, and the one which is actually used is then selected by specifying the appropriate setname before reading the problem.

Record type `IV` and the tolerance records `Tx`, `Rx` can have "=" as the variable name. This provides a default value for the record type, which will be used if no specific information is given for a particular variable.

Note that only linear `BOUND` types can be included in the `SLPDATA` section. Bound types for global entities (discrete variables and special ordered sets) must be provided in the normal `BOUNDS` section and the variables must also appear explicitly in the `COLUMNS` section.

All of the items in the SLPDATA section can be loaded into a model using Xpress NonLinear function calls.

8.4.1 CV (Character variable)

CV setname variable value

The `CV` record defines a character variable. This is only required for user functions which have character arguments (for example, file names). The value field begins with the first non-blank character after the variable name, and the value of the variable is made up of all the characters from that point to

the end of the record. The normal free-format rules do not apply in the value field, and all spacing will be retained exactly as in the original record.

Examples:

```
CV  CVSET1  MyCV1  Program Files\MyLibs\MyLib1
```
This defines the character variable named `MyCV1`. It is required because there is an embedded space in the path name which it holds.

```
CV  CVSET1  MyCV1  Program Files\MyLibs\MyLib1
CV  CVSET2  MyCV1  Program Files\MyLibs\MyLib2
```
This defines the character variable named `MyCV1`. There are two definitions, and the appropriate one is selected by setting the string control variable `XSLP_CVNAME` before calling `XSLPreadprob` to load the problem.

8.4.2 DR (Determining row)

DR variable rowname [*weighting*] [*limit*]

The `DR` record defines the *determining row* for a variable.

In most non-linear problems, there are some variables which are effectively defined by means of an equation in terms of other variables. Such an equation is called a *determining row*. If Xpress NonLinear knows the determining rows for the variables which appear in coefficients, then it can provide better linear approximations for the problem and can then solve it more quickly. Optionally, a non-zero integer value can be included in the *weighting* field. Variables which have weights will generally be evaluated in order of increasing weight. Variables without weights will generally be evaluated after those which do have weights. However, if a variable *A* (with or without a weight) is dependent through its determining row on another variable *B*, then *B* will always be evaluated first. The optional *limit* field provides a variable specific value for `XSLP_CASCADENLIMIT`.

Example:

```
DR  X  Row1
```
This defines `Row1` as the determining row for the variable `X`. If `Row1` is
$X - Y * Z = 6$
then Y and Z will be recalculated first before X is recalculated as $Y * Z + 6$.

8.4.3 EC (Enforced constraint)

EC rowname

The `EC` record defines an *enforced constraint*. Penalty error vectors are never added to enforced constraints, so the effect of such constraints is maintained at all times.

Note that this means the *linearized* version of the enforced constraint will be active, so it is important to appreciate that enforcing too many constraints can easily lead to infeasible linearizations which will make it hard to solve the original nonlinear problem.

Example:

```
EC  Row1
```
This defines `Row1` as an enforced constraint. When the SLP is augmented, no penalty error vectors will be added to the constraint, so the linearized version of `Row1` will constrain the linearized problem in the same sense (L, G or E) as the nonlinear version of `Row1` constrains the original nonlinear problem.

8.4.4 FR (Free variable)

FR boundname variable

An FR record performs the same function in the SLPDATA section as it does in the BOUNDS section. It can be used for bounding variables which do not appear as explicit columns in the matrix.

8.4.5 FX (Fixed variable)

FX boundname variable value

An FX record performs the same function in the SLPDATA section as it does in the BOUNDS section. It can be used for bounding variables which do not appear as explicit columns in the matrix.

8.4.6 IV (Initial value)

IV setname variable [*value* | *= formula*]

An IV record specifies the initial value for a variable. All variables which appear in coefficients or terms, or which have non-linear coefficients, should have an IV record.

A formula provided as the initial value for a variable can contain references to other variables. It will be evaluated based on the initial values of those variables (which may themselves be calculated by formula). It is the user's responsibility to ensure that there are no circular references within the formulae. Formulae are typically used to calculate consistent initial values for dependent variables based on the values of independent variables.

If an IV record is provided for the *equals column* (the column whose name is "=" and which has a fixed value of 1.0), the value provided will be used for all SLP variables which do not have an explicit initial value of their own.

If there is no explicit or implied initial value for an SLP variable, the value of control parameter XSLP_DEFAULTIV will be used.

If the initial value is greater than the upper bound of the variable, the upper bound will be used; if the initial value is less than the lower bound of the variable, the lower bound will be used.

If both a formula and a value are provided, then the explicit value will be used.

Example:

```
IV  IVSET1  Col99  1.4971
IV  IVSET2  Col99  2.5793
```

This sets the initial value of column Col99. The initial value to be used is selected using control parameter XSLP_IVNAME. If no selection is made, the first initial value set found will be used.

If Col99 is bounded in the range $1 \leq$ Col99 ≤ 2 then in the second case (when IVSET2 is selected), an initial value of 2 will be used because the value given is greater than the upper bound.

```
IV  IVSET2  Col98  = Col99 * 2
```

This sets the value of Col98 to twice the initial value of Col99 when IVSET2 is the selected initial value set.

8.4.7 LO (Lower bounded variable)

LO boundname variable value

A LO record performs the same function in the SLPDATA section as it does in the BOUNDS section. It can be used for bounding variables which do not appear as explicit columns in the matrix.

8.4.8 Rx, Tx (Relative and absolute convergence tolerances)

Rx setname variable value

Tx setname variable value

The Tx and Rx records (where "x" is one of the defined tolerance types) define specific tolerances for convergence of the variable. See the section "convergence criteria" for a list of convergence tolerances. The same tolerance set name (*setname*) is used for all the tolerance records.

Example:

```
RA  TOLSET1  Col99  0.005
TA  TOLSET1  Col99  0.05
RI  TOLSET1  Col99  0.015
RA  TOLSET1  Col01  0.01
RA  TOLSET2  Col01  0.015
```

These records set convergence tolerances for variables Col99 and Col01. Tolerances RA (relative convergence tolerance), TA (absolute convergence tolerance) and RI (relative impact tolerance) are set for Col99 using the tolerance set named TOLSET1.
Tolerance RA is set for variable Col01 using tolerance sets named TOLSET1 and TOLSET2.
If control parameter XSLP_TOLNAME is set to the name of a tolerance set before the problem is read using XSLPreadprob, then only the tolerances on records with that tolerance set will be used. If XSLP_TOLNAME is blank or not set, then the name of the set on the first tolerance record will be used.

8.4.9 SB (Initial step bound)

SB setname variable value

An SB record defines the initial step bounds for a variable. Step bounds are symmetric (i.e. the bounds on the delta are $-SB \leq delta \leq +SB$). If a value of 1.0E+20 is used (equivalent to XPRS_PLUSINFINITY in programming), the delta will never have step bounds applied, and will almost always be regarded as converged.

If there is no explicit initial step bound for an SLP variable, a value will be estimated either from the size of the coefficients in the initial linearization, or from the values of the variable during the early SLP iterations. The value of control parameter XSLP_DEFAULTSTEPBOUND provides a lower limit for the step bounds in such cases.

If there is no explicit initial step bound, then the closure convergence tolerance cannot be applied to the variable.

Example:

```
SB  SBSET1  Col99  1.5
SB  SBSET2  Col99  7.5
```

This sets the initial step bound of column Col99. The value to be used is selected using control parameter XSLP_SBNAME. If no selection is made, the first step bound set found will be used.

8.4.10 UF (User function)

UF funcname [= *extname*] (*arguments*) *linkage* [= [*param*1] [= [*param*2] [= [*param*3]]]]

A UF record defines a user function.
The definition includes the list of required arguments, and the linkage or calling mechanism. For details of the fields, see the section on Function Declaration in Xpress NonLinear.

Example:

```
UF  MyFunc  (  DOUBLE  ,  INTEGER  )  DLL  =  UserLib
```

This defines a user function called MyFunc. It takes two arguments (an array of type double precision and an array of type integer). The linkage is DLL (free-standing user library or DLL) and the function is in file UserLib.

8.4.11 UP (Free variable)

UP boundname variable value

An `UP` record performs the same function in the `SLPDATA` section as it does in the `BOUNDS` section. It can be used for bounding variables which do not appear as explicit columns in the matrix.

8.4.12 WT (Explicit row weight)

WT rowname value

The `WT` record is a way of setting the initial penalty weighting for a row. If `value` is positive, then the default initial weight is multiplied by the value given. If `value` is negative, then the absolute value will be used instead of the default weight.

Increasing the penalty weighting of a row makes it less attractive to violate the constraint during the SLP iterations.

Examples:

```
WT  Row1  3
```
This changes the initial weighting on `Row1` by multiplying by 3 the default weight calculated by Xpress-SLP during problem augmentation.

```
WT  Row1  -3
```
This sets the initial weighting on `Row1` to 3.

8.4.13 DL (variable specific Determining row cascade iteration Limit)

DL columnname limit

A `DL` record specifiies a variable specific iteration limit to be emposed on the number of iterations when cascading the variable. This can be used to overwrite the setting of `XSLP_CASCADENLIMIT` for a specific variable.

CHAPTER 9

Xpress-SLP Solution Process

This section gives a brief overview of the sequence of operations within Xpress-SLP once the data has been set up. The positions of the possible user callbacks are also shown.

```
Check if problem is an SLP problem or not. Call the appropriate XPRS library function if not, and DONE.
[Call out to user callback if set by XSLPsetcbslpstart]
Augment the matrix (create the linearized structure) if not already done
If determining row data supplied, calculate cascading order and detect determining columns
DO
  [Call out to user callback if set by XSLPsetcbiterstart]
  If previous solution available, pre-process solution
    Execute line search
    [Call out to user callback if set by XSLPsetcbcascadestart]
    Sequentially update values of SLP variables (cascading) and re-calculate coefficients
    For each variable (in a suitable evaluation order):
      Update solution value (cascading) and re-calculate coefficients
      [Call out to user callback if set by XSLPsetcbcascadevar]
    [Call out to user callback if set by XSLPsetcbcascadeend]
  Update penalties
  Update coefficients, bounds and RHS in linearized matrix
  Solve linearized problem using the Xpress Optimizer
  Recover SLP variable and delta solution values
  Test convergence against specified tolerances and other criteria
  For each variable:
    Test convergence against specified tolerances
    [Call out to user callback if set by XSLPsetcbitervar]
  For each variable with a determining column:
    Check value of determining column and fix variable when necessary, or
    [Call out to user callback if set by XSLPsetcbdrcol]
    Reset variable convergence status if a change is made to a variable
  If not all variables have converged, check for other extended convergence criteria
  If the solution has converged, then BREAK
  For each SLP variable:
    Update history
    Reset step bounds
  [Call out to user callback if set by XSLPsetcbiterend]
  Change row types for DC rows as required
  If SLP iteration limit is reached, then BREAK
ENDDO
[Call out to user callback if set by XSLPsetcbslpend]
```

For MISLP (mixed-integer SLP) problems, the above solution process is normally repeated at each

node. The standard procedure for each node is as follows:

```
Initialize node
[Call out to user callback if set by XSLPsetcbprenode]
Solve node using SLP procedure
If an optimal solution is obtained for the node then
   [Call out to user callback if set by XSLPsetcboptnode]
If an integer optimal solution is obtained for the node then
   [Call out to user callback if set by XSLPsetcbintsol]
When node is completed
   [Call out to user callback if set by XSLPsetcbslpnode]
```

When a problem is destroyed, there is a call out to the user callback set by XSLPsetcbdestroy.

9.1 Analyzing the solution process

Xpress-SLP provides a comprehensive set of callbacks to interact with, and to analyze the solution process. However, there are a set of purpose build options that are intended to assist and make the analysis more efficient.

For infeasible problems, it often helps to identify the source of conflict by running XPRESS' Irreducible Infeasibiliy Set (IIS) finder tool. The set found by IIS often helps to either point to a problem in the original model formulation, or if the infeasibility is a result of conflicting step bounds or linearization updates; please see control XSLP_ANALYZE.

Xpress-SLP can collect the various solutions it generates during the solution pool to an XPRS solution pool object. The solution pool is accessible using the XSLP_SOLUTIONPOOL pointer attribute. The solutions to collect are defined by XSLP_ANALYZE. It is also possible to let XSLP write the collected solutions to disk for easier access.

It is often advantageous to trace a certain variable, constraint or a certain property through the solution process. XSLP_TRACEMASK and XSLP_TRACEMASKOPS allows for collecting detailed information during the solution process, without the need to stop XSLP between iterations.

For in depth debugging purposes or support requests, it is possible to create XSLP save files and linearizations at verious iterations, controlled by XSLP_AUTOSAVE and XSLP_ANALYZE.

9.2 The initial point

The solution process is sensitive to the initial values which are selected for variables in the problem, and particularly so for non-convex problems. It is not uncommon for a general nonlinear problem to have a feasible region which is not connected, and in this case the starting point may largely determine which region, connected set, or basin of attraction the final solution belongs to.

Note that it may not always be beneficial to completely specify an initial point, as the solvers themselves may be able to detect suitable starting values for some or all of the variables.

9.3 Derivatives

Both XSLP and Knitro require the availability of derivative information for the constraints and objective function in order to solve a problem. In the Xpress NonLinear framework, several advanced approaches

to the production of both first and second order derivatives (the Jacobian and Hessian matrices) are available, and which approach is used can be controlled by the user.

9.3.1 Finite Differences

The simplest such method is the use of finite differences, sometimes called numerical derivatives. This is a relatively coarse approximation, in which the function is evaluated in a small neighborhood of the point in question. The standard argument from calculus indicates that an increasingly accurate approximation to the derivative of the function will be found as the size of the neighborhood decreases. This argument ignores the effects of floating point arithmetic, however, which can make it difficult to select values sufficiently small to give a good approximation to the function, and yet sufficiently large to avoid substantial numerical error.

The high performance implementation in XSLP makes use of subexpression caching to improve performance, but finite differences are inherently inefficient. They may however be necessary when the function itself is not known in closed form. When analytic approaches cannot be used, due to the use of expensive black box functions which do not provide derivatives (note that XSLP does allow user functions to provide their own derivatives), the cost of function evaluations may become a dominant factor in solve time. It is important to note that each second order numerical derivative costs twice as much as a first order numerical derivative, and this can make XSLP more attractive than Knitro for such problems.

9.3.2 Symbolic Differentiation

Xpress NonLinear will instead provide analytic derivatives where possible, which are both more accurate and more efficient. There are two major approaches to such calculations, and high quality implementations of both are available in this framework.

A symbolic differentiation engine calculates the derivative of an expression in closed form, using its formula representation. This is a very efficient way of recalculating individual entries of the Jacobian, and is the default approach to providing derivative information to XSLP.

9.3.3 Automatic Differentiation

An automatic differentiation engine in contrast can simultaneously compute multiple derivatives by repeated application of the chain rule. This is a very efficient means of calculating large numbers of Hessian entries, and is the default approach to providing derivative information to Knitro.

9.4 Points of inflection

A point of inflection in a given variable occurs when the first and second order partial derivatives with respect to that variable become zero, but there exist nonzero derivatives of higher order. At such points, the approximations the iterative nonlinear methods create do not encapsulate enough information about the behavior of the function, and both first and second order methods may experience difficulties. For example, consider the following problem

$$\begin{array}{ll} \text{minimize} & x^3 \\ \text{subject to} & -1 \leq x \leq 1 \end{array}$$

for which the optimal solution is -1.

When the initial value of x is varied, XSLP and Knitro produce the solutions presented in Table 9.1 for this problem:

```
Starting point:     -1            0            1
-----------------------------------------------------
Knitro :            -1            0 7.34639e-011
SLP    :            -1           -1           -1
```

Figure 9.1: Effect of an inflection point on solution values.

As a second order method, Knitro examines a local quadratic approximation to the function. Starting at both 0 and 1, this approximation will closely resemble the x^2 function, and so the solution will be attracted to zero. For XSLP, which is a first order method, the approximation at 0 will have a zero gradient. However, XSLP can detect this situation and will perform the analysis required to substitute an appropriate small nonzero (placeholder) value for the derivative during the first iterations. As can be seen, this allows XSLP find an optimal solution in all three cases.

This is only one example of the behaviour of these solvers without further tuning. The long steps which XSLP often takes can be both beneficial and harmful in different contexts. For example, if the function to be optimized includes many local minima, it is possible to see the opposite pattern for XSLP and Knitro. Consider

$$\begin{array}{ll} \text{minimize} & x \sin(100x^2) \\ \text{subject to} & -1 \leq x \leq 1 \end{array}$$

which has many local minima. For this problem, the results obtained are presented in Table 9.2:

```
Starting point:     -1            0            1
-----------------------------------------------------
Knitro :      -0.978883            0    -0.720008
SLP    :       0.506366     0.506366     0.506366
```

Figure 9.2: Local solutions for a function with several local optima

In this case the same long steps made by XSLP lead to it finding the an identical, but unfortunate, local optimum no matter which initial point it begins from.

9.5 Trust regions

In a second order method like Knitro, there is a well-defined merit function which can be used to compare solutions, and which provides a measure of the progress being made by the algorithm. This is a significant advantage over first order methods, in which there is generally no such function.

Despite their speed and resilience to points of inflection, first order methods can also experience difficulties at points in which the current approximation is not well posed. Consider

$$\begin{array}{ll} \text{minimize} & x^2 \\ \text{subject to} & x \text{ free} \end{array}$$

at $x = 1$. A naive linearization is simply

$$\begin{array}{ll} \text{minimize} & 2x \\ \text{subject to} & x \text{ free} \end{array}$$

which is unbounded. To address such situations, XSLP will introduce trust regions to model the neighborhood in which the current approximation is believed to be applicable. When coupled with the use of derivative placeholders described in the previous section, this can lead XSLP to initially make large moves from its starting position.

CHAPTER 10

Handling Infeasibilities

By default, Xpress-SLP will include *penalty error vectors* in the augmented SLP structure. This feature adds explicit positive and negative slack vectors to all constraints (or, optionally, just to equality constraints) which include nonlinear coefficients. In many cases, this is itself enough to retain feasibility. There is also an opportunity to add penalty error vectors to all constraints, but this is not normally required.

During cascading (see next section), Xpress-SLP will ensure that the value of a cascaded variable is never set outside its lower and upper bounds (if these have been specified).

10.1 Infeasibility Analysis in the Xpress Optimizer

For problems which can be solved using the Xpress Optimizer, that is LP, convex QP and QCQP and their MIP counterparts, there is normally no difficulty with establishing feasibility. This is because for these convex problem classes, Xpress can produce global solutions, and any problem declared infeasible is globally infeasible. The concept of local infeasibility is primarily of use in the case of nonlinear problems, and in particular non-convex, nonlinear problems.

When the Xpress Optimizer declares a problem to be infeasible, the tools provided with the Xpress Optimizer console can be used to analyse the infeasibility, and hence to subsequently alter the model to overcome it. One important step in this respect is the ability to retrieve an irreducible infeasible set (IIS) (using the `iis` command). An IIS is a statement of a particular conflict in the model between a set of constraints and bounds, which make the problem certainly infeasible. An IIS is minimal in the sense that if any constraint or bound of the IIS were removed from the subproblem represented by the IIS, the resulting (relaxed) subproblem would be feasible. The Xpress Optimizer also contains a tool to identify the minimum weighted violations of constraints or bounds that would make the problem feasible (called `repairinfeas`).

Both `iis` and `repairinfeas` can be applied to any LP, convex QP, or convex QCQP problem, as well as to their mixed integer counterparts. Please refer to the Xpress Optimizer and Mosel reference manuals for more information.

10.2 Managing Infeasibility with Xpress Knitro

Xpress Knitro has three major controls which govern feasibility.

`XKTR_PARAM_FEASTOL`	This is the relative feasibility tolerance applied to a problem.
`XKTR_PARAM_FEASTOLABS`	This is the corresponding absolute feasibility tolerance.
`XKTR_PARAM_INFEASTOL`	This is the tolerance for declaring a problem infeasible.

The feasibility emphasis control, XKTR_PARAM_BAR_FEASIBLE, can be set for models on which Knitro has encountered difficulties in finding a feasible solution. If it is set to get or get_stay, particular emphasis will be placed upon obtaining feasibility, rather than balancing progress toward feasibility and optimality as is the default.

If one of the built-in interior point methods is used, as determined by XKTR_PARAM_ALGORITHM, the feasibility emphasis control can force the iterates to strictly satisfy inequalities. It does not, however, require Knitro to satisfy all equality constraints at intermediate iterates.

The migration between a pure search for feasibility, and a balanced approach to feasibility and optimality, may be further fine tuned by using the XKTR_PARAM_BAR_SWITCHRULE control. Should a model still fail to converge to a feasible solution, the XKTR_PARAM_BAR_PENCONS control may be used to instruct Knitro to introduce penalty breakers of its own. This option has similar behaviour to the corresponding option in XSLP.

10.3 Managing Infeasibility with Xpress-SLP

There are two sources of infeasibility when XSLP is used

1. Infeasibility introduced by the error of the approximation, most noticeable when significant steps are made in the linearization.
2. Infeasibility introduced by the activation of penalty breakers, where it was not otherwise possible to make a meaningful step in the linearization.

The infeasibility induced by the former diminishes as the solution converges, provided mild assumptions regarding the continuity of the functions describing the model are satisfied. The focus of any analysis of infeasibility in XSLP must therefore most often be on the penalty breakers (also called error vectors).

For some problems, Xpress-SLP may terminate with a solution which is not sufficiently feasible for use in a desired application. The first controls to use to try to resolve such an issue are

XSLP_ECFTOL_A — The absolute linearization feasibility tolerance is compared for each constraint in the original, nonlinear problem to its violation by the current solution.

XSLP_ECFTOL_R — The relative linearization feasibility tolerance is compared for each constraint in the original, nonlinear problem to its violation by the current solution, relative to the maximum absolute value of the positive and negative contributions to the constraint.

10.4 Penalty Infeasibility Breakers in XSLP

Convergence will automatically address any errors introduced by movement within the linearization. When only small movements occur in the solution, then for differentiable functions the drift resulting from motion on the linearization is also limited.

However, it is not always possible to stay within the linearization and still make an improving step. XSLP is often able to resolve such situations automatically by the introduction of penalty infeasibility breakers. These allow the solver to violate the linearized constraints by a small amount. Such variables are associated with large cost penalties in the linearized problems, which prevents the solution process from straying too far from the approximated feasible region.

Note that if penalty breakers are required, the solution process may be very sensitive to the choice of cost penalties placed on the breakers. In most cases, XSLP's constraint analysis will automatically

identify appropriate penalties as needed for each row, but for some problems additional tuning might be required.

Xpress-SLP will attempt to force all penalty breakers to zero in the limit by associating a substantial cost with them in the objective function. Such costs will be increased repeatedly should the penalty breaker remain non-zero over a period of time. The current penalty cost for all such variables is available as XSLP_CURRENTERRORCOST. The control XSLP_ERRORCOST determines the initial value for this cost, while the XSLP_ERRORCOSTFACTOR controls the factor by which it increases if active error vectors remain. The maximum value of the penalty is determined by the control XSLP_ERRORMAXCOST. If the maximum error cost is reached, it is unlikely that XSLP will converge. It is possible in this situation to terminate the solve, by setting bit 11 of XSLP_ALGORITHM.

Some problems may be sensitive to the initial value of XSLP_ERRORCOST. If this value is too small relative to the original objective in the model, feasibility will not be sufficiently strongly encouraged during the solution process. This can cause SLP to explore highly infeasible solutions in the early stages, since the original objective will dominate any consideration of feasibility. It is even possible in this case for unboundedness of the linearizations to occur, although SLP is capable of automatic recovery from such a situation.

When the initial penalty cost is too high, the penalty term will dominate the objective. This in turn will may lead to initially low quality solutions being explored, with the attendant possibility of numerical errors accumulating. The control XSLP_OBJTOPENALTYCOST guides the process which selects an automatic value for XSLP_ERRORCOST, but determining such a value analytically can be difficult. For some difficult problems, there may be significant benefits to selecting the value directly.

Often for infeasible problems, the contribution of the individual constraints to the overall infeasibility is non-uniform. XSLP can automatically associate a weight with each row based upon the magnitude of the terms in the constraint. It is both possible to refine these weights, or alternatively to allow XSLP update them dynamically. The latter case is called escalation, and is controlled by bit 8 of XSLP_ALGORITHM.

Devising appropriate weights manually can be difficult, and in most cases it is preferable to leave the identification of these values to Xpress-SLP. However if it is necessary to do, the output of XSLP may provide hints as to appropriate values if detailed logging is enabled. This can be turned on with XSLP_LOG. The most important points in such output are the active error vectors at each iteration, where the most attractive constraints to modify are those which occur regularly in the log in association with non-zero error vectors.

CHAPTER 11

Cascading

Cascading is the process of recalculating the values of SLP variables to be more consistent with each other. The procedure involves sequencing the designated variables in order of dependence and then, starting from the current solution values, successively recalculating values for the variables, and modifying the stored solution values as required. Normal cascading is only possible if a *determining row* can be identified for each variable to be recalculated. A determining row is an equality constraint which uniquely determines the value of a variable in terms of other variables whose values are already known. Any variable for which there is no determining row will retain its original solution value. Defining a determining row for a column automatically makes the column into an SLP variable.

In extended MPS format, the SLPDATA record type "DR" is used to provide information about determining rows.

In the Xpress NonLinear function library, functions `XSLPaddvars`, `XSLPloadvars`, and `XSLPchgvar` allow the definition of a determining row for a column.

The cascading procedure is as follows:

- Produce an order of evaluation to ensure that variables are cascaded after any variables on which they are dependent.
- After each SLP iteration, evaluate the columns in order, updating coefficients only as required. If a determining row cannot calculate a new value for the SLP variable (for example, because the coefficient of the variable evaluates to zero), then the current value may be left unchanged, or (optionally) the previous value can be used instead.
- If a feedback loop is detected (that is, a determining row for a variable is dependent indirectly on the value of the variable), the evaluation sequence is carried out in the order in which the variables are weighted, or the order in which they are encountered if there is no explicit weighting.
- Check the step bounds, individual bounds and cascaded values for consistency. Adjust the cascaded result to ensure it remains within any explicit or implied bounds.

Normally, the solution value of a variable is exactly equal to its assumed value plus the solution value of its delta. Occasionally, this calculation is not exact (it may vary by up to the LP feasibility tolerance) and the difference may cause problems with the SLP solution path. This is most likely to occur in a quadratic problem when the quadratic part of the objective function contains SLP variables. Xpress NonLinear can re-calculate the value of an SLP variable to be equal to its assumed value plus its delta, rather than using the solution value itself.

`XSLP_CASCADE` is a bitmap which determines whether cascading takes place and whether the recalculation of solution values is extended from the use of determining rows to recalculation of the solution values for all SLP variables, based on the assumed value and the solution value of the delta.

In the following table, in the definitions under **Category**, *error* means the difference between the solution value and the assumed value plus the delta value. Bit settings in `XSLP_CASCADE` are used to determine which category of variable will have its value recalculated as follows:

Bit	Constant name	Category
0	XSLP_CASCADE_ALL	SLP variables with determining rows
1	XSLP_CASCADE_COEF_VAR	Variables appearing in coefficients where the error is greater than the feasibility tolerance
2	XSLP_CASCADE_ALL_COEF_VAR	Variables appearing in coefficients where the error is greater than 1.0E-14
3	XSLP_CASCADE_STRUCT_VAR	Variables not appearing in coefficients where the error is greater than the feasibility tolerance
4	XSLP_CASCADE_ALL_STRUCT_VAR	Variables not appearing in coefficients where the error is greater than 1.0E-14

In the presence of determining rows that include instantiated functions, SLP can attempt to group the corresponding variables together in the cascading order. This can be achieved by setting

Bit	Constant name	Effect
0	XSLP_CASCADE_SECONDARY_GROUPS	Create secondary order groupping DR rows with instantiated user functions together in the order

11.1 Determining rows and determining columns

Normally, Xpress-SLP automatically identifies if the constraint selected as determining row for a variable defines the value of the SLP variable which it determines or not. However, in certain situations, the value of a single another column determines if the determing row defines the variable or not; such a column is called the determining column for the variable.

This situation is typical when the determined and determining column form a bilinear term: x * y + F(Z) = 0 where y is the determined variable, Z is a set of other variables not including x or y, and F is an arbitrary function; in this case x is the determining column. These variable pairs are detected automatically. In case the absolute value of x is smaller than XSLP_DRCOLTOL, then variable y will not be cascaded, instead its value will be fixed and kept at its current value until the value of x becomes larger than the threshold.

Alternatively, the handling of variables for which a determining column has been identified can be customized by using a callback, see XSLPsetcbdrcol.

CHAPTER 12

Convergence criteria

12.1 Convergence criteria

In Xpress-SLP there are two levels of convergence criteria. On the higher level, convergence is driven by the target relative feasibility / validation control XSLP_VALIDATIONTARGET_R, and the target first order validation tolerance XSLP_VALIDATIONTARGET_K. These high level targets drive the traditional SLP convergence measures, of which there are three types for testing test convergence:

- Strict convergence tests on variables
- Extended convergence tests on variables
- Convergence tests on the solution overall

12.2 Convergence overview

12.2.1 Strict Convergence

Three tolerances in XSLP are used to determine whether an individual variable has strictly converged, that is they describe the numerical behaviour of convergence in the formal, mathematical sense.

XSLP_CTOL — The closure tolerance is compared against the movement of a variable relative to its initial step bound.

XSLP_ATOL_A — The absolute delta tolerance is compared against the absolute movement of a variable.

XSLP_ATOL_R — The relative delta tolerance is compared against the movement of a variable relative to its initial value.

12.2.2 Extended Convergence

There are six tolerances in XSLP used to determine whether an individual variable has converged according to the extended definition. These tests essentially measure the quality of the linearization, including the effect of changes to the nonlinear terms that contribute to a variable in the linearization. In order to be deemed to have converged in the extended sense, all terms in which it appears must satisfy at least one of the following:

XSLP_MTOL_A — The absolute matrix tolerance is compared against the approximation error relative only to the absolute value of the variable.

XSLP_MTOL_R The relative matrix tolerance is compared against the approximation error relative to the size of the nonlinear term before any step is taken.

XSLP_ITOL_A The absolute impact tolerance is compared against the approximation error of the nonlinear term.

XSLP_ITOL_R The relative impact tolerance is compared against the approximation error relative to the positive and negative contributions to each constraint.

XSLP_STOL_A The absolute slack impact tolerance is compared against the approximation error, but only for non-binding constraints, which is to say those for which the marginal value is small (as defined by XSLP_MVTOL).

XSLP_STOL_R The relative slack impact tolerance is compared against the approximation error relative to the term's contribution to its constraints, but only for non-binding constraints, which is to say those for which the marginal value is small (as defined by XSLP_MVTOL).

12.2.3 Stopping Criterion

The stopping criterion requires that all variables in the problem have converged in one of the three senses. Detailed information regarding the conditions under which XSLP has terminated can be obtained from the XSLP_STATUS solver attribute. Note that a solution is deemed to have fully converged if all variables have converged in the strict sense. If all variables have converged either in the strict or extended sense, and there are no active step bounds, then the solution is called a practical solution. In contrast, the solution may be called converged if it is feasible and the objective is no longer improving.

The following four control sets can be applied by XSLP to determine whether the objective is stationary, depending on the convergence control parameter XSLP_CONVERGENCEOPS:

VTOL This is the baseline static objective function tolerance, which is compared against the change in the objective over a given number of iterations, relative to the average objective value. Satisfaction of VTOL does not imply convergence of the variables.

- XSLP_VCOUNT This is the number of iterations over which to apply this measure of static objective convergence.
- XSLP_VLIMIT The static objective function test is applied only after at least XSLP_VLIMIT + XSLP_SBSTART XSLP iterations have taken place.
- XSLP_VTOL_A This is the absolute tolerance which is compared to the range of the objective over the last XSLP_VLIMIT iterations.
- XSLP_VTOL_R This is the used for a scaled version of the absolute test which considers the average size of the absolute value of the objective over the previous XSLP_VLIMIT iterations.

OTOL This static objective function tolerance is applied when there are no unconverged variables in active constraints, although some variables with active step bounds might remain. It is compared to the change in the objective over a given number of iterations, relative to the average objective value.

- XSLP_OCOUNT This is the number of iterations over which to apply this measure of static objective convergence.
- XSLP_OTOL_A This is the absolute tolerance which is compared to the range of the objective over the last XSLP_OCOUNT iterations.

XSLP_OTOL_R — This is used for a scaled version of the absolute test which considers the average size of the absolute value of the objective over the previous XSLP_OCOUNT iterations.

XTOL — This static objective function tolerance is applied when a practical solution has been found. It is compared against the change in the objective over a given number of iterations, relative to the average objective value.

- XSLP_XCOUNT — This is the number of iterations over which to apply this measure of static objective convergence.
- XSLP_XLIMIT — This is the maximum number of iterations which can have occurred for this static objective function test to be applied. Once this number is exceeded, the solution is deemed to have converged if all the variables have converged by the strict or extended criteria.
- XSLP_XTOL_A — This is the absolute tolerance which is compared to the range of the objective function over the last XSLP_XLIMIT iterations.
- XSLP_XTOL_R — This is used for a scaled version of the absolute test which considers the average size of the absolute value of the objective over the last XSLP_XLIMIT iterations.

WTOL — The extended convergence continuation tolerance is applied when a practical solution has been found. It is compared to the change in the objective during the previous iteration.

- XSLP_WCOUNT — This is number of iterations over which to calculate this measure of static objective convergence in the relative version of the test.
- XSLP_WTOL_A — This is the absolute tolerance which is compared to the change in the objective in the previous iteration.
- XSLP_WTOL_R — This is used for a scaled version of the test which considers the average size of the absolute value of the objective over the last XSLP_WCOUNT iterations.

12.2.4 Step Bounding

Step bounding in XSLP can be activated in two cases. It may be enabled adaptively in response to variable oscillation, or it may be enabled by after XSLP_SBSTART iterations, by setting XSLP_ALGORITHM appropriately. Two major controls define the behaviour of step bounds:

XSLP_SBSTART — This defines the number of iterations which must occur before XSLP may apply non-essential step bounding. When a linearization is unbounded, XSLP will introduce step bounding regardless of the value of this control.

XSLP_DEFAULTSTEPBOUND — This is the initial size of the step bounds introduced. Depending upon the value of XSLP_ALGORITHM, XSLP may use the iterations before XSLP_SBSTART to refine this initial value on a per variable basis.

12.3 Convergence: technical details

In the following sections we shall use the subscript *0* to refer to values used to build the linear approximation (the *assumed* value) and the subscript *1* to refer to values in the solution to the linear approximation (the *actual* value). We shall also use δ to indicate the change between the assumed and the actual values, so that for example:
$\delta X = X_1 - X_0$.

The tests are described in detail later in this section. Tests are first carried out on each variable in turn, according to the following sequence:

Strict convergence criteria:

1. **Closure tolerance** (CTOL).
   This tests δX against the initial step bound of X.
2. **Delta tolerance** (ATOL)
   This tests δX against X_0.

If the strict convergence tests fail for a variable, it is tested against the extended convergence criteria:

3. **Matrix tolerance** (MTOL)
   This tests whether the effect of a matrix coefficient is adequately approximated by the linearization. It tests the error against the magnitude of the effect.
4. **Impact tolerance** (ITOL)
   This tests whether the effect of a matrix coefficient is adequately approximated by the linearization. It tests the error against the magnitude of the contributions to the constraint.
5. **Slack impact tolerance** (STOL)
   This tests whether the effect of a matrix coefficient is adequately approximated by the linearization and is applied only if the constraint has a negligible marginal value (that is, it is regarded as "not constraining"). The test is the same as for the impact tolerance, but the tolerance values may be different.

The three extended convergence tests are applied simultaneously to all coefficients involving the variable, and each coefficient must pass at least one of the tests if the variable is to be regarded as converged. If any coefficient fails the test, the variable has not converged.

Regardless of whether the variable has passed the system convergence tests or not, if a convergence callback function has been set using `XSLPsetcbitervar` then it is called to allow the user to determine the convergence status of the variable.

6. **User convergence test**
   This test is entirely in the hands of the user and can return one of three conditions: the variable has converged on user criteria; the variable has not converged; or the convergence status of the variable is unchanged from that determined by the system.

Once the tests have been completed for all the variables, there are several possibilities for the convergence status of the solution:

(a) All variables have converged on strict criteria or user criteria.

(b) All variables have converged, some on extended criteria, and there are no active step bounds (that is, there is no delta vector which is at its bound and has a significant reduced cost).

(c) All variables have converged, some on extended criteria, and there are active step bounds (that is, there is at least one delta vector which is at its bound and has a significant reduced cost).

(d) Some variables have not converged, but these have non-constant coefficients only in constraints which are not active (that is, the constraints do not have a significant marginal value);

(e) Some variables have not converged, and at least one has a non-constant coefficient in an active constraint (that is, the constraint has a significant marginal value);

If (a) is true, then the solution has converged on *strict convergence criteria*.

If (b) is true, then the solution has converged on *extended convergence criteria*.

If (c) is true, then the solution is a *practical* solution. That is, the solution is an optimal solution to the linearization and, within the defined tolerances, it is a solution to the original nonlinear problem. It is possible to accept this as the solution to the nonlinear problem, or to continue optimizing to see if a better solution can be obtained.

If (d) or (e) is true, then the solution has not converged. Nevertheless, there are tests which can be applied to establish whether the solution can be regarded as converged, or at least whether there is benefit in continuing with more iterations.

The first convergence test on the solution simply tests the variation in the value of the objective function over a number of SLP iterations:

7. **Objective function convergence test 1** (VTOL)
   This test measures the range of the objective function (the difference between the maximum and minimum values) over a number of SLP iterations, and compares this against the magnitude of the average objective function value. If the range is small, then the solution is deemed to have converged.

Notice that this test says nothing about the convergence of the variables. Indeed, it is almost certain that the solution is not in any sense a practical solution to the original nonlinear problem. However, experience with a particular type of problem may show that the objective function does settle into a narrow range quickly, and is a good indicator of the ultimate *value* obtained. This test can therefore be used in circumstances where only an estimate of the solution value is required, not how it is made up. One example of this is where a set of schedules is being evaluated. If a quick estimate of the value of each schedule can be obtained, then only the most profitable or economical ones need be examined further.

If the convergence status of the variables is as in (d) above, then it may be that the solution is practical and can be regarded as converged:

8. **Objective function convergence test 2** (XTOL)
   If there are no unconverged values in active constraints, then the inaccuracies in the linearization (at least for small errors) are not important. If a constraint is not active, then deleting the constraint does not change the feasibility or optimality of the solution. The convergence test measures the range of the objective function (the difference between the maximum and minimum values) over a number of SLP iterations, and compares this against the magnitude of the average objective function value. If the range is small, then the solution is deemed to have converged.

The difference between this test and the previous one is the requirement for the convergence status of the variables to be (d).

Unless test 7 (VTOL) is being applied, if the convergence status of the variables is (e) then the solution has not converged and another SLP iteration will be carried out.

If the convergence status is (c), then the solution is practical. Because there are active step bounds in the solution, a "better" solution would be obtained to the linearization if the step bounds were relaxed. However, the linearization becomes less accurate the larger the step bounds become, so it might not be the case that a better solution would also be achieved for the nonlinear problem. There are two convergence tests which can be applied to decide whether it is worth continuing with more SLP iterations in the hope of improving the solution:

9. **Objective function convergence test 3** (OTOL)
   If all variables have converged (even if some are converged on extended criteria only, and some of those have active step bounds), the solution is a practical one. If the objective function has not

changed significantly over the last few iterations, then it is reasonable to suppose that the solution will not be significantly improved by continuing with more SLP iterations. The convergence test measures the range of the objective function (the difference between the maximum and minimum values) over a number of SLP iterations, and compares this against the magnitude of the average objective function value. If the range is small, then the solution is deemed to have converged.

10. **Extended convergence continuation test** (WTOL)
Once a solution satisfying (c) has been found, we have a practical solution against which to compare solution values from later SLP iterations. As long as there has been a significant improvement in the objective function, then it is worth continuing. If the objective function over the last few iterations has failed to improve over the practical solution, then the practical solution is restored and the solution is deemed to have converged.

The difference between tests 9 and 10 is that 9 (OTOL) tests for the objective function being stable, whereas 10 (WTOL) tests whether it is actually improving. In either case, if the solution is deemed to have converged, then it has converged to a practical solution.

12.3.1 Closure tolerance (CTOL)

If an initial step bound is provided for a variable, then the closure test measures the significance of the magnitude of the delta compared to the magnitude of the initial step bound. More precisely:

Closure test:

$$ABS(\delta X) \leq B * XSLP\_CTOL$$

where B is the initial step bound for X. If no initial step bound is given for a particular variable, the closure test is not applied to that variable, even if automatic step bounds are applied to it during the solution process.

If a variable passes the closure test, then it is deemed to have converged.

12.3.2 Delta tolerance (ATOL)

The simplest tests for convergence measure whether the actual value of a variable in the solution is significantly different from the assumed value used to build the linear approximation.

The absolute test measures the significance of the magnitude of the delta; the relative test measures the significance of the magnitude of the delta compared to the magnitude of the assumed value. More precisely:

Absolute delta test:

$$ABS(\delta X) \leq XSLP\_ATOL\_A$$

Relative delta test:

$$ABS(\delta X) \leq X_0 * XSLP\_ATOL\_R$$

If a variable passes the absolute or relative delta tests, then it is deemed to have converged.

12.3.3 Matrix tolerance (MTOL)

The matrix tests for convergence measure the linearization error in the effect of a coefficient. The *effect* of a coefficient is its value multiplied by the activity of the column in which it appears.

$$E = V * C$$

where V is the activity of the matrix column in which the coefficient appears, and C is the value of the coefficient. The linearization approximates the effect of the coefficient as

$$E = V * C_0 \; + \; \delta X * C'_0$$

where V is as before, C_0 is the value of the coefficient C calculated using the assumed values for the variables and C'_0 is the value of $\frac{\partial C}{\partial X}$ calculated using the assumed values for the variables.

The error in the effect of the coefficient is given by

$$\delta E = V_1 * C_1 - (V_1 * C_0 + \delta X * C'_0)$$

Absolute matrix test:

$$ABS(\delta E) \leq XSLP\_MTOL\_A$$

Relative matrix test:

$$ABS(\delta E) \leq V_0 * X_0 * XSLP\_MTOL\_R$$

If all the coefficients which involve a given variable pass the absolute or relative matrix tests, then the variable is deemed to have converged.

12.3.4 Impact tolerance (ITOL)

The impact tests for convergence also measure the linearization error in the effect of a coefficient. The effect of a coefficient was described in the previous section. Whereas the matrix test compares the error against the magnitude of the coefficient itself, the impact test compares the error against a measure of the magnitude of the constraint in which it appears. All the elements of the constraint are examined: for each, the contribution to the constraint is evaluated as the element multiplied by the activity of the vector in which it appears; it is then included in a *total positive contribution* or *total negative contribution* depending on the sign of the contribution. If the predicted effect of the coefficient is positive, it is tested against the total positive contribution; if the effect of the coefficient is negative, it is tested against the total negative contribution.

As in the matrix tests, the predicted effect of the coefficient is

$$V * C_0 + \delta X * C'_0$$

and the error is

$$\delta E = V_1 * C_1 - (V_1 * C_0 + \delta X * C'_0)$$

Absolute impact test:

$$ABS(\delta E) \leq XSLP\_ITOL\_A$$

Relative impact test:

$$ABS(\delta E) \leq T_0 * XSLP\_ITOL\_R$$

where

$$T_0 = ABS(\sum_{v \in V} v_0 * c_0)$$

c is the value of the constraint coefficient in the vector v; V is the set of vectors such that $v_0 * c_0 > 0$ if E is positive, or the set of vectors such that $v_0 * c_0 < 0$ if E is negative.

If a coefficient passes the matrix test, then it is deemed to have passed the impact test as well. If all the coefficients which involve a given variable pass the absolute or relative impact tests, then the variable is deemed to have converged.

12.3.5 Slack impact tolerance (STOL)

This test is identical in form to the impact test described in the previous section, but is applied only to constraints whose marginal value is less than `XSLP_MVTOL`. This allows a weaker test to be applied where the constraint is not, or is almost not, binding.

Absolute slack impact test:

$$ABS(\delta E) \leq XSLP\_STOL\_A$$

Relative slack impact test:

$$ABS(\delta E) \leq T_0 * XSLP\_STOL\_R$$

where the items in the expressions are as described in the previous section, and the tests are applied only when

$$ABS(\pi_i) < XSLP\_MVTOL$$

where π_i is the marginal value of the constraint.

If all the coefficients which involve a given variable pass the absolute or relative matrix, impact or slack impact tests, then the variable is deemed to have converged.

12.3.6 Fixed variables due to determining columns smaller than threshold (FX)

Variables having a determining column, that are temporarily fixed due to the absolute value of the determining column being smaller than the threshold `XSLP_DRCOLTOL` are regarded as converged.

12.3.7 User-defined convergence

Regardless of what the Xpress-SLP convergence tests have said about the status of an individual variable, it is possible for the user to set the convergence status for a variable by using a function defined through the `XSLPsetcbitervar` callback registration procedure. The callback function returns an integer result *S* which is interpreted as follows:

$S < 0$	mark variable as unconverged
$S = 0$	leave convergence status of variable unchanged
$S \geq 11$	mark variable as converged with status S

Values of *S* in the range 1 to 10 are interpreted as meaning convergence on the standard system-defined criteria.

If a variable is marked by the user as converged, it is treated as if it has converged on strict criteria.

12.3.8 Static objective function (1) tolerance (VTOL)

This test does not measure convergence of individual variables, and in fact does not in any way imply that the solution has converged. However, it is sometimes useful to be able to terminate an optimization once the objective function appears to have stabilized. One example is where a set of possible schedules are being evaluated and initially only a good estimate of the likely objective function value is required, to eliminate the worst candidates.

The variation in the objective function is defined as

$$\delta Obj = MAX_{Iter}(Obj) - MIN_{Iter}(Obj)$$

where *Iter* is the XSLP_VCOUNT most recent SLP iterations and *Obj* is the corresponding objective function value.

Absolute static objective function (3) test:

$$ABS(\delta Obj) \leq XSLP\_VTOL\_A$$

Relative static objective function (3) test:

$$ABS(\delta Obj) \leq AVG_{Iter}(Obj) * XSLP\_VTOL\_R$$

The static objective function (3) test is applied only after at least XSLP_VLIMIT + XSLP_SBSTART SLP iterations have taken place. Where step bounding is being applied, this ensures that the test is not applied until after step bounding has been introduced.

If the objective function passes the relative or absolute static objective function (3) test then the solution will be deemed to have converged.

12.3.9 Static objective function (2) tolerance (OTOL)

This test does not measure convergence of individual variables. Instead, it measures the significance of the changes in the objective function over recent SLP iterations. It is applied when all the variables interacting with active constraints (those that have a marginal value of at least XSLP_MVTOL) have converged. The rationale is that if the remaining unconverged variables are not involved in active constraints and if the objective function is not changing significantly between iterations, then the solution is more-or-less practical.

The variation in the objective function is defined as

$$\delta Obj = MAX_{Iter}(Obj) - MIN_{Iter}(Obj)$$

where *Iter* is the XSLP_OCOUNT most recent SLP iterations and *Obj* is the corresponding objective function value.

Absolute static objective function (2) test:

$$ABS(\delta Obj) \leq XSLP\_OTOL\_A$$

Relative static objective function (2) test:

$$ABS(\delta Obj) \leq AVG_{Iter}(Obj) * XSLP\_OTOL\_R$$

If the objective function passes the relative or absolute static objective function (2) test then the solution is deemed to have converged.

12.3.10 Static objective function (3) tolerance (XTOL)

It may happen that all the variables have converged, but some have converged on extended criteria (MTOL, ITOL or STOL) and at least one of these is at its step bound. It is therefore possible that an improved result could be obtained by taking another SLP iteration. However, if the objective function has already been stable for several SLP iterations, then there is less likelihood of an improved result, and the converged solution can be accepted.

The static objective function (1) test measures the significance of the changes in the objective function over recent SLP iterations. It is applied when all the variables have converged, but some have converged on extended criteria (MTOL, ITOL or STOL) and at least one of these is at its step bound. Because all

the variables have converged, the solution is already converged but the fact that some variables are at their step bound limit suggests that the objective function could be improved by going further.

The variation in the objective function is defined as

$$\delta Obj = MAX_{Iter}(Obj) - MIN_{Iter}(Obj)$$

where *Iter* is the `XSLP_XCOUNT` most recent SLP iterations and *Obj* is the corresponding objective function value.

Absolute static objective function (1) test:

$$ABS(\delta Obj) \leq XSLP\_XTOL\_A$$

Relative static objective function (1) test:

$$ABS(\delta Obj) \leq AVG_{Iter}(Obj) * XSLP\_XTOL\_R$$

The static objective function (1) test is applied only until `XSLP_XLIMIT` SLP iterations have taken place. After that, if all the variables have converged on strict or extended criteria, the solution is deemed to have converged.

If the objective function passes the relative or absolute static objective function (1) test then the solution is deemed to have converged.

12.3.11 Extended convergence continuation tolerance (WTOL)

This test is applied after a converged solution has been found where at least one variable has converged on extended criteria and is at its step bound limit. As described under XTOL above, it is possible that by continuing with additional SLP iterations, the objective function might improve. The extended convergence continuation test measures whether any improvement is being achieved. If not, then the last converged solution will be restored and the optimization will stop.

For a maximization problem, the improvement in the objective function at the current iteration compared to the objective function at the last converged solution is given by:

$$\delta Obj = Obj - ConvergedObj$$

(for a minimization problem, the sign is reversed).

Absolute extended convergence continuation test:

$$\delta Obj > XSLP\_WTOL\_A$$

Relative extended convergence continuation test:

$$\delta Obj > ABS(ConvergedObj) * XSLP\_WTOL\_R$$

A solution is deemed to have a significantly better objective function value than the converged solution if δObj passes the relative *and* absolute extended convergence continuation tests.

When a solution is found which converges on extended criteria and with active step bounds, the solution is saved and SLP optimization continues until one of the following:

- a new solution is found which converges on some other criterion, in which case the SLP optimization stops with this new solution

- a new solution is found which converges on extended criteria and with active step bounds, and which has a significantly better objective function, in which case this is taken as the new saved solution
- none of the `XSLP_WCOUNT` most recent SLP iterations has a significantly better objective function than the saved solution, in which case the saved solution is restored and the SLP optimization stops

CHAPTER 13

Xpress-SLP Structures

13.1 SLP Matrix Structures

Xpress-SLP augments the original matrix to include additional rows and columns to model some or all of the variables involved in nonlinear relationships, together with first-order derivatives.

The amount and type of augmentation is determined by the bit map control variable XSLP_AUGMENTATION:

Bit 0	Minimal augmentation. All SLP variables appearing in coefficients or matrix entries are provided with a corresponding update row and delta vector.
Bit 1	Even-handed augmentation. All nonlinear expressions are converted into terms. All SLP variables are provided with a corresponding update row and delta vector.
Bit 2	Create penalty error vectors (+ and -) for each equality row of the original problem containing a nonlinear coefficient or term. This can also be implied by the setting of bit 3.
Bit 3	Create penalty error vectors (+ and/or - as required) for each row of the original problem containing a nonlinear coefficient or term. Setting bit 3 to 1 implies the setting of bit 2 to 1 even if it is not explicitly carried out.
Bit 4	Create additional penalty delta vectors to allow the solution to exceed the step bounds at a suitable penalty.
Bit 8	Implement step bounds as constraint rows.
Bit 9	Create error vectors (+ and/or - as required) for each constraining row of the original problem.

If Bits 0-1 are not set, then Xpress-SLP will use standard augmentation: all SLP variables (appearing in coefficients or matrix entries, or variables with non constant coefficients) are provided with a corresponding update row and delta vector.

To avoid too many levels of super- and sub- scripting, we shall use X, Y and Z as variables, $F()$ as a function, and R as the row name. In the matrix structure, column and row names are shown *in italics*.

X_0 is the current estimate ("assumed value") of X. $F'_x(...)$ is the first derivative of F with respect to X.

13.1.1 Augmentation of a nonlinear coefficient

Original matrix structure

	X
R	$F(Y,Z)$

Matrix structure: minimal augmentation (XSLP_AUGMENTATION=1)

	X	***Y***	***Z***	***dY***	***dZ***		
R	$F(Y_0, Z_0)$			$X_0 * F'_y(Y_0, Z_0)$	$X_0 * F'_z(Y_0, Z_0)$		
uY		1		−1		=	Y_0
uZ			1		−1	=	Z_0

The original nonlinear coefficient (X,R) is replaced by its evaluation using the assumed values of the independent variables.

Two vectors and one equality constraint for each independent variable in the coefficient are created if they do not already exist.

The new vectors are:

- The SLP variable (e.g. *Y*)
- The SLP delta variable (e.g. *dY*)

The new constraint is the SLP update row (e.g. *uY*) and is always an equality. The only entries in the update row are the +1 and -1 for the SLP variable and delta variable respectively. The right hand side is the assumed value for the SLP variable.

The entry in the original nonlinear constraint row for each independent variable is the first-order partial derivative of the implied term $X * F(Y, Z)$, evaluated at the assumed values.

The delta variables are bounded by the current values of the corresponding step bounds.

Matrix structure: standard augmentation (XSLP_AUGMENTATION=0)

	X	***Y***	***Z***	***dX***	***dY***	***dZ***		
R	$F(Y_0, Z_0)$				$X_0 * F'_y(Y_0, Z_0)$	$X_0 * F'_z(Y_0, Z_0)$		
uX	1			−1			=	X_0
uY		1			−1		=	Y_0
uZ			1			−1	=	Z_0

The original nonlinear coefficient (X,R) is replaced by its evaluation using the assumed values of the independent variables.

Two vectors and one equality constraint for each independent variable in the coefficient are created if they do not already exist.

The new vectors are:

- The SLP variable (e.g. *Y*)
- The SLP delta variable (e.g. *dY*)

The new constraint is the SLP update row (e.g. *uY*) and is always an equality. The only entries in the update row are the +1 and -1 for the SLP variable and delta variable respectively. The right hand side is the assumed value for the SLP variable.

The entry in the original nonlinear constraint row for each independent variable is the first-order partial derivative of the implied term $X * F(Y, Z)$, evaluated at the assumed values.

The delta variables are bounded by the current values of the corresponding step bounds.

One new vector and one new equality constraint are created for the variable containing the nonlinear coefficient.

The new vector is:

- The SLP delta variable (e.g. *dX*)

The new constraint is the SLP update row (e.g. *uX*) and is always an equality. The only entries in the update row are the +1 and -1 for the original variable and delta variable respectively. The right hand side is the assumed value for the original variable.

The delta variable is bounded by the current values of the corresponding step bounds.

Matrix structure: even-handed augmentation (XSLP_AUGMENTATION=2)

	=	X	Y	Z	dX	dY	dZ		
R	$X_0 * F(Y_0, Z_0)$				$F(Y_0, Z_0)$	$X_0 * F'_y(Y_0, Z_0)$	$X_0 * F'_z(Y_0, Z_0)$		
uX		1			−1			=	X_0
uY			1			−1		=	Y_0
uZ				1			−1	=	Z_0

The coefficient is treated as if it was the term $X * F(Y, Z)$ and is expanded in the same way as a *nonlinear term*.

13.1.2 Augmentation of a nonlinear term

Original matrix structure

	=
R	$F(X, Y, Z)$

The column name = is a reserved name for a column which has a fixed activity of 1.0 and can conveniently be used to hold nonlinear terms, particularly those which cannot be expressed as coefficients of variables.

Matrix structure: all augmentations

	=	X	Y	Z	dX	dY	dZ		
R	$F(X_0, Y_0, Z_0)$				$F'_x(X_0, Y_0, Z_0)$	$F'_y(X_0, Y_0, Z_0)$	$F'_z(X_0, Y_0, Z_0)$		
uX		1			−1			=	X_0
uY			1			−1		=	Y_0
uZ				1			−1	=	Z_0

The original nonlinear coefficient (=,R) is replaced by its evaluation using the assumed values of the independent variables.

Two vectors and one equality constraint for each independent variable in the coefficient are created if they do not already exist.

The new vectors are:

- The SLP variable (e.g. *Y*)
- The SLP delta variable (e.g. *dY*)

The new constraint is the SLP update row (e.g. *uY*) and is always an equality. The only entries in the update row are the +1 and -1 for the SLP variable and delta variable respectively. The right hand side is the assumed value for the SLP variable.

The entry in the original nonlinear constraint row for each independent variable is the first-order partial derivative of the term $F(X, Y, Z)$, evaluated at the assumed values.

The delta variables are bounded by the current values of the corresponding step bounds.

One new vector and one new equality constraint are created for the variable containing the nonlinear coefficient.

The new vector is:

- The SLP delta variable (e.g. *dX*)

The new constraint is the SLP update row (e.g. *uX*) and is always an equality. The only entries in the update row are the +1 and -1 for the original variable and delta variable respectively. The right hand side is the assumed value for the original variable.

The delta variable is bounded by the current values of the corresponding step bounds.

Note that if F(X,Y,Z) = X*F(Y,Z) then this translation is exactly equivalent to that for the nonlinear coefficient described earlier.

13.1.3 Augmentation of a user-defined SLP variable

Typically, this will arise when a variable represents the result of a nonlinear function, and is required to converge, or to be constrained by step-bounding to force convergence. In essence, it would arise from a relationship of the form
$X = F(Y, Z)$

Original matrix structure

	=	X
R	$F(Y,Z)$	−1

Matrix structure: all augmentations

	=	X	Y	Z	dX	dY	dZ		
R	$F(Y_0,Z_0)$	−1				$F'_y(Y_0,Z_0)$	$F'_z(Y_0,Z_0)$		
uX		1			−1			=	X_0
uY			1			−1		=	Y_0
uZ				1			−1	=	Z_0

The Y,Z structures are identical to those which would result from a nonlinear term or coefficient. The X, dX and uX structures effectively define dX as the deviation of X from X0 which can be controlled with step bounds.

The augmented and even-handed structures include more delta vectors, and so allow for more measurement and control of convergence.

Type of structure	Minimal	Standard	Even-handed
Type of variable			
Variables in nonlinear coefficients	Y	Y	Y
Variables with nonlinear coefficients	N	Y	Y
User-defined SLP variable	Y	Y	Y
Nonlinear term	Y	Y	Y

`Y` SLP variable has a delta vector which can be measured and/or controlled for convergence.

`N` SLP variable does not have a delta and cannot be measured and/or controlled for convergence.

There is no mathematical difference between the augmented and even-handed structures.

The even-handed structure is more elegant because it treats all variables in an identical way. However, the original coefficients are lost, because their effect is transferred to the "=" column as a term and so it is not possible to look up the coefficient value in the matrix after the SLP solution process has finished (whether because it has converged or because it has terminated for some other reason). The values of the SLP variables are still accessible in the usual way.

Some of the extended convergence criteria will be less effective because the effects of the individual coefficients may be amalgamated into one term (so, for example, the total positive and negative contributions to a constraint are no longer available).

13.1.4 SLP penalty error vectors

Bits 2, 3 and 9 of control variable XSLP_AUGMENTATION determine whether SLP penalty error vectors are added to constraints. Bit 9 applies penalty error vectors to all constraints; bits 2 and 3 apply them only to constraints containing nonlinear terms. When bit 2 or bit 3 is set, two penalty error vectors are added to each such equality constraint; when bit 3 is set, one penalty error vector is also added to each such inequality constraint. The general form is as follows:

Original matrix structure

	=
R	$F(Y,Z)$

Matrix structure with error vectors

	X	***R+***	***R-***
R	$F(Y,Z)$	+1	−1
P_ERROR		*+Weight*	*+Weight*

For equality rows, two penalty error vectors are added. These have penalty weights in the penalty error row P_ERROR, whose total is transferred to the objective with a cost of XSLP_CURRENTERRORCOST. For inequality rows, only one penalty error vector is added — the one corresponding to the slack is omitted. If any error vectors are used in a solution, the transfer cost from the cost penalty error row will be increased by a factor of XSLP_ERRORCOSTFACTOR up to a maximum of XSLP_ERRORMAXCOST.

Error vectors are ignored when calculating cascaded values.

The presence of error vectors at a non-zero level in an SLP solution normally indicates that the solution is not self-consistent and is therefore not a solution to the nonlinear problem.

Control variable XSLP_ERRORTOL_A is a tolerance on error vectors. Any error vector with a value less than XSLP_ERRORTOL_A will be regarded as having a value of zero.

Bit 9 controls whether error vectors are added to all constraints. If bit 9 is set, then error vectors are added in the same way as for the setting of bit 3, but to all constraints regardless of whether or not they have nonlinear coefficients.

13.2 Xpress-SLP Matrix Name Generation

Xpress-SLP adds rows and columns to the nonlinear problem in order to create a linear approximation. The new rows and columns are given names derived from the row or column to which they are related as follows:

Row or column type	Control parameter containing format	Default format
Update row	XSLP_UPDATEFORMAT	pU_r
Delta vector	XSLP_DELTAFORMAT	pD_c
Penalty delta (below step bound)	XSLP_MINUSDELTAFORMAT	pD-c
Penalty delta (above step bound)	XSLP_PLUSDELTAFORMAT	pD+c
Penalty error (below RHS)	XSLP_MINUSERRORFORMAT	pE-r
Penalty error (above RHS)	XSLP_PLUSERRORFORMAT	pE+r
Row for total of all penalty vectors (error or delta)	XSLP_PENALTYROWFORMAT	pPR_x
Column for standard penalty cost (error or delta)	XSLP_PENALTYCOLFORMAT	pPC_x
LO step bound formulated as a row	XSLP_SBLOROWFORMAT	pSB-c
UP step bound formulated as a row	XSLP_SBUPROWFORMAT	pSB+c

In the default formats:

p	a unique prefix (one or more characters not used as the beginning of any name in the problem).
r	the original row name.
c	the original column name.
x	The penalty row and column vectors are suffixed with "ERR" or "DELT" (for error and delta respectively).

Other characters appear "as is".

The format of one of these generated names can be changed by setting the corresponding control parameter to a formatting string using standard "C"-style conventions. In these cases, the unique prefix is not available and the only obvious choices, apart from constant names, use "%s" to include the original name — for example:

U_%s would create names like U_abcdefghi
U_%-8s would create names like U_abcdefgh (always truncated to 8 characters).

You can use a part of the name by using the XSLP_*OFFSET control parameters (such as XSLP_UPDATEOFFSET) which will offset the start of the original name by the number of characters indicated (so, setting XSLP_UPDATEOFFSET to 1 would produce the name U_bcdefghi).

13.3 Xpress-SLP Statistics

When a matrix is read in using XSLPreadprob, statistics on the model are produced. They should be interpreted as described in the numbered footnotes:

```
Reading Problem xxx                                          (1)
Problem Statistics
        1920 (      0 spare) rows                            (2)
         899 (      0 spare) structural columns              (3)
        6683 (   3000 spare) non-zero elements               (4)
Global Statistics
      0 entities   0 sets    0 set members                   (5)
```

```
Xpress-SLP Statistics:
        3632 coefficients                                   (6)
          14 extended variable arrays                       (7)
           1 user functions                                 (8)
        1011 SLP variables                                  (9)
```

Notes:

1. Standard output from `XPRSreadprob` reading the linear part of the problem
2. Number of rows declared in the ROWS section
3. Number of columns with at least one constant coefficient
4. Number of constant elements
5. Integer and SOS statistics if appropriate
6. Number of non-constant coefficients
7. Number of XVs defined
8. Number of user functions defined
9. Number of variables identified as SLP variables (interacting with a non-linear coefficient)

When the original problem is SLP-presolved prior to augmentation, the following statistics are produced:

```
Xpress-SLP Presolve:
    3 presolve passes                                      (10)
  247 SLP variables newly identified as fixed              (11)
  425 determining rows fixed                               (12)
   32 coefficients identified as fixed                     (13)
   58 columns fixed to zero (56 SLP variables)             (14)
  367 columns fixed to nonzero (360 SLP variables)         (15)
  139 column deltas deleted                                (16)
   34 column bounds tightened (6 SLP variables)            (17)
```

Notes:

10. Presolve is an iterative process. Each iteration refines the problem until no further progress is made. The number of iterations (*presolve passes*) can be limited by using `XSLP_PRESOLVEPASSES`
11. SLP variables which are deduced to be fixed by virtue of constraints in the model (over and above any which are fixed by bounds in the original problem)
12. Number of determining rows which have fixed variables and constant coefficients
13. Number of coefficients which are fixed because they are functions of constants and fixed variables
14. Total number of columns fixed to zero (number of fixed SLP variables shown in brackets)
15. Total number of columns fixed to nonzero values (number of fixed SLP variables shown in brackets)
16. Total number of deltas deleted because the SLP variable is fixed
17. Total number of bounds tightened by virtue of constraints in the model.

If any of these items is zero, it will be omitted. Unless specifically requested by setting additional bits of control XSLP_PRESOLVE, newly fixed variables and tightened bounds are not actually applied to the model. However, they are used in the initial augmentation and during cascading to ensure that the starting points for each iteration are within the tighter bounds.

When the original problem is augmented prior to optimization, the following statistics are produced:

```
Xpress-SLP Augmentation Statistics:
  Columns:
         754 implicit SLP variables                                  (18)
        1010 delta vectors                                           (19)
        2138 penalty error vectors (1177 positive, 961 negative)     (20)
  Rows:
        1370 nonlinear constraints                                   (21)
        1010 update rows                                             (22)
           1 penalty error rows                                      (23)
  Coefficients:
       11862 non-constant coefficients                               (24)
```

Notes:

18. SLP variables appearing only in coefficients and having no constant elements
19. Number of delta vectors created
20. Numbers of penalty error vectors
21. Number of constraints containing nonlinear terms
22. Number of update rows (equals number of delta vectors)
23. Number of rows totaling penalty vectors (error or delta)
24. Number of non-constant coefficients in the linear augmented matrix

- The total number of rows in the augmented matrix is (2) + (22) + (23)
- The total number of columns in the augmented matrix is (3) + (18) + (19) + (20) + (23)
- The total number of elements in the original matrix is (4) + (6)
- The total number of elements in the augmented matrix is (4) + (24) + (19) + 2*(20) + 2*(23)

If the matrix is read in using the XPRSloadxxx and XSLPloadxxx functions then these statistics may not be produced. However, most of the values are accessible through Xpress NonLinear integer attributes using the XSLPgetintattrib function.

13.4 SLP Variable History

Xpress-SLP maintains a history value for each SLP variable. This value indicates the direction in which the variable last moved and the number of consecutive times it moved in the same direction. All variables start with a history value of zero.

Current History	Change in activity of variable	New History
0	>0	1
0	<0	-1
>0	>0	No change unless delta vector is at its bound. If it is, then new value is Current History + 1
>0	<0	-1
<0	<0	No change unless delta vector is at its bound. If it is, then new value is Current History - 1
<0	>0	1
anything	0	No change

Tests of variable movement are based on comparison with absolute and relative (and, if set, closure) tolerances. Any movement within tolerance is regarded as zero.

If the new absolute value of History exceeds the setting of `XSLP_SAMECOUNT`, then the step bound is reset to a larger value (determined by `XSLP_EXPAND`) and History is reset as if it had been zero.

If History and the change in activity are of opposite signs, then the step bound is reset to a smaller value (determined by `XSLP_SHRINK`) and History is reset as if it had been zero.

With the default settings, History will normally be in the range -1 to -3 or +1 to +3.

CHAPTER 14

Xpress NonLinear Formulae

Xpress NonLinear can handle formulae described in three different ways:

`Character strings` The formula is written exactly as it would appear in, for example, the Extended MPS format used for text file input.

`Internal unparsed format` The tokens within the formula are replaced by a {*tokentype*, *tokenvalue*} pair. The list of types and values is in the table below.

`Internal parsed format` The tokens are converted as in the unparsed format, but the order is changed so that the resulting array forms a reverse-Polish execution stack for direct evaluation by the system.

14.1 Parsed and unparsed formulae

All formulae input into Xpress NonLinear are parsed into a reverse-Polish execution stack. Tokens are identified by their type and a value. The table below shows the values used in interface functions.

All formulae are provided in the interface functions as two parallel arrays:
an integer array of token types;
a double array of token values.

The last token type in the array should be an end-of-formula token (`XSLP_EOF`, which evaluates to zero).

If the value required is an integer, it should still be provided in the array of token values as a double precision value.

Even if a token type requires no token value, it is best practice to initialize such values as zeros.

Type	Description	Value
XSLP_COL	column	index of matrix column.
XSLP_CON	constant	(double) value.
XSLP_DEL	delimiter	XSLP_COMMA (1) = comma (",") XSLP_COLON (2) = colon (":")
XSLP_EOF	end of formula	not required: use zero
XSLP_FUN	user function	index of function
XSLP_IFUN	internal function	index of function
XSLP_LB	left bracket	not required: use zero
XSLP_OP	operator	XSLP_UMINUS (1) = unary minus ("-") XSLP_EXPONENT (2) = exponent ("**" or "^") XSLP_MULTIPLY (3) = multiplication ("*") XSLP_DIVIDE (4) = division ("/") XSLP_PLUS (5) = addition ("+") XSLP_MINUS (6) = subtraction ("-")
XSLP_RB	right bracket	not required: use zero
XSLP_VAR	variable	index of variable. Note that variables count from 1, so that the index of matrix column `n` is `n + 1`.

Token type XSLP_COL is used only when passing formulae *into* Xpress NonLinear. Any formulae recovered from Xpress NonLinear will use the XSLP_VAR token type which always count from 1.

When a formula is passed to Xpress NonLinear in "internal unparsed format" — that is, with the formula already converted into tokens — the full range of token types is permitted.

When a formula is passed to Xpress NonLinear in "parsed format" — that is, in reverse Polish — the following rules apply:

XSLP_DEL	comma is optional.
XSLP_FUN	implies a following left-bracket, which is not included explicitly.
XSLP_IFUN	implies a following left-bracket, which is not included explicitly.
XSLP_LB	never used.
XSLP_RB	only used to terminate the list of arguments to a function.

Brackets are not used in the reverse Polish representation of the formula: the order of evaluation is determined by the order of the items on the stack. Functions which need the brackets — for example XSLPgetccoef — fill in brackets as required to achieve the correct evaluation order. The result may not match the formula as originally provided.

14.2 Example of an arithmetic formula

$x^2 + 4y(z - 3)$

Written as an unparsed formula, each token is directly transcribed as follows:

Type	Value
XSLP_VAR	index of x
XSLP_OP	XSLP_EXPONENT
XSLP_CON	2
XSLP_OP	XSLP_PLUS
XSLP_CON	4
XSLP_OP	XSLP_MULTIPLY
XSLP_VAR	index of y
XSLP_OP	XSLP_MULTIPLY
XSLP_LB	0
XSLP_VAR	index of z
XSLP_OP	XSLP_MINUS
XSLP_CON	3
XSLP_RB	0
XSLP_EOF	0

Written as a parsed formula (in reverse Polish), an evaluation order is established first, for example:

x 2 ^ 4 *y* ∗ *z* 3 − ∗ +

and this is then transcribed as follows:

Type	Value
XSLP_VAR	index of x
XSLP_CON	2
XSLP_OP	XSLP_EXPONENT
XSLP_CON	4
XSLP_VAR	index of y
XSLP_OP	XSLP_MULTIPLY
XSLP_VAR	index of z
XSLP_CON	3
XSLP_OP	XSLP_MINUS
XSLP_OP	XSLP_MULTIPLY
XSLP_OP	XSLP_PLUS
XSLP_EOF	0

Notice that the brackets used to establish the order of evaluation in the unparsed formula are not required in the parsed form.

14.3 Example of a formula involving a simple function

$y * MyFunc(z, 3)$

Written as an unparsed formula, each token is directly transcribed as follows:

Type	Value
XSLP_VAR	index of y
XSLP_OP	XSLP_MULTIPLY
XSLP_FUN	index of MyFunc
XSLP_LB	0
XSLP_VAR	index of z
XSLP_DEL	XSLP_COMMA
XSLP_CON	3
XSLP_RB	0
XSLP_EOF	0

Written as a parsed formula (in reverse Polish), an evaluation order is established first, for example:

y) 3 , *z* *MyFunc*(∗

and this is then transcribed as follows:

Type	Value
XSLP_VAR	index of y
XSLP_RB	0
XSLP_CON	3
XSLP_DEL	XSLP_COMMA
XSLP_VAR	index of z
XSLP_FUN	index of MyFunc
XSLP_OP	XSLP_MULTIPLY
XSLP_EOF	0

Notice that the function arguments are in reverse order, and that a right bracket is used as a delimiter to indicate the end of the argument list. The left bracket indicating the start of the argument list is implied by the XSLP_FUN token.

CHAPTER 15

User Functions

15.1 Callbacks and user functions

Callbacks and user functions both provide mechanisms for connecting user-written functions to Xpress NonLinear. However, they have different capabilities and are not interchangeable.

A *callback* is called at a specific point in the SLP optimization process (for example, at the start of each SLP iteration). It has full access to all the problem data and can, in principle, change the values of any items — although not all such changes will necessarily be acted upon immediately or at all.

A *user function* is essentially the same as any other mathematical function, used in a formula to calculate the current value of a coefficient. The function is called when a new value is needed; for efficiency, user functions are not usually called if the value is already known (for example, when the function arguments are the same as on the previous call). Therefore, there is no guarantee that a user function will be called at any specific point in the optimization procedure or at all.

Although a user function is normally free-standing and needs no access to problem or other data apart from that which it receives through its argument list, there are facilities to allow it to access the problem and its data if required. The following limitations should be observed:

1. The function should not make use of any variable data which is not in its list of arguments;
2. The function should not change any of the problem data.

The reasons for these restrictions are as follows:

1. Xpress NonLinear determines which variables are linked to a formula by examining the list of variables and arguments to functions in the formula. If a function were to access and use the value of a variable not in this list, then incorrect relationships would be established, and incorrect or incomplete derivatives would be calculated. The predicted and actual values of the coefficient would then always be open to doubt.
2. Xpress NonLinear generally allows problem data to be changed between function calls, and also by callbacks called from within an Xpress NonLinear function. However, user functions are called at various points during the optimization and no checks are generally made to see if any problem data has changed. The effects of any such changes will therefore at best be unpredictable.

For a description of how to access the problem data from within a user function, see the section on "More complicated user functions" later in this chapter.

15.2 User function interface

In its simplest form, a user function is exactly the same as any other mathematical function: it takes a set of arguments (constants or values of variables) and returns a value as its result. In this form, which is the usual implementation, the function needs no information apart from the values of its arguments. It is possible to create more complicated functions which do use external data in some form: these are discussed at the end of this section.

Xpress NonLinear supports two basic forms of user function. The simple form of function returns a single value, and is treated in essentially the same way as a normal mathematical function. The general form of function returns an array of values and may also perform automatic differentiation.

The main difference between the simple and general form of a user function is in the way the value is returned.

- The simple function calculates and returns one value and is declared as such (for example, `double` in C).
- The general function calculates an array of values. It can either return the array itself (and is declared as such: `double *`), or it can return the results in one of the function arguments, in which case the function itself returns a single (double precision) status value (and is declared as such: `double`).

15.3 User Function declaration in native languages

This section describes how to declare a user function in C, Fortran and so on. The general shape of the declaration is shown. Not all the possible arguments will necessarily be used by any particular function, and the actual arguments required will depend on the way the function is declared to Xpress NonLinear.

15.3.1 User function declaration in C

The `XPRS_CC` calling convention (equivalent to `__stdcall` under Windows) must be used for the function. For example:

```
type XPRS_CC MyFunc(double *InputValues, int *FunctionInfo,
                    char *InputNames, char *ReturnNames
                    double *Deltas, double *ReturnArray);
```

where *`type`* is *`double`* or *`double*`* depending on the nature of the function.

In C++, the function should be declared as having a standard C-style linkage. For example, with Microsoft C++ under Windows:

```
extern "C" type _declspec(dllexport) XPRS_CC
                    MyFunc(double *InputValues, int *FunctionInfo,
                    char *InputNames, char *ReturnNames
                    double *Deltas, double *ReturnArray);
```

If the function is placed in a library, the function name may need to be externalized. If the compiler adds "decoration" to the name of the function, the function may also need to be given an alias which is the original name. For example, with the Microsoft compiler, a definition file can be used, containing the following items:

```
EXPORTS
MyFunc=_MyFunc@12
```

where the name after the equals sign is the original function name preceded by an underscore and followed by the @ sign and the number of bytes in the arguments. As all arguments in Xpress NonLinear external function calls are pointers, each argument represents 4 bytes on a 32-bit platform, and 8 bytes on a 64-bit platform.

A user function can be included in the executable program which calls Xpress NonLinear. In such a case, the user function is declared as usual, but the address of the program is provided using `XSLPchguserfuncaddress` or `XSLPsetuserfuncaddress`. The same technique can also be used when the function has been loaded by the main program and, again, its address is already known.

The `InputNames` and `ReturnNames` arrays, if used, contain a sequence of character strings which are the names, each terminated by a null character.

Any argument omitted from the declaration in Xpress NonLinear will be omitted from the function call.

Any argument declared in Xpress NonLinear as of type `NULL` will generally be passed as a null pointer to the program.

15.4 Simple functions and general functions

A *simple function* is one which returns a single value calculated from its arguments, and does not provide derivatives. A *general function* returns more than one value, because it calculates an array of results, or because it calculates derivatives, or both.

Because of restrictions in the various types of linkage, not all types of function can be declared and used in all languages. Any limitations are described in the appropriate sections.

For simplicity, the functions will be described using only examples in C. Implementation in other languages follows the same rules.

15.4.1 Simple user functions

A simple user function returns only one value and does not calculate derivatives. It therefore does not use the `ReturnNames`, `Deltas` or `ReturnArray` arguments.

The full form of the declaration is:

```
double XPRS_CC MyFunc(double *InputValues, int *FunctionInfo,
                      char *InputNames);
```

`FunctionInfo` can be omitted if the number of arguments is not required, and access to problem information and function objects is not required.
`InputNames` can be omitted if the input values are identified by position and not by name (see "Programming Techniques for User Functions" below).

The function supplies its single result as the return value of the function.

There is no provision for indicating that an error has occurred, so the function must always be able to calculate a value.

15.4.2 General user functions returning an array of values through a reference

General user functions calculate more than one value, and the results are returned as an array. In the first form of a general function, the values are supplied by returning the address of an array which holds the values. See the notes below for restrictions on the use of this method.

The full form of the declaration is:

```
double * XPRS_CC MyFunc(double *InputValues, int *FunctionInfo,
                        char *InputNames, char *ReturnNames
                        double *Deltas);
```

`FunctionInfo` can be omitted if the number of arguments is not required, no derivatives are being calculated, the number of return values is fixed, and access to problem information and function objects is not required. However, it is recommended that `FunctionInfo` is always included.

`InputNames` can be omitted if the input values are identified by position and not by name (see "Programming Techniques for User Functions" below).

`ReturnNames` can be omitted if the return values are identified by position and not by name (see "Programming Techniques for User Functions" below).

`Deltas` must be omitted if no derivatives are calculated.

The function supplies the address of an array of results. This array must be available after the function has returned to its caller, and so is normally a static array. This may mean that the function cannot be called from a multi-threaded optimization, or where multiple instances of the function are required, because the single copy of the array may be overwritten by another call to the function. An alternative method is to use a *function object* which refers to an array specific to the thread or problem being optimized.

`Deltas` is an array with the same number of items as `InputValues`. It is used as an indication of which derivatives (if any) are required on a particular function call. If `Deltas[i]` is zero then a derivative for input variable `i` is not required and must not be returned. If `Deltas[i]` is nonzero then a derivative for input variable `i` is required and must be returned. The total number of nonzero entries in `Deltas` is given in `FunctionInfo[2]`. In particular, if it is zero, then no derivatives are required at all.

When no derivatives are calculated, the array of return values simply contains the results (in the order specified by `ReturnNames` if used).
When derivatives are calculated, the array contains the values and the derivatives as follows (`DVi` is the i^{th} variable for which derivatives are required, which may not be the same as the i^{th} input value):

`Result1`
Derivative of `Result1` w.r.t. `DV1`
Derivative of `Result1` w.r.t. `DV2`
`...`
Derivative of `Result1` w.r.t. `DVn`
`Result2`
Derivative of `Result2` w.r.t. `DV1`
Derivative of `Result2` w.r.t. `DV2`
`...`
Derivative of `Result2` w.r.t. `DVn`
`...`
Derivative of `Resultm` w.r.t. `DVn`

It is therefore important to check whether derivatives are required and, if so, how many.

15.4.3 General user functions returning an array of values through an argument

General user functions calculate more than one value, and the results are returned as an array. In the second form of a general function, the values are supplied by returning the values in an array provided as an argument to the function by the calling program. See the notes below for restrictions on the use of this method.

The full form of the declaration is:

```
double XPRS_CC MyFunc(double *InputValues, int *FunctionInfo,
                      char *InputNames, char *ReturnNames
                      double *Deltas, double *ReturnArray);
```

`FunctionInfo` can be omitted if the number of arguments is not required, no derivatives are being calculated, the number of return values is fixed, and access to problem information and function objects is not required. However, it is recommended that `FunctionInfo` is always included.

`InputNames` can be omitted if the input values are identified by position and not by name (see "Programming Techniques for User Functions" below).

`ReturnNames` can be omitted if the return values are identified by position and not by name (see "Programming Techniques for User Functions" below).

`Deltas` must be omitted if no derivatives are calculated.

The function must supply the results in the array `ReturnArray`. This array is guaranteed to be large enough to hold all the values requested by the calling program. No guarantee is given that the results will be retained between function calls.

`Deltas` is an array with the same number of items as `InputValues`. It is used as an indication of which derivatives (if any) are required on a particular function call. If `Deltas[i]` is zero then a derivative for input variable `i` is not required and must not be returned. If `Deltas[i]` is nonzero then a derivative for input variable `i` is required and must be returned. The total number of nonzero entries in `Deltas` is given in `FunctionInfo[2]`. In particular, if it is zero, then no derivatives are required at all.

When no derivatives are calculated, the array of return values simply contains the results (in the order specified by `ReturnNames` if used).
When derivatives are calculated, the array contains the values and the derivatives as follows (`DVi` is the i^{th} variable for which derivatives are required, which may not be the same as the i^{th} input value):

`Result1`
Derivative of `Result1` w.r.t. `DV1`
Derivative of `Result1` w.r.t. `DV2`
`...`
Derivative of `Result1` w.r.t. `DVn`
`Result2`
Derivative of `Result2` w.r.t. `DV1`
Derivative of `Result2` w.r.t. `DV2`
`...`
Derivative of `Result2` w.r.t. `DVn`
`...`
Derivative of `Resultm` w.r.t. `DVn`

It is therefore important to check whether derivatives are required and, if so, how many.

The return value of the function is a status code indicating whether the function has completed normally. Possible values are:

`0`	No errors: the function has completed normally.
`1`	The function has encountered an error. This will terminate the optimization.
`–1`	The calling function must estimate the function value from the last set of values calculated. This will cause an error if no values are available.

15.5 Programming Techniques for User Functions

This section is principally concerned with the programming of large or complicated user functions, perhaps taking a potentially large number of input values and calculating a large number of results. However, some of the issues raised are also applicable to simpler functions.

The first part describes in more detail some of the possible arguments to the function. The remainder of the section looks at function instances, function objects and direct calls to user functions.

15.5.1 Deltas

The `Deltas` array has the same dimension as `InputValues` and is used to indicate which of the input variables should be used to calculate derivatives. If `Deltas[i]` is zero, then no derivative should be returned for input variable `i`. If `Deltas[i]` is nonzero, then a derivative is required for input variable `i`. The value of `Deltas[i]` can be used as a suggested perturbation for numerical differentiation (a negative sign indicates that if a one-sided derivative is calculated, then a backward one is preferred). If derivatives are calculated analytically, or without requiring a specific perturbation, then `Deltas` can be interpreted simply as an array of flags indicating which derivatives are required.

15.5.2 Return values and `ReturnArray`

The `ReturnArray` array is provided for those user functions which return more than one value, either because they do calculate more than one result, or because they also calculate derivatives. The function must either return the address of an array which holds the values, or pass the values to the calling program through the `ReturnArray` array.

The total number of values returned depends on whether derivatives are being calculated. The FunctionInfo array holds details of the number of input values supplied, the number of return values required (`nRet`) and the number of sets derivatives required (`nDeriv`). The total number of values (and hence the minimum size of the array) is *nRet* ∗ (*nDeriv* + 1). Xpress NonLinear guarantees that `ReturnArray` will be large enough to hold the total number of values requested.

A function which calculates and returns a single value can use the `ReturnArray` array provided that the declarations of the function in Xpress NonLinear and in the native language both include the appropriate argument definition.

functions which use the `ReturnArray` array must also return a status code as their return value. Zero is the normal return value. A value of 1 or greater is an error code which will cause any formula evaluation to stop and will normally interrupt any optimization or other procedure. A value of -1 asks Xpress NonLinear to estimate the function values from the last calculation of the values and partial derivatives. This will produce an error if there is no such set of values.

15.5.3 Returning Derivatives

A multi-valued function which does not calculate its own derivatives will return its results as a one-dimensional array.

As already described, when derivatives are calculated as well, the order is changed, so that the required derivatives follow the value for each result. That is, the order becomes:
$A, \frac{\partial A}{\partial X_1}, \frac{\partial A}{\partial X_2}, \ldots \frac{\partial A}{\partial X_n}, B, \frac{\partial B}{\partial X_1}, \frac{\partial B}{\partial X_2}, \ldots \frac{\partial B}{\partial X_n}, \ldots \frac{\partial Z}{\partial X_n}$
where A, B, Z are the return values, and X_1, X_2, X_n, are the input (independent) variables (in order) for which derivatives have been requested.

Not all calls to a user function necessarily require derivatives to be calculated. Check `FunctionInfo` for the number of derivatives required (it will be zero if only a value calculation is needed), and `Deltas` for the indications as to which independent variables are required to produce derivatives. Xpress NonLinear will not ask for, nor will it expect to receive, derivatives for function arguments which are actually constant in a particular problem. A function which provides uncalled-for derivatives will cause errors in subsequent calculations and may cause other unexpected side-effects if it stores values outside the expected boundaries of the return array.

15.5.4 Function Instances

Xpress NonLinear defines an *instance* of a user function to be a unique combination of function and arguments. For functions which return an array of values, the specific return argument is ignored when determining instances. Thus, given the following formulae:

$f(x) + f(y) + g(x, y : 1)$
$f(y) * f(x) * g(x, y : 2)$
$f(z)$
the following instances are created:
$f(x)$
$f(y)$
$f(z)$
$g(x, y)$
(A function reference of the form $g(x, y : n)$ means that g is a multi-valued function of x and y, and we want the n^{th} return value.)

Xpress NonLinear regards as *complicated* any user function which returns more than one value, which uses input or return names, or which calculates its own derivatives. All complicated functions give rise to function instances, so that each function is called only once for each distinct combination of arguments.

Functions which are not regarded as complicated are normally called each time a value is required. A function of this type can still be made to generate instances by defining its `ExeType` as creating instances (set bit 9 when using the normal library functions, or use the "I" suffix when using file-based input through `XSLPreadprob` or when using `SLPDATA` in Mosel).

Note that conditional re-evaluation of the function is only possible if it generates function instances.

Using function instances can improve the performance of a problem, because the function is called only once for each combination of arguments, and is not re-evaluated if the values have not changed significantly. If the function is computationally intensive, the improvement can be significant.

There are reasons for not wanting to use function instances:

- When the function is fast. It may be as fast to recalculate the value as to work out if evaluation is required.
- When the function is discontinuous. Small changes are estimated by using derivatives. These behave badly across a discontinuity and so it is usually better to evaluate the derivative of a formula by using the whole formula, rather than to calculate it from estimates of the derivatives of each term.
- Function instances do use more memory. Each instance holds a full copy of the last input and output values, and a full set of first-order derivatives. However, the only time when function instances are optional is when there is only one return value, so the extra space is not normally significant.

15.6 Function Derivatives

Xpress NonLinear normally expects to obtain a set of partial derivatives from a user function at a particular base-point and then to use them as required, depending on the evaluation settings for the various functions. If for any reason this is not appropriate, then the integer control parameter `XSLP_EVALUATE` can be set to 1, which will force re-evaluation every time.
A function instance is not re-evaluated if all of its arguments are unchanged.
A simple function which does not have a function instance is evaluated every time.

If `XSLP_EVALUATE` is not set, then it is still possible to by-pass the re-evaluation of a function if the values have not changed significantly since the last evaluation. If the input values to a function have all converged to within their strict convergence tolerance (`CTOL`, `ATOL_A`, `ATOL_R`), and bit 4 of `XSLP_FUNCEVAL` is set to 1, then the existing values and derivatives will continue to be used. At the option of the user, an individual function, or all functions, can be re-evaluated in this way or at each SLP iteration. If a function is not re-evaluated, then all the required values will be calculated from the base

point and the partial derivatives; the input and return values used in making the original function calculation are unchanged.

Bits 3-5 of integer control parameter XSLP_FUNCEVAL determine the nature of function evaluations. The meaning of each bit is as follows:

Bit 3 evaluate functions whenever independent variables change.

Bit 4 evaluate functions when independent variables change outside tolerances.

Bit 5 apply evaluation mode to all functions.

If bits 3-4 are zero, then the settings for the individual functions are used.
If bit 5 is zero, then the settings in bits 3-4 apply only to functions which do not have their own specific evaluation modes set.

Examples:

Bits 3-5 = 1 (set bit 3) Evaluate functions whenever their input arguments (independent variables) change, unless the functions already have their own evaluation options set.

Bits 3-5 = 5 (set bits 3 and 5) Evaluate all functions whenever their input arguments (independent variables) change.

Bits 3-5 = 6 (set bits 4 and 5) Evaluate functions whenever input arguments (independent variables) change outside tolerance. Use existing calculation to estimate values otherwise.

Bits 6-8 of integer control parameter XSLP_FUNCEVAL determine the nature of derivative calculations. The meaning of each bit is as follows:

Bit 6 tangential derivatives.

Bit 7 forward derivatives.

Bit 8 apply evaluation mode to all functions.

If bits 6-7 are zero, then the settings for the individual functions are used.
If bit 8 is zero, then the settings in bits 6-7 apply only to functions which do not have their own specific derivative calculation modes set.

Examples:

Bits 6-8 = 1 (set bit 6) Use tangential derivatives for all functions which do not already have their own derivative options set.

Bits 6-8 = 5 (set bits 6 and 8) Use tangential derivatives for all functions.

Bits 6-8 = 6 (set bits 7 and 8) Use forward derivatives for all functions.

The following constants are provided for setting these bits:

Setting bit 3	XSLP_RECALC
Setting bit 4	XSLP_TOLCALC
Setting bit 5	XSLP_ALLCALCS
Setting bit 6	XSLP_2DERIVATIVE
Setting bit 7	XSLP_1DERIVATIVE
Setting bit 8	XSLP_ALLDERIVATIVES

A function can make its own determination of whether to re-evaluate. If the function has already calculated and returned a full set of values and partial derivatives, then it can request Xpress NonLinear to estimate the values required from those already provided.

The function must be defined as using the `ReturnArray` argument, so that the return value from the function itself is a double precision status value as follows:

`0` normal return. The function has calculated the values and they are in `ReturnArray`.

`1` error return. The function has encountered an unrecoverable error. The values in `ReturnArray` are ignored and the optimization will normally terminate.

`–1` no calculation. Xpress NonLinear should recalculate the values from the previous results. The values in `ReturnArray` are ignored.

15.6.1 Analytic Derivatives of Instantiated User Functions not Returning their own Derivatives

When analytical derivatives are used, SLP will calculate approximated derivatives using finite differences for instantiated functions and use these values when deriving analytical derivatives. Functions returning multiple arguments will always be instantiated, otherwise functions can be forced to be instantiated on a per function basis.

CHAPTER 16

Management of zero placeholder entries

16.1 The augmented matrix structure

During the augmentation process, Xpress-SLP builds additional matrix structure to represent the linear approximation of the nonlinear constraints within the problem (see Xpress-SLP Structures). In effect, it adds a generic structure which approximates the effect of changes to variables in nonlinear expressions, over and above that which would apply if the variables were simply replaced by their current values.

As a very simple example, consider the nonlinear constraint ($R1$, say)
$X * Y \leq 10$

The variables X and Y are replaced by $X_0 + \delta X$ and $Y_0 + \delta Y$ respectively, where X_0 and Y_0 are the values of X and Y at which the approximation will be made.

The original constraint is therefore
$(X_0 + \delta X) * (Y_0 + \delta Y) \leq 10$

Expanding this into individual terms, we have
$X_0 * Y_0 + X_0 * \delta Y + Y_0 * \delta X + \delta X * \delta Y \leq 10$

The first term is constant, the next two terms are linear in δY and δX respectively, and the last term is nonlinear.

The augmented structure deletes the nonlinear term, so that the remaining structure is a linear approximation to the original constraint. The justification for doing this is that if δX or δY (or both) are small, then the error involved in ignoring the term is also small.

The resulting matrix structure has entries of Y_0 in the delta variable δX and X_0 in the delta variable δY. The constant entry $X_0 * Y_0$ is placed in the special "equals" column which has a fixed activity of 1. All these entries are updated at each SLP iteration as the solution process proceeds and the problem is linearized at a new point. The positions of these entries – $(R1, \delta X)$, $(R1, \delta Y)$ and $(R1, =)$ – are known as *placeholders*.

16.2 Derivatives and zero derivatives

At each SLP iteration, the values of the placeholders are re-calculated. In the example in the previous section, the values X_0 in the delta variable δY and Y_0 in the delta variable δX were effectively determined by analytic methods – that is, we differentiated the original formula to determine what values would be required in the placeholders.

In general, analytic differentiation may not be possible: the formula may contain functions which cannot be differentiated (because, for example, they are not smooth or not continuous), or for which the analytic derivatives are not known (because, for example, they are functions providing values from

"black boxes" such as databases or simulators). In such cases, Xpress-SLP approximates the differentiation process by numerical methods. The example in the previous section would have approximate derivatives calculated as follows:

The current value of X (X_0) is perturbed by a small amount (dX), and the value of the formula is recalculated in each case.

$f_d = (X_0 - dX) * Y_0$
$f_u = (X_0 + dX) * Y_0$

$derivative = (f_u - f_d)/(2 * dX)$

In this particular example, the value obtained by numerical methods is the same as the analytic derivative. For more complex functions, there may be a slight difference, depending on the magnitude of dX.

This derivative represents the effect on the constraint of a change in the value of X. Obviously, if Y changes as well, then the combined effect will not be fully represented although, in general, it will be directionally correct.

The problem comes when Y_0 is zero. In such a case, the derivative is calculated as zero, meaning that changing X has no effect on the value of the formula. This can impact in one of two ways: either the value of X never changes because there is no incentive to do so, or it changes by unreasonably large amounts because there is no effect from doing so. If X and Y are linked in some other way, so that Y becomes nonzero when X changes, the approximation using zero as the derivative can cause the optimization process to behave badly.

Xpress-SLP tries to avoid the problem of zero derivatives by using small nonzero values for variables which are in fact zero. In most cases this gives a small nonzero value for the derivative, and hence for the placeholder entry. The model then contains some effect for the change in a variable, even if instantaneously the effect is zero.

The same principle is applied to analytic derivatives, so that the values obtained by either method are broadly similar.

16.3 Placeholder management

The default action of Xpress-SLP is to retain all the calculated values for all the placeholder entries. This includes values which would be zero without the special handling described in the previous section. We will call such values "zero placeholders".

Although retaining all the values gives the best chance of finding a good optimum, the presence of a large dense area of small values often gives rise to considerable numerical instability which adversely affects the optimization process. Xpress-SLP therefore offers a way of deleting small values which is less likely to affect the final outcome whilst improving numerical stability.

Most of the candidate placeholders are in the delta variables (represented by the δX and δY variables above). Various criteria can be selected for deletion of zero placeholder entries without affecting the validity of the basis (and so making the next SLP iteration more costly in time and stability). The criteria are selected using the control parameter `XSLP_ZEROCRITERION` as follows:

- **Bit 0 (=1)** Remove placeholders in nonbasic SLP variables
  This criterion applies to placeholders which are in the SLP variable (not the delta). Any value can be deleted from a nonbasic variable without upsetting the basis, so all eligible zero placeholders can be deleted.
- **Bit 1 (=2)** Remove placeholders in nonbasic delta variables
  Any value can be deleted from a nonbasic variable without upsetting the basis, so all eligible zero placeholders can be deleted.

- **Bit 2 (=4)** Remove placeholders in a basic SLP variable if its update row is nonbasic
  If the update row is nonbasic, then generally the basic SLP variable can be pivoted in the update row, so the basis is still valid if other entries are deleted. The entry in the update row is always 1.0 and will never be deleted.
- **Bit 3 (=8)** Remove placeholders in a basic delta variable if its update row is nonbasic and the corresponding SLP variable is nonbasic
  If the delta is basic and the corresponding SLP variable is nonbasic, then the delta will pivot in the update row (the delta and the SLP variable are the only two variables in the update row), so the basis is still valid if other entries are deleted. The entry in the update row is always -1.0 and will never be deleted.
- **Bit 4 (=16)** Remove placeholders in a basic delta variable if the determining row for the corresponding SLP variable is nonbasic
  If the delta variable is basic and the determining row for the corresponding SLP variable is nonbasic then it is generally possible (although not 100% guaranteed) to pivot the delta variable in the determining row. so the basis is still valid if other entries are deleted. The entry in the determining row is never deleted even if it is otherwise eligible.

The following constants are provided for setting these bits:

Setting bit 0	`XSLP_ZEROCRTIERION_NBSLPVAR`
Setting bit 1	`XSLP_ZEROCRTIERION_NBDELTA`
Setting bit 2	`XSLP_ZEROCRTIERION_SLPVARNBUPDATEROW`
Setting bit 3	`XSLP_ZEROCRTIERION_DELTANBUPSATEROW`
Setting bit 4	`XSLP_ZEROCRTIERION_DELTANBDRROW`

There are two additional control parameters used in this procedure:

- `XSLP_ZEROCRITERIONSTART`
  This is the first SLP iteration at which zero placeholders will be examined for eligibility. Use of this parameter allows a balance to be made between optimality and numerical stability.
- `XSLP_ZEROCRITERIONCOUNT`
  This is the number of consecutive SLP iterations that a placeholder is a zero placeholder before it is deleted. So, if in the earlier example `XSLP_ZEROCRITERIONCOUNT` is 2, the entry in the delta variable *dX* will be deleted only if *Y* was also zero on the previous SLP iteration.

Regardless of the basis status of a variable, its delta, update row and determining row, if a zero placeholder was deleted on the previous SLP iteration, it will always be deleted in the current SLP iteration (keeping a zero matrix entry at zero does not upset the basis).

If the optimization method is barrier, or the basis is not being used, then the bit settings of `XSLP_ZEROCRITERION` are not used as such: if `XSLP_ZEROCRITERION` is nonzero, all zero placeholders will be deleted subject to `XSLP_ZEROCRITERIONCOUNT` and `XSLP_ZEROCRITERIONSTART`.

CHAPTER 17

Special Types of Problem

17.1 Nonlinear objectives

Xpress NonLinear works with nonlinear constraints. If a nonlinear objective is required (except for the special case of a quadratic objective — see below) then the objective should be provided using a constraint in the problem. For example, to optimize `f(x)` where `f` is a nonlinear function and `x` is a set of one or more variables, create the constraint

$$f(x) - X = 0$$

where X is a new variable, and then optimize X.

In general, X should be made a free variable, so that the problem does not converge prematurely on the basis of an unchanging objective function. It is generally important that the objective is not artificially constrained (for example, by bounding X) because this can distort the solution process. Also, as such an objective transfer row is not a real constraint, no error vectors should be added (row can be enforced); feasibility should be provided by the transfer variable X being free.

17.2 Convex Quadratic Programming

Convex quadratic programming (QP) is a special case of nonlinear programming where the constraints are linear but the objective is quadratic (that is, it contains only terms which are constant, variables multiplied by a constant, or products of two variables multiplied by a constant) and convex (convexity is checked by the Xpress Optimizer). It is possible to solve convex quadratic problems using SLP, but it is not usually the best way. The reason is that the solution to a convex QP problem is typically not at a vertex. In SLP a non-vertex solution is achieved by applying step bounds to create additional constraints which surround the solution point, so that ultimately the solution has been obtained within suitable tolerances. Because of the nature of the problem, successive solutions will often swing from one step bound to the other; in such circumstances, the step bounds are reduced on each SLP iteration but it will still take a long time before convergence. In addition, unless the linear approximation is adequately constrained, it will be unbounded because the linear approximation will not recognize the change in direction of the relationship with the derivative as the variable passes through a stationary point. The easiest way to ensure that the linear problem is constrained is to provide realistic upper and lower bounds on all variables.

In Xpress NonLinear, convex quadratic problems can be solved using the quadratic optimizer within the Xpress optimizer package. For pure QP (or MIQP) problems, therefore, SLP is not required. However, the SLP algorithm can be used together with QP to solve problems with a quadratic objective and also nonlinear constraints. The constraints are handled using the normal SLP techniques; the objective is handled by the QP optimizer. If the objective is not convex (not semi-definite), the QP optimizer may not give a solution (with default settings, it will produce an error message); SLP will find a solution but — as always — it may be a local optimum.

If a QP problem is to be solved, then the quadratic component should be input in the normal way (using QMATRIX or QUADOBJ in MPS file format, or the library functions XPRSloadqp or XPRSloadqglobal). Xpress NonLinear will then automatically use the QP optimizer. If the problem is to be solved using the SLP routines throughout, then the objective should be provided via a constraint as described in the previous section.

This applies to quadratically constrained (QCQP and MIQCQP) problems as well.

For a description on when it's more beneficial to use the XPRS library to solve QP or QCQP problems, please see *Selecting the right algorithm for a nonlinear problem - when to use the XPRS library instead of XSLP*.

17.3 Mixed Integer Nonlinear Programming

Mixed Integer Non-Linear Programming (MINLP) is the application of mixed integer techniques to the solution of problems including non-linear relationships. Xpress NonLinear offers a set of components to implement MINLP using Mixed Integer Successive Linear Programming (MISLP).

17.3.1 Mixed Integer SLP

The mixed integer successive linear programming (MISLP) solver is a generalization of the traditional branch and bound procedure to nonlinear programming. The MIP engine is used to control the branch-and-bound algorithm, with each node being evaluated using SLP. MIP then compares the SLP solutions at each node to decide which node to explore next, and to decide when an integer feasible and ultimately optimal solution have been obtained.

MISLP, also known as SLP within MIP, offers nonlinear specific root heuristics controlled by control XSLP_HEURSTRATEGY.

Other generic heuristics are controlled by the respective XPRS heuristics controls.

The branch and bound tree exploration is executed in parallel. Use the XPRS control MIPTHREADS to limit the number of threads used.

Normally, the relaxed problem is solved first, using XSLPminim or XSLPmaxim with the -l flag to ignore the integer elements of the problem. It is possible to go straight into the XSLPglobal routine and allow it to do the initial SLP optimization as well. In that case, ensure that the control parameter XSLP_OBJSENSE is set to +1 (minimization) or -1 (maximization) before calling XSLPglobal.

The actual algorithm employed is controlled by a number of control parameters, as well as offering the possibility of direct user interaction through call-backs at key points in the solution process.

17.3.2 Heuristics for Mixed Integer SLP

For hard MINLP problems, or where a solution must quickly be generated, the root heuristics of MISLP can be executed as stand alone methods. These approaches can be used by changing the value of the control parameter XSLP_MIPALGORITHM.

there are two MISLP heuristics:

1. MIP within SLP. In this, each SLP iteration is optimized using MIP to obtain an integer optimal solution to the linear approximation of the original problem. SLP then compares this MIP solution to the MIP solution of the previous SLP iteration and determines convergence based on the differences between the successive MIP solutions.
2. SLP then MIP. In this, SLP is used to find a converged solution to the relaxed problem. The resulting linearization is then fixed (i.e. the base point and the partial derivatives do not change)

and MIP is run to find an integer optimum. SLP is then run again to find a converged solution to the original problem with these integer settings.

The approach described in (1) seems potentially dangerous, in that changes in the integer variables could have disproportionate effects on the solution and on the values of the SLP variables. There are also question-marks over the use of step-bounding to control convergence, particularly if any of the integer variables are also SLP variables.

The approach described in (2) has the big advantage that MIP is working on a linear problem and so can take advantage of all of the special attributes of such a problem. This means that the solution time is likely to be much faster than the alternatives. However, if the real problem is significantly non-linear, the integer solution to the initial SLP solution may not be a good integer solution to the original problem and so a false optimum may occur.

17.3.3 Fixing or relaxing the values of the SLP variables

The solution process may involve step-bounding to obtain the converged solution. Some MIP solution strategies may want to fix the values of some of the SLP variables before moving on to the MIP part of the process, or they may want to allow the child nodes more freedom than would be allowed by the final settings of the step bounds. Control parameters XSLP_MIPALGORITHM, XSLP_MIPFIXSTEPBOUNDS and XSLP_MIPRELAXSTEPBOUNDS can be used to free, or fix to zero, various categories of step bounds, thus effectively freeing the SLP variables or fixing them to their values in the initial solution.

At each node, step bounds may again be fixed to zero or relaxed or left in the same state as in the solution to the parent node.

XSLP_MIPALGORITHM uses bits 2-3 (for the root node) and 4-5 (for other nodes) to determine which step bounds are fixed to zero (thus fixing the values of the corresponding variables) or freed (thus allowing the variables to change, possibly beyond the point they were restricted to in the parent node).
Set bit 2 (4) of XSLP_MIPALGORITHM to implement relaxation of defined categories of step bounds as determined by XSLP_MIPRELAXSTEPBOUNDS at the root node (at each node).
Set bit 3 (5) of XSLP_MIPALGORITHM to implement fixing of defined categories of step bounds as determined by XSLP_MIPFIXSTEPBOUNDS at the root node (at each node).

Alternatively, specific actions on setting bounds can be carried out by the user callback defined by XSLPsetcbprenode.

The default setting of XSLP_MIPALGORITHM is 17 which relaxes step bounds at all nodes except the root node. The step bounds from the initial SLP optimization are retained for the root node.

XSLP_MIPRELAXSTEPBOUNDS and XSLP_MIPFIXSTEPBOUNDS are bitmaps which determine which categories of SLP variables are processed.

`Bit 1` Process SLP variables which do not appear in coefficients but which do have coefficients (constant or variable) in the original problem.

`Bit 2` Process SLP variables which have coefficients (constant or variable) in the original problem.

`Bit 3` Process SLP variables which appear in coefficients but which do not have coefficients (constant or variable) in the original problem.

`Bit 4` Process SLP variables which appear in coefficients.

In most cases, the default settings (XSLP_MIPFIXSTEPBOUNDS=0, XSLP_MIPRELAXSTEPBOUNDS=15) are appropriate.

17.3.4 Iterating at each node

Any number of SLP iterations can be carried out at each node. The maximum number is set by control parameter XSLP_MIPITERLIMIT and is activated by XSLP_MIPALGORITHM. The significant values for XSLP_MIPITERLIMIT are:

0	Perform an LP optimization with the current linearization. This means that, subject to the step bounds, the SLP variables can take on other values, but the coefficients are not updated.
1	As for 0, but the model is updated after each iteration, so that each node starts with a new linearization based on the solution of its parent.
n>1	Perform up to n SLP iterations, but stop when a termination criterion is satisfied. If no other criteria are set, the SLP will terminate on XSLP_ITERLIMIT or XSLP_MIPITERLIMIT iterations, or when the SLP converges.

After the last MIP node has been evaluated and the MIP procedure has terminated, the final solution can be re-optimized using SLP to obtain a converged solution. This is only necessary if the individual nodes are being terminated on a criterion other than SLP convergence.

17.3.5 Termination criteria at each node

Because the intention at each node is to get a reasonably good estimate for the SLP objective function rather than to obtain a fully converged solution (which is only required at the optimum), it may be possible to set looser but practical termination criteria. The following are provided:

Testing for movement of the objective function
This functions in a similar way to the extended convergence criteria for ordinary SLP convergence, but does not require the SLP variables to have converged in any way. The test is applied once step bounding has been applied (or XSLP_SBSTART SLP iterations have taken place if step bounding is not being used). The node will be terminated at the current iteration if the range of the objective function values over the last *XSLP_MIPOCOUNT* SLP iterations is within *XSLP_MIPOTOL_A* or within *XSLP_MIPOTOL_R* $*$ *OBJ* where *OBJ* is the average value of the objective function over those iterations.

Related control parameters:

XSLP_MIPOTOL_A	Absolute tolerance
XSLP_MIPOTOL_R	Relative tolerance
XSLP_MIPOCOUNT	Number of SLP iterations over which the movement is measured

Testing the objective function against a cutoff
If the objective function is worse by a defined amount than the best integer solution obtained so far, then the SLP will be terminated (and the node will be cut off). The node will be cut off at the current SLP iteration if the objective function for the last *XSLP_MIPCUTOFFCOUNT* SLP iterations are all worse than the best obtained so far, and the difference is greater than *XSLP_MIPCUTOFF_A* and *XSLP_MIPCUTOFF_R* $*$ *OBJ* where *OBJ* is the best integer solution obtained so far.

Related control parameters:

XSLP_MIPCUTOFF_A	Absolute amount by which the objective function is worse
XSLP_MIPCUTOFF_R	Relative amount by which the objective function is worse
XSLP_MIPCUTOFFCOUNT	Number of SLP iterations checked
XSLP_MIPCUTOFFLIMIT	Number of SLP iterations before which the cutoff takes effect

17.3.6 Callbacks

User callbacks are provided as follows:

```
XSLPsetcbintsol(XSLPprob Prob,
                int (*UserFunc)(XSLPprob myProb, void *myObject),
                void *Object);
```

`UserFunc` is called when an integer solution has been obtained. The return value is ignored.

```
XSLPsetcboptnode(XSLPprob Prob,
                 int (*UserFunc)(XSLPprob myProb, void *myObject, int *feas),
                 void *Object);
```

`UserFunc` is called when an optimal solution is obtained at a node.
If the feasibility flag `*feas` is set nonzero or if the function returns a nonzero value, then further processing of the node will be terminated (it is declared infeasible).

```
XSLPsetcbprenode(XSLPprob Prob,
                 int (*UserFunc)(XSLPprob myProb, void *myObject, int *feas),
                 void *Object);
```

`UserFunc` is called at the beginning of each node after the SLP problem has been set up but before any SLP iterations have taken place.
If the feasibility flag `*feas` is set nonzero or if the function returns a nonzero value, then the node will be declared infeasible and cut off. In particular, the SLP optimization at the node will not be performed.

```
XSLPsetcbslpnode(XSLPprob Prob,
                 int (*UserFunc)(XSLPprob myProb, void *myObject, int *feas),
                 void *Object);
```

`UserFunc` is called after each SLP iteration at each node, after the SLP iteration, and after the convergence and termination criteria have been tested.
If the feasibility flag `*feas` is set nonzero or if the function returns a nonzero value, then the node will be declared infeasible and cut off.

17.4 Integer and semi-continuous delta variables

Functions implementing piecewise linear expressions often lead to local stalling due to the partial derivatives not capturing the true nature of the behaviour of the function. Such functions are often implemented as user functions or expressions using the abs function. To provide the Xpress with a better way of evaluating such expressions, it is possible to mark variables (typically the key dependencies of the expression) as having a semi-continuous delta variable with a minimum perturbation size associated, which means the value of any expression that involves this variable is expected to meaningfully change if the variable's value in the current solution is changed by at least of the semi-continuous value of the delta. If a minimum meaningful perturbation is not known, the variable's delta may be set up to being of type explore, when SLP will trial several values up to the provided maximum in case of zero partials are detected. Using exploration deltas may significantly increase the number the formulas the variable is used in are evaluated.

It is important to note that the value with a semi-continuous delta will still be allowed to take any value and make arbitrary steps between iterations, the extra information of the delta variable is solely used as a means of better evaluating the effect of change per variable.

User functions that can only be evaluated at given values (e.g. lookup tables or simulations over integer input) may be modelled with variables with an integer delta variable. If a variable's delta variable is flagged as being integer, with a step value of 'delta', then assuming the variable has an initial value of 'x0', the possible values of the variable are 'x0 + i * delta' where 'i' is an integer number. If no initial value is provided, the lower bound (or zero if no lover bound) is used to start the possible values from.

Variables with a semi-continuous delta are not expected to be harder than the problem without, in fact, the extra information usually aids the solve noticeably.

A model with variables with integer deltas is considered to be hard. An integer delta is expected to be used to model the domain of user function, and should not be used to otherwise model integrality of the original variable. Variables with an integer delta used in constraints tend to make the problem difficult to solve unless their use is balanced by the presence of infeasibility breaker variables (penalty slacks).

To change the type of a delta variable, use 'XSLPchgdeltatype' in the API and the 'setdeltatype' method in Mosel.

If variables with integer deltas are present in the problem, then SLP will run a number of heuristics as part of the solve, please refer to XSLP_GRIDHEURSELECT.

CHAPTER 18

Xpress NonLinear multistart

The feature is an additive feature that minimizes the development overhead and effort of implementing parallel multistart searches. The purpose of multistart is two-fold. Traditionally, multistart is a so called globalization feature. It is important to correctly understand what this technology offers, and what it does not. It offers a convenient and efficient way of exploring a larger feasible space building on top of existing local solver algorithms by the means of perturbing initial points and/or parameters or even the problem statement itself. Multistart can also be viewed as a left-alone feature. In a typical situation, versions of a model react favourably to a set of control settings, dependent on data. Multistart allows for a simple way of combining different control setting scenarios, increasing the robustness of the model.

The base problem is defined as the baseline: as the model is normally loaded it without any multistart information, including problem description, callbacks and controls. A run or a job is defined as a problem instance that needs to be solved as part of multistart.

On completion, the current problem is set up to match that of the winner, allowing examination of the winning strategy and solution using the normal means.

The original prob object is not reused, all runs are mode on a copy of the problem, allowing full customization from the callbacks, including changes to structre.

Callbacks are inherited to the multistart jobs from the master problem and can be customized from the the multistart callbacks. XSLinterrupt has a global scope, and a calling it terminates the multistart search.

Although not intended as the primary use, multistart allows the execution of all supported problem classes, so for example alternate MIP strategies can be used in parallel.

The mutistart job pool is maintained and can be extended until the first maxim / minim with `XSLP_MULTISTART` on. This allows for doing optimizations runs aimed at generating multistart jobs. The multistart pool is dynamic and new jobs can be added on the fly from the jobstart and jobend callbacks.

III. Reference

CHAPTER 19

Problem Attributes

During the optimization process, various properties of the problem being solved are stored and made available to users of the Xpress NonLinear Libraries in the form of *problem attributes*. These can be accessed in much the same manner as 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.

Library users are provided with the following functions for obtaining the values of attributes:

XSLPgetintattrib	XSLPgetdblattrib
XSLPgetptrattrib	XSLPgetstrattrib

The attributes listed in this chapter are all prefixed with XSLP_. It is possible to use the above functions with attributes for the Xpress Optimizer (attributes prefixed with XPRS_). For details of the Optimizer attributes, see the Optimizer manual.

Example of the usage of the functions:

```
XSLPgetintattrib(Prob, XSLP_ITER, &nIter);
printf("The number of SLP iterations is %d\n", nIter);
XSLPgetdblattrib(Prob, XSLP_ERRORCOSTS, &Errors);
printf("and the total error cost is %lg\n", Errors);
```

The following is a list of all the Xpress NonLinear attributes:

XSLP_COEFFICIENTS	Number of nonlinear coefficients	p. 104
XSLP_CURRENTDELTACOST	Current value of penalty cost multiplier for penalty delta vectors	p. 101
XSLP_CURRENTERRORCOST	Current value of penalty cost multiplier for penalty error vectors	p. 101
XSLP_CVS	Number of character variables	p. 104
XSLP_DELTAS	Number of delta vectors created during augmentation	p. 104
XSLP_ECFCOUNT	Number of infeasible constraints found at the point of linearization	p. 104
XSLP_EQUALSCOLUMN	Index of the reserved "=" column	p. 105
XSLP_ERRORCOSTS	Total penalty costs in the solution	p. 101
XSLP_EXPLOREDELTAS	Number of variables with an exploration-type delta set up in the problem	p. 104
XSLP_IFS	Number of internal functions	p. 105
XSLP_IMPLICITVARIABLES	Number of SLP variables appearing only in coefficients	p. 105

XSLP_VARIABLES	Number of SLP variables	p. 118
XSLP_VERSION	Xpress NonLinear major version number	p. 118
XSLP_VERSIONDATE	Date of creation of Xpress NonLinear	p. 121
XSLP_VSOLINDEX	Vertex solution index	p. 103
XSLP_XPRSPROBLEM	The underlying Optimizer problem	p. 120
XSLP_XSLPPROBLEM	The Xpress NonLinear problem	p. 120
XSLP_ZEROESRESET	Number of placeholder entries set to zero	p. 118
XSLP_ZEROESRETAINED	Number of potentially zero placeholders left untouched	p. 118
XSLP_ZEROESTOTAL	Number of potential zero placeholder entries	p. 119

19.1 Double problem attributes

XSLP_CURRENTDELTACOST

Description Current value of penalty cost multiplier for penalty delta vectors

Type Double

Set by routines XSLPmaxim, XSLPminim

See also XSLP_DELTACOST, XSLP_ERRORCOST, XSLP_CURRENTERRORCOST

XSLP_CURRENTERRORCOST

Description Current value of penalty cost multiplier for penalty error vectors

Type Double

Set by routines XSLPmaxim, XSLPminim

See also XSLP_DELTACOST, XSLP_ERRORCOST, XSLP_CURRENTDELTACOST

XSLP_ERRORCOSTS

Description Total penalty costs in the solution

Type Double

Set by routines XSLPmaxim, XSLPminim

XSLP_OBJVAL

Description Objective function value excluding any penalty costs

Type Double

Set by routines XSLPmaxim, XSLPminim

XSLP_PENALTYDELTATOTAL

Description Total activity of penalty delta vectors

Type Double

Set by routines XSLPmaxim, XSLPminim

XSLP_PENALTYDELTAVALUE

Description Total penalty cost attributed to penalty delta vectors

Type Double

Set by routines XSLPmaxim, XSLPminim

XSLP_PENALTYERRORTOTAL

Description Total activity of penalty error vectors

Type Double

Set by routines XSLPmaxim, XSLPminim

XSLP_PENALTYERRORVALUE

Description Total penalty cost attributed to penalty error vectors

Type Double

Set by routines XSLPmaxim, XSLPminim

XSLP_PRIMALINTEGRAL

Description Local primal integral of the solve

Type Double

Set by routines XSLPmaxim, XSLPminim

XSLP_VALIDATIONINDEX_A

Description Absolute validation index

Type Double

Set by routines XSLPvalidate

XSLP_VALIDATIONINDEX_K

Description Relative first order optimality validation index

Type Double

Set by routines `XSLPvalidatekkt`

XSLP_VALIDATIONINDEX_R

Description Relative validation index

Type Double

Set by routines `XSLPvalidate`

XSLP_VSOLINDEX

Description Vertex solution index

Type Double

Notes The *vertex solution index* (`VSOLINDEX`) is a measure of how nearly the converged solution to a problem is at a vertex (that is, at the intersection of a set of constraints) of the feasible region.

Where the solution is in the middle of a face, the solution will in general have been achieved through the use of step bounds. The `VSOLINDEX` is the fraction of delta vectors which are *not* at a bound in the solution. Therefore, a value of 1.0 means that no delta is at a step bound and therefore the solution is at a vertex of the feasible region. Smaller values indicate that there are deltas at step bounds and so the solution is further from being a vertex solution.

19.2 Integer problem attributes

XSLP_COEFFICIENTS

Description Number of nonlinear coefficients

Type Integer

Set by routines XSLPaddcoefs, XSLPchgcoef, XSLPloadcoefs, XSLPreadprob

XSLP_CVS

Description Number of character variables

Type Integer

Set by routines XSLPreadprob

XSLP_DELTAS

Description Number of delta vectors created during augmentation

Type Integer

Set by routines XSLPconstruct

XSLP_ECFCOUNT

Description Number of infeasible constraints found at the point of linearization

Type Integer

Set by routines XSLPmaxim, XSLPminim

See also XSLP_ECFCHECK, XSLP_ECFTOL_A, XSLP_ECFTOL_R

XSLP_EXPLOREDELTAS

Description Number of variables with an exploration-type delta set up in the problem

Type Integer

Set by routines XSLPconstruct

XSLP_EQUALSCOLUMN

Description Index of the reserved "=" column

Type Integer

Note If there had been no "=" column present, it will be assumed that the user needs the index to add nonlinear terms into the problem that are not coefficients, and an "=" columns will be added to the problem, whose index is then returned. Please note, that this means that a call to XSLPgetintattrib with this attribute might make a slight modification to the problem itself.

Set by routines `XSLPconstruct`, `XSLPreadprob`

XSLP_IFS

Description Number of internal functions

Type Integer

Set by routines `XSLPcreateprob`

XSLP_IMPLICITVARIABLES

Description Number of SLP variables appearing only in coefficients

Type Integer

Set by routines `XSLPconstruct`

XSLP_INTEGERDELTAS

Description Number of variables set up with an integer delta in the problem

Type Integer

Set by routines `XSLPconstruct`

XSLP_INTERNALFUNCCALLS

Description Number of calls made to internal functions

Type Integer

Set by routines `XSLPcascade`, `XSLPconstruct`, `XSLPevaluatecoef`, `XSLPevaluateformula`, `XSLPmaxim`, `XSLPminim`

XSLP_ITER

Description SLP iteration count

Type Integer

Set by routines XSLPmaxim, XSLPminim

XSLP_JOBID

Description Unique identifier for the current job

Type Integer

Note Assigned when a job is created, and can be used to identify jobs in callbacks. Note that all callback receives an optional job name that can be assigned at job creation time.

Set by routines XSLPmaxim, XSLPminim

XSLP_KEEPBESTITER

Description The iteration in which the returned solution has been found.

Type Integer

Note A zero value indicates no solution or the filter option is off. A value of '-1' indicates the initial solution has been returned.

Set by routines XSLPmaxim, XSLPminim

XSLP_MINORVERSION

Description Xpress NonLinear minor version number

Type Integer

Set by routines XSLPinit

XSLP_MINUSPENALTYERRORS

Description Number of negative penalty error vectors

Type Integer

Set by routines XSLPconstruct

XSLP_MIPITER

Description Total number of SLP iterations in MISLP

Type Integer

Set by routines XSLPglobal

XSLP_MIPNODES

Description Number of nodes explored in MISLP. This includes any nodes for which a non-linear solve has been carried out.

Type Integer

Set by routines XSLPglobal

XSLP_MIPSOLS

Description Number of integer solutions found in MISLP. This includes solutions found during the tree search or any heuristics.

Type Integer

Set by routines XSLPglobal

XSLP_MODELCOLS

Description Number of model columns in the problem

Type Integer

Note This is the number of columns currently in the problem without any augmentation, i.e. the number of columns that describe the algebraic definition of the problem. These columns always precede the augmentation columns in order. If the problem is presolved, this may be smaller than the number of original columns in the problem. To access the number of original columns, use XSLP_ORIGINALCOLS.

XSLP_MODELROWS

Description Number of model rows in the problem

Type Integer

Note This is the number of rows currently in the problem without any augmentation, i.e. the number of rows that describe the algebraic definition of the problem. These rows always precede the augmentation rows in order. If the problem is presolved, this may be smaller than the number of original rows in the problem. To access the number of original rows, use XSLP_ORIGINALROWS.

XSLP_MSSTATUS

Description Status of the mutlistart search

Type Integer

Note The value matches that of the winner job if the multistart search completes and a feasible solution has been found. If no solution is found, it is set to XSLP_NLPSTATUS_INFEASIBLE. If the search is terminated early, it is set to XSLP_NLPSTATUS_UNFINISHED (thought in which case the winner if any is still synchronized to the base problem and the solution and XSLP_NLPSTATUS is available).

XSLP_NLPSTATUS

Description The solution status of the problem.

Type Integer

Values

0	Optimization unstarted (XSLP_NLPSTATUS_UNSTARTED)
1	Solution found (XSLP_NLPSTATUS_SOLUTION)
2	Globally optimal (XSLP_NLPSTATUS_OPTIMAL)
3	No solution found (XSLP_NLPSTATUS_NOSOLUTION)
4	Proven infeasible (XSLP_NLPSTATUS_INFEASIBLE)
5	Locally unbounded (XSLP_NLPSTATUS_UNBOUNDED)
6	Problem could not be solved due to numerical issues. (XSLP_NLPSTATUS_UNFINISHED)
7	Unsolved (XSLP_NLPSTATUS_UNSOLVED)

Default value 0

Set by routines XSLPminim, XSLPmaxim, XSLPglobal

XSLP_NONCONSTANTCOEFF

Description Number of coefficients in the augmented problem that might change between SLP iterations

Type Integer

Set by routines XSLPconstruct

XSLP_NONLINEARCONSTRAINTS

Description Number of nonlinear constraints in the problem

Type Integer

Set by routines `XSLPconstruct`

XSLP_ORIGINALCOLS

Description Number of model columns in the problem

Type Integer

Note The number of columns in the original matrix before presolveing without any augmentation columns.

XSLP_ORIGINALROWS

Description Number of model rows in the problem

Type Integer

Note The number of rows in the original matric before presolveing without any augmentation rows.

XSLP_PENALTYDELTACOLUMN

Description Index of column costing the penalty delta row

Type Integer

Note This index always counts from 1. It is zero if there is no penalty delta row.

Set by routines `XSLPconstruct`

XSLP_PENALTYDELTAROW

Description Index of equality row holding the penalties for delta vectors

Type Integer

Note This index always counts from 1. It is zero if there are no penalty delta vectors.

Set by routines `XSLPconstruct`

XSLP_PENALTYDELTAS

Description Number of penalty delta vectors

Type Integer

Set by routines `XSLPconstruct`

XSLP_PENALTYERRORCOLUMN

Description Index of column costing the penalty error row

Type Integer

Note This index always counts from 1. It is zero if there is no penalty error row.

Set by routines `XSLPconstruct`

XSLP_PENALTYERRORROW

Description Index of equality row holding the penalties for penalty error vectors

Type Integer

Note This index always counts from 1. It is zero if there are no penalty error vectors.

Set by routines `XSLPconstruct`

XSLP_PENALTYERRORS

Description Number of penalty error vectors

Type Integer

Set by routines `XSLPconstruct`

XSLP_PLUSPENALTYERRORS

Description Number of positive penalty error vectors

Type Integer

Set by routines `XSLPconstruct`

XSLP_PRESOLVEDELETEDDELTA

Description Number of potential delta variables deleted by XSLPpresolve

Type Integer

Note A potential delta variable is deleted when an SLP variable is identified as not interacting in a nonlinear way with any constraints (that is, it appears only in linear constraints, or is fixed).

Set by routines XSLPpresolve

See also XSLP_PRESOLVEFIXEDCOEF, XSLP_PRESOLVEFIXEDDR, XSLP_PRESOLVEFIXEDNZCOL, XSLP_PRESOLVEFIXEDSLPVAR, XSLP_PRESOLVEFIXEDZCOL, XSLP_PRESOLVETIGHTENED, XSLP_PRESOLVEELIMINATIONS

XSLP_PRESOLVEELIMINATIONS

Description Number of SLP variables eliminated by XSLPpresolve

Type Integer

Set by routines XSLPpresolve

See also XSLP_PRESOLVEDELETEDDELTA, XSLP_PRESOLVEFIXEDDR, XSLP_PRESOLVEFIXEDNZCOL, XSLP_PRESOLVEFIXEDSLPVAR, XSLP_PRESOLVEFIXEDZCOL, XSLP_PRESOLVETIGHTENED

XSLP_PRESOLVEFIXEDCOEF

Description Number of SLP coefficients fixed by XSLPpresolve

Type Integer

Set by routines XSLPpresolve

See also XSLP_PRESOLVEDELETEDDELTA, XSLP_PRESOLVEFIXEDDR, XSLP_PRESOLVEFIXEDNZCOL, XSLP_PRESOLVEFIXEDSLPVAR, XSLP_PRESOLVEFIXEDZCOL, XSLP_PRESOLVETIGHTENED, XSLP_PRESOLVEELIMINATIONS

XSLP_PRESOLVEFIXEDDR

Description Number of determining rows fixed by XSLPpresolve

Type Integer

Set by routines XSLPpresolve

See also XSLP_PRESOLVEDELETEDDELTA, XSLP_PRESOLVEFIXEDCOEF, XSLP_PRESOLVEFIXEDNZCOL, XSLP_PRESOLVEFIXEDSLPVAR, XSLP_PRESOLVEFIXEDZCOL, XSLP_PRESOLVETIGHTENED, XSLP_PRESOLVEELIMINATIONS

XSLP_PRESOLVEFIXEDNZCOL

Description Number of variables fixed to a nonzero value by XSLPpresolve

Type Integer

Set by routines XSLPpresolve

See also XSLP_PRESOLVEDELETEDDELTA, XSLP_PRESOLVEFIXEDCOEF, XSLP_PRESOLVEFIXEDDR, XSLP_PRESOLVEFIXEDSLPVAR, XSLP_PRESOLVEFIXEDZCOL, XSLP_PRESOLVETIGHTENED, XSLP_PRESOLVEELIMINATIONS

XSLP_PRESOLVEFIXEDSLPVAR

Description Number of SLP variables fixed by XSLPpresolve

Type Integer

Set by routines XSLPpresolve

See also XSLP_PRESOLVEDELETEDDELTA, XSLP_PRESOLVEFIXEDCOEF, XSLP_PRESOLVEFIXEDDR, XSLP_PRESOLVEFIXEDNZCOL, XSLP_PRESOLVEFIXEDZCOL, XSLP_PRESOLVETIGHTENED, XSLP_PRESOLVEELIMINATIONS

XSLP_PRESOLVEFIXEDZCOL

Description Number of variables fixed at zero by XSLPpresolve

Type Integer

Set by routines XSLPpresolve

See also XSLP_PRESOLVEDELETEDDELTA, XSLP_PRESOLVEFIXEDCOEF, XSLP_PRESOLVEFIXEDDR, XSLP_PRESOLVEFIXEDNZCOL, XSLP_PRESOLVEFIXEDSLPVAR, XSLP_PRESOLVETIGHTENED, XSLP_PRESOLVEELIMINATIONS

XSLP_PRESOLVEPASSES

Description Number of passes made by the SLP nonlinear presolve procedure

Type Integer

Set by routines XSLPpresolve

XSLP_PRESOLVESTATE

Description Indicates if the problem is presolved

Type Integer

Values

0	The problem is not presolved
1	The problem is presolved, but no columns or rows have been removed from the problem
2	The problem is fully presolved, and the column and row indices do not match the original problem

Set by routines XSLPmaxim, XSLPminim, XSLPpresolve

XSLP_PRESOLVETIGHTENED

Description Number of bounds tightened by XSLPpresolve

Type Integer

Set by routines XSLPpresolve

See also XSLP_PRESOLVEDELETEDDELTA, XSLP_PRESOLVEFIXEDCOEF, XSLP_PRESOLVEFIXEDDR, XSLP_PRESOLVEFIXEDNZCOL, XSLP_PRESOLVEFIXEDSLPVAR, XSLP_PRESOLVEFIXEDZCOL, XSLP_PRESOLVEELIMINATIONS

XSLP_SBXCONVERGED

Description Number of step-bounded variables converged only on extended criteria

Type Integer

Set by routines XSLPmaxim, XSLPminim

XSLP_SEMICONTDELTAS

Description Number of variables with a minimum perturbation step set up in the problem

Type Integer

Set by routines XSLPconstruct

XSLP_SOLVERSELECTED

Description Includes information of which Xpress solver has been used to solve the problem

Type Integer

Values

−1	Unset
0	Xpress-SLP
1	Knitro (Artelys)
2	Xpress Optimizer

Default value -1

Set by routines XSLPmaxim, XSLPminim

Note The following constants are provided:

```
0 XSLP_SOLVER_XSLP
1 XSLP_SOLVER_KNITRO
2 XSLP_SOLVER_OPTIMIZER
```

XSLP_SOLSTATUS

Description Indicates the type of solution returned by the solver.

Type Integer

Values

0	No solution available.
1	A solution with no dual information.
2	A locally optimal solution with dual information.
3	A globally optimal solution without dual information.
4	A globally optimal solution with dual information.

Default value 0

Set by routines XSLPminim, XSLPmaxim, XSLPglobal

XSLP_STATUS

Description Bitmap holding the problem convergence status

Type Integer

Values

Bit	Meaning
0	Converged on objective function with no unconverged values in active constraints.
1	Converged on objective function with some variables converged on extended criteria only.
2	LP solution is infeasible.
3	LP solution is unfinished (not optimal or infeasible).
4	SLP terminated on maximum SLP iterations.
5	SLP is integer infeasible.
6	SLP converged with residual penalty errors.
7	Converged on objective.
9	SLP terminated on max time.
10	SLP terminated by user.
11	Some variables are linked to active constraints.
12	No unconverged values in active constraints.
13	OTOL is satisfied - range of objective change small, active step bounds.
14	VTOL is satisfied - range of objective change is small.
15	XTOL is satisfied - range of objective change small, no unconverged in active.
16	WTOL is satisfied - convergence continuation.
17	ERRORTOL satisfied - penalties not increased further.
18	EVTOL satisfied - penalties not increased further.
19	There were iterations where the solution had to be polished.
20	There were iterations where the solution polishing failed.
21	There were iterations where rows were enforced.
22	Terminated due to XSLP_INFEASLIMIT.

Note

A value of zero after SLP optimization means that the solution is fully converged.

The following constants are provided for checking these bits:

Setting bit 0	`XSLP_STATUS_CONVERGEDOBJUCC`
Setting bit 1	`XSLP_STATUS_CONVERGEDOBJSBX`
Setting bit 2	`XSLP_STATUS_LPINFEASIBLE`
Setting bit 3	`XSLP_STATUS_LPUNFINISHED`
Setting bit 4	`XSLP_STATUS_MAXSLPITERATIONS`
Setting bit 5	`XSLP_STATUS_INTEGERINFEASIBLE`
Setting bit 6	`XSLP_STATUS_RESIDUALPENALTIES`
Setting bit 7	`XSLP_STATUS_CONVERGEDOBJOBJ`
Setting bit 9	`XSLP_STATUS_MAXTIME`
Setting bit 10	`XSLP_STATUS_USER`
Setting bit 11	`XSLP_STATUS_VARSLINKEDINACTIVE`
Setting bit 12	`XSLP_STATUS_NOVARSINACTIVE`
Setting bit 13	`XSLP_STATUS_OTOL`
Setting bit 14	`XSLP_STATUS_VTOL`
Setting bit 15	`XSLP_STATUS_XTOL`
Setting bit 16	`XSLP_STATUS_WTOL`
Setting bit 17	`XSLP_STATUS_ERROTOL`
Setting bit 18	`XSLP_STATUS_EVTOL`
Setting bit 19	`XSLP_STATUS_POLISHED`
Setting bit 20	`XSLP_STATUS_POLISH_FAILURE`
Setting bit 21	`XSLP_STATUS_ENFORCED`
Setting bit 22	`XSLP_STATUS_CONSECUTIVE_INFEAS`

Set by routines XSLPmaxim, XSLPminim

XSLP_STOPSTATUS

Description Status of the optimization process.

Type Integer

Note Possible values are:

Value	Description
XSLP_STOP_NONE	no interruption - the solve completed normally
XSLP_STOP_TIMELIMIT	time limit hit
XSLP_STOP_CTRLC	control C hit
XSLP_STOP_NODELIMIT	node limit hit
XSLP_STOP_ITERLIMIT	iteration limit hit
XSLP_STOP_MIPGAP	MIP gap is sufficiently small
XSLP_STOP_SOLLIMIT	solution limit hit
XSLP_STOP_USER	user interrupt.

Set by routines XSLPnlpoptimize, XSLPmaxim, XSLPminim, XSLPglobal.

XSLP_TOLSETS

Description Number of tolerance sets

Type Integer

Set by routines XSLPaddtolsets, XSLPchgtolset, XSLPloadtolsets, XSLPreadprob

XSLP_TOTALEVALUATIONERRORS

Description The total number of evaluation errors during the solve

Type Integer

Set by routines

Set by routines XSLPnlpoptimize, XSLPmaxim, XSLPminim, XSLPglobal.

XSLP_UCCONSTRAINEDCOUNT

Description Number of unconverged variables with coefficients in constraining rows

Type Integer

Set by routines XSLPmaxim, XSLPminim

XSLP_UFINSTANCES

Description Number of user function instances

Type Integer

Set by routines XSLPconstruct

XSLP_UFS

Description Number of user functions

Type Integer

Set by routines XSLPadduserfunction, XSLPdeluserfunction, XSLPreadprob

XSLP_UNCONVERGED

Description Number of unconverged values

Type Integer

Note Prior to the first iteration this will return -1.

Set by routines XSLPmaxim, XSLPminim

XSLP_USEDERIVATIVES

Description Indicates whether numeric or analytic derivatives were used to create the linear approximations and solve the problem

Type Integer

Values

0	numeric derivatives.
1	analytic derivatives for all formulae unless otherwise specified.

Set by routines XSLPconstruct

XSLP_USERFUNCCALLS

Description Number of calls made to user functions

Type Integer

Set by routines XSLPcascade, XSLPconstruct, XSLPevaluatecoef, XSLPevaluateformula, XSLPmaxim, XSLPminim

XSLP_VARIABLES

Description Number of SLP variables

Type Integer

Set by routines XSLPconstruct

XSLP_VERSION

Description Xpress NonLinear major version number

Type Integer

Set by routines XSLPinit

XSLP_ZEROESRESET

Description Number of placeholder entries set to zero

Type Integer

Note For an explanation of deletion of placeholder entries in the matrix see *Management of zero placeholder entries*.

Set by routines XSLPmaxim, XSLPminim

See also XSLP_ZEROCRITERIONCOUNT, XSLP_ZEROCRITERIONSTART, XSLP_ZEROESRETAINED, XSLP_ZEROESTOTAL, *Management of zero placeholder entries*

XSLP_ZEROESRETAINED

Description Number of potentially zero placeholders left untouched

Type Integer

Note For an explanation of deletion of placeholder entries in the matrix see *Management of zero placeholder entries*.

Set by routines XSLPmaxim, XSLPminim

See also XSLP_ZEROCRITERIONCOUNT, XSLP_ZEROCRITERIONSTART, XSLP_ZEROESRESET, XSLP_ZEROESTOTAL, *Management of zero placeholder entries*

XSLP_ZEROESTOTAL

Description Number of potential zero placeholder entries

Type Integer

Note For an explanation of deletion of placeholder entries in the matrix see *Management of zero placeholder entries*.

Set by routines XSLPmaxim, XSLPminim

See also XSLP_ZEROCRITERIONCOUNT, XSLP_ZEROCRITERIONSTART, XSLP_ZEROESRESET, XSLP_ZEROESRETAINED, *Management of zero placeholder entries*

19.3 Reference (pointer) problem attributes

The reference attributes are void pointers whose size (32 or 64 bit) depends on the platform.

XSLP_MIPPROBLEM

Description The underlying Optimizer MIP problem. `XSLP_MIPPROBLEM` is a reference of type XPRSprob, and should be used in MISLP callbacks to access MIP-specific Optimizer values (such as node and parent numbers).

Type Reference

Set by routines `XSLPglobal`

XSLP_SOLUTIONPOOL

Description The underlying solution pool. `XSLP_SOLUTIONPOOL` is a reference of type XPRSmipsolpool. Change control `XSLP_ANALYZE` to record the solutions into the pool.

Type Reference

Set by routines `XSLPminim`, `XSLPmaxim`

XSLP_XPRSPROBLEM

Description The underlying Optimizer problem

Type Reference

Set by routines `XSLPcreateprob`

XSLP_XSLPPROBLEM

Description The Xpress NonLinear problem

Type Reference

Set by routines `XSLPcreateprob`

19.4 String problem attributes

XSLP_VERSIONDATE

Description Date of creation of Xpress NonLinear

Type String

Note The format of the date is dd mmm yyyy.

Set by routines `XSLPinit`

CHAPTER 20

Control Parameters

Various controls exist within Xpress NonLinear to govern the solution procedure and the form of the output. Some of these take integer values and act as switches between various types of behavior. Many are tolerances on values related to the convergence criteria; these are all double precision. There are also a few controls which are character strings, setting names for structures. Any of these may be altered by the user to enhance performance of the SLP algorithm. In most cases, 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.

Users of the Xpress NonLinear function library are provided with the following set of functions for setting and obtaining control values:

XSLPgetintcontrol	XSLPgetdblcontrol	XSLPgetstrcontrol
XSLPsetintcontrol	XSLPsetdblcontrol	XSLPsetstrcontrol

All the controls as listed in this chapter are prefixed with XSLP_. It is possible to use the above functions with control parameters for the Xpress Optimizer (controls prefixed with XPRS_). For details of the Optimizer controls, see the Optimizer manual.

Example of the usage of the functions:

```
XSLPgetintcontrol(Prob, XSLP_PRESOLVE, &presolve);
printf("The value of PRESOLVE was %d\n", presolve);
XSLPsetintcontrol(Prob, XSLP_PRESOLVE, 1-presolve);
printf("The value of PRESOLVE is now %d\n", 1-presolve);
```

The following is a list of all the Xpress NonLinear controls:

XSLP_ESCALATION	Factor for increasing cost multiplier on individual penalty error vectors	p. 143
XSLP_ETOL_A	Absolute tolerance on penalty vectors	p. 143
XSLP_ETOL_R	Relative tolerance on penalty vectors	p. 143
XSLP_EVALUATE	Evaluation strategy for user functions	p. 180
XSLP_EVTOL_A	Absolute tolerance on total penalty costs	p. 144
XSLP_EVTOL_R	Relative tolerance on total penalty costs	p. 144
XSLP_EXPAND	Multiplier to increase a step bound	p. 145
XSLP_FEASTOLTARGET	When set, this defines a target feasibility tolerance to which the linearizations are solved to	p. 145
XSLP_FILTER	Bit map for controlling solution updates	p. 180
XSLP_FINDIV	Option for running a heuristic to find a feasible initial point	p. 181
XSLP_FUNCEVAL	Bit map for determining the method of evaluating user functions and their derivatives	p. 181
XSLP_GRANULARITY	Base for calculating penalty costs	p. 145
XSLP_GRIDHEURSELECT	Bit map selectin which heuristics to run if the problem has variable with an integer delta	p. 182
XSLP_HESSIAN	Second order differentiation mode when using analytical derivatives	p. 183
XSLP_HEURSTRATEGY	Branch and Bound: This specifies the MINLP heuristic strategy. On some problems it is worth trying more comprehensive heuristic strategies by setting HEURSTRATEGY to 2 or 3.	p. 182
XSLP_INFEASLIMIT	The maximum number of consecutive infeasible SLP iterations which can occur before Xpress-SLP terminates	p. 183
XSLP_INFINITY	Value returned by a divide-by-zero in a formula	p. 146
XSLP_ITERFALLBACKOPS	Alternative LP level control values for numerically challengeing problems	p. 208
XSLP_ITERLIMIT	The maximum number of SLP iterations	p. 183
XSLP_ITOL_A	Absolute impact convergence tolerance	p. 146
XSLP_ITOL_R	Relative impact convergence tolerance	p. 147
XSLP_IVNAME	Name of the set of initial values to be used	p. 209
XSLP_JACOBIAN	First order differentiation mode when using analytical derivatives	p. 184
XSLP_LINQUADBR	Use linear and quadratic constraints and objective function to further reduce bounds on all variables	p. 184
XSLP_LOG	Level of printing during SLP iterations	p. 184
XSLP_LSITERLIMIT	Number of iterations in the line search	p. 185
XSLP_LSPATTERNLIMIT	Number of iterations in the pattern search preceding the line search	p. 185
XSLP_LSSTART	Iteration in which to active the line search	p. 185

20.1 Double control parameters

XSLP_ATOL_A

Description Absolute delta convergence tolerance

Type Double

Note The absolute delta convergence criterion assesses the change in value of a variable (δX) against the absolute delta convergence tolerance. If
$\delta X < XSLP\_ATOL\_A$
then the variable has converged on the absolute delta convergence criterion. When the value is set to be negative, the value is adjusted automatically by SLP, based on the feasibility target XSLP_VALIDATIONTARGET_R. Good values for the control are usually fall between 1e-3 and 1e-6.

Default value -1.0

See also Convergence Criteria, XSLP_ATOL_R

XSLP_ATOL_R

Description Relative delta convergence tolerance

Type Double

Note The relative delta convergence criterion assesses the change in value of a variable (δX) relative to the value of the variable (X), against the relative delta convergence tolerance. If
$\delta X < X * XSLP\_ATOL\_R$
then the variable has converged on the relative delta convergence criterion. When the value is set to be negative, the value is adjusted automatically by SLP, based on the feasibility target XSLP_VALIDATIONTARGET_R. Good values for the control are usually fall between 1e-3 and 1e-6.

Default value -1.0

See also Convergence Criteria, XSLP_ATOL_A

XSLP_BARSTALLINGTOL

Description Required change in the objective when progress is measured in barrier iterations without crossover

Type Double

Note Minumum objective variability change required in relation to control XSLP_BARSTALLINGOBJLIMIT for the iterations to be regarded as making progress. The net objective, error cost and error sum are taken into account.

Default value 0.05

Affects routines XSLPmaxim, XSLPminim

See also XSLP_BARCROSSOVERSTART, XSLP_BARLIMIT, XSLP_BARSTARTOPS, XSLP_BARSTALLINGLIMIT, XSLP_BARSTALLINGOBJLIMIT

XSLP_CASCADETOL_PA

Description Absolute cascading print tolerance

Type Double

Note The change to the value of a variable as a result of cascading is only printed if the change is deemed significant. The change is tested against: absolute and relative convergence tolerance and absolute and relative cascading print tolerance. The change is printed only if all tests fail. The absolute cascading print criterion measures the change in value of a variable (δX) against the absolute cascading print tolerance. If
$\delta X <$ *XSLP_CASCADETOL_PA*
then the change is within the absolute cascading print tolerance and will not be printed. XSLP_LOG must be at least 5 for this control to have an effect.

Default value 0.01

See also Cascading, XSLP_CASCADETOL_PR

Affects routines XSLPcascade

XSLP_CASCADETOL_PR

Description Relative cascading print tolerance

Type Double

Note The change to the value of a variable as a result of cascading is only printed if the change is deemed significant. The change is tested against: absolute and relative convergence tolerance and absolute and relative cascading print tolerance. The change is printed only if all tests fail. The relative cascading print criterion measures the change in value of a variable (δX) relative to the value of the variable (X), against the relative cascading print tolerance. If
$\delta X < X * XSLP\_CASCADETOL\_PR$
then the change is within the relative cascading print tolerance and will not be printed. XSLP_LOG must be at least 5 for this control to have an effect.

Default value 0.01

See also Cascading, XSLP_CASCADETOL_PA

Affects routines XSLPcascade

XSLP_CDTOL_A

Description Absolute tolerance for deducing constant derivatives

Type Double

Note The absolute tolerance test for constant derivatives is used as follows:
If the value of the user function at point X_0 is Y_0 and the values at $(X_0 - \delta X)$ and $(X_0 + \delta X)$ are Y_d and Y_u respectively, then the numerical derivatives at X_0 are:
"down" derivative $D_d = (Y_0 - Y_d)/\delta X$
"up" derivative $D_u = (Y_u - Y_0)/\delta X$

If $abs(D_d - D_u) \leq XSLP\_CDTOL\_A$
then the derivative is regarded as constant.

Default value 1.0e-08

See also XSLP_CDTOL_R

XSLP_CDTOL_R

Description Relative tolerance for deducing constant derivatives

Type Double

Note The relative tolerance test for constant derivatives is used as follows:
If the value of the user function at point X_0 is Y_0 and the values at $(X_0 - \delta X)$ and $(X_0 + \delta X)$ are Y_d and Y_u respectively, then the numerical derivatives at X_0 are:
"down" derivative $D_d = (Y_0 - Y_d)/\delta X$
"up" derivative $D_u = (Y_u - Y_0)/\delta X$

If $abs(D_d - D_u) \leq XSLP\_CDTOL\_R * abs(Y_d + Y_u)/2$
then the derivative is regarded as constant.

Default value 1.0e-08

See also XSLP_CDTOL_A

XSLP_CLAMPSHRINK

Description Shrink ratio used to impose strict convergence on variables converged in extended criteria only

Type Double

Note If the solution has converged but there are variables converged on extended criteria only, the XSLP_CLAMPSHRINK acts as a shrinking ratio on the step bounds and the problem is optimized (if necessary multiple times), with the purpose of expediting strict convergence on all variables. XSLP_ALGORITHM controls if this shrinking is applied at all, and if shrinking is applied to of the variables converged on extended criteria only with active step bounds only, or if on all variables.

Default value 0.3

See also XSLP_ALGORITHM, XSLP_CLAMPVALIDATIONTOL_A, XSLP_CLAMPVALIDATIONTOL_R

XSLP_CLAMPVALIDATIONTOL_A

Description Absolute validation tolerance for applying XSLP_CLAMPSHRINK

Type Double

Note If set and the absolute validation value is larger than this value, then control XSLP_CLAMPSHRINK is checked once the solution has converged, but there are variables converged on extended criteria only.

Default value 0.0 (not set)

See also XSLP_ALGORITHM, XSLP_CLAMPSHRINK, XSLP_CLAMPVALIDATIONTOL_R

XSLP_CLAMPVALIDATIONTOL_R

Description Relative validation tolerance for applying XSLP_CLAMPSHRINK

Type Double

Note If set and the relative validation value is larger than this value, then control XSLP_CLAMPSHRINK is checked once the solution has converged, but there are variables converged on extended criteria only.

Default value 0.0 (not set)

See also XSLP_ALGORITHM, XSLP_CLAMPSHRINK, XSLP_CLAMPVALIDATIONTOL_A

XSLP_CTOL

Description Closure convergence tolerance

Type Double

Notes The closure convergence criterion measures the change in value of a variable (δX) relative to the value of its initial step bound (B), against the closure convergence tolerance. If
$\delta X < B * XSLP\_CTOL$
then the variable has converged on the closure convergence criterion.
If no explicit initial step bound is provided, then the test will not be applied and the variable can never converge on the closure criterion. When the value is set to be negative, the value is adjusted automatically by SLP, based on the feasibility target XSLP_VALIDATIONTARGET_R. Good values for the control are usually fall between 1e-3 and 1e-6.

Default value -1.0

See also Convergence Criteria, XSLP_ATOL_A, XSLP_ATOL_R

XSLP_DAMP

Description Damping factor for updating values of variables

Type Double

Note The damping factor sets the next *assumed value* for a variable based on the previous assumed value (X_0) and the *actual value* (X_1). The new assumed value is given by $X_1 * XSLP\_DAMP + X_0 * (1 - XSLP\_DAMP)$

Default value 1

See also Xpress-SLP Solution Process, XSLP_DAMPEXPAND XSLP_DAMPMAX, XSLP_DAMPMIN, XSLP_DAMPSHRINK, XSLP_DAMPSTART

Affects routines XSLPmaxim, XSLPminim

XSLP_DAMPEXPAND

Description Multiplier to increase damping factor during dynamic damping

Type Double

Note If dynamic damping is enabled, the damping factor for a variable will be increased if successive changes are in the same direction. More precisely, if there are XSLP_SAMEDAMP successive changes in the same direction for a variable, then the damping factor (D) for the variable will be reset to
$D * XSLP\_DAMPEXPAND + XSLP\_DAMPMAX * (1 - XSLP\_DAMPEXPAND)$

Default value 1

See also Xpress-SLP Solution Process, XSLP_ALGORITHM, XSLP_DAMP, XSLP_DAMPMAX, XSLP_DAMPMIN, XSLP_DAMPSHRINK, XSLP_DAMPSTART, XSLP_SAMEDAMP

Affects routines XSLPmaxim, XSLPminim

XSLP_DAMPMAX

Description Maximum value for the damping factor of a variable during dynamic damping

Type Double

Note If dynamic damping is enabled, the damping factor for a variable will be increased if successive changes are in the same direction. More precisely, if there are XSLP_SAMEDAMP successive changes in the same direction for a variable, then the damping factor (D) for the variable will be reset to
$D * XSLP\_DAMPEXPAND + XSLP\_DAMPMAX * (1 - XSLP\_DAMPEXPAND)$

Default value 1

See also Xpress-SLP Solution Process, XSLP_ALGORITHM, XSLP_DAMP, XSLP_DAMPEXPAND, XSLP_DAMPMIN, XSLP_DAMPSHRINK, XSLP_DAMPSTART, XSLP_SAMEDAMP

Affects routines XSLPmaxim, XSLPminim

XSLP_DAMPMIN

Description Minimum value for the damping factor of a variable during dynamic damping

Type Double

Note If dynamic damping is enabled, the damping factor for a variable will be decreased if successive changes are in the opposite direction. More precisely, the damping factor (*D*) for the variable will be reset to
D ∗ *XSLP_DAMPSHRINK* + *XSLP_DAMPMIN* ∗ (1 − *XSLP_DAMPEXPAND*)

Default value 1

See also Xpress-SLP Solution Process, XSLP_ALGORITHM, XSLP_DAMP, XSLP_DAMPEXPAND, XSLP_DAMPMAX, XSLP_DAMPSHRINK, XSLP_DAMPSTART

Affects routines XSLPmaxim, XSLPminim

XSLP_DAMPSHRINK

Description Multiplier to decrease damping factor during dynamic damping

Type Double

Note If dynamic damping is enabled, the damping factor for a variable will be decreased if successive changes are in the opposite direction. More precisely, the damping factor (*D*) for the variable will be reset to
D ∗ *XSLP_DAMPSHRINK* + *XSLP_DAMPMIN* ∗ (1 − *XSLP_DAMPEXPAND*)

Default value 1

See also Xpress-SLP Solution Process, XSLP_ALGORITHM, XSLP_DAMP, XSLP_DAMPEXPAND, XSLP_DAMPMAX, XSLP_DAMPMIN, XSLP_DAMPSTART

Affects routines XSLPmaxim, XSLPminim

XSLP_DEFAULTIV

Description Default initial value for an SLP variable if none is explicitly given

Type Double

Note If no initial value is given for an SLP variable, then the initial value provided for the "equals column" will be used. If no such value has been provided, then XSLP_DEFAULTIV will be used. If this is above the upper bound for the variable, then the upper bound will be used; if it is below the lower bound for the variable, then the lower bound will be used.

Default value 100

Affects routines XSLPconstruct

XSLP_DEFAULTSTEPBOUND

Description Minimum initial value for the step bound of an SLP variable if none is explicitly given

Type Double

Notes If no initial step bound value is given for an SLP variable, this will be used as a minimum value. If the algorithm is estimating step bounds, then the step bound actually used for a variable may be larger than the default.
A default initial step bound is ignored when testing for the closure tolerance XSLP_CTOL: if there is no specific value, then the test will not be applied.

Default value 16

See also XSLP_CTOL

Affects routines XSLPconstruct

XSLP_DELTA_A

Description Absolute perturbation of values for calculating numerical derivatives

Type Double

Note First-order derivatives are calculated by perturbing the value of each variable in turn by a small amount. The amount is determined by the absolute and relative delta factors as follows:
$XSLP\_DELTA\_A + abs(X) * XSLP\_DELTA\_R$
where (X) is the current value of the variable. If the perturbation takes the variable outside a bound, then the perturbation normally made only in the opposite direction.

Default value 0.001

Affects routines XSLPmaxim, XSLPminim

See also XSLP_DELTA_R

XSLP_DELTA_R

Description Relative perturbation of values for calculating numerical derivatives

Type Double

Note First-order derivatives are calculated by perturbing the value of each variable in turn by a small amount. The amount is determined by the absolute and relative delta factors as follows:
$XSLP\_DELTA\_A + abs(X) * XSLP\_DELTA\_R$
where (X) is the current value of the variable. If the perturbation takes the variable outside a bound, then the perturbation normally made only in the opposite direction.

Default value 0.001

Affects routines XSLPmaxim, XSLPminim

See also XSLP_DELTA_A

XSLP_DELTA_X

Description Minimum absolute value of delta coefficients to be retained

Type Double

Notes If the value of a coefficient in a delta column is less than this value, it will be reset to zero. Larger values of XSLP_DELTA_X will result in matrices with fewer elements, which may be easier to solve. However, there will be increased likelihood of local optima as some of the small relationships between variables and constraints are deleted. There may also be increased difficulties with singular bases resulting from deletion of pivot elements from the matrix.

Default value 1.0e-6

Affects routines XSLPmaxim, XSLPminim

XSLP_DELTA_Z

Description Tolerance used when calculating derivatives

Type Double

Notes If the absolute value of a variable is less than this value, then a value of XSLP_DELTA_Z will be used instead for calculating derivatives.
If a nonzero derivative is calculated for a formula which always results in a matrix coefficient less than XSLP_DELTA_Z, then a larger value will be substituted so that at least one of the coefficients is XSLP_DELTA_Z in magnitude.
If XSLP_DELTAZLIMIT is set to a positive number, then when that number of iterations have passed, values smaller than XSLP_DELTA_Z will be set to zero.

Default value 0.00001

Affects routines XSLPmaxim, XSLPminim

See also XSLP_DELTAZLIMIT, XSLP_DELTA_ZERO

XSLP_DELTA_ZERO

Description Absolute zero acceptance tolerance used when calculating derivatives

Type Double

Notes Provides an override value for the XSLP_DELTA_Z behavior. Derivatives smaller than XSLP_DELTA_ZERO will not be substituted by XSLP_DELTA_Z, defining a range in which derivatives are deemed nonzero and are affected by XSLP_DELTA_Z.
A negative value means that this tolerance will not be applied.

Default value -1.0 (not applied)

Affects routines XSLPmaxim, XSLPminim

See also XSLP_DELTAZLIMIT, XSLP_DELTA_Z

XSLP_DELTACOST

Description Initial penalty cost multiplier for penalty delta vectors

Type Double

Note If penalty delta vectors are used, this parameter sets the initial cost factor. If there are active penalty delta vectors, then the penalty cost may be increased.

Default value 200

Affects routines XSLPmaxim, XSLPminim

See also XSLP_AUGMENTATION, XSLP_DELTACOSTFACTOR, XSLP_DELTAMAXCOST, XSLP_ERRORCOST

XSLP_DELTACOSTFACTOR

Description Factor for increasing cost multiplier on total penalty delta vectors

Type Double

Note If there are active penalty delta vectors, then the penalty cost multiplier will be increased by a factor of XSLP_DELTACOSTFACTOR up to a maximum of XSLP_DELTAMAXCOST

Default value 1.3

Affects routines XSLPmaxim, XSLPminim

See also XSLP_AUGMENTATION, XSLP_DELTACOST, XSLP_DELTAMAXCOST, XSLP_ERRORCOST

XSLP_DELTAMAXCOST

Description Maximum penalty cost multiplier for penalty delta vectors

Type Double

Note If there are active penalty delta vectors, then the penalty cost multiplier will be increased by a factor of XSLP_DELTACOSTFACTOR up to a maximum of XSLP_DELTAMAXCOST

Default value infinite

Affects routines XSLPmaxim, XSLPminim

See also XSLP_AUGMENTATION, XSLP_DELTACOST, XSLP_DELTACOSTFACTOR, XSLP_ERRORCOST

XSLP_DJTOL

Description Tolerance on DJ value for determining if a variable is at its step bound

Type Double

Note If a variable is at its step bound and within the absolute delta tolerance XSLP_ATOL_A or closure tolerance XSLP_CTOL then the step bounds will not be further reduced. If the DJ is greater in magnitude than XSLP_DJTOL then the step bound may be relaxed if it meets the necessary criteria.

Default value 1.0e-6

Affects routines XSLPmaxim, XSLPminim

See also XSLP_ATOL_A, XSLP_CTOL

XSLP_DRCOLTOL

Description The minimum absolute magnitude of a determining column, for which the determined variable is still regarded as well defined

Type Double

Notes This control affects the cascading procedure. Please see Chapter *Cascading* for more information.

Default value 0

See also XSLP_CASCADE

Affects routines XSLPconstruct XSLPcascade

XSLP_ECFTOL_A

Description Absolute tolerance on testing feasibility at the point of linearization

Type Double

Notes The extended convergence criteria test how well the linearization approximates the true problem. They depend on the point of linearization being a reasonable approximation — in particular, that it should be reasonably close to feasibility. Each constraint is tested at the point of linearization, and the total positive and negative contributions to the constraint from the columns in the problem are calculated. A feasibility tolerance is calculated as the largest of $XSLP_ECFTOL_A$ and
$max(abs(Positive), abs(Negative)) * XSLP\_ECFTOL\_R$
If the calculated infeasibility is greater than the tolerance, the point of linearization is regarded as infeasible and the extended convergence criteria will not be applied. When the value is set to be negative, the value is adjusted automatically by SLP, based on the feasibility target XSLP_VALIDATIONTARGET_R. Good values for the control are usually fall between 1e-1 and 1e-6.

Default value -1.0

Affects routines XSLPmaxim, XSLPminim

See also Convergence criteria, XSLP_ECFCHECK, XSLP_ECFCOUNT, XSLP_ECFTOL_R

XSLP_ECFTOL_R

Description Relative tolerance on testing feasibility at the point of linearization

Type Double

Notes The extended convergence criteria test how well the linearization approximates the true problem. They depend on the point of linearization being a reasonable approximation — in particular, that it should be reasonably close to feasibility. Each constraint is tested at the point of linearization, and the total positive and negative contributions to the constraint from the columns in the problem are calculated. A feasibility tolerance is calculated as the largest of $XSLP_ECFTOL_A$ and
$max(abs(Positive), abs(Negative)) * XSLP\_ECFTOL\_R$
If the calculated infeasibility is greater than the tolerance, the point of linearization is regarded as infeasible and the extended convergence criteria will not be applied. When the value is set to be negative, the value is adjusted automatically by SLP, based on the feasibility target XSLP_VALIDATIONTARGET_R. Good values for the control are usually fall between 1e-1 and 1e-6.

Default value -1.0

Affects routines XSLPmaxim, XSLPminim

See also Convergence criteria, XSLP_ECFCHECK, XSLP_ECFCOUNT, XSLP_ECFTOL_A

XSLP_ENFORCECOSTSHRINK

Description Factor by which to decrease the current penalty multiplier when enforcing rows.

Type Double

Notes When feasiblity of a row cannot be achieved by increasing the penalty cost on its error variable, removing the variable (fixing it to zero) can force the row to be satisfied, as set by XSLP_ENFORCEMAXCOST. After the error variables have been removed (which is equivalent to setting to row to be enforced) the penalties on the remaining error variables are rebalanced to allow for a reduction in the size of the penalties in the objective in order to achive better numerical behaviour.

Default value 0.00001

Affects routines XSLPmaxim, XSLPminim

See also XSLP_ENFORCEMAXCOST

XSLP_ENFORCEMAXCOST

Description Maximum penalty cost in the objective before enforcing most violating rows

Type Double

Notes When feasiblity of a row cannot be achieved by increasing the penalty cost on its error variable, removing the variable (fixing it to zero) can force the row to be satisfied. After the error variables have been removed (which is equivalent to setting to row to be enforced) the penalties on the remaining error variables are rebalanced to allow for a reduction in the size of the penalties in the objective in order to achive better numerical behaviour, controlled by XSLP_ENFORCECOSTSHRINK.

Default value 10000000000

Affects routines XSLPmaxim, XSLPminim

See also XSLP_ENFORCECOSTSHRINK

XSLP_ERRORCOST

Description Initial penalty cost multiplier for penalty error vectors

Type Double

Note If penalty error vectors are used, this parameter sets the initial cost factor. If there are active penalty error vectors, then the penalty cost may be increased.

Default value 200

Affects routines XSLPmaxim, XSLPminim

See also XSLP_AUGMENTATION, XSLP_DELTACOST, XSLP_ERRORCOSTFACTOR, XSLP_ERRORMAXCOST

XSLP_ERRORCOSTFACTOR

Description Factor for increasing cost multiplier on total penalty error vectors

Type Double

Note If there are active penalty error vectors, then the penalty cost multiplier will be increased by a factor of XSLP_ERRORCOSTFACTOR up to a maximum of XSLP_ERRORMAXCOST

Default value 1.3

Affects routines XSLPmaxim, XSLPminim

See also XSLP_AUGMENTATION, XSLP_DELTACOST, XSLP_ERRORCOST, XSLP_ERRORMAXCOST

XSLP_ERRORMAXCOST

Description Maximum penalty cost multiplier for penalty error vectors

Type Double

Note If there are active penalty error vectors, then the penalty cost multiplier will be increased by a factor of `XSLP_ERRORCOSTFACTOR` up to a maximum of `XSLP_ERRORMAXCOST`

Default value infinite

Affects routines `XSLPmaxim`, `XSLPminim`

See also `XSLP_AUGMENTATION`, `XSLP_DELTACOST`, `XSLP_ERRORCOST`, `XSLP_ERRORCOSTFACTOR`

XSLP_ERRORTOL_A

Description Absolute tolerance for error vectors

Type Double

Note The solution will be regarded as having no active error vectors if one of the following applies:
every penalty error vector and penalty delta vector has an activity less than *XSLP_ERRORTOL_A*;
the sum of the cost contributions from all the penalty error and penalty delta vectors is less than *XSLP_EVTOL_A*;
the sum of the cost contributions from all the penalty error and penalty delta vectors is less than $XSLP\_EVTOL\_R * Obj$ where *Obj* is the current objective function value.

Default value 0.00001

Affects routines `XSLPmaxim`, `XSLPminim`

See also `XSLP_EVTOL_A`, `XSLP_EVTOL_R`

XSLP_ERRORTOL_P

Description Absolute tolerance for printing error vectors

Type Double

Note The solution log includes a print of penalty delta and penalty error vectors with an activity greater than *XSLP_ERRORTOL_P*.

Default value 0.0001

Affects routines `XSLPmaxim`, `XSLPminim`

XSLP_ESCALATION

Description Factor for increasing cost multiplier on individual penalty error vectors

Type Double

Note If penalty cost escalation is activated in XSLP_ALGORITHM then the penalty cost multiplier will be increased by a factor of XSLP_ESCALATION for any active error vector up to a maximum of XSLP_MAXWEIGHT.

Default value 1.25

Affects routines XSLPmaxim, XSLPminim

See also XSLP_ALGORITHM, XSLP_DELTACOST, XSLP_ERRORCOST, XSLP_MAXWEIGHT

XSLP_ETOL_A

Description Absolute tolerance on penalty vectors

Type Double

Note For each penalty error vector, the contribution to its constraint is calculated, together with the total positive and negative contributions to the constraint from other vectors. If its contribution is less than *XSLP_ETOL_A* or less than *Positive* $*$ *XSLP_ETOL_R* or less than *abs*(*Negative*) $*$ *XSLP_ETOL_R* then it will be regarded as insignificant and will not have its penalty increased. When the value is set to be negative, the value is adjusted automatically by SLP, based on the feasibility target XSLP_VALIDATIONTARGET_R. Good values for the control are usually fall between 1e-3 and 1e-6.

Default value -1.0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_ETOL_R XSLP_DELTACOST, XSLP_ERRORCOST, XSLP_ESCALATION

XSLP_ETOL_R

Description Relative tolerance on penalty vectors

Type Double

Note For each penalty error vector, the contribution to its constraint is calculated, together with the total positive and negative contributions to the constraint from other vectors. If its contribution is less than *XSLP_ETOL_A* or less than *Positive* $*$ *XSLP_ETOL_R* or less than *abs*(*Negative*) $*$ *XSLP_ETOL_R* then it will be regarded as insignificant and will not have its penalty increased. When the value is set to be negative, the value is adjusted automatically by SLP, based on the feasibility target XSLP_VALIDATIONTARGET_R. Good values for the control are usually fall between 1e-3 and 1e-6.

Default value -1.0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_ETOL_A XSLP_DELTACOST, XSLP_ERRORCOST, XSLP_ESCALATION

XSLP_EVTOL_A

Description Absolute tolerance on total penalty costs

Type Double

Note The solution will be regarded as having no active error vectors if one of the following applies:
every penalty error vector and penalty delta vector has an activity less than *XSLP_ERRORTOL_A*;
the sum of the cost contributions from all the penalty error and penalty delta vectors is less than *XSLP_EVTOL_A*;
the sum of the cost contributions from all the penalty error and penalty delta vectors is less than *XSLP_EVTOL_R* $*$ *Obj* where *Obj* is the current objective function value. When the value is set to be negative, the value is adjusted automatically by SLP, based on the feasibility target XSLP_VALIDATIONTARGET_R. Good values for the control are usually fall between 1e-2 and 1e-6, but normally a magnitude larger than XSLP_ETOL_A.

Default value -1.0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_ERRORTOL_A, XSLP_EVTOL_R

XSLP_EVTOL_R

Description Relative tolerance on total penalty costs

Type Double

Note The solution will be regarded as having no active error vectors if one of the following applies:
every penalty error vector and penalty delta vector has an activity less than *XSLP_ERRORTOL_A*;
the sum of the cost contributions from all the penalty error and penalty delta vectors is less than *XSLP_EVTOL_A*;
the sum of the cost contributions from all the penalty error and penalty delta vectors is less than *XSLP_EVTOL_R* $*$ *Obj* where *Obj* is the current objective function value. When the value is set to be negative, the value is adjusted automatically by SLP, based on the feasibility target XSLP_VALIDATIONTARGET_R. Good values for the control are usually fall between 1e-2 and 1e-6, but normally a magnitude larger than XSLP_ETOL_R.

Default value -1.0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_ERRORTOL_A, XSLP_EVTOL_A

XSLP_EXPAND

Description Multiplier to increase a step bound

Type Double

Note If step bounding is enabled, the step bound for a variable will be increased if successive changes are in the same direction. More precisely, if there are XSLP_SAMECOUNT successive changes reaching the step bound and in the same direction for a variable, then the step bound (B) for the variable will be reset to $B * XSLP\_EXPAND$.

Default value 2

Affects routines XSLPmaxim, XSLPminim

See also XSLP_SHRINK, XSLP_SHRINKBIAS, XSLP_SAMECOUNT

XSLP_FEASTOLTARGET

Description When set, this defines a target feasibility tolerance to which the linearizations are solved to

Type Double

Note This is a soft version of XPRS_FEASTOL, and will dynamically revert back to XPRS_FEASTOL if the desired accuracy could not be achieved.

Default value 0 (ignored, not set)

Affects routines XSLPmaxim, XSLPminim

See also XSLP_OPTIMALITYTOLTARGET,

XSLP_GRANULARITY

Description Base for calculating penalty costs

Type Double

Note If XSLP_GRANULARITY >1, then initial penalty costs will be powers of XSLP_GRANULARITY.

Default value 4

Affects routines XSLPconstruct

See also XSLP_MAXWEIGHT, XSLP_MINWEIGHT

XSLP_INFINITY

Description Value returned by a divide-by-zero in a formula

Type Double

Default value 1.0e+10

XSLP_ITOL_A

Description Absolute impact convergence tolerance

Type Double

Note The absolute impact convergence criterion assesses the change in the effect of a coefficient in a constraint. The *effect* of a coefficient is its value multiplied by the activity of the column in which it appears.

$$E = X * C$$

where X is the activity of the matrix column in which the coefficient appears, and C is the value of the coefficient. The linearization approximates the effect of the coefficient as

$$E_1 = X * C_0 \; + \; \delta X * C'_0$$

where X is as before, C_0 is the value of the coefficient C calculated using the assumed values for the variables and C'_0 is the value of $\frac{\partial C}{\partial X}$ calculated using the assumed values for the variables.
If C_1 is the value of the coefficient C calculated using the actual values for the variables, then the error in the effect of the coefficient is given by

$$\delta E = X * C_1 - (X * C_0 \; + \; \delta X * C'_0)$$

If $\delta E <$ *XSLP_ITOL_A*
then the variable has passed the absolute impact convergence criterion for this coefficient. If a variable which has not converged on strict (closure or delta) criteria passes the (relative or absolute) impact or matrix criteria for all the coefficients in which it appears, then it is deemed to have converged. When the value is set to be negative, the value is adjusted automatically by SLP, based on the feasibility target `XSLP_VALIDATIONTARGET_R`. Good values for the control are usually fall between 1e-3 and 1e-6.

Default value -1.0

Affects routines `XSLPmaxim`, `XSLPminim`

See also `XSLP_ITOL_R`, `XSLP_MTOL_A`, `XSLP_MTOL_R`, `XSLP_STOL_A`, `XSLP_STOL_R`

XSLP_ITOL_R

Description Relative impact convergence tolerance

Type Double

Note The relative impact convergence criterion assesses the change in the effect of a coefficient in a constraint in relation to the magnitude of the constituents of the constraint. The *effect* of a coefficient is its value multiplied by the activity of the column in which it appears.

$$E = X * C$$

where X is the activity of the matrix column in which the coefficient appears, and C is the value of the coefficient. The linearization approximates the effect of the coefficient as

$$E_1 = X * C_0 \ + \ \delta X * C'_0$$

where X is as before, C_0 is the value of the coefficient C calculated using the assumed values for the variables and C'_0 is the value of $\frac{\partial C}{\partial X}$ calculated using the assumed values for the variables.
If C_1 is the value of the coefficient C calculated using the actual values for the variables, then the error in the effect of the coefficient is given by

$$\delta E = X * C_1 - (X * C_0 \ + \ \delta X * C'_0)$$

All the elements of the constraint are examined, excluding delta and error vectors: for each, the contribution to the constraint is evaluated as the element multiplied by the activity of the vector in which it appears; it is then included in a *total positive contribution* or *total negative contribution* depending on the sign of the contribution. If the predicted effect of the coefficient is positive, it is tested against the total positive contribution; if the effect of the coefficient is negative, it is tested against the total negative contribution. If T_0 is the total positive or total negative contribution to the constraint (as appropriate)
and $\delta E < T_0 * XSLP\_ITOL\_R$
then the variable has passed the relative impact convergence criterion for this coefficient.
If a variable which has not converged on strict (closure or delta) criteria passes the (relative or absolute) impact or matrix criteria for all the coefficients in which it appears, then it is deemed to have converged. When the value is set to be negative, the value is adjusted automatically by SLP, based on the feasibility target `XSLP_VALIDATIONTARGET_R`. Good values for the control are usually fall between 1e-3 and 1e-6.

Default value -1.0

Affects routines `XSLPmaxim`, `XSLPminim`

See also `XSLP_ITOL_A`, `XSLP_MTOL_A`, `XSLP_MTOL_R`, `XSLP_STOL_A`, `XSLP_STOL_R`

XSLP_MATRIXTOL

Description Provides an override value for XPRS_MATRIXTOL, which controls the smallest magnitude of matrix coefficents

Type Double

Note Any value smaller than XSLP_MATRIXTOL in magnitude will not be loaded into the linearization. This only applies to the matrix coefficients; bounds, right hand sides and objectives are not affected.

Default value 1e-30

Affects routines XSLPconstruct, XSLPmaxim, XSLPminim

XSLP_MAXWEIGHT

Description Maximum penalty weight for delta or error vectors

Type Double

Note When penalty vectors are created, or when their weight is increased by escalation, the maximum weight that will be used is given by XSLP_MAXWEIGHT.

Default value 100

Affects routines XSLPconstruct, XSLPmaxim, XSLPminim

See also XSLP_ALGORITHM, XSLP_AUGMENTATION, XSLP_ESCALATION, XSLP_MINWEIGHT

XSLP_MEMORYFACTOR

Description Factor for expanding size of dynamic arrays in memory

Type Double

Note When a dynamic array has to be increased in size, the new space allocated will be XSLP_MEMORYFACTOR times as big as the previous size. A larger value may result in improved performance because arrays need to be re-sized and moved less frequently; however, more memory may be required under such circumstances because not all of the previous memory area can be re-used efficiently.

Default value 1.6

See also Memory control variables XSLP_MEM* Memory control variables XSLP_MEM*

XSLP_MERITLAMBDA

Description Factor by which the net objective is taken into account in the merit function

Type Double

Note The merit function is evaluated in the original, non-augmented / linearized space of the problem. A solution is deemed improved, if either feasibility improved, or if feasibility is not deteriorated but the net objective is improved, or if the combination of the two is improved, where the value of the XSLP_MERITLAMBDA control is used to combine the two measures. A nonpositive value indicates that the combined effect should not be checked.

Default value 0.0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_FILTER XSLP_LSITERLIMIT XSLP_LSPATTERNLIMIT

XSLP_MINSBFACTOR

Description Factor by which step bounds can be decreased beneath XSLP_ATOL_A

Type Double

Note Normally, step bounds are not decreased beneath XSLP_ATOL_A, as such variables are treated as converged. However, it may be beneficial to decrease step bounds further, as individual variable value changes might affect the convergence of other variables in the model, even if the variablke itself is deemed converged.

Default value 1.0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_ATOL_A

XSLP_MINWEIGHT

Description Minimum penalty weight for delta or error vectors

Type Double

Note When penalty vectors are created, the minimum weight that will be used is given by XSLP_MINWEIGHT.

Default value 0.01

Affects routines XSLPconstruct, XSLPmaxim, XSLPminim

See also XSLP_AUGMENTATION, XSLP_MAXWEIGHT

XSLP_MIPCUTOFF_A

Description Absolute objective function cutoff for MIP termination

Type Double

Note If the objective function is worse by a defined amount than the best integer solution obtained so far, then the SLP will be terminated (and the node will be cut off). The node will be cut off at the current SLP iteration if the objective function for the last XSLP_MIPCUTOFFCOUNT SLP iterations are all worse than the best obtained so far, and the difference is greater than *XSLP_MIPCUTOFF_A* and *OBJ* $*$ *XSLP_MIPCUTOFF_R* where *OBJ* is the best integer solution obtained so far.
The MIP cutoff tests are only applied after XSLP_MIPCUTOFFLIMIT SLP iterations at the current node.

Default value 0.0001

Affects routines XSLPglobal

See also XSLP_MIPCUTOFFCOUNT, XSLP_MIPCUTOFFLIMIT, XSLP_MIPCUTOFF_R

XSLP_MIPCUTOFF_R

Description Absolute objective function cutoff for MIP termination

Type Double

Note If the objective function is worse by a defined amount than the best integer solution obtained so far, then the SLP will be terminated (and the node will be cut off). The node will be cut off at the current SLP iteration if the objective function for the last XSLP_MIPCUTOFFCOUNT SLP iterations are all worse than the best obtained so far, and the difference is greater than *XSLP_MIPCUTOFF_A* and *OBJ* $*$ *XSLP_MIPCUTOFF_R* where *OBJ* is the best integer solution obtained so far.
The MIP cutoff tests are only applied after XSLP_MIPCUTOFFLIMIT SLP iterations at the current node.

Default value 0.0001

Affects routines XSLPglobal

See also XSLP_MIPCUTOFFCOUNT, XSLP_MIPCUTOFFLIMIT, XSLP_MIPCUTOFF_A

XSLP_MIPERRORTOL_A

Description Absolute penalty error cost tolerance for MIP cut-off

Type Double

Note The penalty error cost test is applied at each node where there are active penalties in the solution. If XSLP_MIPERRORTOL_A is nonzero and the absolute value of the penalty costs is greater than XSLP_MIPERRORTOL_A, the node will be declared infeasible. If XSLP_MIPERRORTOL_A is zero then no test is made and the node will not be declared infeasible on this criterion.

Default value 0 (inactive)

Affects routines XSLPglobal

See also XSLP_MIPERRORTOL_R

XSLP_MIPERRORTOL_R

Description Relative penalty error cost tolerance for MIP cut-off

Type Double

Note The penalty error cost test is applied at each node where there are active penalties in the solution. If XSLP_MIPERRORTOL_R is nonzero and the absolute value of the penalty costs is greater than *XSLP_MIPERRORTOL_R* $*$ *abs*(*Obj*) where *Obj* is the value of the objective function, then the node will be declared infeasible. If XSLP_MIPERRORTOL_R is zero then no test is made and the node will not be declared infeasible on this criterion.

Default value 0 (inactive)

Affects routines XSLPglobal

See also XSLP_MIPERRORTOL_A

XSLP_MIPOTOL_A

Description Absolute objective function tolerance for MIP termination

Type Double

Note The objective function test for MIP termination is applied only when step bounding has been applied (or XSLP_SBSTART SLP iterations have taken place if step bounding is not being used). The node will be terminated at the current SLP iteration if the range of the objective function values over the last XSLP_MIPOCOUNT SLP iterations is within *XSLP_MIPOTOL_A* or within *OBJ* $*$ *XSLP_MIPOTOL_R* where *OBJ* is the average value of the objective function over those iterations.

Default value 0.00001

Affects routines XSLPglobal

See also XSLP_MIPOCOUNT XSLP_MIPOTOL_R XSLP_SBSTART

XSLP_MIPOTOL_R

Description Relative objective function tolerance for MIP termination

Type Double

Note The objective function test for MIP termination is applied only when step bounding has been applied (or XSLP_SBSTART SLP iterations have taken place if step bounding is not being used). The node will be terminated at the current SLP iteration if the range of the objective function values over the last XSLP_MIPOCOUNT SLP iterations is within *XSLP_MIPOTOL_A* or within *OBJ* $*$ *XSLP_MIPOTOL_R* where *OBJ* is the average value of the objective function over those iterations.

Default value 0.00001

Affects routines XSLPglobal

See also XSLP_MIPOCOUNT XSLP_MIPOTOL_A XSLP_SBSTART

XSLP_MSMAXBOUNDRANGE

Description Defines the maximum range inside which initial points are generated by multistart presets

Type Double

Note The is the maximum range in which initial points are generated; the actual range is expected to be smaller as bounds are domains are also considered.

Default value 1000

Affects routines XSLPminim, XSLPmaxim

See also XSLP_MULTISTART

XSLP_MTOL_A

Description Absolute effective matrix element convergence tolerance

Type Double

Note The absolute effective matrix element convergence criterion assesses the change in the effect of a coefficient in a constraint. The *effect* of a coefficient is its value multiplied by the activity of the column in which it appears.

$$E = X * C$$

where X is the activity of the matrix column in which the coefficient appears, and C is the value of the coefficient. The linearization approximates the effect of the coefficient as

$$E = X * C_0 \; + \; \delta X * C_0'$$

where V is as before, C_0 is the value of the coefficient C calculated using the assumed values for the variables and C_0' is the value of $\frac{\partial C}{\partial X}$ calculated using the assumed values for the variables.
If C_1 is the value of the coefficient C calculated using the actual values for the variables, then the error in the effect of the coefficient is given by

$$\delta E = X * C_1 - (X * C_0 \; + \; \delta X * C_0')$$

If $\delta E < X * XSLP\_MTOL\_A$
then the variable has passed the absolute effective matrix element convergence criterion for this coefficient.
If a variable which has not converged on strict (closure or delta) criteria passes the (relative or absolute) impact or matrix criteria for all the coefficients in which it appears, then it is deemed to have converged. When the value is set to be negative, the value is adjusted automatically by SLP, based on the feasibility target XSLP_VALIDATIONTARGET_R. Good values for the control are usually fall between 1e-3 and 1e-6.

Default value -1.0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_ITOL_A, XSLP_ITOL_R, XSLP_MTOL_R, XSLP_STOL_A, XSLP_STOL_R

XSLP_MTOL_R

Description Relative effective matrix element convergence tolerance

Type Double

Note The relative effective matrix element convergence criterion assesses the change in the effect of a coefficient in a constraint relative to the magnitude of the coefficient. The *effect* of a coefficient is its value multiplied by the activity of the column in which it appears.

$$E = X * C$$

where X is the activity of the matrix column in which the coefficient appears, and C is the value of the coefficient. The linearization approximates the effect of the coefficient as

$$E_1 = X * C_0 \ + \ \delta X * C'_0$$

where V is as before, C_0 is the value of the coefficient C calculated using the assumed values for the variables and C'_0 is the value of $\frac{\partial C}{\partial X}$ calculated using the assumed values for the variables.
If C_1 is the value of the coefficient C calculated using the actual values for the variables, then the error in the effect of the coefficient is given by

$$\delta E = X * C_1 - (X * C_0 \ + \ \delta X * C'_0)$$

If $\delta E < E_1 * XSLP\_MTOL\_R$
then the variable has passed the relative effective matrix element convergence criterion for this coefficient.
If a variable which has not converged on strict (closure or delta) criteria passes the (relative or absolute) impact or matrix criteria for all the coefficients in which it appears, then it is deemed to have converged. When the value is set to be negative, the value is adjusted automatically by SLP, based on the feasibility target XSLP_VALIDATIONTARGET_R. Good values for the control are usually fall between 1e-3 and 1e-6.

Default value -1.0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_ITOL_A, XSLP_ITOL_R, XSLP_MTOL_A, XSLP_STOL_A, XSLP_STOL_R

XSLP_MVTOL

Description Marginal value tolerance for determining if a constraint is slack

Type Double

Note If the absolute value of the marginal value of a constraint is less than XSLP_MVTOL, then (1) the constraint is regarded as not constraining for the purposes of the slack tolerance convergence criteria;
(2) the constraint is not regarded as an *active constraint* when identifying unconverged variables in active constraints. When the value is set to be negative, the value is adjusted automatically by SLP, based on the feasibility target XSLP_VALIDATIONTARGET_R. Good values for the control are usually fall between 1e-3 and 1e-6.

Default value -1.0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_STOL_A, XSLP_STOL_R

XSLP_OBJSENSE

Description Objective function sense

Type Double

Note XSLP_OBJSENSE is set to +1 for minimization and to -1 for maximization. It is automatically set by XSLPmaxim and XSLPminim; it must be set by the user before calling XSLPnlpoptimize.

Set by routines XSLPmaxim, XSLPminim

Default value +1

Affects routines XSLPmaxim, XSLPminim, XSLPnlpoptimize

XSLP_OBJTOPENALTYCOST

Description Factor to estimate initial penalty costs from objective function

Type Double

Notes The setting of initial penalty error costs can affect the path of the optimization and, indeed, whether a solution is achieved at all. If the penalty costs are too low, then unbounded solutions may result although Xpress-SLP will increase the costs in an attempt to recover. If the penalty costs are too high, then the requirement to achieve feasibility of the linearized constraints may be too strong to allow the system to explore the nonlinear feasible region. Low penalty costs can result in many SLP iterations, as feasibility of the nonlinear constraints is not achieved until the penalty costs become high enough; high penalty costs force feasibility of the linearizations, and so tend to find local optima close to an initial feasible point. Xpress-SLP can analyze the problem to estimate the size of penalty costs required to avoid an initial unbounded solution. XSLP_OBJTOPENALTYCOST can be used in conjunction with this procedure to scale the costs and give an appropriate initial value for balancing the requirements of feasibility and optimality.
Not all models are amenable to the Xpress-SLP analysis. As the analysis is initially concerned with establishing a cost level to avoid unboundedness, a model which is sufficiently constrained will never show unboundedness regardless of the cost. Also, as the analysis is done at the start of the optimization to establish a penalty cost, significant changes in the coefficients, or a high degree of nonlinearity, may invalidate the initial analysis.
A setting for XSLP_OBJTOPENALTYCOST of zero disables the analysis. A setting of 3 or 4 has proved successful for many models. If XSLP_OBJTOPENALTYCOST cannot be used because of the problem structure, its effect can still be emulated by some initial experiments to establish the cost required to avoid unboundedness, and then manually applying a suitable factor. If the problem is initially unbounded, then the penalty cost will be increased until either it reaches its maximum or the problem becomes bounded.

Default value 0

Affects routines XSLPmaxim, XSLPminim

XSLP_OPTIMALITYTOLTARGET

Description When set, this defines a target optimality tolerance to which the linearizations are solved to

Type Double

Note This is a soft version of XPRS_OPTIMALITYTOL, and will dynamically revert back to XPRS_OPTIMALITYTOL if the desired accuracy could not be achieved.

Default value 0 (ignored, not set)

Affects routines XSLPmaxim, XSLPminim

See also XSLP_FEASTOLTARGET,

XSLP_OTOL_A

Description Absolute static objective (2) convergence tolerance

Type Double

Note The static objective (2) convergence criterion does not measure convergence of individual variables. Instead, it measures the significance of the changes in the objective function over recent SLP iterations. It is applied when all the variables interacting with active constraints (those that have a marginal value of at least XSLP_MVTOL) have converged. The rationale is that if the remaining unconverged variables are not involved in active constraints and if the objective function is not changing significantly between iterations, then the solution is more-or-less practical.
The variation in the objective function is defined as

$$\delta Obj = MAX_{Iter}(Obj) - MIN_{Iter}(Obj)$$

where *Iter* is the XSLP_OCOUNT most recent SLP iterations and *Obj* is the corresponding objective function value.
If $ABS(\delta Obj) \leq XSLP\_OTOL\_A$
then the problem has converged on the absolute static objective (2) convergence criterion. The static objective function (2) test is applied only if XSLP_OCOUNT is at least 2. When the value is set to be negative, the value is adjusted automatically by SLP, based on the optimality target XSLP_VALIDATIONTARGET_K. Good values for the control are usually fall between 1e-3 and 1e-6.

Default value -1.0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_OCOUNT, XSLP_OTOL_R

XSLP_OTOL_R

Description Relative static objective (2) convergence tolerance

Type Double

Note The static objective (2) convergence criterion does not measure convergence of individual variables. Instead, it measures the significance of the changes in the objective function over recent SLP iterations. It is applied when all the variables interacting with active constraints (those that have a marginal value of at least XSLP_MVTOL) have converged. The rationale is that if the remaining unconverged variables are not involved in active constraints and if the objective function is not changing significantly between iterations, then the solution is more-or-less practical.
The variation in the objective function is defined as

$$\delta Obj = MAX_{Iter}(Obj) - MIN_{Iter}(Obj)$$

where *Iter* is the XSLP_OCOUNT most recent SLP iterations and *Obj* is the corresponding objective function value.
If $ABS(\delta Obj) \leq AVG_{Iter}(Obj) * XSLP\_OTOL\_R$
then the problem has converged on the relative static objective (2) convergence criterion.
The static objective function (2) test is applied only if XSLP_OCOUNT is at least 2. When the value is set to be negative, the value is adjusted automatically by SLP, based on the optimality target XSLP_VALIDATIONTARGET_K. Good values for the control are usually fall between 1e-3 and 1e-6.

Default value -1.0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_OCOUNT, XSLP_OTOL_A

XSLP_PRESOLVEZERO

Description Minimum absolute value for a variable which is identified as nonzero during SLP presolve

Type Double

Note During the SLP (nonlinear)presolve, a variable may be identified as being nonzero (for example, because it is used as a divisor). A bound of plus or minus XSLP_PRESOLVEZERO will be applied to the variable if it is identified as non-negative or non-positive.

Default value 1.0E-09

Affects routines XSLPpresolve

XSLP_PRIMALINTEGRALREF

Description Reference solution value to take into account when calculating the primal integral

Type Double

Note When a global optimum is known, this can used to calculate a globally valid primal integral. It can also be used to indicate the target objective value still to be taken into account in the integral.

Default value XPRS_PLUSINFINITY

Affects routines XSLPminim, XSLPmaxim

XSLP_SHRINK

Description Multiplier to reduce a step bound

Type Double

Note If step bounding is enabled, the step bound for a variable will be decreased if successive changes are in opposite directions. The step bound (*B*) for the variable will be reset to $B * XSLP\_SHRINK$.
If the step bound is already below the strict (delta or closure) tolerances, it will not be reduced further.

Default value 0.5

Affects routines XSLPmaxim, XSLPminim

See also XSLP_EXPAND, XSLP_SHRINKBIAS, XSLP_SAMECOUNT

XSLP_SHRINKBIAS

Description Defines an overwrite / adjustment of step bounds for improving iterations

Type Double

Note Positive values overwrite XSLP_SHRINK only if the objective is improving. A negative value is used to scale all step bounds in improving iterations.

Default value 0 (ignored, not set)

Affects routines XSLPminim, XSLPmaxim

See also XSLP_SHRINK, XSLP_EXPAND, XSLP_SAMECOUNT

XSLP_STOL_A

Description Absolute slack convergence tolerance

Type Double

Note The slack convergence criterion is identical to the impact convergence criterion, except that the tolerances used are XSLP_STOL_A (instead of XSLP_ITOL_A) and XSLP_STOL_R (instead of XSLP_ITOL_R). See XSLP_ITOL_A for a description of the test. When the value is set to be negative, the value is adjusted automatically by SLP, based on the feasibility target XSLP_VALIDATIONTARGET_R. Good values for the control are usually fall between 1e-3 and 1e-6.

Default value -1.0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_ITOL_A, XSLP_ITOL_R, XSLP_MTOL_A, XSLP_MTOL_R, XSLP_STOL_R

XSLP_STOL_R

Description Relative slack convergence tolerance

Type Double

Note The slack convergence criterion is identical to the impact convergence criterion, except that the tolerances used are XSLP_STOL_A (instead of XSLP_ITOL_A) and XSLP_STOL_R (instead of XSLP_ITOL_R). See XSLP_ITOL_R for a description of the test. When the value is set to be negative, the value is adjusted automatically by SLP, based on the feasibility target XSLP_VALIDATIONTARGET_R. Good values for the control are usually fall between 1e-3 and 1e-6.

Default value -1.0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_ITOL_A, XSLP_ITOL_R, XSLP_MTOL_A, XSLP_MTOL_R, XSLP_STOL_A

XSLP_VALIDATIONTARGET_R

Description Feasiblity target tolerance

Type Double

Note Primary feasiblity control for SLP. When the relevant feasibility based convergence controls are left at their default values, SLP will adjust their value to match the target. The control defines a target value, that may not necessarily be attainable.

Default value 1e-6

Affects routines XSLPmaxim, XSLPminim

See also XSLP_VALIDATIONTARGET_K

XSLP_VALIDATIONTARGET_K

Description Optimality target tolerance

Type Double

Note Primary optimality control for SLP. When the relevant optimality based convergence controls are left at their default values, SLP will adjust their value to match the target. The control defines a target value, that may not necessarily be attainable for problem with no strong constraint qualifications.

Default value 1e-6

Affects routines XSLPmaxim, XSLPminim

See also XSLP_VALIDATIONTARGET_R

XSLP_VALIDATIONTOL_A

Description Absolute tolerance for the XSLPvalidate procedure

Type Double

Note `XSLPvalidate` checks the feasibility of a converged solution against relative and absolute tolerances for each constraint. The left hand side and the right hand side of the constraint are calculated using the converged solution values. If the calculated values imply that the constraint is infeasible, then the difference (*D*) is tested against the absolute and relative validation tolerances.
If $D < XSLP\_VALIDATIONTOL\_A$
then the constraint is within the absolute validation tolerance. The total positive (*TPos*) and negative contributions (*TNeg*) to the left hand side are also calculated.
If $D < MAX(ABS(TPos), ABS(TNeg)) * XSLP\_VALIDATIONTOL\_A$
then the constraint is within the relative validation tolerance. For each constraint which is outside both the absolute and relative validation tolerances, validation factors are calculated which are the factors by which the infeasibility exceeds the corresponding validation tolerance; the smaller factor is printed in the validation report.
The validation index `XSLP_VALIDATIONINDEX_A` is the largest of these factors which is an absolute validation factor multiplied by the absolute validation tolerance; the validation index `XSLP_VALIDATIONINDEX_R` is the largest of these factors which is a relative validation factor multiplied by the relative validation tolerance.

Default value 0.00001

Affects routines `XSLPvalidate`

See also `XSLP_VALIDATIONINDEX_A`, `XSLP_VALIDATIONINDEX_R`, `XSLP_VALIDATIONTOL_R`

XSLP_VALIDATIONTOL_R

Description Relative tolerance for the XSLPvalidate procedure

Type Double

Note `XSLPvalidate` checks the feasibility of a converged solution against relative and absolute tolerances for each constraint. The left hand side and the right hand side of the constraint are calculated using the converged solution values. If the calculated values imply that the constraint is infeasible, then the difference (*D*) is tested against the absolute and relative validation tolerances.
If $D < XSLP\_VALIDATIONTOL\_A$
then the constraint is within the absolute validation tolerance. The total positive (*TPos*) and negative contributions (*TNeg*) to the left hand side are also calculated.
If $D < MAX(ABS(TPos), ABS(TNeg)) * XSLP\_VALIDATIONTOL\_R$
then the constraint is within the relative validation tolerance. For each constraint which is outside both the absolute and relative validation tolerances, validation factors are calculated which are the factors by which the infeasibility exceeds the corresponding validation tolerance; the smaller factor is printed in the validation report.
The validation index `XSLP_VALIDATIONINDEX_A` is the largest of these factors which is an absolute validation factor multiplied by the absolute validation tolerance; the validation index

XSLP_VALIDATIONINDEX_R is the largest of these factors which is a relative validation factor multiplied by the relative validation tolerance.

Default value 0.00001

Affects routines XSLPvalidate

See also XSLP_VALIDATIONINDEX_A, XSLP_VALIDATIONINDEX_R, XSLP_VALIDATIONTOL_A

XSLP_VTOL_A

Description Absolute static objective (3) convergence tolerance

Type Double

Note The static objective (3) convergence criterion does not measure convergence of individual variables, and in fact does not in any way imply that the solution has converged. However, it is sometimes useful to be able to terminate an optimization once the objective function appears to have stabilized. One example is where a set of possible schedules are being evaluated and initially only a good estimate of the likely objective function value is required, to eliminate the worst candidates.
The variation in the objective function is defined as

$$\delta Obj = MAX_{Iter}(Obj) - MIN_{Iter}(Obj)$$

where *Iter* is the XSLP_VCOUNT most recent SLP iterations and *Obj* is the corresponding objective function value.
If $ABS(\delta Obj) \leq XSLP\_VTOL\_A$
then the problem has converged on the absolute static objective function (3) criterion.
The static objective function (3) test is applied only if after at least XSLP_VLIMIT + XSLP_SBSTART SLP iterations have taken place and only if XSLP_VCOUNT is at least 2. Where step bounding is being used, this ensures that the test is not applied until after step bounding has been introduced. When the value is set to be negative, the value is adjusted automatically by SLP, based on the optimality target XSLP_VALIDATIONTARGET_K. Good values for the control are usually fall between 1e-3 and 1e-6.

Default value -1.0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_SBSTART, XSLP_VCOUNT, XSLP_VLIMIT, XSLP_VTOL_R

XSLP_VTOL_R

Description Relative static objective (3) convergence tolerance

Type Double

Note The static objective (3) convergence criterion does not measure convergence of individual variables, and in fact does not in any way imply that the solution has converged. However, it is sometimes useful to be able to terminate an optimization once the objective function appears to have stabilized. One example is where a set of possible schedules are being evaluated and

initially only a good estimate of the likely objective function value is required, to eliminate the worst candidates.
The variation in the objective function is defined as

$$\delta Obj = MAX_{Iter}(Obj) - MIN_{Iter}(Obj)$$

where *Iter* is the XSLP_VCOUNT most recent SLP iterations and *Obj* is the corresponding objective function value.
If $ABS(\delta Obj) \leq AVG_{Iter}(Obj) * XSLP\_VTOL\_R$
then the problem has converged on the absolute static objective function (3) criterion.
The static objective function (3) test is applied only if after at least XSLP_VLIMIT + XSLP_SBSTART SLP iterations have taken place and only if XSLP_VCOUNT is at least 2. Where step bounding is being used, this ensures that the test is not applied until after step bounding has been introduced. When the value is set to be negative, the value is adjusted automatically by SLP, based on the optimality target XSLP_VALIDATIONTARGET_K. Good values for the control are usually fall between 1e-3 and 1e-6.

Default value -1.0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_SBSTART, XSLP_VCOUNT, XSLP_VLIMIT, XSLP_VTOL_A

XSLP_WTOL_A

Description Absolute extended convergence continuation tolerance

Type Double

Note It may happen that all the variables have converged, but some have converged on extended criteria and at least one of these variables is at its step bound. This means that, at least in the linearization, if the variable were to be allowed to move further the objective function would improve. This does not necessarily imply that the same is true of the original problem, but it is still possible that an improved result could be obtained by taking another SLP iteration.

The extended convergence continuation criterion is applied after a converged solution has been found where at least one variable has converged on extended criteria and is at its step bound limit. The extended convergence continuation test measures whether any improvement is being achieved when additional SLP iterations are carried out. If not, then the last converged solution will be restored and the optimization will stop.
For a maximization problem, the improvement in the objective function at the current iteration compared to the objective function at the last converged solution is given by:
$\delta Obj = Obj - LastConvergedObj$
For a minimization problem, the sign is reversed.
If $\delta Obj > XSLP\_WTOL\_A$ and
$\delta Obj > ABS(ConvergedObj) * XSLP\_WTOL\_R$ then the solution is deemed to have a significantly better objective function value than the converged solution.

When a solution is found which converges on extended criteria and with active step bounds, the solution is saved and SLP optimization continues until one of the following:
(1) a new solution is found which converges on some other criterion, in which case the SLP optimization stops with this new solution;
(2) a new solution is found which converges on extended criteria and with active step bounds, and which has a significantly better objective function, in which case this is taken as the new saved solution;

(3) none of the XSLP_WCOUNT most recent SLP iterations has a significantly better objective function than the saved solution, in which case the saved solution is restored and the SLP optimization stops.

When the value is set to be negative, the value is adjusted automatically by SLP, based on the optimality target XSLP_VALIDATIONTARGET_K. Good values for the control are usually fall between 1e-3 and 1e-6.

Default value -1.0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_WCOUNT, XSLP_WTOL_R

XSLP_WTOL_R

Description Relative extended convergence continuation tolerance

Type Double

Note It may happen that all the variables have converged, but some have converged on extended criteria and at least one of these variables is at its step bound. This means that, at least in the linearization, if the variable were to be allowed to move further the objective function would improve. This does not necessarily imply that the same is true of the original problem, but it is still possible that an improved result could be obtained by taking another SLP iteration.

The extended convergence continuation criterion is applied after a converged solution has been found where at least one variable has converged on extended criteria and is at its step bound limit. The extended convergence continuation test measures whether any improvement is being achieved when additional SLP iterations are carried out. If not, then the last converged solution will be restored and the optimization will stop.
For a maximization problem, the improvement in the objective function at the current iteration compared to the objective function at the last converged solution is given by:
$\delta Obj = Obj - LastConvergedObj$
For a minimization problem, the sign is reversed.
If $\delta Obj > XSLP\_WTOL\_A$ and
$\delta Obj > ABS(ConvergedObj) * XSLP\_WTOL\_R$ then the solution is deemed to have a significantly better objective function value than the converged solution.

If XSLP_WCOUNT is greater than zero, and a solution is found which converges on extended criteria and with active step bounds, the solution is saved and SLP optimization continues until one of the following:
(1) a new solution is found which converges on some other criterion, in which case the SLP optimization stops with this new solution;
(2) a new solution is found which converges on extended criteria and with active step bounds, and which has a significantly better objective function, in which case this is taken as the new saved solution;
(3) none of the XSLP_WCOUNT most recent SLP iterations has a significantly better objective function than the saved solution, in which case the saved solution is restored and the SLP optimization stops.

When the value is set to be negative, the value is adjusted automatically by SLP, based on the optimality target XSLP_VALIDATIONTARGET_K. Good values for the control are usually fall between 1e-4 and 1e-6.

Default value -1.0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_WCOUNT, XSLP_WTOL_A

XSLP_XTOL_A

Description Absolute static objective function (1) tolerance

Type Double

Note It may happen that all the variables have converged, but some have converged on extended criteria and at least one of these variables is at its step bound. This means that, at least in the linearization, if the variable were to be allowed to move further the objective function would improve. This does not necessarily imply that the same is true of the original problem, but it is still possible that an improved result could be obtained by taking another SLP iteration. However, if the objective function has already been stable for several SLP iterations, then there is less likelihood of an improved result, and the converged solution can be accepted.

The static objective function (1) test measures the significance of the changes in the objective function over recent SLP iterations. It is applied when all the variables have converged, but some have converged on extended criteria and at least one of these variables is at its step bound. Because all the variables have converged, the solution is already converged but the fact that some variables are at their step bound limit suggests that the objective function could be improved by going further.

The variation in the objective function is defined as
$\delta Obj = MAX_{Iter}(Obj) - MIN_{Iter}(Obj)$
where *Iter* is the XSLP_XCOUNT most recent SLP iterations and *Obj* is the corresponding objective function value.

If $ABS(\delta Obj) \leq XSLP\_XTOL\_A$
then the objective function is deemed to be static according to the absolute static objective function (1) criterion.
If $ABS(\delta Obj) \leq AVG_{Iter}(Obj) * XSLP\_XTOL\_R$
then the objective function is deemed to be static according to the relative static objective function (1) criterion.

The static objective function (1) test is applied only until XSLP_XLIMIT SLP iterations have taken place. After that, if all the variables have converged on strict or extended criteria, the solution is deemed to have converged.

If the objective function passes the relative or absolute static objective function (1) test then the solution is deemed to have converged.

When the value is set to be negative, the value is adjusted automatically by SLP, based on the optimality target XSLP_VALIDATIONTARGET_K. Good values for the control are usually fall between 1e-3 and 1e-6.

Default value -1.0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_XCOUNT, XSLP_XLIMIT, XSLP_XTOL_R

XSLP_XTOL_R

Description Relative static objective function (1) tolerance

Type Double

Note It may happen that all the variables have converged, but some have converged on extended criteria and at least one of these variables is at its step bound. This means that, at least in the linearization, if the variable were to be allowed to move further the objective function would improve. This does not necessarily imply that the same is true of the original problem, but it is still possible that an improved result could be obtained by taking another SLP iteration. However, if the objective function has already been stable for several SLP iterations, then there is less likelihood of an improved result, and the converged solution can be accepted.

The static objective function (1) test measures the significance of the changes in the objective function over recent SLP iterations. It is applied when all the variables have converged, but some have converged on extended criteria and at least one of these variables is at its step bound. Because all the variables have converged, the solution is already converged but the fact that some variables are at their step bound limit suggests that the objective function could be improved by going further.

The variation in the objective function is defined as
$\delta Obj = MAX_{Iter}(Obj) - MIN_{Iter}(Obj)$
where *Iter* is the XSLP_XCOUNT most recent SLP iterations and *Obj* is the corresponding objective function value.

If $ABS(\delta Obj) \leq XSLP\_XTOL\_A$
then the objective function is deemed to be static according to the absolute static objective function (1) criterion.
If $ABS(\delta Obj) \leq AVG_{Iter}(Obj) * XSLP\_XTOL\_R$
then the objective function is deemed to be static according to the relative static objective function (1) criterion.

The static objective function (1) test is applied only until XSLP_XLIMIT SLP iterations have taken place. After that, if all the variables have converged on strict or extended criteria, the solution is deemed to have converged.

If the objective function passes the relative or absolute static objective function (1) test then the solution is deemed to have converged.

When the value is set to be negative, the value is adjusted automatically by SLP, based on the optimality target XSLP_VALIDATIONTARGET_K. Good values for the control are usually fall between 1e-4 and 1e-6.

Default value -1.0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_XCOUNT, XSLP_XLIMIT, XSLP_XTOL_A

XSLP_ZERO

Description Absolute tolerance

Type Double

Note If a value is below XSLP_ZERO in magnitude, then it will be regarded as zero in certain formula calculations:
an attempt to divide by such a value will give a "divide by zero" error;
an exponent of a negative number will produce a "negative number, fractional exponent" error if the exponent differs from an integer by more than XSLP_ZERO.

Default value 1.0E-10

Affects routines `XSLPevaluatecoef`, `XSLPevaluateformula`

20.2 Integer control parameters

XSLP_ALGORITHM

Description Bit map describing the SLP algorithm(s) to be used

Type Integer

Values

Bit	Meaning
0	Do not apply step bounds.
1	Apply step bounds to SLP delta vectors only when required.
2	Estimate step bounds from early SLP iterations.
3	Use dynamic damping.
4	Do not update values which are converged within strict tolerance.
5	Retain previous value when cascading if determining row is zero.
6	Reset XSLP_DELTA_Z to zero when converged and continue SLP.
7	Quick convergence check.
8	Escalate penalties.
9	Use the primal simplex algorithm when all error vectors become inactive.
11	Continue optimizing after penalty cost reaches maximum.
12	Accept a solution which has converged even if there are still significant active penalty error vectors.
13	Skip the solution polishing step if the LP postsolve returns a slightly infeasible, but claimed optimal solution.
14	Step bounds are updated to accomodate cascaded values (otherwise cascaded values are pushed to respect step bounds).
15	Apply clamping when converged on extended criteria only with some variables having active step bounds.
16	Apply clamping when converged on extended criteria only.

Notes

`Bit 0:` Do not apply step bounds. The default algorithm uses step bounds to force convergence. Step bounds may not be appropriate if dynamic damping is used.

`Bit 1:` Apply step bounds to SLP delta vectors only when required. Step bounds can be applied to all vectors simultaneously, or applied only when oscillation of the delta vector (change in sign between successive SLP iterations) is detected.

`Bit 2:` Estimate step bounds from early SLP iterations. If initial step bounds are not being explicitly provided, this gives a good method of calculating reasonable values. Values will tend to be larger rather than smaller, to reduce the risk of infeasibility caused by excessive tightness of the step bounds.

`Bit 3:` Use dynamic damping. Dynamic damping is sometimes an alternative to step bounding as a means of encouraging convergence, but it does not have the same power to force convergence as do step bounds.

`Bit 4:` Do not update values which are converged within strict tolerance. Models which are numerically unstable may benefit from this setting, which does not update values which have effectively hardly changed. If a variable subsequently does move outside its strict convergence tolerance, it will be updated as usual.

`Bit 5:` Retain previous value when cascading if determining row is zero. If the determining row is zero (that is, all the coefficients interacting with it are either zero or in columns with a

zero activity), then it is impossible to calculate a new value for the vector being cascaded. The choice is to use the solution value as it is, or to revert to the assumed value

`Bit 6:` Reset `XSLP_DELTA_Z` to zero when converged and continue SLP. One of the mechanisms to avoid local optima is to retain small non-zero coefficients between delta vectors and constraints, even when the coefficient should strictly be zero. If this option is set, then a converged solution will be continued with zero coefficients as appropriate.

`Bit 7:` Quick convergence check. Normally, each variable is checked against all convergence criteria until either a criterion is found which it passes, or it is declared "not converged". Later (extended convergence) criteria are more expensive to test and, once an unconverged variable has been found, the overall convergence status of the solution has been established. The quick convergence check carries out checks on the strict criteria, but omits checks on the extended criteria when an unconverged variable has been found.

`Bit 8:` Escalate penalties. Constraint penalties are increased after each SLP iteration where penalty vectors are present in the solution. Escalation applies an additional scaling factor to the penalty costs for active errors. This helps to prevent successive solutions becoming "stuck" because of a particular constraint, because its cost will be raised so that other constraints may become more attractive to violate instead and thus open up a new region to explore.

`Bit 9:` Use the primal simplex algorithm when all error vectors become inactive. The primal simplex algorithm often performs better than dual during the final stages of SLP optimization when there are relatively few basis changes between successive solutions. As it is impossible to establish in advance when the final stages are being reached, the disappearance of error vectors from the solution is used as a proxy.

`Bit 11:` Continue optimizing after penalty cost reaches maximum. Normally if the penalty cost reaches its maximum (by default the value of `XPRS_PLUSINFINITY`), the optimization will terminate with an unconverged solution. If the maximum value is set to a smaller value, then it may make sense to continue, using other means to determine when to stop.

`Bit 12:` Accept a solution which has converged even if there are still significant active penalty error vectors. Normally, the optimization will continue if there are active penalty vectors in the solution. However, it may be that there is no feasible solution (and so active penalties will always be present). Setting bit 12 means that, if other convergence criteria are met, then the solution will be accepted as converged and the optimization will stop.

`Bit 13:` Due to the nature of the SLP linearizations, and in particular because of the large differences in the objective function (model objective against penalty costs) some dual reductions in the linear presolver might introduce numerically instable reductions that cause slight infeasibilities to appear in postsolve. It is typically more efficient to remove these infeasibilities with an extra call to the linear optimizer; compared to switching these reductions off, which usually has a significant cost in performance. This bit is provided for numerically very hard problems, when the polishing step proves to be too expensive (XSLP will report these if any in the final log summary).

`Bit 14:` Normally, cascading will respect the step bounds of the SLP variable being cascaded. However, allowing the cascaded value to fall outside the step bounds (i.e. expanding the step bounds) can lead to better linearizations, as cascading will set better values for the SLP variables regarding their determining rows; note, that this later strategy might interfere with convergence of the cascaded variables.

`Bit 15:` When clamping is applied, then in any iteration when the solution would normally be deemed converged on extended criteria only, an extra step bound shrinking step is applied to help imposing strict convergence. In this variant, clamping is only applied on variables that have converged on extended criteria only and have active step bounds.

`Bit 16:` When clamping is applied, then in any iteration when the solution would normally be deemed converged on extended criteria only, an extra step bound shrinking step is applied

to help imposing strict convergence. In this variant, clamping is applied on all variables that have converged on extended criteria only.

The following constants are provided for setting these bits:

Setting bit 0	XSLP_NOSTEPBOUNDS
Setting bit 1	XSLP_STEPBOUNDSASREQUIRED
Setting bit 2	XSLP_ESTIMATESTEPBOUNDS
Setting bit 3	XSLP_DYNAMICDAMPING
Setting bit 4	XSLP_HOLDVALUES
Setting bit 5	XSLP_RETAINPREVIOUSVALUE
Setting bit 6	XSLP_RESETDELTAZ
Setting bit 7	XSLP_QUICKCONVERGENCECHECK
Setting bit 8	XSLP_ESCALATEPENALTIES
Setting bit 9	XSLP_SWITCHTOPRIMAL
Setting bit 11	XSLP_MAXCOSTOPTION
Setting bit 12	XSLP_RESIDUALERRORS
Setting bit 13	XSLP_NOLPPOLISHING
Setting bit 14	XSLP_CASCADEDBOUNDS
Setting bit 15	XSLP_CLAMPEXTENDEDACTIVESB
Setting bit 16	XSLP_CLAMPEXTENDEDALL

Recommended setting: Bits 1, 2, 5, 7 and usually bits 8 and 9.

Default value 166 (sets bits 1, 2, 5, 7)

Affects routines XSLPmaxim, XSLPminim

See also XSLP_DELTA_Z, XSLP_ERRORMAXCOST, XSLP_ESCALATION, XSLP_CLAMPSHRINK

XSLP_ANALYZE

Description Bit map activating additional options supporting model / solution path analyzis

Type Integer

Values

Bit	Meaning
0	Add solutions of the linearizations to the solution pool.
1	Add cascaded solutions to the solution pool.
2	Add line search solutions to the solution pool.
3	Include an extended iteration summary.
4	Run infeasibility analysis on infeasible iterations.
5	Save the solutions collected in the pool to disk.
6	Write the linearizations to disk at every XSLP_AUTOSAVE iterations.
7	Write the initial basis of the linearizations to disk at every XSLP_AUTOSAVE iterations.
8	Create an XSLP save file at every XSLP_AUTOSAVE iterations.

Note The solution pool can be accessed using the memory attribute XSLP_SOLUTIONPOOL. Normally, the value of this control does not affect the solution process itself. However, bit 3 (extended summary) will cause SLP to do more function evaluations, and the presence of non-deterministic user functions might cause changes in the solution process. These options are off by default due to performance considerations. The following constants are provided for setting these bits:

Setting bit 0	XSLP_ANALYZE_RECORDLINEARIZATION
Setting bit 1	XSLP_ANALYZE_RECORDCASCADE
Setting bit 2	XSLP_ANALYZE_RECORDLINESEARCH
Setting bit 3	XSLP_ANALYZE_EXTENDEDFINALSUMMARY
Setting bit 4	XSLP_ANALYZE_INFEASIBLE_ITERATION
Setting bit 5	XSLP_ANALYZE_AUTOSAVEPOOL
Setting bit 6	XSLP_ANALYZE_SAVELINEARIZATIONS
Setting bit 7	XSLP_ANALYZE_SAVEITERBASIS
Setting bit 8	XSLP_ANALYZE_SAVEFILE

Default value 0

See also XSLP_AUTOSAVE

XSLP_AUGMENTATION

Description Bit map describing the SLP augmentation method(s) to be used

Type Integer

Values

Bit	Meaning
0	Minimum augmentation.
1	Even handed augmentation.
2	Penalty error vectors on all non-linear equality constraints.
3	Penalty error vectors on all non-linear inequality constraints.
4	Penalty vectors to exceed step bounds.
5	Use arithmetic means to estimate penalty weights.
6	Estimate step bounds from values of row coefficients.
7	Estimate step bounds from absolute values of row coefficients.
8	Row-based step bounds.
9	Penalty error vectors on all constraints.
10	Intial values do not imply an SLP variable.

Notes

Bit 0: Minimum augmentation. Standard augmentation includes delta vectors for all variables involved in nonlinear terms (in non-constant coefficients or as vectors containing non-constant coefficients). Minimum augmentation includes delta vectors only for variables in non-constant coefficients. This produces a smaller linearization, but there is less control on convergence, because convergence control (for example, step bounding) cannot be applied to variables without deltas.

Bit 1: Even handed augmentation. Standard augmentation treats variables which appear in non-constant coefficients in a different way from those which contain non-constant coefficients. Even-handed augmentation treats them all in the same way by replacing each non-constant coefficient C in a vector V by a new coefficient $C * V$ in the "equals" column (which has a fixed activity of 1) and creating delta vectors for all types of variable in the same way.

Bit 2: Penalty error vectors on all non-linear equality constraints. The linearization of a nonlinear equality constraint is inevitably an approximation and so will not generally be feasible except at the point of linearization. Adding penalty error vectors allows the linear approximation to be violated at a cost and so ensures that the linearized constraint is feasible.

Bit 3: Penalty error vectors on all non-linear inequality constraints. The linearization of a nonlinear constraint is inevitably an approximation and so may not be feasible except at the

point of linearization. Adding penalty error vectors allows the linear approximation to be violated at a cost and so ensures that the linearized constraint is feasible.

`Bit 4`: Penalty vectors to exceed step bounds. Although it has rarely been found necessary or desirable in practice, Xpress-SLP allows step bounds to be violated at a cost. This may help with feasibility but it generally slows down or prevents convergence, so it should be used only if found absolutely necessary.

`Bit 5`: Use arithmetic means to estimate penalty weights. Penalty weights are estimated from the magnitude of the elements in the constraint or interacting rows. Geometric means are normally used, so that a few excessively large or small values do not distort the weights significantly. Arithmetic means will value the coefficients more equally.

`Bit 6`: Estimate step bounds from values of row coefficients. If step bounds are to be imposed from the start, the best approach is to provide explicit values for the bounds. Alternatively, Xpress-SLP can estimate the values from the range of estimated coefficient sizes in the relevant rows.

`Bit 7`: Estimate step bounds from absolute values of row coefficients. If step bounds are to be imposed from the start, the best approach is to provide explicit values for the bounds. Alternatively, Xpress-SLP can estimate the values from the largest estimated magnitude of the coefficients in the relevant rows.

`Bit 8`: Row-based step bounds. Step bounds are normally applied as bounds on the delta variables. Some applications may find that using explicit rows to bound the delta vectors gives better results.

`Bit 9`: Penalty error vectors on all constraints. If the linear portion of the underlying model may actually be infeasible, then applying penalty vectors to all rows may allow identification of the infeasibility and may also allow a useful solution to be found.

`Bit 10`: Having an initial value will not cause the augmentation to include the corresponding delta variable; i.e. treat the variable as an SLP variable. Useful to provide initial values necessary in the first linearization in case of a minimal augmentation, or as a convenience option when it's easiest to set an initial value for all variables for some reason.

The following constants are provided for setting these bits:

Setting bit 0	`XSLP_MINIMUMAUGMENTATION`
Setting bit 1	`XSLP_EVENHANDEDAUGMENTATION`
Setting bit 2	`XSLP_EQUALITYERRORVECTORS`
Setting bit 3	`XSLP_ALLERRORVECTORS`
Setting bit 4	`XSLP_PENALTYDELTAVECTORS`
Setting bit 5	`XSLP_AMEANWEIGHT`
Setting bit 6	`XSLP_SBFROMVALUES`
Setting bit 7	`XSLP_SBFROMABSVALUES`
Setting bit 8	`XSLP_STEPBOUNDROWS`
Setting bit 9	`XSLP_ALLROWERRORVECTORS`
Setting bit 10	`XSLP_NOUPDATEIFONLYIV`

The recommended setting is bits 2 and 3 (penalty vectors on all nonlinear constraints).

Default value 12 (sets bits 2 and 3)

Affects routines `XSLPconstruct`

XSLP_AUTOSAVE

Description Frequency with which to save the model

Type Integer

Note A value of zero means that the model will not automatically be saved. A positive value of `n` will save model information at every `n`th SLP iteration as requested by XSLP_ANALYZIS.

Default value 0

Affects routines `XSLPmaxim`, `XSLPminim`

See also `XSLP_ANALYZE`

XSLP_BARCROSSOVERSTART

Description Default crossover activation behaviour for barrier start

Type Integer

Note When `XSLP_BARLIMIT` is set, `XSLP_BARCROSSOVERSTART` offers an overwrite control on when crossover is applied. A positive value indicates that crossover should be disabled in iterations smaller than `XSLP_BARCROSSOVERSTART` and should be enabled afterwards, or when stalling is detected as described in `XSLP_BARSTARTOPS`. A value of 0 indicates to respect the value of `XPRS_CROSSOVER` and only overwrite its value when stalling is detected. A value of -1 indicates to always rely on the value of `XPRS_CROSSOVER`.

Default value 0

Affects routines `XSLPmaxim`, `XSLPminim`

See also `XSLP_BARLIMIT`, `XSLP_BARSTARTOPS`, `XSLP_BARSTALLINGLIMIT`, `XSLP_BARSTALLINGOBJLIMIT`, `XSLP_BARSTALLINGTOL`

XSLP_BARLIMIT

Description Number of initial SLP iterations using the barrier method

Type Integer

Note Particularly for larger models, using the Newton barrier method is faster in the earlier SLP iterations. Later on, when the basis information becomes more useful, a simplex method generally performs better. `XSLP_BARLIMIT` sets the number of SLP iterations which will be performed using the Newton barrier method.

Default value 0

Affects routines `XSLPmaxim`, `XSLPminim`

See also `XSLP_BARCROSSOVERSTART`, `XSLP_BARSTARTOPS`, `XSLP_BARSTALLINGLIMIT`, `XSLP_BARSTALLINGOBJLIMIT`, `XSLP_BARSTALLINGTOL`

XSLP_BARSTALLINGLIMIT

Description Number of iterations to allow numerical failures in barrier before switching to dual

Type Integer

Note On large problems, it may be beneficial to warm start progress by running a number of iterations with the barrier solver as specified by XSLP_BARLIMIT. On some numerically difficult problems, the barrier may stop prematurely due to numerical issues. Such solves can sometimes be finished if crossover is applied. After XSLP_BARSTALLINGLIMIT such attempts, SLP will automatically switch to use the dual simplex.

Default value 3

Affects routines XSLPmaxim, XSLPminim

See also XSLP_BARCROSSOVERSTART, XSLP_BARLIMIT, XSLP_BARSTARTOPS, XSLP_BARSTALLINGOBJLIMIT, XSLP_BARSTALLINGTOL

XSLP_BARSTALLINGOBJLIMIT

Description Number of iterations over which to measure the objective change for barrier iterations with no crossover

Type Integer

Note On large problems, it may be beneficial to warm start progress by running a number of iterations with the barrier solver without crossover by setting XSLP_BARLIMIT to a positive value and setting XPRS_CROSSOVER to 0. A potential drawback is slower convergence due to the interior point provided by the barrier solve keeping a higher number of variables active. This may lead to stalling in progress, negating the benefit of using the barrier. When in the last XSLP_BARSTALLINGOBJLIMIT iterations no significant progress has been made, crossover is automatically enabled.

Default value 3

Affects routines XSLPmaxim, XSLPminim

See also XSLP_BARCROSSOVERSTART, XSLP_BARLIMIT, XSLP_BARSTARTOPS, XSLP_BARSTALLINGLIMIT, XSLP_BARSTALLINGTOL

XSLP_BARSTARTOPS

Description Controls behaviour when the barrier is used to solve the linearizations

Type Integer

Values

Bit	Meaning
0	Check objective progress when no crossover is applied.
1	Fall back to dual simplex if too many numerical problems are reported by the barrier.
2	If a non-vertex converged solution found by barrier without crossover can be returned as a final solution.

Note The following constants are provided for setting these bits:

Setting bit 0	BARSTARTOPS_STALLING_OBJECTIVE
Setting bit 1	BARSTARTOPS_STALLING_NUMERICAL
Setting bit 2	BARSTARTOPS_ALLOWINTERIORSOLUTION

Default value -1

Affects routines XSLPmaxim, XSLPminim

See also XSLP_BARCROSSOVERSTART, XSLP_BARLIMIT, XSLP_BARSTALLINGLIMIT, XSLP_BARSTALLINGOBJLIMIT, XSLP_BARSTALLINGTOL

XSLP_CALCTHREADS

Description Number of threads used for formula and derivatives evaluations

Type Integer

Note When beneficial, SLP can calculate formula values and partial derivative information in parallel.

Default value -1 (automatically determined)

Affects routines XSLPmaxim, XSLPmaxim

See also XSLP_THREADS,

XSLP_CASCADE

Description Bit map describing the cascading to be used

Type Integer

Values

Bit	Meaning
0	Apply cascading to all variables with determining rows.
1	Apply cascading to SLP variables which appear in coefficients and which would change by more than XPRS_FEASTOL.
2	Apply cascading to all SLP variables which appear in coefficients.
3	Apply cascading to SLP variables which are structural and which would change by more than XPRS_FEASTOL.
4	Apply cascading to all SLP variables which are structural.
5	Create secondary order groupping DR rows with instantiated user functions together in the order.

Note Normal cascading (bit 0) uses determining rows to recalculate the values of variables to be consistent with values already available or already recalculated.
Other bit settings are normally required only in quadratic programming where some of the SLP variables are in the objective function. The values of such variables may need to be corrected if the corresponding update row is slightly infeasible. The following constants are provided for setting these bits:

Setting bit 0	XSLP_CASCADE_ALL
Setting bit 1	XSLP_CASCADE_COEF_VAR
Setting bit 2	XSLP_CASCADE_ALL_COEF_VAR
Setting bit 3	XSLP_CASCADE_STRUCT_VAR
Setting bit 4	XSLP_CASCADE_ALL_STRUCT_VAR
Setting bit 5	XSLP_CASCADE_SECONDARY_GROUPS

Default value 1

Affects routines XSLPcascade

XSLP_CASCADENLIMIT

Description Maximum number of iterations for cascading with non-linear determining rows

Type Integer

Note Re-calculation of the value of a variable uses a modification of the Newton-Raphson method. The maximum number of steps in the method is set by XSLP_CASCADENLIMIT. If the maximum number of steps is taken without reaching a converged value, the best value found will be used.

Default value 10

Affects routines XSLPcascade

See also XSLP_CASCADE

XSLP_CONTROL

Description Bit map describing which Xpress NonLinear functions also activate the corresponding Optimizer Library function

Type Integer

Values

Bit	Meaning
0	Xpress NonLinear problem management functions do NOT invoke the corresponding Optimizer Library function for the underlying linear problem.
1	XSLPcopycontrols does NOT invoke XPRScopycontrols.
2	XSLPcopycallbacks does NOT invoke XPRScopycallbacks.
3	XSLPcopyprob does NOT invoke XPRScopyprob.
4	XSLPsetdefaults does NOT invoke XPRSsetdefaults.
5	XSLPsave does NOT invoke XPRSsave.
6	XSLPrestore does NOT invoke XPRSrestore.

Note The problem management functions are:
XSLPcopyprob to copy from an existing problem;
XSLPcopycontrols and XSLPcopycallbacks to copy the current controls and callbacks from an existing problem;
XSLPsetdefaults to reset the controls to their default values;
XSLPsave and XSLPrestore for saving and restoring a problem.

Default value 0 (no bits set)

Affects routines XSLPcopycontrols, XSLPcopycallbacks, XSLPcopyprob, XSLPrestore, XSLPsave, XSLPsetdefaults

XSLP_CONVERGENCEOPS

Description Bit map describing which convergence tests should be carried out

Type Integer

Values

Bit	Meaning
0	Execute the closure tolerance checks.
1	Execute the delta tolerance checks.
2	Execute the matrix tolerance checks.
3	Execute the impact tolerance checks.
4	Execute the slack impact tolerance checks.
5	Check for user provided convergence.
6	Execute the objective range checks.
7	Execute the objective range + constraint activity check.
8	Execute the objective range + active step bound check.
9	Execute the convergence continuation check.
10	Take scaling of individual variables / rows into account.
11	Execute the validation target convergence checks.
12	Execute the first order optimality target convergence checks.

Note Provides fine tuned control (over setting the related convergence tolerances) of which convergence checks are carried out.

The following constants are provided for setting these bits:

Setting bit 0	XSLP_CONVERGEBIT_CTOL
Setting bit 1	XSLP_CONVERGEBIT_ATOL
Setting bit 2	XSLP_CONVERGEBIT_MTOL
Setting bit 3	XSLP_CONVERGEBIT_ITOL
Setting bit 4	XSLP_CONVERGEBIT_STOL
Setting bit 5	XSLP_CONVERGEBIT_USER
Setting bit 6	XSLP_CONVERGEBIT_VTOL
Setting bit 7	XSLP_CONVERGEBIT_XTOL
Setting bit 8	XSLP_CONVERGEBIT_OTOL
Setting bit 9	XSLP_CONVERGEBIT_WTOL
Setting bit 10	XSLP_CONVERGEBIT_EXTENDEDSCALING
Setting bit 11	CONVERGEBIT_VALIDATION
Setting bit 12	CONVERGEBIT_VALIDATION_K

Default value 7167 (bits 0-9 and 11-12 are set)

Affects routines XSLPmaxim, XSLPminim

XSLP_DAMPSTART

Description SLP iteration at which damping is activated

Type Integer

Note If damping is used as part of the SLP algorithm, it can be delayed until a specified SLP iteration. This may be appropriate when damping is used to encourage convergence after an un-damped algorithm has failed to converge.

Default value 0

Affects routines XSLPmaxim, XSLPmaxim

See also XSLP_ALGORITHM, XSLP_DAMPEXPAND, XSLP_DAMPMAX, XSLP_DAMPMIN, XSLP_DAMPSHRINK

XSLP_DCLIMIT

Description Default iteration delay for delayed constraints

Type Integer

Note If a delayed constraint does not have an explicit delay, then the value of XSLP_DCLIMIT will be used.

Default value 5

Affects routines XSLPmaxim, XSLPminim

XSLP_DCLOG

Description Amount of logging information for activcation of delayed constraints

Type Integer

Note If XSLP_DCLOG is set to 1, then a message will be produced for each DC as it is activated.

Default value 0

Affects routines XSLPmaxim, XSLPminim

XSLP_DELAYUPDATEROWS

Description Number of SLP iterations before update rows are fully activated

Type Integer

Notes Update rows are an integral part of the augmented matrix used to create linear approximations of the nonlinear problem. However, if determining rows are present, then it is possible for some updated values to be calculated during cascading, and the corresponding update rows are then not required. When SLP variables have explicit bounds, and particularly when step bounding is enforced, update rows become important to the solutions actually obtained. It is therefore normal practice to delay update rows for only a few initial SLP iterations.
Update rows can only be delayed for variables which are not structural (that is, they do not have explicit coefficients in the original problem) and for which determining rows are provided.

Default value 2

Affects routines XSLPmaxim, XSLPminim

XSLP_DELTAOFFSET

Description Position of first character of SLP variable name used to create name of delta vector

Type Integer

Note During augmentation, a delta vector, and possibly penalty delta vectors, are created for each SLP variable. They are created with names derived from the corresponding SLP variable. Customized naming is possible using XSLP_DELTAFORMAT etc to define a format and XSLP_DELTAOFFSET to define the first character (counting from zero) of the variable name to be used.

Default value 0

Affects routines XSLPconstruct

See also XSLP_DELTAFORMAT, XSLP_MINUSDELTAFORMAT, XSLP_PLUSDELTAFORMAT

XSLP_DELTAZLIMIT

Description Number of SLP iterations during which to apply XSLP_DELTA_Z

Type Integer

Note XSLP_DELTA_Z is used to retain small derivatives which would otherwise be regarded as zero. This is helpful in avoiding local optima, but may make the linearized problem more difficult to solve because of the number of small nonzero elements in the resulting matrix. XSLP_DELTAZLIMIT can be set to a nonzero value, which is then the number of iterations for which XSLP_DELTA_Z will be used. After that, small derivatives will be set to zero. A negative value indicates no automatic perturbations to the derivatives in any situation.

Default value 0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_DELTA_Z

XSLP_DERIVATIVES

Description Bitmap describing the method of calculating derivatives

Type Integer

Values

Bit	Meaning
0	analytic derivatives where possible
1	avoid embedding numerical derivatives of instantiated functions into analytic derivatives

Notes If no bits are set then numerical derivatives are calculated using finite differences.
Analytic derivatives cannot be used for formulae involving discontinuous functions. They may not work well with functions which are not smooth (such as `MAX`), or where the derivative changes very quickly with the value of the variable (such as `LOG` of small values).
Both first and second order analytic derivatives can either be calculated as symbolic formulas, or by the means of auto-differentiation, with the exception that the second order symbolic derivatives require that the first order derivatives are also calculated using the symbolic method.

Default value 1

Affects routines `XSLPconstruct`, `XSLPmaxim`, `XSLPminim`

See also `XSLP_JACOBIAN`, `XSLP_HESSIAN`

XSLP_DETERMINISTIC

Description Determines if the parallel features of SLP should be guaranteed to be deterministic

Type Integer

Note Determinism can only be guaranteed if no callbacks are used, or if in the presence of callbacks the effect of the callbacks only depend on local information provided by SLP.

Default value 1

Affects routines `XSLPminim`, `XSLPmaxim`

See also `XSLP_MULTISTART_POOLSIZE`,

XSLP_ECFCHECK

Description Check feasibility at the point of linearization for extended convergence criteria

Type Integer

Values

0	no check (extended criteria are always used);
1	check until one infeasible constraint is found;
2	check all constraints.

Notes The extended convergence criteria measure the accuracy of the solution of the linear approximation compared to the solution of the original nonlinear problem. For this to work, the linear approximation needs to be reasonably good at the point of linearization. In particular, it needs to be reasonably close to feasibility.
XSLP_ECFCHECK is used to determine what checking of feasibility is carried out at the point of linearization. If the point of linearization at the start of an SLP iteration is deemed to be infeasible, then the extended convergence criteria are not used to decide convergence at the end of that SLP iteration.
If all that is required is to decide that the point of linearization is not feasible, then the search can stop after the first infeasible constraint is found (parameter is set to 1). If the actual number of infeasible constraints is required, then XSLP_ECFCHECK should be set to 2, and all constraints will be checked.
The number of infeasible constraints found at the point of linearization is returned in XSLP_ECFCOUNT.

Default value 1

Affects routines Convergence criteria, XSLPmaxim, XSLPminim

See also XSLP_ECFCOUNT, XSLP_ECFTOL_A, XSLP_ECFTOL_R

XSLP_ECHOXPRSMESSAGES

Description Controls if the XSLP message callback should relay messages from the XPRS library.

Type Integer

Note In case the XSLP and XPRS logs are handled the same way by an application, setting this control to 1 makes it sufficient to implement the XSLP messaging callback only.

Default value 0

XSLP_ERROROFFSET

Description Position of first character of constraint name used to create name of penalty error vectors

Type Integer

Note During augmentation, penalty error vectors may be created for some or all of the constraints. The vectors are created with names derived from the corresponding constraint name. Customized naming is possible using XSLP_MINUSERRORFORMAT and XSLP_PLUSERRORFORMAT to define a format and XSLP_ERROROFFSET to define the first character (counting from zero) of the constraint name to be used.

Default value 0

Affects routines XSLPconstruct

See also XSLP_MINUSERRORFORMAT, XSLP_PLUSERRORFORMAT

XSLP_EVALUATE

Description Evaluation strategy for user functions

Type Integer

Values

0	use derivatives where possible;
1	always re-evaluate.

Note If a user function returns derivatives or returns more than one value, then it is possible for Xpress NonLinear to estimate the value of the function from its derivatives if the new point of evaluation is sufficiently close to the original. Setting XSLP_EVALUATE to 1 will force re-evaluation of all functions regardless of how much or little the point of evaluation has changed.

Default value 0

Affects routines XSLPevaluatecoef, XSLPevaluateformula

See also XSLP_FUNCEVAL

XSLP_FILTER

Description Bit map for controlling solution updates

Type Integer

Values

Bit	Meaning
0	retrain solution best according to the merit function.
1	check cascaded solutions against improvements in the merit function.
2	force minimum step sizes in line search.
3	accept the trust region step is the line search returns a zero step size.

Notes Bits 0 determine if XSLPgetslpsol should return the final converged solution, or the solution which had the best value according to the merit function.
If bit 1 is set, a cascaded solution which does not improve the merit function will be rejected (XSLP will revert to the solution of the linearization).
Bits 2-3 determine the strategy for when the step direction is not improving according to the merit function.
The following constants are provided for setting these bits:

Setting bit 0	XSLP_FILTER_KEEPBEST
Setting bit 1	XSLP_FILTER_CASCADE
Setting bit 2	XSLP_FILTER_ZEROLINESEARCH
Setting bit 3	XSLP_FILTER_ZEROLINESEARCHTR

Default value 11 (bit 0,1,3)

Affects routines XSLPmaxim, XSLPminim, XSLPcascade

See also XSLP_MERITLAMBDA, XSLP_CASCADE, XSLP_LSSTART, XSLP_LSITERLIMIT, XSLP_LSPATTERNLIMIT

XSLP_FINDIV

Description Option for running a heuristic to find a feasible initial point

Type Integer

Values

−1	Automatic (default).
0	Disable the heuristic.
1	Enable the heuristic.

Notes The procedure uses bound reduction (and, up to an extent, probing) to obtain a point in the initial bounding box that is feasible for the bound reduction techniques.
If an initial point is already specified and is found not to violate bound reduction, then the heuristic is not run and the given point is used as the initial solution.

Default value -1

Affects routines XSLPmaxim, XSLPminim

XSLP_FUNCEVAL

Description Bit map for determining the method of evaluating user functions and their derivatives

Type Integer

Values

Bit	Meaning
3	evaluate function whenever independent variables change.
4	evaluate function when independent variables change outside tolerances.
5	application of bits 3-4: 0 = functions which do not have a defined re-evaluation mode;1 = all functions.
6	tangential derivatives.
7	forward derivatives
8	application of bits 6-7: 0 = functions which do not have a defined derivative mode;1 = all functions.

Notes Bits 3-4 determine the type of function re-evaluation. If both bits are zero, then the settings for each individual function are used.
If bit 3 or bit 4 is set, then bit 5 defines which functions the setting applies to. If it is set to 1, then it applies to all functions. Otherwise, it applies only to functions which do not have an explicit setting of their own.
Bits 6-7 determine the type of calculation for numerical derivatives. If both bits are zero, then the settings for each individual function are used.
If bit 6 or bit 7 is set, then bit 8 defines which functions the setting applies to. If it is set to 1, then it applies to all functions. Otherwise, it applies only to functions which do not have an explicit setting of their own.

The following constants are provided for setting these bits:

Setting bit 3	XSLP_RECALC
Setting bit 4	XSLP_TOLCALC
Setting bit 5	XSLP_ALLCALCS
Setting bit 6	XSLP_2DERIVATIVE
Setting bit 7	XSLP_1DERIVATIVE
Setting bit 8	XSLP_ALLDERIVATIVES

Default value 0

Affects routines XSLPevaluatecoef, XSLPevaluateformula

See also XSLP_EVALUATE

XSLP_GRIDHEURSELECT

Description Bit map selectin which heuristics to run if the problem has variable with an integer delta

Type Integer

Values

Bit	Meaning
0	Enumeration: try all combinations.
1	Simple search heuristics.
2	Simulated annealing.

Note A value of 0 indicates that integer deltas are only taken into consideration during the SLP iterations.

Note The enumeration option can be useful for cases where the number of possible values of the variables with an integer delta is small.

Default value 3 (bits 1-2 are set)

Affects routines XSLPmaxim, XSLPminim

XSLP_HEURSTRATEGY

Description Branch and Bound: This specifies the MINLP 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.
4	Run all heuristics without effort limits.

Default value −1

Affects routines XSLPminim, XSLPmaxim.

XSLP_HESSIAN

Description Second order differentiation mode when using analytical derivatives

Type Integer

Values

`-1,0`	automatic selection
`1`	numerical derivatives (finite difference)
`2`	symbolic differentiation
`3`	automatic differentiation

Note Symbolic mode differentiation for the second order derivatives is only available when `XSLP_JACOBIAN` is also set to symbolic mode.

Default value -1

See also `XSLP_DERIVATIVES`, `XSLP_JACOBIAN`

XSLP_INFEASLIMIT

Description The maximum number of consecutive infeasible SLP iterations which can occur before Xpress-SLP terminates

Type Integer

Note An infeasible solution to an SLP iteration means that is likely that Xpress-SLP will create a poor linear approximation for the next SLP iteration. Sometimes, small infeasibilities arise because of numerical difficulties and do not seriously affect the solution process. However, if successive solutions remain infeasible, it is unlikely that Xpress-SLP will be able to find a feasible converged solution. `XSLP_INFEASLIMIT` sets the number of successive SLP iterations which must take place before Xpress-SLP terminates with a status of "infeasible solution".

Default value 3

Affects routines `XSLPmaxim`, `XSLPminim`

XSLP_ITERLIMIT

Description The maximum number of SLP iterations

Type Integer

Note If Xpress-SLP reaches `XSLP_ITERLIMIT` without finding a converged solution, it will stop. For MISLP, the limit is on the number of SLP iterations at each node.

Default value 1000

Affects routines `XSLPglobal`, `XSLPmaxim`, `XSLPminim`

XSLP_JACOBIAN

Description First order differentiation mode when using analytical derivatives

Type Integer

Values

`-1,0`	automatic selection
`1`	numerical derivatives (finite difference)
`2`	symbolic differentiation
`3`	automatic differentiation

Note Symbolic mode differentiation for the second order derivatives is only available when `XSLP_JACOBIAN` is set to symbolic mode.

Default value -1

See also `XSLP_DERIVATIVES`, `XSLP_HESSIAN`

XSLP_LINQUADBR

Description Use linear and quadratic constraints and objective function to further reduce bounds on all variables

Type Integer

Values

`-1`	automatic selection
`0`	disable
`1`	enable

Note While bound reduction is effective when performed on nonlinear, nonquadratic constraints and objective function, it can be useful to obtain tightened bounds from linear and quadratic constraints, as the corresponding variables may appear in other nonlinear constraints. This option then allows for a slightly more expensive bound reduction procedure, at the benefit of further reduction in the problem's bounds.

Default value -1

See also `XSLP_PRESOLVEOPS`, `XSLP_PROBING`

XSLP_LOG

Description Level of printing during SLP iterations

Type Integer

Values

−1	none
0	minimal
1	normal: iteration, penalty vectors
2	omit from convergence log any variables which have converged
3	omit from convergence log any variables which have already converged (except variables on step bounds)
4	include all variables in convergence log
5	include user function call communications in the log

Default value 0

Affects routines XSLPmaxim, XSLPminim

XSLP_LSITERLIMIT

Description Number of iterations in the line search

Type Integer

Notes The line search attempts to refine the step size suggested by the trust region step bounds. The line search is a local method; the control sets a maximum on the number of model evaluations during the line search.

Default value 0

See also XSLP_LSPATTERNLIMIT, XSLP_LSSTART, XSLP_LSZEROLIMIT, XSLP_FILTER

Affects routines XSLPmaxim, XSLPminim

XSLP_LSPATTERNLIMIT

Description Number of iterations in the pattern search preceding the line search

Type Integer

Notes When positive, defines the number of samples taken along the step size suggested by the trust region step bounds before initiating the line search. Useful for highly non-convex problems.

Default value 0

See also XSLP_LSITERLIMIT, XSLP_LSSTART, XSLP_LSZEROLIMIT, XSLP_FILTER

Affects routines XSLPmaxim, XSLPminim

XSLP_LSSTART

Description Iteration in which to active the line search

Type Integer

Notes

Default value 8

See also XSLP_LSITERLIMIT, XSLP_LSPATTERNLIMIT, XSLP_LSZEROLIMIT, XSLP_FILTER

Affects routines XSLPmaxim, XSLPminim

XSLP_LSZEROLIMIT

Description Maximum number of zero length line search steps before line search is deactivated

Type Integer

Notes When the line search repeatedly returns a zero step size, counteracted by bits set on XSLP_FILTER, the effort spent in line search is redundant, and line search will be deactivated after XSLP_LSZEROLIMIT consecutive such iteration.

Default value 5

See also XSLP_LSITERLIMIT, XSLP_LSPATTERNLIMIT, XSLP_LSSTART, XSLP_FILTER

Affects routines XSLPmaxim, XSLPminim

XSLP_MAXTIME

Description The maximum time in seconds that the SLP optimization will run before it terminates

Type Integer

Notes The (elapsed) time is measured from the beginning of the first SLP optimization.
If XSLP_MAXTIME is negative, Xpress NonLinear will terminate after (−XSLP_MAXTIME) seconds. If it is positive, Xpress NonLinear will terminate in MISLP after XSLP_MAXTIME seconds or as soon as an integer solution has been found thereafter.

Default value 0

Affects routines XSLPglobal, XSLPmaxim, XSLPminim

XSLP_MIPALGORITHM

Description Bitmap describing the MISLP algorithms to be used

Type Integer

Values

Bit	Meaning
0	Solve initial SLP to convergence.
2	Relax step bounds according to XSLP_MIPRELAXSTEPBOUNDS after initial node.
3	Fix step bounds according to XSLP_MIPFIXSTEPBOUNDS after initial node.
4	Relax step bounds according to XSLP_MIPRELAXSTEPBOUNDS at each node.
5	Fix step bounds according to XSLP_MIPFIXSTEPBOUNDS at each node.
6	Limit iterations at each node to XSLP_MIPITERLIMIT.
7	Relax step bounds according to XSLP_MIPRELAXSTEPBOUNDS after MIP solution is found.
8	Fix step bounds according to XSLP_MIPFIXSTEPBOUNDS after MIP solution is found.
9	Use MIP at each SLP iteration instead of SLP at each node.
10	Use MIP on converged SLP solution and then SLP on the resulting MIP solution.

Notes

XSLP_MIPALGORITHM determines the strategy of XSLPglobal for solving MINLP problems. The recommended approach is to solve the problem first without reference to the global variables. This can be handled automatically by setting bit 0 of XSLP_MIPALGORITHM; if done manually, then optimize using the "l" option to prevent the Optimizer presolve from changing the problem.
Some versions of the optimizer re-run the initial node as part of the global search; it is possible to initiate a new SLP optimization at this point by relaxing or fixing step bounds (use bits 2 and 3). If step bounds are fixed for a class of variable, then the variables in that class will not change their value in any child node.
At each node, it is possible to relax or fix step bounds. It is recommended that step bounds are relaxed, so that the new problem can be solved starting from its parent, but without undue restrictions cased by step bounding (use bit 4). Exceptionally, it may be preferable to restrict the freedom of child nodes by relaxing fewer types of step bound or fixing the values of some classes of variable (use bit 5).
When the optimal node has been found, it is possible to fix the global variables and then re-optimize with SLP. Step bounds can be relaxed or fixed for this optimization as well (use bits 7 and 8).
Although it is ultimately necessary to solve the optimal node to convergence, individual nodes can be truncated after XSLP_MIPITERLIMIT SLP iterations. Set bit 6 to activate this feature.
The normal MISLP algorithm uses SLP at each node. One alternative strategy is to use the MIP optimizer for solving each SLP iteration. Set bit 9 to implement this strategy ("MIP within SLP").
Another strategy is to solve the problem to convergence ignoring the nature of the global variables. Then, fixing the linearization, use MIP to find the optimal setting of the global variables. Then, fixing the global variables, but varying the linearization, solve to convergence. Set bit 10 to implement this strategy ("SLP then MIP").
For mode details about MISLP algorithms and strategies, see the separate section.

The following constants are provided for setting these bits:

Setting bit 0	XSLP_MIPINITIALSLP
Setting bit 1	XSLP_MIPFINALSLP
Setting bit 2	XSLP_MIPINITIALRELAXSLP
Setting bit 3	XSLP_MIPINITIALFIXSLP
Setting bit 4	XSLP_MIPNODERELAXSLP
Setting bit 5	XSLP_MIPNODEFIXSLP
Setting bit 6	XSLP_MIPNODELIMITSLP
Setting bit 7	XSLP_MIPFINALRELAXSLP
Setting bit 8	XSLP_MIPFINALFIXSLP
Setting bit 9	XSLP_MIPWITHINSLP
Setting bit 10	XSLP_SLPTHENMIP
Setting bit 11	XSLP_NOFINALROUNDING

Default value 17 (bits 0 and4 are set)

Affects routines XSLPglobal

See also XSLP_ALGORITHM, XSLP_MIPFIXSTEPBOUNDS, XSLP_MIPITERLIMIT, XSLP_MIPRELAXSTEPBOUNDS

XSLP_MIPCUTOFFCOUNT

Description Number of SLP iterations to check when considering a node for cutting off

Type Integer

Notes If the objective function is worse by a defined amount than the best integer solution obtained so far, then the SLP will be terminated (and the node will be cut off). The node will be cut off at the current SLP iteration if the objective function for the last XSLP_MIPCUTOFFCOUNT SLP iterations are all worse than the best obtained so far, and the difference is greater than $XSLP\_MIPCUTOFF\_A$ and $OBJ * XSLP\_MIPCUTOFF\_R$ where OBJ is the best integer solution obtained so far.
The test is not applied until at least XSLP_MIPCUTOFFLIMIT SLP iterations have been carried out at the current node.

Default value 5

Affects routines XSLPglobal

See also XSLP_MIPCUTOFF_A, XSLP_MIPCUTOFF_R, XSLP_MIPCUTOFFLIMIT

XSLP_MIPCUTOFFLIMIT

Description Number of SLP iterations to check when considering a node for cutting off

Type Integer

Notes If the objective function is worse by a defined amount than the best integer solution obtained so far, then the SLP will be terminated (and the node will be cut off). The node will be cut off at the current SLP iteration if the objective function for the last XSLP_MIPCUTOFFCOUNT SLP iterations are all worse than the best obtained so far, and the difference is greater than $XSLP\_MIPCUTOFF\_A$ and $OBJ * XSLP\_MIPCUTOFF\_R$ where OBJ is the best integer solution obtained so far.

The test is not applied until at least XSLP_MIPCUTOFFLIMIT SLP iterations have been carried out at the current node.

Default value 10

Affects routines XSLPglobal

See also XSLP_MIPCUTOFF_A, XSLP_MIPCUTOFF_R, XSLP_MIPCUTOFFCOUNT

XSLP_MIPDEFAULTALGORITHM

Description Default algorithm to be used during the global search in MISLP

Type Integer

Note The default algorithm used within SLP during the MISLP optimization can be set using XSLP_MIPDEFAULTALGORITHM. It will not necessarily be the same as the one best suited to the initial SLP optimization.

Default value 3 (primal simplex)

Affects routines XSLPglobal

See also XPRS_DEFAULTALG, XSLP_MIPALGORITHM

XSLP_MIPFIXSTEPBOUNDS

Description Bitmap describing the step-bound fixing strategy during MISLP

Type Integer

Values

Bit	Meaning
0	Fix step bounds on structural SLP variables which are not in coefficients.
1	Fix step bounds on all structural SLP variables.
2	Fix step bounds on SLP variables appearing only in coefficients.
3	Fix step bounds on SLP variables appearing in coefficients.

Note At any node (including the initial and optimal nodes) it is possible to fix the step bounds of classes of variables so that the variables themselves will not change. This may help with convergence, but it does increase the chance of a local optimum because of excessive artificial restrictions on the variables.

Default value 0

Affects routines XSLPglobal

See also XSLP_MIPALGORITHM, XSLP_MIPRELAXSTEPBOUNDS

XSLP_MIPITERLIMIT

Description Maximum number of SLP iterations at each node

Type Integer

Note If bit 6 of XSLP_MIPALGORITHM is set, then the number of iterations at each node will be limited to XSLP_MIPITERLIMIT.

Default value 0

Affects routines XSLPglobal

See also XSLP_ITERLIMIT, XSLP_MIPALGORITHM

XSLP_MIPLOG

Description Frequency with which MIP status is printed

Type Integer

Note By default (zero or negative value) the MIP status is printed after syncronization points. If XSLP_MIPLOG is set to a positive integer, then the current MIP status (node number, best value, best bound) is printed every XSLP_MIPLOG nodes.

Default value 0 (deterministic logging)

Affects routines XSLPglobal

See also XSLP_LOG, XSLP_SLPLOG

XSLP_MIPOCOUNT

Description Number of SLP iterations at each node over which to measure objective function variation

Type Integer

Note The objective function test for MIP termination is applied only when step bounding has been applied (or XSLP_SBSTART SLP iterations have taken place if step bounding is not being used). The node will be terminated at the current SLP iteration if the range of the objective function values over the last XSLP_MIPOCOUNT SLP iterations is within *XSLP_MIPOTOL_A* or within *OBJ* $*$ *XSLP_MIPOTOL_R* where *OBJ* is the average value of the objective function over those iterations.

Default value 5

Affects routines XSLPglobal

See also XSLP_MIPOTOL_A XSLP_MIPOTOL_R XSLP_SBSTART

XSLP_MIPRELAXSTEPBOUNDS

Description Bitmap describing the step-bound relaxation strategy during MISLP

Type Integer

Values

Bit	Meaning
0	Relax step bounds on structural SLP variables which are not in coefficients.
1	Relax step bounds on all structural SLP variables.
2	Relax step bounds on SLP variables appearing only in coefficients.
3	Relax step bounds on SLP variables appearing in coefficients.

Note At any node (including the initial and optimal nodes) it is possible to relax the step bounds of classes of variables so that the variables themselves are completely free to change. This may help with finding a global optimum, but it may also increase the solution time, because more SLP iterations are necessary at each node to obtain a converged solution.

Default value 15 (relax all types)

Affects routines XSLPglobal

See also XSLP_MIPALGORITHM, XSLP_MIPFIXSTEPBOUNDS

XSLP_MULTISTART

Description The multistart master control. Defines if the multistart search is to be initiated, or if only the baseline model is to be solved.

Type Integer

Values

−1	Depends on if any multistart jobs have been added.
0	Multistart is off.
1	Multistart is on.

Note By default, the multistart search will always be initiated if multistart jobs have been added to the problem. The (original) base problem is not part of the multisearch job pool. To make it so, add an job with no extra settings (template job). It might be useful to load multiple template jobs, and customize them from callbacks.

Default value -1

Affects routines XSLPminim, XSLPmaxim

See also XSLP_MULTISTART_MAXSOLVES, XSLP_MULTISTART_MAXTIME

XSLP_MULTISTART_MAXSOLVES

Description The maximum number of jobs to create during the multistart search.

Type Integer

Note This control can be increased on the fly during the mutlistart search: for example, if a job gets refused by a user callback, the callback may increase this limit to account for the rejected job.

Default value 0 (no upper limit)

Affects routines XSLPminim, XSLPmaxim

See also XSLP_MULTISTART, XSLP_MULTISTART_MAXTIME

XSLP_MULTISTART_MAXTIME

Description The maximum total time to be spent in the mutlistart search.

Type Integer

Note XSLP_MAXTIME applies on a per job instance basis. There will be some time spent even after XSLP_MULTISTART_MAXTIME has elapsed, while the running jobs get terminated and their results collected.

Default value 0 (no upper limit)

Affects routines XSLPminim, XSLPmaxim

See also XSLP_MULTISTART, XSLP_MULTISTART_MAXSOLVES

XSLP_MULTISTART_POOLSIZE

Description The maximum number of problem objects allowed to pool up before synchronization in the deterministic multistart.

Type Integer

Default value 2

Note Deterministic multistart is ensured by guaranteeing that the multistart solve results are evaluated in the same order every time. Solves that finish too soon can be pooled until all earlier started solves finish, allowing the system to start solving other multistart instances in the meantime on idle threads. Larger pool sizes will provide better speedups, but will require larger amounts of memory. Positive values are interpreted as a multiplier on the maximum number of active threads used, while negative values are interpreted as an absolute limit (and the absolute value is used). A value of zero will mean no result pooling.

Affects routines XSLPminim, XSLPmaxim

See also XSLP_MULTISTART, XSLP_DETERMINISTIC

XSLP_MULTISTART_SEED

Description Random seed used for the automatic generation of initial point when loading multistart presets

Type Integer

Default value 0

Affects routines XSLPminim, XSLPmaxim

See also XSLP_MULTISTART

XSLP_MULTISTART_THREADS

Description The maximum number of threads to be used in multistart

Type Integer

Default value -1

Note The current hard upper limit on the number of threads to be sued in multistart is 64.

Affects routines XSLPminim, XSLPmaxim

See also XSLP_MULTISTART

XSLP_OCOUNT

Description Number of SLP iterations over which to measure objective function variation for static objective (2) convergence criterion

Type Integer

Note The static objective (2) convergence criterion does not measure convergence of individual variables. Instead, it measures the significance of the changes in the objective function over recent SLP iterations. It is applied when all the variables interacting with active constraints (those that have a marginal value of at least XSLP_MVTOL) have converged. The rationale is that if the remaining unconverged variables are not involved in active constraints and if the objective function is not changing significantly between iterations, then the solution is more-or-less practical.
The variation in the objective function is defined as

$$\delta Obj = MAX_{Iter}(Obj) - MIN_{Iter}(Obj)$$

where *Iter* is the XSLP_OCOUNT most recent SLP iterations and *Obj* is the corresponding objective function value.
If $ABS(\delta Obj) \leq XSLP\_OTOL\_A$
then the problem has converged on the absolute static objective (2) convergence criterion.
The static objective function (2) test is applied only if XSLP_OCOUNT is at least 2.

Default value 5

Affects routines XSLPmaxim, XSLPminim

See also XSLP_OTOL_A XSLP_OTOL_R

XSLP_PENALTYINFOSTART

Description Iteration from which to record row penalty information

Type Integer

Note Information about the size (current and total) of active penalties of each row and the number of times a penalty vector has been active is recorded starting at the SLP iteration number given by XSLP_PENALTYINFOSTART.

Default value 3

XSLP_POSTSOLVE

Description This control determines whether postsolving should be performed automatically

Type Integer

Values

0	Do not automatically postsolve.
1	Postsolve automatically.

Default value 0

See also XSLP_PRESOLVE

XSLP_PRESOLVE

Description This control determines whether presolving should be performed prior to starting the main algorithm

Type Integer

Values

0	Disable SLP presolve.
1	Activate SLP presolve.
2	Low memory presolve. Original problem is not restored by postsolve and dual solution may not be completely postsolved.

Note The Xpress NonLinear nonlinear presolve (which is carried out once, before augmentation) is independent of the Optimizer presolve (which is carried out during each SLP iteration).

Default value 1

Affects routines XSLPconstruct, XSLPpresolve

See also XSLP_PRESOLVELEVEL, XSLP_PRESOLVEOPS, XSLP_REFORMULATE, XSLP_PRESOLVEPASSLIMIT

XSLP_PRESOLVELEVEL

Description This control determines the level of changes presolve may carry out on the problem

Type Integer

Values

`XSLP_PRESOLVELEVEL_LOCALIZED` Individual rows only presolve, no nonlinear transformations.

`XSLP_PRESOLVELEVEL_BASIC` Individual rows and bounds only presolve, no nonlinear transformations.

`XSLP_PRESOLVELEVEL_LINEAR` Presolve allowing changing problem dimension, no nonlinear transformations.

`XSLP_PRESOLVELEVEL_FULL` Full presolve.

Note `XSLP_PRESOLVEOPS` and `XSLP_REFORMULATE` controls the operations carried out in presolve. XSLP_PRESOLVELEVEL controls how those operations may change the problem.

Default value XSLP_PRESOLVELEVEL_FULL

Affects routines `XSLPconstruct`, `XSLPpresolve`

See also `XSLP_PRESOLVE`, `XSLP_PRESOLVEOPS`, `XSLP_REFORMULATE`, `XSLP_PRESOLVEPASSLIMIT`

XSLP_PRESOLVEOPS

Description Bitmap indicating the SLP presolve actions to be taken

Type Integer

Values

Bit	Meaning
0	Generic SLP presolve.
1	Explicitly fix columns identified as fixed to zero.
2	Explicitly fix all columns identified as fixed.
3	SLP bound tightening.
4	MISLP bound tightening.
5	Bound tightening based on function domains.
8	Do not presolve coefficients.
9	Do not remove delta variables.
10	Avoid reductions that can not be dual postsolved.
11	Allow eliminations on determined variables.

Note The Xpress NonLinear nonlinear presolve (which is carried out once, before augmentation) is independent of the Optimizer presolve (which is carried out during each SLP iteration).

Default value 24

Affects routines `XSLPconstruct`, `XSLPpresolve`

See also `XSLP_PRESOLVELEVEL`, `XSLP_PRESOLVE`, `XSLP_PRESOLVEOPS`, `XSLP_PRESOLVEPASSLIMIT`, `XSLP_REFORMULATE`

XSLP_PRESOLVEPASSLIMIT

Description Maximum number of passes through the problem to improve SLP bounds

Type Integer

Note The Xpress NonLinear nonlinear presolve (which is carried out once, before augmentation) is independent of the Optimizer presolve (which is carried out during each SLP iteration). The procedure carries out a number of passes through the SLP problem, seeking to tighten implied bounds or to identify fixed values. `XSLP_PRESOLVEPASSLIMIT` can be used to change the maximum number of passes carried out.

Default value 20

Affects routines `XSLPpresolve`

See also `XSLP_PRESOLVE`

XSLP_PROBING

Description This control determines whether probing on a subset of variables should be performed prior to starting the main algorithm. Probing runs multiple times bound reduction in order to further tighten the bounding box.

Type Integer

Values

`−1`	Automatic.
`0`	Disable SLP probing.
`1`	Activate SLP probing only on binary variables.
`2`	Activate SLP probing only on binary or unbounded integer variables.
`3`	Activate SLP probing only on binary or integer variables.
`4`	Activate SLP probing only on binary, integer variables, and unbounded continuous variables.
`5`	Activate SLP probing on any variable.

Default value -1: XSLP sets the probing level based on the problem size

Note The Xpress NonLinear nonlinear probing, which is carried out once, is independent of the Optimizer presolve (which is carried out during each SLP iteration). The probing level allows for probing on an expanding set of variables, allowing for probing on all variables (level 5) or only those for which probing is more likely to be useful (binary variables).

Affects routines `XSLPpresolve`

See also `XSLP_PRESOLVEOPS`,

XSLP_REFORMULATE

Description Controls the problem reformulations carried out before augmentation. This allows SLP to take advantage of dedicated algorithms for special problem classes.

Type Integer

Values

Bit	Meaning
0	Solve convex quadratic objectives using the XPRS library .
1	Convert non-convex quadratic objectives to SLP constructs .
2	Solve convex quadratic constraints using the XPRS library.
3	Convert non-convex QCQP constraints to SLP constructs.
4	Convexity of a quadratic only problem may be checked by calling the optimizer to solve the instance.

Default value -1: All structures are checked against reformulation

Note The reformulation is part of XSLP presolve, and is only carried out if XSLP_PRESOLVE is nonzero. The following constants are provided for setting these bits:

Setting bit 0	XSLP_REFORMULATE_SLP2QP
Setting bit 1	XSLP_REFORMULATE_QP2SLP
Setting bit 2	XSLP_REFORMULATE_SLP2QCQP
Setting bit 3	XSLP_REFORMULATE_QCQP2SLP
Setting bit 4	XSLP_REFORMULATE_QPSOLVE

Affects routines XSLPconstruct, XSLPminim, XSLPmaxim, XSLPreminim, XSLPremaxim, XSLPnlpoptimize, XSLPglobal

XSLP_SAMECOUNT

Description Number of steps reaching the step bound in the same direction before step bounds are increased

Type Integer

Note If step bounding is enabled, the step bound for a variable will be increased if successive changes are in the same direction. More precisely, if there are XSLP_SAMECOUNT successive changes reaching the step bound and in the same direction for a variable, then the step bound (B) for the variable will be reset to
$B * XSLP\_EXPAND$.

Default value 3

Affects routines XSLPmaxim, XSLPminim

See also XSLP_EXPAND

XSLP_SAMEDAMP

Description Number of steps in same direction before damping factor is increased

Type Integer

Note If dynamic damping is enabled, the damping factor for a variable will be increased if successive changes are in the same direction. More precisely, if there are XSLP_SAMEDAMP successive changes in the same direction for a variable, then the damping factor (D) for the variable will be reset to
$D * XSLP\_DAMPEXPAND + XSLP\_DAMPMAX * (1 - XSLP\_DAMPEXPAND)$

Default value 3

See also Xpress-SLP Solution Process, XSLP_ALGORITHM, XSLP_DAMP, XSLP_DAMPMAX

Affects routines XSLPmaxim, XSLPminim

XSLP_SBROWOFFSET

Description Position of first character of SLP variable name used to create name of SLP lower and upper step bound rows

Type Integer

Note During augmentation, a delta vector is created for each SLP variable. Step bounds are provided for each delta variable, either using explicit bounds, or by using rows to provide lower and upper bounds. If such rows are used, they are created with names derived from the corresponding SLP variable. Customized naming is possible using XSLP_SBLOROWFORMAT and XSLP_SBUPROWFORMAT to define a format and XSLP_SBROWOFFSET to define the first character (counting from zero) of the variable name to be used.

Default value 0

Affects routines XSLPconstruct

See also XSLP_SBLOROWFORMAT, XSLP_SBUPROWFORMAT

XSLP_SBSTART

Description SLP iteration after which step bounds are first applied

Type Integer

Note If step bounds are used, they can be applied for the whole of the SLP optimization process, or started after a number of SLP iterations. In general, it is better not to apply step bounds from the start unless one of the following applies:
(1) the initial estimates are known to be good, and explicit values can be provided for initial step bounds on all variables; or
(2) the problem is unbounded unless all variables are step-bounded.

Default value 8

Affects routines XSLPmaxim, XSLPminim

XSLP_SCALE

Description When to re-scale the SLP problem

Type Integer

Values

0	No re-scaling.
1	Re-scale every SLP iteration up to XSLP_SCALECOUNT iterations after the end of barrier optimization.
2	Re-scale every SLP iteration up to XSLP_SCALECOUNT iterations in total.
3	Re-scale every SLP iteration until primal simplex is automatically invoked.
4	Re-scale every SLP iteration.
5	Re-scale every XSLP_SCALECOUNT SLP iterations.
6	Re-scale every XSLP_SCALECOUNT SLP iterations after the end of barrier optimization.

Note During the SLP optimization, matrix entries can change considerably in magnitude, even when the formulae in the coefficients are not very nonlinear. Re-scaling of the matrix can reduce numerical errors, but may increase the time taken to achieve convergence.

Default value 1

Affects routines XSLPmaxim, XSLPminim

See also XSLP_SCALECOUNT

XSLP_SCALECOUNT

Description Iteration limit used in determining when to re-scale the SLP matrix

Type Integer

Notes If XSLP_SCALE is set to 1 or 2, then XSLP_SCALECOUNT determines the number of iterations (after the end of barrier optimization or in total) in which the matrix is automatically re-scaled.

Default value 0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_SCALE

XSLP_SOLVER

Description First order differentiation mode when using analytical derivatives

Type Integer

Values

−1	automatic selection, based on model characteristics and solver availability
0	use Xpress-SLP (always available)
1	use Knitro if available

Note The presence of Knitro is detected automatically. Knitro can be used to solve any problem loaded into XSLP, independently from how the problem was loaded. XSLP_SOLVER is set to automatic, XSLP will be selected if any SLP specific construct has been loaded (these are ignored if Knitro is selcetd manually).

Default value -1

XSLP_SLPLOG

Description Frequency with which SLP status is printed

Type Integer

Note If XSLP_LOG is set to zero (minimal logging) then a nonzero value for XSLP_SLPLOG defines the frequency (in SLP iterations) when summary information is printed out.

Default value 1

Affects routines XSLPglobal, XSLPmaxim, XSLPminim

See also XSLP_LOG, XSLP_MIPLOG

XSLP_STOPOUTOFRANGE

Description Stop optimization and return error code if internal function argument is out of range

Type Integer

Note If XSLP_STOPOUTOFRANGE is set to 1, then if an internal function receives an argument which is out of its allowable range (for example, *LOG* of a negative number), an error code is set and the optimization is terminated.

Default value 0

Affects routines XSLPevaluatecoef, XSLPevaluateformula XSLPmaxim, XSLPminim

XSLP_THREADS

Description Default number of threads to be used

Type Integer

Note Overall thread control value, used to determine the number of threads used where parallel calculations are possible.

Default value -1 (automatically determined)

Affects routines XSLPmaxim, XSLPmaxim

See also XSLP_CALCTHREADS, XSLP_MULTISTART_THREADS,

XSLP_TIMEPRINT

Description Print additional timings during SLP optimization

Type Integer

Note Date and time printing can be useful for identifying slow procedures during the SLP optimization. Setting XSLP_TIMEPRINT to 1 prints times at additional points during the optimization.

Default value 0

Affects routines XSLPmaxim, XSLPminim

XSLP_THREADSAFEUSERFUNC

Description Defines if user functions are allowed to be called in parallel

Type Integer

Note Date and time printing can be useful for identifying slow procedures during the SLP optimization. Setting XSLP_TIMEPRINT to 1 prints times at additional points during the optimization.

Values

0	user function are not thread safe, and will not be called in parallel
1	user functions are thread safe, and may be called in parallel

Default value 0 (no parallel user function calls)

Affects routines XSLPmaxim, XSLPminim

XSLP_TRACEMASKOPS

Description Controls the information printed for XSLP_TRACEMASK. The order in which the information is printed is determined by the order of bits in XSLP_TRACEMASKOPS.

Type Integer

Values

Bit	Meaning
0	The variable name is used as a mask, not as an exact fit.
1	Use mask to trace rows.
2	Use mask to trace columns.
3	Use mask to trace cascaded SLP variables.
4	Show row / column category.
5	Trace slack values.
6	Trace dual values.
7	Trace row penalty multiplier.
8	Trace variable values (as returned by the linearization).
9	Trace reduced costs.
10	Trace slp value (value used in linearization and cascaded).
11	Trace step bounds.
12	Trace convergence status.
13	Trace line search.

Default value -1: all bits are set

Note The following constants are provided for setting these bits:

Setting bit 0	XSLP_TRACEMASK_GENERALFIT
Setting bit 1	XSLP_TRACEMASK_ROWS
Setting bit 2	XSLP_TRACEMASK_COLS
Setting bit 3	XSLP_TRACEMASK_CASCADE
Setting bit 4	XSLP_TRACEMASK_TYPE
Setting bit 5	XSLP_TRACEMASK_SLACK
Setting bit 6	XSLP_TRACEMASK_DUAL
Setting bit 7	XSLP_TRACEMASK_WEIGHT
Setting bit 8	XSLP_TRACEMASK_SOLUTION
Setting bit 9	XSLP_TRACEMASK_REDUCEDCOST
Setting bit 10	XSLP_TRACEMASK_SLPVALUE
Setting bit 11	XSLP_TRACEMASK_STEPBOUND
Setting bit 12	XSLP_TRACEMASK_CONVERGE
Setting bit 13	XSLP_TRACEMASK_LINESEARCH

Affects routines XSLPminim, XSLPmaxim, XSLPreminim, XSLPremaxim, XSLPnlpoptimize, XSLPglobal

XSLP_UNFINISHEDLIMIT

Description Number of times within one SLP iteration that an unfinished LP optimization will be continued

Type Integer

Note If the optimization of the current linear approximation terminates with an "unfinished" status (for example, because it has reached maximum iterations), Xpress-SLP will attempt to continue using the primal simplex algorithm. This process will be repeated for up to XSLP_UNFINISHEDLIMIT successive LP optimizations within any one SLP iteration. If the limit is reached, Xpress-SLP will terminate with XSLP_STATUS set to XSLP_LPUNFINISHED

Default value 3

Affects routines XSLPglobal, XSLPmaxim, XSLPminim

XSLP_UPDATEOFFSET

Description Position of first character of SLP variable name used to create name of SLP update row

Type Integer

Note During augmentation, one or more delta vectors are created for each SLP variable. The values of these are linked to that of the variable through an *update row* which is created as part of the augmentation procedure. Update rows are created with names derived from the corresponding SLP variable. Customized naming is possible using XSLP_UPDATEFORMAT to define a format and XSLP_UPDATEOFFSET to define the first character (counting from zero) of the variable name to be used.

Default value 0

Affects routines XSLPconstruct

See also XSLP_UPDATEFORMAT

XSLP_VCOUNT

Description Number of SLP iterations over which to measure static objective (3) convergence

Type Integer

Note The static objective (3) convergence criterion does not measure convergence of individual variables, and in fact does not in any way imply that the solution has converged. However, it is sometimes useful to be able to terminate an optimization once the objective function appears to have stabilized. One example is where a set of possible schedules are being evaluated and initially only a good estimate of the likely objective function value is required, to eliminate the worst candidates.
The variation in the objective function is defined as

$$\delta Obj = MAX_{Iter}(Obj) - MIN_{Iter}(Obj)$$

where *Iter* is the `XSLP_VCOUNT` most recent SLP iterations and *Obj* is the corresponding objective function value.
If $ABS(\delta Obj) \leq XSLP\_VTOL\_A$
then the problem has converged on the absolute static objective function (3) criterion.
The static objective function (3) test is applied only if after at least `XSLP_VLIMIT` + `XSLP_SBSTART` SLP iterations have taken place and only if `XSLP_VCOUNT` is at least 2. Where step bounding is being used, this ensures that the test is not applied until after step bounding has been introduced.

Default value 0

Affects routines `XSLPmaxim`, `XSLPminim`

See also `XSLP_SBSTART`, `XSLP_VLIMIT`, `XSLP_VTOL_A`, `XSLP_VTOL_R`

XSLP_VLIMIT

Description Number of SLP iterations after which static objective (3) convergence testing starts

Type Integer

Note The static objective (3) convergence criterion does not measure convergence of individual variables, and in fact does not in any way imply that the solution has converged. However, it is sometimes useful to be able to terminate an optimization once the objective function appears to have stabilized. One example is where a set of possible schedules are being evaluated and initially only a good estimate of the likely objective function value is required, to eliminate the worst candidates.
The variation in the objective function is defined as

$$\delta Obj = MAX_{Iter}(Obj) - MIN_{Iter}(Obj)$$

where *Iter* is the `XSLP_VCOUNT` most recent SLP iterations and *Obj* is the corresponding objective function value.
If $ABS(\delta Obj) \leq XSLP\_VTOL\_A$
then the problem has converged on the absolute static objective function (3) criterion.

The static objective function (3) test is applied only if after at least XSLP_VLIMIT + XSLP_SBSTART SLP iterations have taken place and only if XSLP_VCOUNT is at least 2. Where step bounding is being used, this ensures that the test is not applied until after step bounding has been introduced.

Default value 0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_SBSTART, XSLP_VCOUNT, XSLP_VTOL_A, XSLP_VTOL_R

XSLP_WCOUNT

Description Number of SLP iterations over which to measure the objective for the extended convergence continuation criterion

Type Integer

Note It may happen that all the variables have converged, but some have converged on extended criteria and at least one of these variables is at its step bound. This means that, at least in the linearization, if the variable were to be allowed to move further the objective function would improve. This does not necessarily imply that the same is true of the original problem, but it is still possible that an improved result could be obtained by taking another SLP iteration.

The extended convergence continuation criterion is applied after a converged solution has been found where at least one variable has converged on extended criteria and is at its step bound limit. The extended convergence continuation test measures whether any improvement is being achieved when additional SLP iterations are carried out. If not, then the last converged solution will be restored and the optimization will stop.
For a maximization problem, the improvement in the objective function at the current iteration compared to the objective function at the last converged solution is given by:
$\delta Obj = Obj - LastConvergedObj$
For a minimization problem, the sign is reversed.
If $\delta Obj > XSLP\_WTOL\_A$ and
$\delta Obj > ABS(ConvergedObj) * XSLP\_WTOL\_R$ then the solution is deemed to have a significantly better objective function value than the converged solution.

When a solution is found which converges on extended criteria and with active step bounds, the solution is saved and SLP optimization continues until one of the following:
(1) a new solution is found which converges on some other criterion, in which case the SLP optimization stops with this new solution;
(2) a new solution is found which converges on extended criteria and with active step bounds, and which has a significantly better objective function, in which case this is taken as the new saved solution;
(3) none of the XSLP_WCOUNT most recent SLP iterations has a significantly better objective function than the saved solution, in which case the saved solution is restored and the SLP optimization stops.

If XSLP_WCOUNT is zero, then the extended convergence continuation criterion is disabled.

Default value 0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_WTOL_A, XSLP_WTOL_R

XSLP_XCOUNT

Description Number of SLP iterations over which to measure static objective (1) convergence

Type Integer

Note It may happen that all the variables have converged, but some have converged on extended criteria and at least one of these variables is at its step bound. This means that, at least in the linearization, if the variable were to be allowed to move further the objective function would improve. This does not necessarily imply that the same is true of the original problem, but it is still possible that an improved result could be obtained by taking another SLP iteration. However, if the objective function has already been stable for several SLP iterations, then there is less likelihood of an improved result, and the converged solution can be accepted.

The static objective function (1) test measures the significance of the changes in the objective function over recent SLP iterations. It is applied when all the variables have converged, but some have converged on extended criteria and at least one of these variables is at its step bound. Because all the variables have converged, the solution is already converged but the fact that some variables are at their step bound limit suggests that the objective function could be improved by going further.

The variation in the objective function is defined as
$\delta Obj = MAX_{Iter}(Obj) - MIN_{Iter}(Obj)$
where *Iter* is the XSLP_XCOUNT most recent SLP iterations and *Obj* is the corresponding objective function value.

If $ABS(\delta Obj) \leq XSLP\_XTOL\_A$
then the objective function is deemed to be static according to the absolute static objective function (1) criterion.
If $ABS(\delta Obj) \leq AVG_{Iter}(Obj) * XSLP\_XTOL\_R$
then the objective function is deemed to be static according to the relative static objective function (1) criterion.

The static objective function (1) test is applied only until XSLP_XLIMIT SLP iterations have taken place. After that, if all the variables have converged on strict or extended criteria, the solution is deemed to have converged.

If the objective function passes the relative or absolute static objective function (1) test then the solution is deemed to have converged.

Default value 5

Affects routines XSLPmaxim, XSLPminim

See also XSLP_XLIMIT, XSLP_XTOL_A, XSLP_XTOL_R

XSLP_XLIMIT

Description Number of SLP iterations up to which static objective (1) convergence testing starts

Type Integer

Note It may happen that all the variables have converged, but some have converged on extended criteria and at least one of these variables is at its step bound. This means that, at least in the linearization, if the variable were to be allowed to move further the objective function would

improve. This does not necessarily imply that the same is true of the original problem, but it is still possible that an improved result could be obtained by taking another SLP iteration. However, if the objective function has already been stable for several SLP iterations, then there is less likelihood of an improved result, and the converged solution can be accepted.

The static objective function (1) test measures the significance of the changes in the objective function over recent SLP iterations. It is applied when all the variables have converged, but some have converged on extended criteria and at least one of these variables is at its step bound. Because all the variables have converged, the solution is already converged but the fact that some variables are at their step bound limit suggests that the objective function could be improved by going further.

The variation in the objective function is defined as
$\delta Obj = MAX_{Iter}(Obj) - MIN_{Iter}(Obj)$
where *Iter* is the XSLP_XCOUNT most recent SLP iterations and *Obj* is the corresponding objective function value.

If $ABS(\delta Obj) \leq XSLP\_XTOL\_A$
then the objective function is deemed to be static according to the absolute static objective function (1) criterion.
If $ABS(\delta Obj) \leq AVG_{Iter}(Obj) * XSLP\_XTOL\_R$
then the objective function is deemed to be static according to the relative static objective function (1) criterion.

The static objective function (1) test is applied only until XSLP_XLIMIT SLP iterations have taken place. After that, if all the variables have converged on strict or extended criteria, the solution is deemed to have converged.

If the objective function passes the relative or absolute static objective function (1) test then the solution is deemed to have converged.

Default value 100

Affects routines XSLPmaxim, XSLPminim

See also XSLP_XCOUNT, XSLP_XTOL_A, XSLP_XTOL_R

XSLP_ZEROCRITERION

Description Bitmap determining the behavior of the placeholder deletion procedure

Type Integer

Values

Bit	Meaning
0	(=1) Remove placeholders in nonbasic SLP variables
1	(=2) Remove placeholders in nonbasic delta variables
2	(=4) Remove placeholders in a basic SLP variable if its update row is nonbasic
3	(=8) Remove placeholders in a basic delta variable if its update row is nonbasic and the corresponding SLP variable is nonbasic
4	(=16) Remove placeholders in a basic delta variable if the determining row for the corresponding SLP variable is nonbasic
5	(=32) Print information about zero placeholders

Note For an explanation of deletion of placeholder entries in the matrix see *Management of zero placeholder entries*.

The following constants are provided for setting these bits:

Setting bit 0	XSLP_ZEROCRTIERION_NBSLPVAR
Setting bit 1	XSLP_ZEROCRTIERION_NBDELTA
Setting bit 2	XSLP_ZEROCRTIERION_SLPVARNBUPDATEROW
Setting bit 3	XSLP_ZEROCRTIERION_DELTANBUPSATEROW
Setting bit 4	XSLP_ZEROCRTIERION_DELTANBDRROW
Setting bit 5	XSLP_ZEROCRTIERION_PRINT

Default value 0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_ZEROCRITERIONCOUNT, XSLP_ZEROCRITERIONSTART, *Management of zero placeholder entries*

XSLP_ZEROCRITERIONCOUNT

Description Number of consecutive times a placeholder entry is zero before being considered for deletion

Type Integer

Note For an explanation of deletion of placeholder entries in the matrix see *Management of zero placeholder entries*.

Default value 0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_ZEROCRITERION, XSLP_ZEROCRITERIONSTART, *Management of zero placeholder entries*

XSLP_ZEROCRITERIONSTART

Description SLP iteration at which criteria for deletion of placeholder entries are first activated.

Type Integer

Note For an explanation of deletion of placeholder entries in the matrix see *Management of zero placeholder entries*.

Default value 0

Affects routines XSLPmaxim, XSLPminim

See also XSLP_ZEROCRITERION, XSLP_ZEROCRITERIONCOUNT, *Management of zero placeholder entries*

20.3 String control parameters

XSLP_CVNAME

Description Name of the set of character variables to be used

Type String

Notes This variable may be required for input from a file using XSLPreadprob if there is more than one set of character variables in the file. If no name is set, then the first set of character variables will be used, and the name will be set accordingly.
This variable may also be required for output using XSLPwriteprob where character variables are included in the problem. If it is not set, then a default name will be used.

Set by routines XSLPreadprob

Default value none

Affects routines XSLPreadprob, XSLPwriteprob

See also XSLP_IVNAME, XSLP_SBNAME, XSLP_TOLNAME

XSLP_DELTAFORMAT

Description Formatting string for creation of names for SLP delta vectors

Type String

Note This control can be used to create a specific naming structure for delta vectors. The structure follows the normal C-style printf form, and can contain printing characters plus one %s string. This will be replaced by sequential characters from the name of the variable starting at position XSLP_DELTAOFFSET.

Default value pD_%s
where p is a unique prefix for names in the current problem

Affects routines XSLPconstruct

See also XSLP_DELTAOFFSET

XSLP_ITERFALLBACKOPS

Description Alternative LP level control values for numerically challengeing problems

Type String

Notes When set, this control provides alternative ways of solving a linearization called adaptive iteration solves. This can be useful for numerically challenging problems that either solve to a non-satisfactory accuracy (relative to XSLP_FEASTOLTARGET) with the default solves, or that can incorrectly report infeasibility or unboundedness. In such cases, the solve will try the

controls listed by XSLP_ITERFALLBACKOPS until a satisfactory solution is found or all options are exhausted.

The individual controls for each solve are separated by a comma (','), while the set of controls for an attepmt by a colon (':') . Example: 'XPRS_DEFAULTALG=3 : XPRS_BARORDER = 2, XPRS_PRESOVLE = 0' will try primal in one solve, and the homogenous barrier with presolve turned off in an other. Optimizer flags are not inherited by the solve, so use XPRS_DEFAULTALG for selecting an LP solver to use.

The resulting LP solves are carried out in a parallel manner, using XSLP_MULTISTART_THREADS number of threads.

The result of the adaptive solves are always deterministic.

Once a satisfactory solution is found, remaining solves are progressed only as far as necessary to guarantee determinism, so it is beneficial to list more promising control sets first.

Default value none

Affects routines XSLPminim, XSLPmaxim, XSLPnlpoptimize

See also XSLP_FEASTOLTARGET,

XSLP_IVNAME

Description Name of the set of initial values to be used

Type String

Notes This variable may be required for input from a file using XSLPreadprob if there is more than one set of initial values in the file. If no name is set, then the first set of initial values will be used, and the name will be set accordingly.
This variable may also be required for output using XSLPwriteprob where initial values are included in the problem. If it is not set, then a default name will be used.

Set by routines XSLPreadprob

Default value none

Affects routines XSLPreadprob, XSLPwriteprob

See also XSLP_CVNAME, XSLP_SBNAME, XSLP_TOLNAME

XSLP_MINUSDELTAFORMAT

Description Formatting string for creation of names for SLP negative penalty delta vectors

Type String

Note This control can be used to create a specific naming structure for negative penalty delta vectors. The structure follows the normal C-style printf form, and can contain printing characters plus one %s string. This will be replaced by sequential characters from the name of the variable starting at position XSLP_DELTAOFFSET.

Default value pD-%s
where p is a unique prefix for names in the current problem

Affects routines XSLPconstruct

See also XSLP_DELTAOFFSET

XSLP_MINUSERRORFORMAT

Description Formatting string for creation of names for SLP negative penalty error vectors

Type String

Note This control can be used to create a specific naming structure for negative penalty error vectors. The structure follows the normal C-style printf form, and can contain printing characters plus one %s string. This will be replaced by sequential characters from the name of the variable starting at position XSLP_ERROROFFSET.

Default value pE-%s
where p is a unique prefix for names in the current problem

Affects routines XSLPconstruct

See also XSLP_ERROROFFSET

XSLP_PENALTYCOLFORMAT

Description Formatting string for creation of the names of the SLP penalty transfer vectors

Type String

Note This control can be used to create a specific naming structure for the penalty transfer vectors which transfer penalty costs into the objective. The structure follows the normal C-style printf form, and can contain printing characters plus one %s string. This will be replaced by "DELT" for the penalty delta transfer vector and "ERR" for the penalty error transfer vector.

Default value pPC_%s
where p is a unique prefix for names in the current problem

Affects routines XSLPconstruct

XSLP_PENALTYROWFORMAT

Description Formatting string for creation of the names of the SLP penalty rows

Type String

Note This control can be used to create a specific naming structure for the penalty rows which total the penalty costs for the objective. The structure follows the normal C-style printf form, and can contain printing characters plus one %s string. This will be replaced by "DELT" for the penalty delta row and "ERR" for the penalty error row.

Default value pPR_%s
where p is a unique prefix for names in the current problem

Affects routines XSLPconstruct

XSLP_PLUSDELTAFORMAT

Description Formatting string for creation of names for SLP positive penalty delta vectors

Type String

Note This control can be used to create a specific naming structure for positive penalty delta vectors. The structure follows the normal C-style printf form, and can contain printing characters plus one %s string. This will be replaced by sequential characters from the name of the variable starting at position XSLP_DELTAOFFSET.

Default value pD+%s
where p is a unique prefix for names in the current problem

Affects routines XSLPconstruct

See also XSLP_DELTAOFFSET

XSLP_PLUSERRORFORMAT

Description Formatting string for creation of names for SLP positive penalty error vectors

Type String

Note This control can be used to create a specific naming structure for positive penalty error vectors. The structure follows the normal C-style printf form, and can contain printing characters plus one %s string. This will be replaced by sequential characters from the name of the variable starting at position XSLP_ERROROFFSET.

Default value pE+%s
where p is a unique prefix for names in the current problem

Affects routines XSLPconstruct

See also XSLP_ERROROFFSET

XSLP_SBLOROWFORMAT

Description Formatting string for creation of names for SLP lower step bound rows

Type String

Note This control can be used to create a specific naming structure for lower limits on step bounds modeled as rows. The structure follows the normal C-style printf form, and can contain printing characters plus one %s string. This will be replaced by sequential characters from the name of the variable starting at position XSLP_SBROWOFFSET.

Default value pSB-%s
where p is a unique prefix for names in the current problem

Affects routines XSLPconstruct

See also XSLP_SBROWOFFSET

XSLP_SBNAME

Description Name of the set of initial step bounds to be used

Type String

Notes This variable may be required for input from a file using XSLPreadprob if there is more than one set of initial step bounds in the file. If no name is set, then the first set of initial step bounds will be used, and the name will be set accordingly.
This variable may also be required for output using XSLPwriteprob where initial step bounds are included in the problem. If it is not set, then a default name will be used.

Set by routines XSLPreadprob

Default value none

Affects routines XSLPreadprob, XSLPwriteprob

See also XSLP_CVNAME, XSLP_IVNAME, XSLP_TOLNAME

XSLP_SBUPROWFORMAT

Description Formatting string for creation of names for SLP upper step bound rows

Type String

Note This control can be used to create a specific naming structure for upper limits on step bounds modeled as rows. The structure follows the normal C-style printf form, and can contain printing characters plus one %s string. This will be replaced by sequential characters from the name of the variable starting at position XSLP_SBROWOFFSET.

Default value pSB+%s
where p is a unique prefix for names in the current problem

Affects routines XSLPconstruct

See also XSLP_SBROWOFFSET

XSLP_TOLNAME

Description Name of the set of tolerance sets to be used

Type String

Notes This variable may be required for input from a file using XSLPreadprob if there is more than one set of tolerance sets in the file. If no name is set, then the first set of tolerance sets will be used, and the name will be set accordingly.
This variable may also be required for output using XSLPwriteprob where tolerance sets are included in the problem. If it is not set, then a default name will be used.

Set by routines XSLPreadprob

Default value none

Affects routines XSLPreadprob, XSLPwriteprob

See also XSLP_CVNAME, XSLP_IVNAME, XSLP_SBNAME

XSLP_TRACEMASK

Description Mask of variable or row names that are to be traced through the SLP iterates

Type String

Notes If the mask is nonempty, variables and rows matching the mask are listed after each SLP iteration and each cascade, allowing for a convenient means to observe how certain variables change through the iterates. This feasture is provided for tuning and model debugging purposes. The actual information printed is controlled by XSLP_TRACEMASKOPS.
The string in the tracemask may contain several variable or row names, separated by a whitespace. Wildcards may also be used.

Default value none: no tracing

Affects routines XSLPminim, XSLPmaxim, XSLPreminim, XSLPremaxim, XSLPnlpoptimize, XSLPglobal

See also XSLP_TRACEMASKOPS

XSLP_UPDATEFORMAT

Description Formatting string for creation of names for SLP update rows

Type String

Note This control can be used to create a specific naming structure for update rows. The structure follows the normal C-style printf form, and can contain printing characters plus one %s string. This will be replaced by sequential characters from the name of the variable starting at position XSLP_UPDATEOFFSET.

Default value pU_%s
where p is a unique prefix for names in the current problem

Affects routines XSLPconstruct

See also XSLP_UPDATEOFFSET

20.4 Knitro controls

All Knitro controls are available with an 'X' pre-tag. For example the Knitro integer control 'KTR_PARAM_ALGORITHM' can be set using XSLPsetintcontrol using the control ID defined as 'XKTR_PARAM_ALGORITHM'. Please refer to the Xpress Knitro manual for the description of the Knitro controls.

CHAPTER 21

Library functions and the programming interface

21.1 Counting

All Xpress NonLinear entities are numbered from 1. The 0^{th} item is defined, and is an empty entity of the appropriate type. Therefore, whenever an Xpress NonLinear function returns a zero value, it means that there is no data of that type.

In parsed and unparsed function arrays, the indices always count from 1. This includes types `XSLP_VAR` and `XSLP_CONSTRAINT`: the index is the matrix column or row index +1.

Note that for *input* of function arrays, types `XSLP_COL` and `XSLP_ROW` can be used, but will be converted into standard `XSLP_VAR` or `XSLP_CONSTRAINT` references. When a function array is returned from Xpress NonLinear, the `XSLP_VAR` or `XSLP_CONSTRAINT` type will always be used.

21.2 The Xpress NonLinear problem pointer

Xpress NonLinear uses the same concept as the Optimizer library, with a "pointer to a problem". The optimizer problem must be initialized first in the normal way. Then the corresponding Xpress NonLinear problem must be initialized, including a pointer to the underlying optimizer problem. For example:

```
{
...
XPRSprob prob=NULL;
XSLPprob SLPprob=NULL;

XPRSinit("");
XSLPinit();
XPRScreateprob(&prob);
XSLPcreateprob(&SLPprob,&prob);
...
}
```

At the end of the program, the Xpress NonLinear problem should be destroyed. You are responsible for destroying the underlying XPRSprob linear problem afterwards. For example:

```
{
...
  XSLPdestroyprob(SLPprob);
  XPRSdestroyprob(prob);
  XSLPfree();
  XPRSfree();
  ...
}
```

The following functions are provided to manage Xpress NonLinear problems. See the documentation below on the individual functions for more details.

`XSLPcopycontrols(XSLPprob prob1, XSLPprob prob2)`
Copy the settings of control variables

`XSLPcopycallbacks(XSLPprob prob1, XSLPprob prob2)`
Copy the callback settings

`XSLPcopyprob(XSLPprob prob1, XSLPprob prob2, char *ProbName)`
Copy a problem completely

`XSLPcreateprob(XSLPprob *prob1, XPRSprob *prob2)`
Create an Xpress NonLinear problem

`XSLPdestroyprob(XSLPprob prob1)`
Delete an Xpress NonLinear problem from memory

`XSLPrestore(XSLPprob prob1)`
Restore Xpress NonLinear data structures from file

`XSLPsave(XSLPprob prob1)`
Save Xpress NonLinear data structures to file

21.3 The `XSLPload...` functions

The `XSLPload...` functions can be used to load an Xpress NonLinear problem directly into the Xpress data structures. Because there are so many additional items which can be loaded apart from the basic (linear) matrix, the loading process is divided into several functions.

The best practice is to load the linear part of the problem irst, using the normal Optimizer Library functions `XPRSloadlp` or `XPRSloadglobal`. Then the appropriate parts of the Xpress NonLinear problem can be loaded. After all the `XSLPload...` functions have been called, `XSLPconstruct` should be called to create the SLP matrix and data structures. If `XSLPconstruct` is not invoked before a call to one of the Xpress NonLinear optimization routines, then it will be called by the optimization routine itself.

All of these functions initialize their data areas. Therefore, if a second call is made to the same function for the same problem, the previous data will be deleted. If you want to include additional data of the same type, then use the corresponding `XSLPadd...` function.

It is possible to remove parts of the SLP strcutures with the various `XSLPdel` functions, and `XSLPunconstruct` can also be used to remove the augmentation.

Xpress NonLinear is compatible with the Xpress quadratic programming optimizer. `XPRSloadqp` and `XPRSloadqglobal` can be used to load quadratic problems (or quadratically constrained problmes using `XPRSloadqcqp` and `XPRSloadqcqpglobal`). The quadratic objective will be optimized using the Xpress quadratic optimizer; the nonlinear constraints will be handled with the normal SLP procedures. Please note, that this separation is only useful for a convex quadratic objective and convex quadratic inequality constraints. All nonconvex quadratic matrices should be handled as SLP strctures.

For a description on when it's more beneficial to use the XPRS library to solve QP or QCQP problems, please see *Selecting the right algorithm for a nonlinear problem - when to use the XPRS library instead of XSLP*.

21.4 Library functions

A large number of routines are available for Library users of Xpress NonLinear, 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. Library users have access to a set of functions providing advanced control over their program's interaction with the SLP module and catering for more complicated problem development.

XSLPaddcoefs

Purpose

Add non-linear coefficients to the SLP problem

Synopsis

```
int XPRS_CC XSLPaddcoefs(XSLPprob Prob, int nSLPCoef, int *RowIndex, int
      *ColIndex, double *Factor, int *FormulaStart, int Parsed, int *Type,
      double *Value);
```

Arguments

`Prob`	The current SLP problem.
`nSLPCoef`	Number of non-linear coefficients to be added.
`RowIndex`	Integer array holding index of row for the coefficient.
`ColIndex`	Integer array holding index of column for the coefficient.
`Factor`	Double array holding factor by which formula is scaled. If this is `NULL`, then a value of 1.0 will be used.
`FormulaStart`	Integer array of length `nSLPCoef+1` holding the start position in the arrays `Type` and `Value` of the formula for the coefficients. `FormulaStart[nSLPCoef]` should be set to the next position after the end of the last formula.
`Parsed`	Integer indicating whether the token arrays are formatted as internal unparsed (`Parsed`=0) or internal parsed reverse Polish (`Parsed`=1).
`Type`	Array of token types providing the formula for each coefficient.
`Value`	Array of values corresponding to the types in `Type`.

Example

Assume that the rows and columns of `Prob` are named `Row1`, `Row2` ..., `Col1`, `Col2` ... The following example adds coefficients representing:
`Col2 * Col3 + Col6 * Col2^2` into `Row1` and
`Col2 ^ 2` into `Row3`.

```
int RowIndex[3], ColIndex[3], FormulaStart[4], Type[8];
int n, nSLPCoef;
double Value[8];

RowIndex[0] = 1; ColIndex[0] = 2;
RowIndex[1] = 1; ColIndex[1] = 6;
RowIndex[2] = 3; ColIndex[2] = 2;

n = nSLPCoef = 0;
FormulaStart[nSLPCoef++] = n;
Type[n] = XSLP_COL; Value[n++] = 3;
Type[n++] = XSLP_EOF;

FormulaStart[nSLPCoef++] = n;
Type[n] = XSLP_COL; Value[n++] = 2;
Type[n] = XSLP_COL; Value[n++] = 2;
Type[n] = XSLP_OP;  Value[n++] = XSLP_MULTIPLY;
Type[n++] = XSLP_EOF;

FormulaStart[nSLPCoef++] = n;
Type[n] = XSLP_COL; Value[n++] = 2;
Type[n++] = XSLP_EOF;
```

```
FormulaStart[nSLPCoef] = n;

XSLPaddcoefs(Prob, nSLPCoef, RowIndex, ColIndex,
             NULL, FormulaStart, 1, Type, Value);
```

The first coefficient in `Row1` is in `Col2` and has the formula `Col3`, so it represents `Col2 * Col3`.

The second coefficient in `Row1` is in `Col6` and has the formula `Col2 * Col2` so it represents `Col6 * Col2^2`. The formulae are described as *parsed* (`Parsed`=1), so the formula is written as
`Col2 Col2 *`
rather than the unparsed form
`Col2 * Col2`

The last coefficient, in `Row3`, is in `Col2` and has the formula `Col2`, so it represents `Col2 * Col2`.

Further information

The j^{th} coefficient is made up of two parts: `Factor` and `Formula`. `Factor` is a constant multiplier, which can be provided in the `Factor` array. If Xpress NonLinear can identify a constant factor in `Formula`, then it will use that as well, to minimize the size of the formula which has to be calculated. `Formula` is made up of a list of tokens in `Type` and `Value` starting at `FormulaStart[j]`. The tokens follow the rules for parsed or unparsed formulae as indicated by the setting of `Parsed`. The formula must be terminated with an `XSLP_EOF` token. If several coefficients share the same formula, they can have the same value in `FormulaStart`. For possible token types and values see the chapter on "Formula Parsing".

The `XSLPadd...` functions load additional items into the SLP problem. The corresponding `XSLPload...` functions delete any existing items first.

The behaviour for existing coefficients is additive: the formula defined in the parameters are added to any existing formula coefficients. However, due to performance considerations, such duplications should be avoided when possible.

Related topics

`XSLPchgcoef`, `XSLPchgccoef`, `XSLPdelcoefs`, `XSLPgetcoefformula`, `XSLPgetccoef`, `XSLPloadcoefs`

XSLPadddfs

Purpose

Add a set of distribution factors

Synopsis

```
int XSLP_CC XSLPadddfs(XSLPprob Prob, int nDF, const int *ColIndex, const
      int *RowIndex, const double *Value)
```

Arguments

`Prob`	The current SLP problem.
`nDF`	The number of distribution factors.
`ColIndex`	Array of indices of columns whose distribution factor is to be changed.
`RowIndex`	Array of indices of the rows where each distribution factor applies.
`Value`	Array of double precision variables holding the new values of the distribution factors.

Example

The following example adds distribution factors as follows:
column 282 in row 134 = 0.1
column 282 in row 136 = 0.15
column 285 in row 133 = 1.0.

```
int ColIndex[3], RowIndex[3];
double Value[3];
ColIndex[0] = 282;  RowIndex[0] = 134; Value[0] = 0.1;
ColIndex[1] = 282;  RowIndex[1] = 136; Value[1] = 0.15;
ColIndex[2] = 285;  RowIndex[2] = 133; Value[2] = 1.0;
XSLPadddfs(prob,3,ColIndex,RowIndex,Value);
```

Further information

The *distribution factor* of a column in a row is the matrix coefficient of the corresponding delta vector in the row. Distribution factors are used in conventional recursion models, and are essentially normalized first-order derivatives. Xpress-SLP can accept distribution factors instead of initial values, provided that the values of the variables involved can all be calculated after optimization using determining rows, or by a callback.

The `XSLPadd...` functions load additional items into the SLP problem. The corresponding `XSLPload...` functions delete any existing items first.

Related topics

`XSLPchgdf`, `XSLPgetdf`, `XSLPloaddfs`

XSLPaddtolsets

Purpose

Add sets of standard tolerance values to an SLP problem

Synopsis

```
int XPRS_CC XSLPaddtolsets(XSLPprob Prob, int nSLPTol, double *SLPTol);
```

Arguments

`Prob`	The current SLP problem.
`nSLPTol`	The number of tolerance sets to be added.
`SLPTol`	Double array of (`nSLPTol * 9`) items containing the 9 tolerance values for each set in order.

Example

The following example creates two tolerance sets: the first has values of 0.005 for all tolerances; the second has values of 0.001 for relative tolerances (numbers 2,4,6,8), values of 0.01 for absolute tolerances (numbers 1,3,5,7) and zero for the closure tolerance (number 0).

```
double SLPTol[18];
for (i=0;i<9;i++) SLPTol[i] = 0.005;
SLPTol[9] = 0;
for (i=10;i<18;i=i+2) SLPTol[i] = 0.01;
for (i=11;i<18;i=i+2) SLPTol[i] = 0.001;
XSLPaddtolsets(Prob, 2, SLPTol);
```

Further information

A tolerance set is an array of 9 values containing the following tolerances:

Entry / Bit	Tolerance	XSLP constant	XSLP bit constant
0	Closure tolerance (TC)	`XSLP_TOLSET_TC`	`XSLP_TOLSETBIT_TC`
1	Absolute delta tolerance (TA)	`XSLP_TOLSET_TA`	`XSLP_TOLSETBIT_TA`
2	Relative delta tolerance (RA)	`XSLP_TOLSET_RA`	`XSLP_TOLSETBIT_RA`
3	Absolute coefficient tolerance (TM)	`XSLP_TOLSET_TM`	`XSLP_TOLSETBIT_TM`
4	Relative coefficient tolerance (RM)	`XSLP_TOLSET_RM`	`XSLP_TOLSETBIT_RM`
5	Absolute impact tolerance (TI)	`XSLP_TOLSET_TI`	`XSLP_TOLSETBIT_TI`
6	Relative impact tolerance (RI)	`XSLP_TOLSET_RI`	`XSLP_TOLSETBIT_RI`
7	Absolute slack tolerance (TS)	`XSLP_TOLSET_TS`	`XSLP_TOLSETBIT_TS`
8	Relative slack tolerance (RS)	`XSLP_TOLSET_RS`	`XSLP_TOLSETBIT_RS`

The XSLP_TOLSET constants can be used to access the corresponding entry in the value arrays, while the XSLP_TOLSETBIT constants are used to set or retrieve which tolerance values are used for a given SLP variable.

Once created, a tolerance set can be used to set the tolerances for any SLP variable.

If a tolerance value is zero, then the default tolerance will be used instead. To force the use of a tolerance, use the `XSLPchgtolset` function and set the `Status` variable appropriately.

See the section Convergence criteria for a fuller description of tolerances and their uses.

The `XSLPadd...` functions load additional items into the SLP problem. The corresponding `XSLPload...` functions delete any existing items first.

Related topics

`XSLPchgtolset`, `XSLPdeltolsets`, `XSLPgettolset`, `XSLPloadtolsets`

XSLPadduserfunction

Purpose

Add user function definitions to an SLP problem.

Synopsis

```
int XPRS_CC XSLPadduserfunction(XSLPprob Prob, const char * FunctionName,
      int FunctionType, int nInput, int nOutput, int Options, void *
      FunctionPointer,void * UserContext, int * FunctionTokenId);
```

Arguments

`Prob` The current SLP problem.

`FunctionName` The name of the function as it appears in text formula expressions.

`FunctionType` The type of the user function, one of

1	(XSLP_USERFUNCTION_MAP)	function takes double, returns double.
2	(XSLP_USERFUNCTION_VECMAP)	function takes double array, returns double.
3	(XSLP_USERFUNCTION_MULTIMAP)	function takes double array, returns double array.
4	(XSLP_USERFUNCTION_MAPDELTA)	function takes double, returns double and delta.
5	(XSLP_USERFUNCTION_VECMAPDELTA)	function takes double array, returns double and deltas.
6	(XSLP_USERFUNCTION_MULTIMAPDELTA)	function takes double array, returns double array and deltas.

`nInput` Number of arguments the user function takes.

`nOutput` Number of return arguments for the function.

`Options` Options as a bitmap to the user function

`XSLP_INSTANCEFUNCTION` always istantiate the function.

`FunctionPointer` Pointer of the suer function to call.

`UserContext` Context pointer to provide the user function with.

`FunctionTokenId` The token id of the user function added, to be used in the Value array when defining formulas and using with XSLP_USERFUNCTION.

Further information

The function declarations expected for the user functions are defined by the `FunctionType` argument.

The function of type `XSLP_USERFUNCTION_MAP` expects a function in the form of 'double XPRS_CC F(double Value, void *Context)'.

The function of type `XSLP_USERFUNCTION_VECMAP` expects a function in the form of 'double XPRS_CC F(double *Value, void *Context)'.

The function of type `XSLP_USERFUNCTION_MULTIMAP` expects a function in the form of 'int XPRS_CC F(double *Value, double *Out, void *Context)'.

The function of type `XSLP_USERFUNCTION_MAPDELTA` expects a function in the form of 'int XPRS_CC F(double Value, double Delta, double *Evaluation, double *Partial, void *Context)'.

The function of type `XSLP_USERFUNCTION_VECMAPDELTA` expects a function in the form of 'int XPRS_CC F(double *Value, double *Deltas, double *Evaluation, double *Partials, void *Context)'.

The function of type `XSLP_USERFUNCTION_MULTIMAPDELTA` expects a function in the form of 'int XPRS_CC F(double *Value, double *Deltas, double *Out, void *Context)'.

Related topics

Function Declaration in Xpress NonLinear, `XSLPdeluserfunction`, `XSLPimportlibfunc`

XSLPaddvars

Purpose

Add SLP variables defined as matrix columns to an SLP problem

Synopsis

```
int XPRS_CC XSLPaddvars(XSLPprob Prob, int nSLPVar, int *ColIndex, int
      *VarType, int *DetRow, int *SeqNum, int *TolIndex, double *InitValue,
      double *StepBound);
```

Arguments

`Prob`	The current SLP problem.
`nSLPVar`	The number of SLP variables to be added.
`ColIndex`	Integer array holding the index of the matrix column corresponding to each SLP variable.
`VarType`	Bitmap giving information about the SLP variable as follows: `Bit 1` Variable has a delta vector; `Bit 2` Variable has an initial value; `Bit 14` Variable is the reserved "=" column; May be `NULL` if not required.
`DetRow`	Integer array holding the index of the determining row for each SLP variable (a negative value means there is no determining row) May be `NULL` if not required.
`SeqNum`	Integer array holding the index sequence number for cascading for each SLP variable (a zero value means there is no pre-defined order for this variable) May be `NULL` if not required.
`TolIndex`	Integer array holding the index of the tolerance set for each SLP variable (a zero value means the default tolerances are used) May be `NULL` if not required.
`InitValue`	Double array holding the initial value for each SLP variable (use the `VarType` bit map to indicate if a value is being provided) May be `NULL` if not required.
`StepBound`	Double array holding the initial step bound size for each SLP variable (a zero value means that no initial step bound size has been specified). If a value of `XPRS_PLUSINFINITY` is used for a value in `StepBound`, the delta will never have step bounds applied, and will almost always be regarded as converged. May be `NULL` if not required.

Example

The following example loads two SLP variables into the problem. They correspond to columns 23 and 25 of the underlying LP problem. Column 25 has an initial value of 1.42; column 23 has no specific initial value

```
int ColIndex[2], VarType[2];
double InitValue[2];

ColIndex[0] = 23; VarType[0] = 0;
ColIndex[1] = 25; Vartype[1] = 2; InitValue[1] = 1.42;

XSLPaddvars(Prob, 2, ColIndex, VarType, NULL, NULL,
            NULL, InitValue, NULL);
```

`InitValue` is not set for the first variable, because it is not used (`VarType` = 0). Bit 1 of `VarType` is set for the second variable to indicate that the initial value has been set.

The arrays for determining rows, sequence numbers, tolerance sets and step bounds are not used at all, and so have been passed to the function as `NULL`.

Further information

The `XSLPadd...` functions load additional items into the SLP problem. The corresponding `XSLPload...` functions delete any existing items first.

Related topics

`XSLPchgvar`, `XSLPdelvars`, `XSLPgetvar`, `XSLPloadvars`

XSLPcalcslacks

Purpose

Calculate the slack values for the provided solution in the non-linear problem

Synopsis

```
int XPRS_CC XSLPcalcslacks(XSLPprob Prob, const double * dSol, double *
      Slacks);
```

Arguments

`Prob`	The current SLP problem.
`dSol`	The solution for which the slacks are requested for.
`Slacks`	Vector of length NROWS to return the slack in.

Related topics

`XSLPvalidate`, `XSLPvalidaterow`

XSLPcascade

Purpose

Re-calculate consistent values for SLP variables based on the current values of the remaining variables.

Synopsis

```
int XPRS_CC XSLPcascade(XSLPprob Prob);
```

Argument

`Prob` The current SLP problem.

Example

The following example changes the solution value for column 91, and then re-calculates the values of those dependent on it.

```
int ColNum;
double Value;

ColNum = 91;
XSLPgetvar(Prob, ColNum, NULL, NULL, NULL, NULL,
           NULL, NULL, &Value, NULL, NULL, NULL,
           NULL, NULL, NULL, NULL, NULL);

Value = Value + 1.42;
XSLPchgvar(Prob, ColNum, NULL, NULL, NULL, NULL,
           NULL, NULL, &Value, NULL, NULL, NULL,
           NULL);
XSLPcascade(Prob);
```

`XSLPgetvar` and `XSLPchgvar` are being used to get and change the current value of a single variable.

Provided no other values have been changed since the last execution of `XSLPcascade`, values will be changed only for variables which depend on column 91.

Further information

See the section on cascading for an extended discussion of the types of cascading which can be performed.

`XSLPcascade` is called automatically during the SLP iteration process and so it is not normally necessary to perform an explicit cascade calculation.

The variables are re-calculated in accordance with the order generated by `XSLPcascadeorder`.

Related topics

`XSLPcascadeorder`, `XSLP_CASCADE`, `XSLP_CASCADENLIMIT`, `XSLP_CASCADETOL_PA`, `XSLP_CASCADETOL_PR`

XSLPcascadeorder

Purpose

Establish a re-calculation sequence for SLP variables with determining rows.

Synopsis

```
int XPRS_CC XSLPcascadeorder(XSLPprob Prob);
```

Argument

`Prob` The current SLP problem.

Example

Assuming that all variables are SLP variables, the following example sets default values for the variables, creates the re-calculation order and then calls `XSLPcascade` to calculate consistent values for the dependent variables.

```
int ColNum;
for (ColNum=1;ColNum<=nCol;ColNum++)
  XSLPchgvar(Prob, ColNum, NULL, NULL, NULL, NULL,
             NULL, NULL, &DefaultValue[ColNum], NULL, NULL, NULL,
             NULL);
XSLPcascadeorder(Prob);
XSLPcascade(Prob);
```

Further information

`XSLPcascadeorder` is called automatically at the start of the SLP iteration process and so it is not normally necessary to perform an explicit cascade ordering.

Related topics

`XSLPcascade`

XSLPchgcascadenlimit

Purpose

Set a variable specific cascade iteration limit

Synopsis

```
int XPRS_CC XSLPchgcascadenlimit(XSLPprob Prob, int iCol, int
      CascadeNLimit);
```

Arguments

`Prob`	The current SLP problem.
`iCol`	The index of the column corresponding to the SLP variable for which the cascading limit is to be imposed.
`CascadeNLimit`	The new cascading iteration limit.

Further information

A value set by this function will overwrite the value of `XSLP_CASCADENLIMIT` for this variable. To remove any previous value set by this function, use an iteration limit of 0.

Related topics

`XSLPcascadeorder`, `XSLP_CASCADE`, `XSLP_CASCADENLIMIT`, `XSLP_CASCADETOL_PA`, `XSLP_CASCADETOL_PR`

XSLPchgccoef

Purpose

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

Synopsis

```
int XPRS_CC XSLPchgccoef(XSLPprob Prob, int RowIndex, int ColIndex, double
      *Factor, char *Formula);
```

Arguments

`Prob`	The current SLP problem.
`RowIndex`	The index of the matrix row for the coefficient.
`ColIndex`	The index of the matrix column for the coefficient.
`Factor`	Address of a double precision variable holding the constant multiplier for the formula. If `Factor` is `NULL`, a value of 1.0 will be used.
`Formula`	Character string holding the formula with the tokens separated by spaces.

Example

Assuming that the columns of the matrix are named `Col1`, `Col2`, etc, the following example puts the formula `2.5*sin(Col1)` into the coefficient in row 1, column 3.

```
char *Formula="sin ( Col1 )";
double Factor;

Factor = 2.5;
XSLPchgccoef(Prob, 1, 3, &Factor, Formula);
```

Note that all the tokens in the formula (including mathematical operators and separators) are separated by one or more spaces.

Further information

If the coefficient already exists as a constant or formula, it will be changed into the new coefficient. If it does not exist, it will be added to the problem.

A coefficient is made up of two parts: `Factor` and `Formula`. `Factor` is a constant multiplier which can be provided in the `Factor` variable. If Xpress NonLinear can identify a constant factor in the Formula, then it will use that as well, to minimize the size of the formula which has to be calculated.

This function can only be used if all the operands in the formula can be correctly identified as constants, existing columns, character variables or functions. Therefore, if a formula refers to a new column, that new item must be added to the Xpress NonLinear problem first.

Related topics

`XSLPaddcoefs`, `XSLPdelcoefs`, `XSLPchgcoef`, `XSLPgetcoefformula`, `XSLPloadcoefs`

XSLPchgcoef

Purpose

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

Synopsis

```
int XPRS_CC XSLPchgcoef(XSLPprob Prob, int RowIndex, int ColIndex, double
      *Factor, int Parsed, int *Type, double *Value);
```

Arguments

`Prob`	The current SLP problem.
`RowIndex`	The index of the matrix row for the coefficient.
`ColIndex`	The index of the matrix column for the coefficient.
`Factor`	Address of a double precision variable holding the constant multiplier for the formula. If `Factor` is `NULL`, a value of 1.0 will be used.
`Parsed`	Integer indicating the whether the token arrays are formatted as internal unparsed (`Parsed`=0) or internal parsed reverse Polish (`Parsed`=1).
`Type`	Array of token types providing the description and formula for each item.
`Value`	Array of values corresponding to the types in `Type`.

Example

Assuming that the columns of the matrix are named `Col1`, `Col2`, etc, the following example puts the formula `2.5*sin(Col1)` into the coefficient in row 1, column 3.

```
int n, iSin, Type[4];
double Value[4];
double Factor;

XSLPgetindex(Prob, XSLP_INTERNALFUNCNAMESNOCASE,
             "sin", &iSin);

n = 0;
Type[n] = XSLP_IFUN; Value[n++] = iSin;
Type[n] = XSLP_VAR;  Value[n++] = 1;
Type[n++] = XSLP_RB;
Type[n++] = XSLP_EOF;

Factor = 2.5;
XSLPchgcoef(Prob, 1, 3, &Factor, 0, Type, Value);
```

`XSLPgetindex` is used to retrieve the index for the internal function `sin`. The "nocase" version matches the function name regardless of the (upper or lower) case of the name.

Token type `XSLP_VAR` always counts from 1, so `Col1` is always 1.

The formula is written in unparsed form (`Parsed` = 0) and so it is provided as tokens in the same order as they would appear if the formula were written in character form.

Further information

If the coefficient already exists as a constant or formula, it will be changed into the new coefficient. If it does not exist, it will be added to the problem.

A coefficient is made up of two parts: `Factor` and `Formula`. `Factor` is a constant multiplier which can be provided in the `Factor` variable. If Xpress NonLinear can identify a constant factor in the Formula, then it will use that as well, to minimize the size of the formula which has to be calculated.

Related topics

`XSLPaddcoefs`, `XSLPchgccoef`, `XSLPdelcoefs`, `XSLPgetcoefformula`, `XSLPloadcoefs`

XSLPchgdeltatype

Purpose

Changes the type of the delta assigned to a nonlinear variable

Synopsis

```
int XPRS_CC XSLPchgdeltatype(XSLPprob Prob, int nVar, int Vars[], int
      DeltaTypes[], double Values[]);
```

Arguments

`Prob` The current SLP problem.

`nVar` The number of SLP variables to change the delta type for.

`Vars` Indices of the variables to change the deltas for.

`DeltaTypes` Type if the delta variable:

- `0` Differentiable variable, default.
- `1` Variable defined over the grid size given in `Values`.
- `2` Variable where a minimum perturbation size given in `Values` may be required before a significant change in the problem is achieved.
- `3` Variable where a meaningful step size should automatically be detected, with an upper limit given in `Values`.

`Values` Grid or minimum step sizes for the variables.

Further information

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

Related topics

`XSLP_SEMICONTDELTAS`, `XSLP_INTEGERDELTAS`, `XSLP_EXPLOREDELTAS`

XSLPchgdf

Purpose

Set or change a distribution factor

Synopsis

```
int XSLP_CC XSLPchgdf(XSLPprob Prob, int ColIndex, int RowIndex, const
      double *Value)
```

Arguments

`Prob`	The current SLP problem.
`ColIndex`	The index of the column whose distribution factor is to be set or changed.
`RowIndex`	The index of the row where the distribution applies.
`Value`	Address of a double precision variable holding the new value of the distribution factor. May be `NULL` if not required.

Example

The following example retrieves the value of the distribution factor for column 282 in row 134 and changes it to be twice as large.

```
double Value;
XSLPgetdf(prob,282,134,&Value);
Value = Value * 2;
XSLPchgdf(prob,282,134,&Value);
```

Further information

The *distribution factor* of a column in a row is the matrix coefficient of the corresponding delta vector in the row. Distribution factors are used in conventional recursion models, and are essentially normalized first-order derivatives. Xpress NonLinear can accept distribution factors instead of initial values, provided that the values of the variables involved can all be calculated after optimization using determining rows, or by a callback.

Related topics

`XSLPadddfs`, `XSLPgetdf`, `XSLPloaddfs`

XSLPchgrowstatus

Purpose

Change the status setting of a constraint

Synopsis

```
int XPRS_CC XSLPchgrowstatus(XSLPprob Prob, int RowIndex, int *Status);
```

Arguments

`Prob`	The current SLP problem.
`RowIndex`	The index of the matrix row to be changed.
`Status`	Address of an integer holding a bitmap with the new status settings. If the status is to be changed, always get the current status first (use `XSLPgetrowstatus`) and then change settings as required. The only settings likely to be changed are: `Bit 11` Set if row must not have a penalty error vector. This is the equivalent of an enforced constraint (SLPDATA type EC).

Example

The following example changes the status of row 9 to be an enforced constraint.

```
int RowIndex, Status;
RowIndex = 9;
XSLPgetrowstatus(Prob,RowIndex,&Status);
Status = Status | (1<<11);
XSLPchgrowstatus(Prob,RowIndex,&Status);
```

Further information

If `Status` is `NULL` the current status will remain unchanged.

Related topics

`XSLPgetrowstatus`

XSLPchgrowwt

Purpose

Set or change the initial penalty error weight for a row

Synopsis

```
int XSLP_CC XSLPchgrowwt(XSLPprob Prob, int RowIndex, const double *Value)
```

Arguments

`Prob`	The current SLP problem.
`RowIndex`	The index of the row whose weight is to be set or changed.
`Value`	Address of a double precision variable holding the new value of the weight. May be `NULL` if not required.

Example

The following example sets the initial weight of row number 2 to a fixed value of 3.6 and the initial weight of row 4 to a value twice the calculated default value.

```
double Value;
Value = -3.6;
XSLPchgrowwt(Prob,2,&Value);
Value = 2.0;
XSLPchgrowwt(Prob,4,&Value);
```

Further information

A positive value is interpreted as a multiplier of the default row weight calculated by Xpress-SLP.

A negative value is interpreted as a fixed value: the absolute value is used directly as the row weight.

The initial row weight is used only when the augmented structure is created.

Related topics

`XSLPgetrowwt`

XSLPchgtolset

Purpose

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

Synopsis

```
int XPRS_CC XSLPchgtolset(XSLPprob Prob, int nSLPTol, int *Status, double
      *Tols);
```

Arguments

`Prob`	The current SLP problem.
`nSLPTol`	Tolerance set for which values are to be changed. A zero value for `nSLPTol` will create a new set.
`Status`	Address of an integer holding a bitmap describing which tolerances are active in this set. See below for the settings.
`Tols`	Array of 9 double precision values holding the values for the corresponding tolerances.

Example

The following example creates a new tolerance set with the default values for all tolerances except the relative delta tolerance, which is set to 0.005. It then changes the value of the absolute delta and absolute impact tolerances in tolerance set 6 to 0.015

```
int Status;
double Tols[9];

Tols[2] = 0.005;
Status = 1<<2;
XSLPchgtolset(Prob, 0, &Status, Tols);
Tols[1] = Tols[5] = 0.015;
Status = 1<<1 | 1<<5;
XSLPchgtolset(Prob, 6, &Status, Tols);
```

Further information

The bits in `Status` are set to indicate that the corresponding tolerance is to be changed in the tolerance set. The meaning of the bits is as follows:

Entry / Bit	Tolerance	XSLP constant	XSLP bit constant
0	Closure tolerance (TC)	`XSLP_TOLSET_TC`	`XSLP_TOLSETBIT_TC`
1	Absolute delta tolerance (TA)	`XSLP_TOLSET_TA`	`XSLP_TOLSETBIT_TA`
2	Relative delta tolerance (RA)	`XSLP_TOLSET_RA`	`XSLP_TOLSETBIT_RA`
3	Absolute coefficient tolerance (TM)	`XSLP_TOLSET_TM`	`XSLP_TOLSETBIT_TM`
4	Relative coefficient tolerance (RM)	`XSLP_TOLSET_RM`	`XSLP_TOLSETBIT_RM`
5	Absolute impact tolerance (TI)	`XSLP_TOLSET_TI`	`XSLP_TOLSETBIT_TI`
6	Relative impact tolerance (RI)	`XSLP_TOLSET_RI`	`XSLP_TOLSETBIT_RI`
7	Absolute slack tolerance (TS)	`XSLP_TOLSET_TS`	`XSLP_TOLSETBIT_TS`
8	Relative slack tolerance (RS)	`XSLP_TOLSET_RS`	`XSLP_TOLSETBIT_RS`

The XSLP_TOLSET constants can be used to access the corresponding entry in the value arrays, while the XSLP_TOLSETBIT constants are used to set or retrieve which tolerance values are used for a given SLP variable. The members of the `Tols` array corresponding to nonzero bit settings in `Status` will be used to change the tolerance set. So, for example, if bit 3 is set in `Status`, then `Tols[3]` will replace the current value of the absolute coefficient tolerance. If a bit is not set in `Status`, the value of the corresponding element of `Tols` is unimportant.

Related topics

XSLPaddtolsets, XSLPdeltolsets, XSLPgettolset, XSLPloadtolsets

XSLPchgvar

Purpose

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

Synopsis

```
int XPRS_CC XSLPchgvar(XSLPprob Prob, int ColIndex, int *DetRow, double
      *InitStepBound, double *StepBound, double *Penalty, double *Damp,
      double *InitValue, double *Value, int *TolSet, int *History, int
      *Converged, int *VarType);
```

Arguments

`Prob` The current SLP problem.

`ColIndex` The index of the matrix column.

`DetRow` Address of an integer holding the index of the determining row. Use -1 if there is no determining row. May be `NULL` if not required.

`InitStepBound` Address of a double precision variable holding the initial step bound size. May be `NULL` if not required.

`StepBound` Address of a double precision variable holding the current step bound size. Use zero to disable the step bounds. May be `NULL` if not required.

`Penalty` Address of a double precision variable holding the weighting of the penalty cost for exceeding the step bounds. May be `NULL` if not required.

`Damp` Address of a double precision variable holding the damping factor for the variable. May be `NULL` if not required.

`InitValue` Address of a double precision variable holding the initial value for the variable. May be `NULL` if not required.

`Value` Address of a double precision variable holding the current value for the variable. May be `NULL` if not required.

`TolSet` Address of an integer holding the index of the tolerance set for this variable. Use zero if there is no specific tolerance set. May be `NULL` if not required.

`History` Address of an integer holding the history value for this variable. May be `NULL` if not required.

`Converged` Address of an integer holding the convergence status for this variable. May be `NULL` if not required.

`VarType` Address of an integer holding a bitmap defining the existence of certain properties for this variable:
`Bit 1:` Variable has a delta vector
`Bit 2:` Variable has an initial value
`Bit 14:` Variable is the reserved "=" column
May be `NULL` if not required.

Example

The following example sets an initial value of 1.42 and tolerance set 2 for column 25 in the matrix.

```
double InitialValue;
int VarType, TolSet;

InitialValue = 1.42;
TolSet = 2;
VarType = 1<<1 | 1<<2;

XSLPchgvar(Prob, 25, NULL, NULL, NULL, NULL,
           NULL, &InitialValue, NULL, &TolSet,
```

```
                    NULL, NULL, &VarType);
```

Note that bits 1 and 2 of `VarType` are set, indicating that the variable has a delta vector and an initial value. For columns already defined as SLP variables, use `XSLPgetvar` to obtain the current value of `VarType` because other bits may already have been set by the system.

Further information

If any of the arguments is `NULL` then the corresponding information for the variable will be left unaltered. If the information is new (i.e. the column was not previously defined as an SLP variable) then the default values will be used.

Changing `Value`, `History` or `Converged` is only effective during SLP iterations.

Changing `InitValue` and `InitStepBound` is only effective before `XSLPconstruct`.

If a value of `XPRS_PLUSINFINITY` is used in the value for `StepBound` or `InitStepBound`, the delta will never have step bounds applied, and will almost always be regarded as converged.

Related topics

`XSLPaddvars`, `XSLPdelvars`, `XSLPgetvar`, `XSLPloadvars`

XSLPconstruct

Purpose

Create the full augmented SLP matrix and data structures, ready for optimization

Synopsis

```
int XPRS_CC XSLPconstruct(XSLPprob Prob);
```

Argument

`Prob` The current SLP problem.

Example

The following example constructs the augmented matrix and then outputs the result in MPS format to a file called augment.mat

```
/* creation and/or loading of data */
/* precedes this segment of code   */
...
XSLPconstruct(Prob);
XSLPwriteprob(Prob,"augment","l");
```

The "l" flag causes output of the current linear problem (which is now the augmented structure and the current linearization) rather than the original nonlinear problem.

Further information

`XSLPconstruct` adds new rows and columns to the SLP matrix and calculates initial values for the non-linear coefficients. Which rows and columns are added will depend on the setting of `XSLP_AUGMENTATION`. Names for the new rows and columns are generated automatically, based on the existing names and the string control variables `XSLP_xxxFORMAT`.

Once `XSLPconstruct` has been called, no new rows, columns or non-linear coefficients can be added to the problem. Any rows or columns which will be required must be added first. Non-linear coefficients must not be changed; constant matrix elements can generally be changed after `XSLPconstruct`, but not after `XSLPpresolve` if used.

`XSLPconstruct` is called automatically by the SLP optimization procedure, and so only needs to be called explicitly if changes need to be made between the augmentation and the optimization.

Related topics

`XSLPpresolve`

XSLPcopycallbacks

Purpose

Copy the user-defined callbacks from one SLP problem to another

Synopsis

```
int XPRS_CC XSLPcopycallbacks(XSLPprob NewProb, XSLPprob OldProb);
```

Arguments

`NewProb`	The SLP problem to receive the callbacks.
`OldProb`	The SLP problem from which the callbacks are to be copied.

Example

The following example creates a new problem and copies only the Xpress NonLinear callbacks from the existing problem (not the Optimizer library ones).

```
XSLPprob nProb;
XPRSprob xProb;
int Control;

XSLPcreateprob(&nProb, &xProb);

Control = 1<<2;
XSLPsetintcontrol(Prob, XSLP_CONTROL, Control);
XSLPcopycallbacks(nProb, Prob);
```

Note that `XSLP_CONTROL` is set in the *old* problem, not the new one.

Further information

Normally `XSLPcopycallbacks` copies both the Xpress NonLinear callbacks and the Optimizer Library callbacks for the underlying problem. If only the Xpress NonLinear callbacks are required, set the integer control variable `XSLP_CONTROL` appropriately.

Related topics

`XSLP_CONTROL`

XSLPcopycontrols

Purpose

Copy the values of the control variables from one SLP problem to another

Synopsis

```
int XPRS_CC XSLPcopycontrols(XSLPprob NewProb, XSLPprob OldProb);
```

Arguments

`NewProb`	The SLP problem to receive the controls.
`OldProb`	The SLP problem from which the controls are to be copied.

Example

The following example creates a new problem and copies only the Xpress NonLinear controls from the existing problem (not the Optimizer library ones).

```
XSLPprob nProb;
XPRSprob xProb;
int Control;

XSLPcreateprob(&nProb, &xProb);

Control = 1<<1;
XSLPsetintcontrol(Prob, XSLP_CONTROL, Control);
XSLPcopycontrols(nProb, Prob);
```

Note that `XSLP_CONTROL` is set in the *old* problem, not the new one.

Further information

Normally `XSLPcopycontrols` copies both the Xpress NonLinear controls and the Optimizer Library controls for the underlying problem. If only the Xpress NonLinear controls are required, set the integer control variable `XSLP_CONTROL` appropriately.

Related topics

`XSLP_CONTROL`

XSLPcopyprob

Purpose

Copy an existing SLP problem to another

Synopsis

```
int XPRS_CC XSLPcopyprob(XSLPprob NewProb, XSLPprob OldProb, char
      *ProbName);
```

Arguments

`NewProb`	The SLP problem to receive the copy.
`OldProb`	The SLP problem from which to copy.
`ProbName`	The name to be given to the problem.

Example

The following example creates a new Xpress NonLinear problem and then copies an existing problem to it. The new problem is named "ANewProblem".

```
XSLPprob nProb;
XPRSprob xProb;

XSLPcreateprob(&nProb, &xProb);
XSLPcopyprob(nProb, Prob, "ANewProblem");
```

Further information

Normally `XSLPcopyprob` copies both the Xpress NonLinear problem and the underlying Optimizer Library problem. If only the Xpress NonLinear problem is required, set the integer control variable `XSLP_CONTROL` appropriately.

This function does not copy controls or callbacks. These must be copied separately using `XSLPcopycontrols` and `XSLPcopycallbacks` if required.

Related topics

`XSLP_CONTROL`

XSLPcreateprob

Purpose

Create a new SLP problem

Synopsis

```
int XPRS_CC XSLPcreateprob(XSLPprob *Prob, XPRSprob *xProb);
```

Arguments

`Prob`	The address of the SLP problem variable.
`xProb`	The address of the underlying Optimizer Library problem variable.

Example

The following example creates an optimizer problem, and then a new Xpress NonLinear problem.

```
XSLPprob nProb;
XPRSprob xProb;

XPRScreateprob(&xProb);
XSLPcreateprob(&nProb, &xProb);
```

Further information

An Xpress NonLinear problem includes an underlying optimizer problem which is used to solve the successive linear approximations. The user is responsible for creating and destroying the underlying linear problem, and can also access it using the normal optimizer library functions. When an SLP problem is to be created, the underlying problem is created first, and the SLP problem is then created, knowing the address of the underlying problem.

Related topics

`XSLPdestroyprob`

XSLPdelcoefs

Purpose

Delete coefficients from the current problem

Synopsis

```
int XPRS_CC XSLPdelcoefs(XSLPprob Prob, in nSLPCoef, int *RowIndex, int
      *ColIndex);
```

Arguments

`Prob`	The current SLP problem.
`nSLPCoef`	Number of SLP coefficients to delete.
`RowIndex`	Row indices of the SLP coefficients to delete.
`ColIndex`	Column indices of the SLP coefficients to delete.

Related topics

`XSLPaddcoefs`, `XSLPchgcoef`, `XSLPchgccoef`, `XSLPgetcoefformula`, `XSLPgetccoef`, `XSLPloadcoefs`

XSLPdeltolsets

Purpose

Delete tolerance sets from the current problem

Synopsis

```
int XPRS_CC XSLPdeltolsets(XSLPprob Prob, int nTolSet, int *TolSetIndex);
```

Arguments

`Prob`	The current SLP problem.
`nTolSet`	Number of tolerance sets to delete.
`TolSetIndex`	Indices of tolerance sets to delete.

Related topics

`XSLPaddtolsets`, `XSLPchgtolset`, `XSLPgettolset`, `XSLPloadtolsets`

XSLPdeluserfunction

Purpose

Delete a user function from the current problem

Synopsis

```
int XPRS_CC XSLPdeluserfunction(XSLPprob prob, int FunctionTokenId);
```

Arguments

`Prob` The current SLP problem.

`FunctionTokenId` The identifier of the user function as returned by `XSLPadduserfunction`.

Related topics

`XSLPadduserfunction`, `XSLPimportlibfunc`

XSLPdelvars

Purpose

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

Synopsis

```
int XPRS_CC XSLPdelvars(XSLPprob prob, int nCol, int *ColIndex);
```

Arguments

`Prob`	The current SLP problem.
`nCol`	Number SLP variables to be converted to linear columns.
`ColIndex`	Column indices of the SLP vars to be converted to linear ones.

Further information

The SLP variables to be converted to linear, non SLP columns must not be in use by any other SLP structure (coefficients, initial value formulae, delayed columns). Use the appropriate deletion or change functions to remove them first.

Related topics

`XSLPaddvars`, `XSLPchgvar`, `XSLPgetvar`, `XSLPloadvars`

XSLPdestroyprob

Purpose

Delete an SLP problem and release all the associated memory

Synopsis

```
int XPRS_CC XSLPdestroyprob(XSLPprob Prob);
```

Argument

`Prob` The SLP problem.

Example

The following example creates an SLP problem and then destroys it together with the underlying optimizer problem.

```
XSLPprob nProb;
XPRSprob xProb;

XPRScreateprob(&xProb);
XSLPcreateprob(&nProb, &xProb);
...
XSLPdestroyprob(nProb);
XPRSdestroyprob(xProb);
```

Further information

When you have finished with the SLP problem, it should be "destroyed" so that the memory used by the problem can be released. Note that this does not destroy the underlying optimizer problem, so a call to `XPRSdestroyprob` should follow `XSLPdestroyprob` as and when you have finished with the underlying optimizer problem.

Related topics

`XSLPcreateprob`

XSLPevaluatecoef

Purpose

Evaluate a coefficient using the current values of the variables

Synopsis

```
int XPRS_CC XSLPevaluatecoef(XSLPprob Prob, int RowIndex, int ColIndex,
      double *dValue);
```

Arguments

`Prob`	The current SLP problem.
`RowIndex`	Integer index of the row.
`ColIndex`	Integer index of the column.
`Value`	Address of a double precision value to receive the result of the calculation.

Example

The following example sets the value of column 5 to 1.42 and then calculates the coefficient in row 2, column 3. If the coefficient depends on column 5, then a value of 1.42 will be used in the calculation.

```
double Value, dValue;

Value = 1.42;
XSLPchgvar(Prob, 5, NULL, NULL, NULL, NULL,
           NULL, NULL, &Value, NULL, NULL, NULL,
           NULL);
XSLPevaluatecoef(Prob, 2, 3, &dValue);
```

Further information

The values of the variables are obtained from the solution, or from the `Value` setting of an SLP variable (see `XSLPchgvar` and `XSLPgetvar`).

Related topics

`XSLPchgvar`, `XSLPevaluateformula` `XSLPgetvar`

XSLPevaluateformula

Purpose

Evaluate a formula using the current values of the variables

Synopsis

```
int XPRS_CC XSLPevaluateformula(XSLPprob Prob, int Parsed, int *Type,
       double *Value, double *dValue);
```

Arguments

`Prob`	The current SLP problem.
`Parsed`	integer indicating whether the formula of the item is in internal unparsed format (`Parsed`=0) or parsed (reverse Polish) format (`Parsed`=1).
`Type`	Integer array of token types for the formula.
`Value`	Double array of values corresponding to `Type`.
`dValue`	Address of a double precision value to receive the result of the calculation.

Example

The following example calculates the value of column 3 divided by column 6.

```
int n, Type[10];
double dValue, Value[10];

n = 0;
Type[n] = XSLP_COL; Value[n++] = 3;
Type[n] = XSLP_COL; Value[n++] = 6;
Type[n] = XSLP_OP;  Value[n++] = XSLP_DIVIDE;
Type[n++] = XSLP_EOF;

XSLPevaluateformula(Prob, 1, Type, Value, &dValue);
```

Further information

The formula in `Type` and `Value` must be terminated by an `XSLP_EOF` token.

The formula cannot include "complicated" functions, such as user functions which return more than one value

Related topics

`XSLPevaluatecoef`

XSLPfixpenalties

Purpose

Fixe the values of the error vectors

Synopsis

```
int XPRS_CC XSLPfixpenalties(XSLPprob Prob, int *Status);
```

Arguments

`Prob`	The current SLP problem.
`Status`	Return status after fixing the penalty variables: 0 is successful, nonzero otherwise.

Further information

The function fixes the values of all error vectors on their current values. It also removes their objective cost contribution.

The function is intended to support post optimization analysis, by removing any possible direct effect of the error vectors from the dual and reduced cost values.

The XSLPfixpenalties will automatically reoptimize the linearization. However, as the XSLP convergence and infeasibility checks (regarding the original non-linear problem) will not be carried out, this function will not update the SLP solution itself. The updated values will be accessible using XPRSgetlpsolution instead.

XSLPfree

Purpose

Free any memory allocated by Xpress NonLinear and close any open Xpress NonLinear files

Synopsis

```
int XPRS_CC XSLPfree(void);
```

Example

The following code frees the Xpress NonLinear memory and then frees the optimizer memory:

```
XSLPfree();
XPRSfree();
```

Further information

A call to `XSLPfree` only frees the items specific to Xpress NonLinear. `XPRSfree` must be called after `XSLPfree` to free the optimizer structures.

Related topics

`XSLPinit`

XSLPgetbanner

Purpose

Retrieve the Xpress NonLinear banner and copyright messages

Synopsis

```
int XPRS_CC XSLPgetbanner(char *Banner);
```

Argument

`Banner` Character buffer to hold the banner. This will be at most 256 characters including the null terminator.

Example

The following example retrieves the Xpress NonLinear banner and prints it

```
char Buffer[260];
XSLPgetbanner(Buffer];
printf("%s\n",Buffer);
```

Further information

Note that `XSLPgetbanner` does not take the normal `Prob` argument.

If `XSLPgetbanner` is called before `XPRSinit`, then it will return only the Xpress NonLinear information; otherwise it will include the `XPRSgetbanner` information as well.

XSLPgetccoef

Purpose

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

Synopsis

```
int XPRS_CC XSLPgetccoef(XSLPprob Prob, int RowIndex, int ColIndex, double
      *Factor, char *Formula, int fLen);
```

Arguments

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

Return value

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

Example

The following example displays the formula for the coefficient in row 2, column 3:

```
char Buffer[60];
double Factor;
int Code;

Code = XSLPgetccoef(Prob, 2, 3, &Factor, Buffer, 60);
switch (Code) {
case 0:  printf("\nFormula is %s",Buffer);
         printf("\nFactor = %lg",Factor);
         break;
case 1:  printf("\nFormula is too long for the buffer");
         break;
default: printf("\nError accessing coefficient");
         break;
}
```

Further information

If the requested coefficient is constant, then `Factor` will be set to 1.0 and the value will be formatted in `Formula`.

If the length of the formula would exceed `fLen-1`, the formula is truncated to the last token that will fit, and the (partial) formula is terminated with a null character.

Related topics

XSLPchgccoef, XSLPchgcoef, XSLPgetcoefformula

XSLPgetcoefformula

Purpose

Retrieve a single matrix coefficient as a formula split into tokens

Synopsis

```
int XPRS_CC XSLPgetcoefformula(XSLPprob Prob, int RowIndex, int ColIndex,
      double *Factor, int Parsed, int BufferSize, int *TokenCount, int
      *Type, double *Value);
```

Arguments

`Prob`	The current SLP problem.
`RowIndex`	Integer holding the row index for the coefficient.
`ColIndex`	Integer holding the column index for the coefficient.
`Factor`	Address of a double precision variable to receive the value of the constant factor multiplying the formula in the coefficient.
`Parsed`	Integer indicating whether the formula of the item is to be returned in internal unparsed format (`Parsed`=0) or parsed (reverse Polish) format (`Parsed`=1).
`BufferSize`	Maximum number of tokens to return, i.e. length of the Type and Value arrays.
`TokenCount`	Number of tokens returned in Type and Value.
`Type`	Integer array to hold the token types for the formula.
`Value`	Double array of values corresponding to `Type`.

Example

The following example displays the formula for the coefficient in row 2, column 3 in unparsed form:

```
int n, Type[10];
double Value[10];
int TokenCount;

XSLPgetcoefformula(Prob, 2, 3, &Factor, 0, 10, &TokenCount, Type, Value);

for (n=0;Type[n] != XSLP_EOF;n++)
  printf("\nType=%-3d  Value=%lg",Type[n],Value[n]);
```

Further information

The `Type` and `Value` arrays are terminated by an `XSLP_EOF` token.

If the requested coefficient is constant, then `Factor` will be set to 1.0 and the value will be returned with token type `XSLP_CON`.

Related topics

`XSLPchgccoef`, `XSLPchgcoef`, `XSLPgetccoef`

XSLPgetcoefs

Purpose

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

Synopsis

```
int XPRS_CC XSLPgetcoefs(XSLPprob Prob, int *nCoef, int *RowIndices, int
      *ColIndices);
```

Arguments

`Prob`	The current SLP problem.
`nCoef`	Integer used to return the total number of nonlinear coefficients in the problem.
`RowIndices`	Integer array used for returning the row positions of the coefficients. May be NULL if not required.
`ColIndices`	Integer array used for returning the column positions of the coefficients. May be NULL if not required.

Related topics

`XSLPgetccoef`, `XSLPgetcoefformula`

XSLPgetcolinfo

Purpose

Get current column information.

Synopsis

```
int XSLP_CC XSLPgetcolinfo(XSLPprob Prob, int InfoType, int ColIndex,
      XSLPalltype *Info);
```

Arguments

`Prob`	The current SLP problem
`InfoType`	Type of information (see below)
`ColIndex`	Index of the column whose information is to be handled
`Info`	Address of information to be set or retrieved

Further information

If the data is not available, the type of the returned Info is set to `XSLPtype_undefined`.

Please refer to the header file `xslp.h` for the definition of XSLPalltype.

The following constants are provided for column information handling:

`XSLP_COLINFO_VALUE`	Get the current value of the column
`XSLP_COLINFO_RDJ`	Get the current reduced cost of the column
`XSLP_COLINFO_DELTAINDEX`	Get the delta variable index associated to the column
`XSLP_COLINFO_DELTA`	Get the delta value (change since previous value) of the column
`XSLP_COLINFO_DELTADJ`	Get the delta variables reduced cost
`XSLP_COLINFO_UPDATEROW`	Get the index of the update (or step bound) row associated to the column
`XSLP_COLINFO_SB`	Get the step bound on the variable
`XSLP_COLINFO_SBDUAL`	Get the dual multiplier of the step bound row for the variable

XSLPgetdblattrib

Purpose

Retrieve the value of a double precision problem attribute

Synopsis

```
int XPRS_CC XSLPgetdblattrib(XSLPprob Prob, int Param, double *dValue);
```

Arguments

`Prob`	The current SLP problem.
`Param`	attribute (SLP or optimizer) whose value is to be returned.
`dValue`	Address of a double precision variable to receive the value.

Example

The following example retrieves the value of the Xpress NonLinear attribute `XSLP_CURRENTDELTACOST` and of the optimizer attribute `XPRS_LPOBJVAL`:

```
double DeltaCost, ObjVal;
XSLPgetdblattrib(Prob, XSLP_CURRENTDELTACOST, &DeltaCost);
XSLPgetdblattrib(Prob, XPRS_LPOBJVAL, &ObjVal);
```

Further information

Both SLP and optimizer attributes can be retrieved using this function. If an optimizer attribute is requested, the return value will be the same as that from `XPRSgetdblattrib`.

Related topics

`XSLPgetintattrib`, `XSLPgetstrattrib`

XSLPgetdblcontrol

Purpose

Retrieve the value of a double precision problem control

Synopsis

```
int XPRS_CC XSLPgetdblcontrol(XSLPprob Prob, int Param, double *dValue);
```

Arguments

`Prob`	The current SLP problem.
`Param`	control (SLP or optimizer) whose value is to be returned.
`dValue`	Address of a double precision variable to receive the value.

Example

The following example retrieves the value of the Xpress NonLinear control `XSLP_CTOL` and of the optimizer control `XPRS_FEASTOL`:

```
double CTol, FeasTol;
XSLPgetdblcontrol(Prob, XSLP_CTOL, &CTol);
XSLPgetdblcontrol(Prob, XPRS_FEASTOL, &FeasTol);
```

Further information

Both SLP and optimizer controls can be retrieved using this function. If an optimizer control is requested, the return value will be the same as that from `XPRSgetdblcontrol`.

Related topics

`XSLPgetintcontrol`, `XSLPgetstrcontrol`, `XSLPsetdblcontrol`

XSLPgetdf

Purpose

Get a distribution factor

Synopsis

```
int XSLP_CC XSLPgetdf(XSLPprob Prob, int ColIndex, int RowIndex, double
      *Value)
```

Arguments

`Prob`	The current SLP problem.
`ColIndex`	The index of the column whose distribution factor is to be retrieved.
`RowIndex`	The index of the row from which the distribution factor is to be taken.
`Value`	Address of a double precision variable to receive the value of the distribution factor. May be `NULL` if not required.

Example

The following example retrieves the value of the distribution factor for column 282 in row 134 and changes it to be twice as large.

```
double Value;
XSLPgetdf(prob,282,134,&Value);
Value = Value * 2;
XSLPchgdf(prob,282,134,&Value);
```

Further information

The *distribution factor* of a column in a row is the matrix coefficient of the corresponding delta vector in the row. Distribution factors are used in conventional recursion models, and are essentially normalized first-order derivatives. Xpress-SLP can accept distribution factors instead of initial values, provided that the values of the variables involved can all be calculated after optimization using determining rows, or by a callback.

Related topics

`XSLPadddfs`, `XSLPchgdf`, `XSLPloaddfs`

XSLPgetindex

Purpose

Retrieve the index of an Xpress NonLinear entity with a given name

Synopsis

```
int XPRS_CC XSLPgetindex(XSLPprob Prob, int Type, char *cName, int *Index);
```

Arguments

`Prob` The current SLP problem.

`Type` Type of entity. The following are defined:

`XSLP_CVNAMES`	`(=3)`	Character variables;
`XSLP_USERFUNCNAMES`	`(=5)`	User functions;
`XSLP_INTERNALFUNCNAMES`	`(=6)`	Internal functions;
`XSLP_USERFUNCNAMESNOCASE`	`(=7)`	User functions, case insensitive;
`XSLP_INTERNALFUNCNAMESNOCASE`	`(=8)`	Internal functions, case insensitive;

The constants 1 (for row names) and 2 (for column names) may also be used.

`cName` Character string containing the name, terminated by a null character.

`Index` Integer to receive the index of the item.

Example

The following example retrieves the index of the internal `SIN` function using both an upper-case and a lower case version of the name.

```
int UpperIndex, LowerIndex;
XSLPgetindex(Prob, XSLP_INTERNALFUNCNAMESNOCASE,
             "SIN", &UpperIndex);
XSLPgetindex(Prob, XSLP_INTERNALFUNCNAMESNOCASE,
             "sin", &LowerIndex);
```

`UpperIndex` and `LowerIndex` will contain the same value because the search was made using case-insensitive matching.

Further information

All entities count from 1. This includes the use of 1 or 2 (row or column) for `Type`. A value of zero returned in `Index` means there is no matching item. The case-insensitive types will find the first match regardless of the case of `cName` or of the defined function.

XSLPgetintattrib

Purpose

Retrieve the value of an integer problem attribute

Synopsis

```
int XPRS_CC XSLPgetintattrib(XSLPprob Prob, int Param, int *iValue);
```

Arguments

`Prob`	The current SLP problem.
`Param`	attribute (SLP or optimizer) whose value is to be returned.
`iValue`	Address of an integer variable to receive the value.

Example

The following example retrieves the value of the Xpress NonLinear attribute `XSLP_CVS` and of the optimizer attribute `XPRS_COLS`:

```
int nCV, nCol;
XSLPgetintattrib(Prob, XSLP_CVS, &nCV);
XSLPgetintattrib(Prob, XPRS_COLS, &nCol);
```

Further information

Both SLP and optimizer attributes can be retrieved using this function. If an optimizer attribute is requested, the return value will be the same as that from `XPRSgetintattrib`.

Related topics

`XSLPgetdblattrib`, `XSLPgetstrattrib`

XSLPgetintcontrol

Purpose

Retrieve the value of an integer problem control

Synopsis

```
int XPRS_CC XSLPgetintcontrol(XSLPprob Prob, int Param, int *iValue);
```

Arguments

`Prob`	The current SLP problem.
`Param`	control (SLP or optimizer) whose value is to be returned.
`iValue`	Address of an integer variable to receive the value.

Example

The following example retrieves the value of the Xpress NonLinear control `XSLP_ALGORITHM` and of the optimizer control `XPRS_DEFAULTALG`:

```
int Algorithm, DefaultAlg;
XSLPgetintcontrol(Prob, XSLP_ALGORITHM, &Algorithm);
XSLPgetintcontrol(Prob, XPRS_DEFAULTALG, &DefaultAlg);
```

Further information

Both SLP and optimizer controls can be retrieved using this function. If an optimizer control is requested, the return value will be the same as that from `XPRSgetintcontrol`.

Related topics

`XSLPgetdblcontrol`, `XSLPgetstrcontrol`, `XSLPsetintcontrol`

XSLPgetlasterror

Purpose

Retrieve the error message corresponding to the last Xpress NonLinear error during an SLP run

Synopsis

```
int XPRS_CC XSLPgetlasterror(XSLPprob Prob, int *Code, char *Buffer);
```

Arguments

`Prob`	The current SLP problem.
`Code`	Address of an integer to receive the message number of the last error. May be `NULL` if not required.
`Buffer`	Character buffer to receive the error message. The error message will never be longer than 256 characters. May be `NULL` if not required.

Example

The following example checks the return code from reading a matrix. If the code is nonzero then an error has occurred, and the error number is retrieved for further processing.

```
int Error, Code;
if (Error=XSLPreadprob(Prob, "Matrix", "")) {
  XSLPgetlasterror(Prob, &Code, NULL);
  MyErrorHandler(Code);
}
```

Further information

In general, Xpress NonLinear functions return a value of 32 to indicate a non-recoverable error. `XSLPgetlasterror` can retrieve the actual error number and message. In case no SLP error code was retuned, the function will check the underlying XPRS libary for any errors reported.

XSLPgetptrattrib

Purpose

Retrieve the value of a problem pointer attribute

Synopsis

```
int XPRS_CC XSLPgetptrattrib(XSLPprob Prob, int Param, void **Value);
```

Arguments

`Prob`	The current SLP problem.
`Param`	attribute whose value is to be returned.
`Value`	Address of a pointer to receive the value.

Example

The following example retrieves the value of the Xpress NonLinear pointer attribute `XSLP_XPRSPROBLEM` which is the underlying optimizer problem pointer:

```
XPRSprob xprob;
XSLPgetptrattrib(Prob, XSLP_XPRSPROBLEM, &xprob);
```

Further information

This function is normally used to retrieve the underlying optimizer problem pointer, as shown in the example.

Related topics

`XSLPgetdblattrib`, `XSLPgetintattrib`, `XSLPgetstrattrib`

XSLPgetrowinfo

Purpose

Get current row information.

Synopsis

```
int XSLP_CC XSLPgetrowinfo(XSLPprob Prob, int InfoType, int RowIndex,
      XSLPalltype *Info);
```

Arguments

`Prob`	The current SLP problem
`InfoType`	Type of information (see below)
`RowIndex`	Index of the row whose information is to be handled
`Info`	Address of information to be set or retrieved

Further information

If the data is not available, the type of the returned Info is set to `XSLPtype_undefined`.

Please refer to the header file `xslp.h` for the definition of XSLPalltype.

The following constants are provided for row information handling:

`XSLP_ROWINFO_SLACK`	Get the current slack value of the row
`XSLP_ROWINFO_DUAL`	Get the current dual multiplier of the row
`XSLP_ROWINFO_NUMPENALTYERRORS`	Get the number of times the penalty error vector has been active for the row
`XSLP_ROWINFO_MAXPENALTYERROR`	Get the maximum size of the penalty error vector activity for the row
`XSLP_ROWINFO_TOTALPENALTYERROR`	Get the total size of the penalty error vector activity for the row
`XSLP_ROWINFO_CURRENTPENALTYERROR`	Get the size of the penalty error vector activity in the current iteration for the row
`XSLP_ROWINFO_CURRENTPENALTYFACTOR`	Set the size of the penalty error factor for the current iteration for the row
`XSLP_ROWINFO_PENALTYCOLUMNPLUS`	Get the index of the positive penalty column for the row (+)
`XSLP_ROWINFO_PENALTYCOLUMNPLUSVALUE`	Get the value of the positive penalty column for the row (+)
`XSLP_ROWINFO_PENALTYCOLUMNPLUSDJ`	Get the reduced cost of the positive penalty column for the row (+)
`XSLP_ROWINFO_PENALTYCOLUMNMINUS`	Get the index of the negative penalty column for the row (-)
`XSLP_ROWINFO_PENALTYCOLUMNMINUSVALUE`	Get the value of the negative penalty column for the row (-)
`XSLP_ROWINFO_PENALTYCOLUMNMINUSDJ`	Get the reduced cost of the negative penalty column for the row (-)

XSLPgetrowstatus

Purpose

Retrieve the status setting of a constraint

Synopsis

```
int XPRS_CC XSLPgetrowstatus(XSLPprob Prob, int RowIndex, int *Status);
```

Arguments

`Prob`	The current SLP problem.
`RowIndex`	The index of the matrix row whose data is to be obtained.
`Status`	Address of an integer holding a bitmap to receive the status settings.

Example

This recovers the status of the rows of the matrix of the current problem and reports those which are flagged as enforced constraints.

```
int iRow, nRow, Status;
XSLPgetintattrib(Prob, XPRS_ROWS, &nRow);
for (iRow=0;iRow<nRow;iRow++) {
  XSLPgetrowstatus(Prob, iRow, &Status);
  if (Status & 0x800) printf("\nRow %d is enforced");
}
```

Further information

See the section on bitmap settings for details on the possible information in `Status`.

Related topics

`XSLPchgrowstatus`

XSLPgetrowwt

Purpose

Get the initial penalty error weight for a row

Synopsis

```
int XSLP_CC XSLPgetrowwt(XSLPprob Prob, int RowIndex, double *Value)
```

Arguments

`Prob`	The current SLP problem.
`RowIndex`	The index of the row whose weight is to be retrieved.
`Value`	Address of a double precision variable to receive the value of the weight.

Example

The following example gets the initial weight of row number 2.

```
double Value;
XSLPgetrowwt(Prob,2,&Value)
```

Further information

The initial row weight is used only when the augmented structure is created. After that, the current weighting can be accessed using `XSLPgetrowinfo`.

Related topics

`XSLPchgrowwt`, `XSLPgetrowinfo`

XSLPgetslpsol

Purpose

Obtain the solution values for the most recent SLP iteration

Synopsis

```
int XPRS_CC XSLPgetslpsol(XSLPprob Prob, double *x, double *slack, double
      *dual, double *dj);
```

Arguments

`Prob`	The current SLP problem.
`x`	Double array of length `XSLP_ORIGINALCOLS` to hold the values of the primal variables. May be `NULL` if not required.
`slack`	Double array of length `XSLP_ORIGINALROWS` to hold the values of the slack variables. May be `NULL` if not required.
`dual`	Double array of length `XSLP_ORIGINALROWS` to hold the values of the dual variables. May be `NULL` if not required.
`dj`	Double array of length `XSLP_ORIGINALCOLS` to hold the recuded costs of the primal variables. May be `NULL` if not required.

Example

The following code fragment recovers the values and reduced costs of the primal variables from the most recent SLP iteration:

```
XSLPprob prob;
int nCol;
double *val, *dj;
XSLPgetintattrib(prob,XSLP_ORIGINALCOLS,&nCol);
val = malloc(nCol*sizeof(double));
dj = malloc(nCol*sizeof(double));
XSLPgetslpsol(prob,val,NULL,NULL,dj);
```

Further information

`XSLPgetslpsol` can be called at any time after an SLP iteration has completed, and will return the same values even if the problem is subsequently changed. `XSLPgetslpsol` returns values for the columns and rows originally in the problem and not for any augmentation rows or columns. To access the values of any augmentation columns or rows, use `XPRSgetlpsol`; accessing the augmented solution is only recommended if `XSLP_PRESOLVELEVEL` indicates that the problem dimensions should not be changed in presolve.

XSLPgetstrattrib

Purpose

Retrieve the value of a string problem attribute

Synopsis

```
int XPRS_CC XSLPgetstrattrib(XSLPprob Prob, int Param, char *cValue);
```

Arguments

`Prob`	The current SLP problem.
`Param`	attribute (SLP or optimizer) whose value is to be returned.
`cValue`	Character buffer to receive the value.

Example

The following example retrieves the value of the Xpress NonLinear attribute `XSLP_VERSIONDATE` and of the optimizer attribute `XPRS_MATRIXNAME`:

```
char VersionDate[200], MatrixName[200];
XSLPgetstrattrib(Prob, XSLP_VERSIONDATE, VersionDate);
XSLPgetstrattrib(Prob, XPRS_MATRIXNAME, MatrixName);
```

Further information

Both SLP and optimizer attributes can be retrieved using this function. If an optimizer attribute is requested, the return value will be the same as that from `XPRSgetstrattrib`.

Related topics

`XSLPgetdblattrib`, `XSLPgetintattrib`

XSLPgetstrcontrol

Purpose

Retrieve the value of a string problem control

Synopsis

```
int XPRS_CC XSLPgetstrcontrol(XSLPprob Prob, int Param, char *cValue);
```

Arguments

`Prob`	The current SLP problem.
`Param`	control (SLP or optimizer) whose value is to be returned.
`cValue`	Character buffer to receive the value.

Example

The following example retrieves the value of the Xpress NonLinear control `XSLP_CVNAME` and of the optimizer control `XPRS_MPSOBJNAME`:

```
char CVName[200], ObjName[200];
XSLPgetstrcontrol(Prob, XSLP_CVNAME, CVName);
XSLPgetstrcontrol(Prob, XPRS_MPSOBJNAME, ObjName);
```

Further information

Both SLP and optimizer controls can be retrieved using this function. If an optimizer control is requested, the return value will be the same as that from `XPRSgetstrcontrol`.

Related topics

`XSLPgetdblcontrol`, `XSLPgetintcontrol`, `XSLPsetstrcontrol`

XSLPgettolset

Purpose

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

Synopsis

```
int XPRS_CC XSLPgettolset(XSLPprob Prob, int nSLPTol, int *Status, double
      *Tols);
```

Arguments

`Prob`	The current SLP problem.
`nSLPTol`	The index of the tolerance set.
`Status`	Address of integer to receive the bit-map of status settings. May be `NULL` if not required.
`Tols`	Array of 9 double-precision values to hold the tolerances. May be `NULL` if not required.

Example

The following example retrieves the values for tolerance set 3 and prints those which are set:

```
double Tols[9];
int i, Status;
XSLPgettolset(Prob, 3, &Status, Tols);
for (i=0;i<9;i++)
  if (Status & (1<<i))
    printf("\nTolerance %d = %lg",i,Tols[i]);
```

Further information

If `Status` or `Tols` is `NULL`, then the corresponding information will not be returned.

If `Tols` is not `NULL`, then a set of 9 values will always be returned. `Status` indicates which of these values is active as follows. Bit `n` of `Status` is set if `Tols[n]` is active, where `n` is:

Entry / Bit	Tolerance	XSLP constant	XSLP bit constant
0	Closure tolerance (TC)	`XSLP_TOLSET_TC`	`XSLP_TOLSETBIT_TC`
1	Absolute delta tolerance (TA)	`XSLP_TOLSET_TA`	`XSLP_TOLSETBIT_TA`
2	Relative delta tolerance (RA)	`XSLP_TOLSET_RA`	`XSLP_TOLSETBIT_RA`
3	Absolute coefficient tolerance (TM)	`XSLP_TOLSET_TM`	`XSLP_TOLSETBIT_TM`
4	Relative coefficient tolerance (RM)	`XSLP_TOLSET_RM`	`XSLP_TOLSETBIT_RM`
5	Absolute impact tolerance (TI)	`XSLP_TOLSET_TI`	`XSLP_TOLSETBIT_TI`
6	Relative impact tolerance (RI)	`XSLP_TOLSET_RI`	`XSLP_TOLSETBIT_RI`
7	Absolute slack tolerance (TS)	`XSLP_TOLSET_TS`	`XSLP_TOLSETBIT_TS`
8	Relative slack tolerance (RS)	`XSLP_TOLSET_RS`	`XSLP_TOLSETBIT_RS`

The XSLP_TOLSET constants can be used to access the corresponding entry in the value arrays, while the XSLP_TOLSETBIT constants are used to set or retrieve which tolerance values are used for a given SLP variable.

Related topics

Related topics

`XSLPaddtolsets`, `XSLPchgtolset`, `XSLPdeltolsets`, `XSLPloadtolsets`

XSLPgetvar

Purpose

Retrieve information about an SLP variable

Synopsis

```
int XPRS_CC XSLPgetvar(XSLPprob prob, int ColIndex, int *DetRow, double
      *InitStepBound, double *StepBound, double *Penalty, double *Damp,
      double *InitValue, double *Value, int *TolSet, int *History, int
      *Converged, int *VarType, int *Delta, int *PenaltyDelta, int
      *UpdateRow, double *OldValue);
```

Arguments

`Prob` The current SLP problem.

`ColIndex` The index of the column.

`DetRow` Address of an integer to receive the index of the determining row. May be `NULL` if not required.

`InitStepBound` Address of a double precision variable to receive the value of the initial step bound of the variable. May be `NULL` if not required.

`StepBound` Address of a double precision variable to receive the value of the current step bound of the variable. May be `NULL` if not required.

`Penalty` Address of a double precision variable to receive the value of the penalty delta weighting of the variable. May be `NULL` if not required.

`Damp` Address of a double precision variable to receive the value of the current damping factor of the variable. May be `NULL` if not required.

`InitValue` Address of a double precision variable to receive the value of the initial value of the variable. May be `NULL` if not required.

`Value` Address of a double precision variable to receive the current activity of the variable. May be `NULL` if not required.

`TolSet` Address of an integer to receive the index of the tolerance set of the variable. May be `NULL` if not required.

`History` Address of an integer to receive the SLP history of the variable. May be `NULL` if not required.

`Converged` Address of an integer to receive the convergence status of the variable as defined in the "Convergence Criteria" section (The returned value will match the numbering of the tolerances). May be `NULL` if not required.

`VarType` Address of an integer to receive the status settings (a bitmap defining the existence of certain properties for this variable). The following bits are defined:
`Bit 1:` Variable has a delta vector
`Bit 2:` Variable has an initial value
`Bit 14:` Variable is the reserved "=" column
Other bits are reserved for internal use. May be `NULL` if not required.

`Delta` Address of an integer to receive the index of the delta vector for the variable. May be `NULL` if not required.

`PenaltyDelta` Address of an integer to receive the index of the first penalty delta vector for the variable. The second penalty delta immediately follows the first. May be `NULL` if not required.

`UpdateRow` Address of an integer to receive the index of the update row for the variable. May be `NULL` if not required.

`OldValue` Address of a double precision variable to receive the value of the variable at the previous SLP iteration. May be `NULL` if not required.

Example

The following example retrieves the current value, convergence history and status for column 3.

```
int Status, History;
double Value;

XSLPgetvar(Prob, 3, NULL, NULL, NULL,
           NULL, NULL, NULL, &Value,
           NULL, &History, &Converged,
           NULL, NULL, NULL, NULL, NULL);
```

Further information

If `ColIndex` refers to a column which is not an SLP variable, then all the return values will indicate that there is no corresponding data.

`DetRow` will be set to -1 if there is no determining row.

`Delta`, `PenaltyDelta` and `UpdateRow` will be set to -1 if there is no corresponding item.

Related topics

`XSLPaddvars`, `XSLPchgvar`, `XSLPdelvars`, `XSLPloadvars`

XSLPglobal

Purpose

Initiate the Xpress NonLinear mixed integer SLP (MISLP) algorithm

Synopsis

```
int XPRS_CC XSLPglobal(XSLPprob Prob);
```

Argument

`Prob` The current SLP problem.

Example

The following example optimizes the problem and then finds the integer solution.

```
XSLPmaxim(Prob,"");
XSLPglobal(Prob);
```

Further information

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

It is recommended that `XSLPminim` or `XSLPmaxim` is used first to obtain a converged solution to the relaxed problem. If this is not done, ensure that `XSLP_OBJSENSE` is set appropriately.

See the chapter on Mixed Integer Non-Linear Programming for more information about the Xpress NonLinear MISLP algorithms.

Related topics

`XSLPmaxim`, `XSLPminim`, `XSLP_MIPALGORITHM`, `XSLP_OBJSENSE`

XSLPimportlibfunc

Purpose

Imports a function from a library file to be called as a user function

Synopsis

```
int XPRS_CC XSLPimportlibfunc(XSLPprob Prob,const char * LibName, const
      char * FunctionName, void ** FuncPointer, int * Status );
```

Arguments

`Prob` The current SLP problem.

`LibName` Filename of the library.

`FunctionName` Fucntion anme inside the library.

`FuncPointer` Function pointer to return the loaded function.

`Status` Outcome of the load operation

`0`	success.
`1`	library file not found.
`2`	library function in library file not found.

Further information

On systems where necessary, Xpress will hold the handle of the library opened and free up when the problem object `Prob` is destroyed.

Related topics

`XSLPadduserfunction`, `XSLPdeluserfunction`

XSLPinit

Purpose

Initializes the Xpress NonLinear system

Synopsis

```
int XPRS_CC XSLPinit();
```

Argument

```
none
```

Example

The following example initiates the Xpress NonLinear system and prints the banner.

```
char Buffer[256];
XPRSinit();
XSLPinit();
XSLPgetbanner(Buffer);
```

`XPRSinit` initializes the Xpress optimizer; `XSLPinit` then initializes the SLP module, so that the banner contains information from both systems.

Further information

`XSLPinit` must be the first call to the Xpress NonLinear system except for `XSLPgetbanner` and `XSLPgetversion`. It initializes any global parts of the system if required. The call to `XSLPinit` must be preceded by a call to `XPRSinit` to initialize the Optimizer Library part of the system first.

Related topics

`XSLPfree`

XSLPinterrupt

Purpose

Interrupts the current SLP optimization

Synopsis

```
int XPRS_CC XSLPinterrupt(int Reason);
```

Arguments

`Prob`	The current SLP problem.
`Reason`	Interrupt code to be propagated.

Further information

Provides functionality to stop the SLP optimization process from inside a callback. The following constants are provided for the paramter value:

Value 1	`XSLP_STOP_TIMELIMIT`
Value 2	`XSLP_STOP_CTRLC`
Value 3	`XSLP_STOP_NODELIMIT`
Value 4	`XSLP_STOP_ITERLIMIT`
Value 5	`XSLP_STOP_MIPGAP`
Value 6	`XSLP_STOP_SOLLIMIT`
Value 9	`XSLP_STOP_USER`

XSLPitemname

Purpose

Retrieves the name of an Xpress NonLinear entity or the value of a function token as a character string.

Synopsis

```
int XPRS_CC XSLPitemname(XSLPprob Prob, int Type, double Value, char
      *Buffer);
```

Arguments

`Prob`	The current SLP problem.
`Type`	Integer holding the type of Xpress NonLinear entity. This can be any one of the token types described in the section on Formula Parsing.
`Value`	Double precision value holding the index or value of the token. The use and meaning of the value is as described in the section on Formula Parsing.
`Buffer`	Character buffer to hold the result, which will be terminated with a null character.

Example

The following example displays the formula for the coefficient in row 2, column 3 in unparsed form:

```
int n, Type[10];
double Value[10];
char Buffer[60];
int TokenCount;

XSLPgetcoefformula(Prob, 2, 3, &Factor, 0, 10, &TokenCount, Type, Value);

printf("\n");
for (n=0;Type[n] != XSLP_EOF;n++) {
  XSLPitemname(Prob, Type[n], Value[n], Buffer);
  printf(" %s", Buffer);
}
```

Further information

If a name has not been provided for an Xpress NonLinear entity, then an internally-generated name will be used.

Numerical values will be formatted as fixed-point or floating-point depending on their size.

XSLPloadcoefs

Purpose

Load non-linear coefficients into the SLP problem

Synopsis

```
int XPRS_CC XSLPloadcoefs(XSLPprob Prob, int nSLPCoef, int *RowIndex, int
      *ColIndex, double *Factor, int *FormulaStart, int Parsed, int *Type,
      double *Value);
```

Arguments

`Prob`	The current SLP problem.
`nSLPCoef`	Number of non-linear coefficients to be loaded.
`RowIndex`	Integer array holding index of row for the coefficient.
`ColIndex`	Integer array holding index of column for the coefficient.
`Factor`	Double array holding factor by which formula is scaled. If this is `NULL`, then a value of 1.0 will be used.
`FormulaStart`	Integer array of length `nSLPCoef+1` holding the start position in the arrays `Type` and `Value` of the formula for the coefficients. `FormulaStart[nSLPCoef]` should be set to the next position after the end of the last formula.
`Parsed`	Integer indicating whether the token arrays are formatted as internal unparsed (`Parsed`=0) or internal parsed reverse Polish (`Parsed`=1).
`Type`	Array of token types providing the formula for each coefficient.
`Value`	Array of values corresponding to the types in `Type`.

Example

Assume that the rows and columns of `Prob` are named `Row1`, `Row2` ..., `Col1`, `Col2` ... The following example loads coefficients representing:
`Col2 * Col3 + Col6 * Col2^2` into `Row1` and
`Col2 ^ 2` into `Row3`.

```
int RowIndex[3], ColIndex[3], FormulaStart[4], Type[8];
int n, nSLPCoef;
double Value[8];

RowIndex[0] = 1; ColIndex[0] = 2;
RowIndex[1] = 1; ColIndex[1] = 6;
RowIndex[2] = 3; ColIndex[2] = 2;

n = nSLPCoef = 0;
FormulaStart[nSLPCoef++] = n;
Type[n] = XSLP_COL; Value[n++] = 3;
Type[n++] = XSLP_EOF;

FormulaStart[nSLPCoef++] = n;
Type[n] = XSLP_COL; Value[n++] = 2;
Type[n] = XSLP_COL; Value[n++] = 2;
Type[n] = XSLP_OP;  Value[n++] = XSLP_MULTIPLY;
Type[n++] = XSLP_EOF;

FormulaStart[nSLPCoef++] = n;
Type[n] = XSLP_COL; Value[n++] = 2;
Type[n++] = XSLP_EOF;
```

```
FormulaStart[nSLPCoef] = n;

XSLPloadcoefs(Prob, nSLPCoef, RowIndex, ColIndex,
              NULL, FormulaStart, 1, Type, Value);
```

The first coefficient in `Row1` is in `Col2` and has the formula `Col3`, so it represents `Col2 * Col3`.

The second coefficient in `Row1` is in `Col6` and has the formula `Col2 * Col2` so it represents `Col6 * Col2^2`. The formulae are described as *parsed* (`Parsed`=1), so the formula is written as
`Col2 Col2 *`
rather than the unparsed form
`Col2 * Col2`

The last coefficient, in `Row3`, is in `Col2` and has the formula `Col2`, so it represents `Col2 * Col2`.

Further information

The j^{th} coefficient is made up of two parts: `Factor` and `Formula`. `Factor` is a constant multiplier, which can be provided in the `Factor` array. If Xpress NonLinear can identify a constant factor in `Formula`, then it will use that as well, to minimize the size of the formula which has to be calculated. `Formula` is made up of a list of tokens in `Type` and `Value` starting at `FormulaStart[j]`. The tokens follow the rules for parsed or unparsed formulae as indicated by the setting of `Parsed`. The formula must be terminated with an `XSLP_EOF` token. If several coefficients share the same formula, they can have the same value in `FormulaStart`. For possible token types and values see the chapter on "Formula Parsing".

The `XSLPload...` functions load items into the SLP problem. Any existing items of the same type are deleted first. The corresponding `XSLPadd...` functions add or replace items leaving other items of the same type unchanged.

Related topics

`XSLPaddcoefs`, `XSLPchgcoef`, `XSLPchgccoef`, `XSLPgetcoefformula`, `XSLPgetccoef`

XSLPloaddfs

Purpose

Load a set of distribution factors

Synopsis

```
int XSLP_CC XSLPloaddfs(XSLPprob Prob, int nDF, const int *ColIndex, const
      int *RowIndex, const double *Value)
```

Arguments

`Prob`	The current SLP problem.
`nDF`	The number of distribution factors.
`ColIndex`	Array of indices of columns whose distribution factor is to be changed.
`RowIndex`	Array of indices of the rows where each distribution factor applies.
`Value`	Array of double precision variables holding the new values of the distribution factors.

Example

The following example loads distribution factors as follows:
column 282 in row 134 = 0.1
column 282 in row 136 = 0.15
column 285 in row 133 = 1.0.
Any other first-order derivative placeholders are set to `XSLP_DELTA_Z`.

```
int ColIndex[3], RowIndex[3];
double Value[3];
ColIndex[0] = 282;  RowIndex[0] = 134; Value[0] = 0.1;
ColIndex[1] = 282;  RowIndex[1] = 136; Value[1] = 0.15;
ColIndex[2] = 285;  RowIndex[2] = 133; Value[2] = 1.0;
XSLPloaddfs(prob,3,ColIndex,RowIndex,Value);
```

Further information

The *distribution factor* of a column in a row is the matrix coefficient of the corresponding delta vector in the row. Distribution factors are used in conventional recursion models, and are essentially normalized first-order derivatives. Xpress-SLP can accept distribution factors instead of initial values, provided that the values of the variables involved can all be calculated after optimization using determining rows, or by a callback.

The `XSLPadd...` functions load additional items into the SLP problem. The corresponding `XSLPload...` functions delete any existing items first.

Related topics

`XSLPadddfs`, `XSLPchgdf`, `XSLPgetdf`

XSLPloadtolsets

Purpose

Load sets of standard tolerance values into an SLP problem

Synopsis

```
int XPRS_CC XSLPloadtolsets(XSLPprob Prob, int nSLPTol, double *SLPTol);
```

Arguments

`Prob`	The current SLP problem.
`nSLPTol`	The number of tolerance sets to be loaded.
`SLPTol`	Double array of (`nSLPTol * 9`) items containing the 9 tolerance values for each set in order.

Example

The following example creates two tolerance sets: the first has values of 0.005 for all tolerances; the second has values of 0.001 for relative tolerances (numbers 2,4,6,8), values of 0.01 for absolute tolerances (numbers 1,3,5,7) and zero for the closure tolerance (number 0).

```
double SLPTol[18];
for (i=0;i<9;i++) SLPTol[i] = 0.005;
SLPTol[9] = 0;
for (i=10;i<18;i=i+2) SLPTol[i] = 0.01;
for (i=11;i<18;i=i+2) SLPTol[i] = 0.001;
XSLPloadtolsets(Prob, 2, SLPTol);
```

Further information

A tolerance set is an array of 9 values containing the following tolerances:

Entry / Bit	Tolerance	XSLP constant	XSLP bit constant
0	Closure tolerance (TC)	`XSLP_TOLSET_TC`	`XSLP_TOLSETBIT_TC`
1	Absolute delta tolerance (TA)	`XSLP_TOLSET_TA`	`XSLP_TOLSETBIT_TA`
2	Relative delta tolerance (RA)	`XSLP_TOLSET_RA`	`XSLP_TOLSETBIT_RA`
3	Absolute coefficient tolerance (TM)	`XSLP_TOLSET_TM`	`XSLP_TOLSETBIT_TM`
4	Relative coefficient tolerance (RM)	`XSLP_TOLSET_RM`	`XSLP_TOLSETBIT_RM`
5	Absolute impact tolerance (TI)	`XSLP_TOLSET_TI`	`XSLP_TOLSETBIT_TI`
6	Relative impact tolerance (RI)	`XSLP_TOLSET_RI`	`XSLP_TOLSETBIT_RI`
7	Absolute slack tolerance (TS)	`XSLP_TOLSET_TS`	`XSLP_TOLSETBIT_TS`
8	Relative slack tolerance (RS)	`XSLP_TOLSET_RS`	`XSLP_TOLSETBIT_RS`

The XSLP_TOLSET constants can be used to access the corresponding entry in the value arrays, while the XSLP_TOLSETBIT constants are used to set or retrieve which tolerance values are used for a given SLP variable.

Once created, a tolerance set can be used to set the tolerances for any SLP variable.

If a tolerance value is zero, then the default tolerance will be used instead. To force the use of a tolerance, use the `XSLPchgtolset` function and set the `Status` variable appropriately.

See the section "Convergence Criteria" for a fuller description of tolerances and their uses.

The `XSLPload...` functions load items into the SLP problem. Any existing items of the same type are deleted first. The corresponding `XSLPadd...` functions add or replace items leaving other items of the same type unchanged.

Related topics

`XSLPaddtolsets`, `XSLPdeltolsets`, `XSLPchgtolset`, `XSLPgettolset`

XSLPloadvars

Purpose

Load SLP variables defined as matrix columns into an SLP problem

Synopsis

```
int XPRS_CC XSLPloadvars(XSLPprob Prob, int nSLPVar, int *ColIndex, int
      *VarType, int *DetRow, int *SeqNum, int *TolIndex, double *InitValue,
      double *StepBound);
```

Arguments

`Prob`	The current SLP problem.
`nSLPVar`	The number of SLP variables to be loaded.
`ColIndex`	Integer array holding the index of the matrix column corresponding to each SLP variable.
`VarType`	Bitmap giving information about the SLP variable as follows: `Bit 1` Variable has a delta vector; `Bit 2` Variable has an initial value; `Bit 14` Variable is the reserved "=" column; May be `NULL` if not required.
`DetRow`	Integer array holding the index of the determining row for each SLP variable (a negative value means there is no determining row) May be `NULL` if not required.
`SeqNum`	Integer array holding the index sequence number for cascading for each SLP variable (a zero value means there is no pre-defined order for this variable) May be `NULL` if not required.
`TolIndex`	Integer array holding the index of the tolerance set for each SLP variable (a zero value means the default tolerances are used) May be `NULL` if not required.
`InitValue`	Double array holding the initial value for each SLP variable (use the `VarType` bit map to indicate if a value is being provided) May be `NULL` if not required.
`StepBound`	Double array holding the initial step bound size for each SLP variable (a zero value means that no initial step bound size has been specified). If a value of `XPRS_PLUSINFINITY` is used for a value in `StepBound`, the delta will never have step bounds applied, and will almost always be regarded as converged. May be `NULL` if not required.

Example

The following example loads two SLP variables into the problem. They correspond to columns 23 and 25 of the underlying LP problem. Column 25 has an initial value of 1.42; column 23 has no specific initial value

```
int ColIndex[2], VarType[2];
double InitValue[2];

ColIndex[0] = 23; VarType[0] = 0;
ColIndex[1] = 25; Vartype[1] = 2; InitValue[1] = 1.42;

XSLPloadvars(Prob, 2, ColIndex, VarType, NULL, NULL,
             NULL, InitValue, NULL);
```

`InitValue` is not set for the first variable, because it is not used (`VarType` = 0). Bit 1 of `VarType` is set for the second variable to indicate that the initial value has been set.

The arrays for determining rows, sequence numbers, tolerance sets and step bounds are not used at all, and so have been passed to the function as `NULL`.

Further information

The `XSLPload...` functions load items into the SLP problem. Any existing items of the same type are deleted first. The corresponding `XSLPadd...` functions add or replace items leaving other items of the same type unchanged.

Related topics

`XSLPaddvars`, `XSLPchgvar`, `XSLPdelvars`, `XSLPgetvar`

XSLPmaxim

Purpose

Maximize an SLP problem

Synopsis

```
int XPRS_CC XSLPmaxim(XSLPprob Prob, char *Flags);
```

Arguments

`Prob` The current SLP problem.

`Flags` These have the same meaning as for `XPRSmaxim`.

Example

The following example reads an SLP problem from file and then maximizes it using the primal simplex optimizer.

```
XSLPreadprob("Matrix","");
XSLPmaxim(Prob,"p");
```

Related controls

Integer

`XSLP_ALGORITHM` Bit map determining the SLP algorithm(s) used in the optimization.

`XSLP_AUGMENTATION` Bit map determining the type of augmentation used to create the linearization.

`XSLP_CASCADE` Bit map determining the type of cascading (recalculation of SLP variable values) used during the SLP optimization.

`XSLP_LOG` Determines the amount of iteration logging information produced.

`XSLP_PRESOLVE` Bit map determining the type of nonlinear presolve used before the SLP optimization starts.

Further information

If `XSLPconstruct` has not already been called, it will be called first, using the augmentation defined by the control variable `XSLP_AUGMENTATION`. If determining rows are provided, then cascading will be invoked in accordance with the setting of the control variable `XSLP_CASCADE`.

Related topics

`XSLPconstruct`, `XSLPglobal`, `XSLPminim`, `XSLPnlpoptimize`, `XSLPpresolve`

XSLPminim

Purpose

Minimize an SLP problem

Synopsis

```
int XPRS_CC XSLPminim(XSLPprob Prob, char *Flags);
```

Arguments

`Prob` The current SLP problem.

`Flags` These have the same meaning as for `XPRSminim`.

Example

The following example reads an SLP problem from file and then minimizes it using the Newton barrier optimizer.

```
XSLPreadprob("Matrix","");
XSLPminim(Prob,"b");
```

Related controls

Integer

`XSLP_ALGORITHM` Bit map determining the SLP algorithm(s) used in the optimization.

`XSLP_AUGMENTATION` Bit map determining the type of augmentation used to create the linearization.

`XSLP_CASCADE` Bit map determining the type of cascading (recalculation of SLP variable values) used during the SLP optimization.

`XSLP_LOG` Determines the amount of iteration logging information produced.

`XSLP_PRESOLVE` Bit map determining the type of nonlinear presolve used before the SLP optimization starts.

Further information

If `XSLPconstruct` has not already been called, it will be called first, using the augmentation defined by the control variable `XSLP_AUGMENTATION`. If determining rows are provided, then cascading will be invoked in accordance with the setting of the control variable `XSLP_CASCADE`.

Related topics

`XSLPconstruct`, `XSLPglobal`, `XSLPmaxim`, `XSLPnlpoptimize`, `XSLPpresolve`

XSLPmsaddcustompreset

Purpose

A combined version of XSLPmsaddjob and XSLPmsaddpreset. The preset described is loaded, topped up with the specific settings supplied

Synopsis

```
int XSLP_CC XSLPmsaddcustompreset( XSLPprob Prob, const char *sDescription,
       const int Preset, const int Count, const int nIVs, const int *IVCols,
       const double *IVValues, const int nIntControls, const int
       *IntControlIndices, const int *IntControlValues, const int
       nDblControls, const int *DblControlIndices, const double
       *DblControlValues, void *pJobObject);
```

Arguments

`Prob` The current SLP problem.

`sDescription` Text description of the job. Used for messaging, may be NULL if not required.

`Preset` Which preset to load.

`Count` Maximum number of jobs to be added to the multistart pool.

`nIVs` Number of initial values to set.

`IVCols` Indices of the variables for which to set an initial value. May be NULL if nIVs is zero.

`IVValues` Initial values for the variables for which to set an initial value. May be NULL if nIVs is zero.

`nIntControls` Number of integer controls to set.

`IntControlIndices` The indices of the integer controls to be set. May be NULL if nIntControls is zero.

`IntControlValues` The values of the integer controls to be set. May be NULL if nIntControls is zero.

`nDblControls` Number of double controls to set.

`DblControlIndices` The indices of the double controls to be set. May be NULL if nDblControls is zero.

`DblControlValues` The values of the double controls to be set. May be NULL if nDblControls is zero.

`pJobObject` Job specific user context pointer to be passed to the multistart callbacks.

Further information

This function allows for repeatedly calling the same multistart preset (e.g. initial values) using different basic controls.

Related topics

`XSLPmsaddpreset`, `XSLPmsaddjob`, `XSLPmsclear`

XSLPmsaddjob

Purpose

Adds a multistart job to the multistart pool

Synopsis

```
int XSLP_CC XSLPmsaddjob( XSLPprob Prob, const char *sDescription, const
      int nIVs, const int *IVCols, const double *IVValues, const int
      nIntControls, const int *IntControlIndices, const int
      *IntControlValues, const int nDblControls, const int
      *DblControlIndices, const double *DblControlValues, void
      *pJobObject);
```

Arguments

`Prob` The current SLP problem.

`sDescription` Text description of the job. Used for messaging, may be NULL if not required.

`nIVs` Number of initial values to set.

`IVCols` Indices of the variables for which to set an initial value. May be NULL if nIVs is zero.

`IVValues` Initial values for the variables for which to set an initial value. May be NULL if nIVs is zero.

`nIntControls` Number of integer controls to set.

`IntControlIndices` The indices of the integer controls to be set. May be NULL if nIntControls is zero.

`IntControlValues` The values of the integer controls to be set. May be NULL if nIntControls is zero.

`nDblControls` Number of double controls to set.

`DblControlIndices` The indices of the double controls to be set. May be NULL if nDblControls is zero.

`DblControlValues` The values of the double controls to be set. May be NULL if nDblControls is zero.

`pJobObject` Job specific user context pointer to be passed to the multistart callbacks.

Further information

Adds a mutistart job, applying the specified initial point and option combinations on top of the base problem, i.e. the options and initial values specified to the function is applied on top of the existing settigns.

This function allows for loading empty template jobs, that can then be identified using the pJobObject variable.

Related topics

`XSLPmsaddpreset`, `XSLPmsaddcustompreset`, `XSLPmsclear`

XSLPmsaddpreset

Purpose

Loads a preset of jobs into the multistart job pool.

Synopsis

```
int XSLP_CC XSLPmsaddpreset( XSLPprob Prob, const char *sDescription, const
      int Preset, const int Count, void *pJobObject);
```

Arguments

`Prob`	The current SLP problem.
`sDescription`	Text description of the preset. Used for messaging, may be NULL if not required.
`Preset`	Which preset to load.
`Count`	Maximum number of jobs to be added to the multistart pool.
`pJobObject`	Job specific user context pointer to be passed to the multistart callbacks.

Further information

The following presets are defined:

`XSLP_MSSET_INITIALVALUES`: generate Count number of random base points.

`XSLP_MSPRESET_SOLVERS`: load all solvers.

`XSLP_MSPRESET_SLPCONTROLSBASIC`: load the most typical SLP tuning settings. A maximum of Count jobs are loaded.

`XSLP_MSPRESET_SLPCONROLSEXTENSIVE`: load a comprehensive set of SLP tuning settings. A maximum of Count jobs are loaded.

`XSLP_MSPRESET_KNITROBASIC`: load the most typical Knitro tuning settings. A maximum of Count jobs are loaded.

`XSLP_MSPRESET_KNITROEXTENSIVE`: load a comprehensive set of Knitro tuning settings. A maximum of Count jobs are loaded.

`XSLP_MSSET_INITIALFILTERED`: generate Count number of random base points, filtered by a merit function centred on initial feasibility.

`XSLP_MSSET_INITIALDYNAMIC`: generate Count number of random base points, that are then refined and combined further by any solution found during the search.

See `XSLP_MSMAXBOUNDRANGE` for controlling the range in which initial values are generated.

Related topics

`XSLPmsaddjob`, `XSLPmsaddcustompreset`, `XSLPmsclear`

XSLPmsclear

Purpose

Removes all scheduled jobs from the multistart job pool

Synopsis

```
int XSLP_CC XSLPmsclear( XSLPprob Prob);
```

Argument

`Prob` The current SLP problem.

Related topics

`XSLPmsaddjob`, `XSLPmsaddpreset`, `XSLPmsaddcustompreset`

XSLPnlpoptimize

Purpose

Maximize or minimize an SLP problem

Synopsis

```
int XPRS_CC XSLPnlpoptimize(XSLPprob Prob, char *Flags);
```

Arguments

`Prob` The current SLP problem.

`Flags` These have the same meaning as for `XPRSmaxim` and `XPRSminim`.

Related controls

Double

`XSLP_OBJSENSE` Determines the direction of optimization: +1 is for minimization, -1 is for maximization.

Integer

`XSLP_ALGORITHM` Bit map determining the SLP algorithm(s) used in the optimization.

`XSLP_AUGMENTATION` Bit map determining the type of augmentation used to create the linearization.

`XSLP_CASCADE` Bit map determining the type of cascading (recalculation of SLP variable values) used during the SLP optimization.

`XSLP_LOG` Determines the amount of iteration logging information produced.

`XSLP_PRESOLVE` Bit map determining the type of nonlinear presolve used before the SLP optimization starts.

Further information

`XSLPnlpoptimize` is equivalent to `XSLPmaxim` (if `XSLP_OBJSENSE` = -1) or `XSLPminim` (if `XSLP_OBJSENSE` = +1).

If `XSLPconstruct` has not already been called, it will be called first, using the augmentation defined by the control variable `XSLP_AUGMENTATION`. If determining rows are provided, then cascading will be invoked in accordance with the setting of the control variable `XSLP_CASCADE`.

Related topics

`XSLPconstruct`, `XSLPglobal`, `XSLPmaxim`, `XSLPminim`, `XSLPpresolve`

XSLPpostsolve

Purpose

Restores the problem to its pre-solve state

Synopsis

```
int XPRS_CC XSLPpostsolve(XSLPprob Prob);
```

Argument

`Prob` The current SLP problem.

Related controls

Integer

`XSLP_POSTSOLVE` Determines if postsolve is applied automatically.

Further information

If Xpress-SLP was used to solve the problem, postsolve will unconstruct the problem before postsolving (including any reformulation that might have been applied).

Related topics

`XSLP_POSTSOLVE`

XSLPpresolve

Purpose

Perform a nonlinear presolve on the problem

Synopsis

```
int XPRS_CC XSLPpresolve(XSLPprob Prob);
```

Argument

`Prob` The current SLP problem.

Example

The following example reads a problem from file, sets the presolve control, presolves the problem and then maximizes it.

```
XSLPreadprob(Prob, "Matrix", "");
XSLPsetintcontrol(Prob, XSLP_PRESOLVE, 1);
XSLPpresolve(Prob);
XSLPmaximize(Prob,"");
```

Related controls

Integer

`XSLP_PRESOLVE` Bitmap containing nonlinear presolve options.

Further information

If bit 1 of `XSLP_PRESOLVE` is not set, no nonlinear presolve will be performed. Otherwise, the presolve will be performed in accordance with the bit settings.. `XSLPpresolve` is called automatically by `XSLPconstruct`, so there is no need to call it explicitly unless there is a requirement to interrupt the process between presolve and optimization. `XSLPpresolve` must be called before `XSLPconstruct` or any of the SLP optimization procedures..

Related topics

`XSLP_PRESOLVE`

XSLPprintmemory

Purpose

Print the dimensions and memory allocations for a problem

Synopsis

```
int XPRS_CC XSLPprintmemory(XSLPprob prob);
```

Argument

`Prob` The current SLP problem.

Example

The following example loads a problem from file and then prints the dimensions of the arrays.

```
XSLPreadprob(Prob, "Matrix1", "");
XSLPprintmemory(Prob);
```

The output is similar to the following:

```
Arrays and dimensions:
Array    Item  Used  Max  Allocated   Memory
         Size Items Items    Memory   Control
MemList    28   103   129        4K
String      1  8779 13107       13K   XSLP_MEM_STRING
Xv         16     2  1000       16K   XSLP_MEM_XV
Xvitem     48    11  1000       47K   XSLP_MEM_XVITEM
....
```

Further information

`XSLPprintmemory` lists the current sizes and amounts used of the variable arrays in the current problem. For each array, the size of each item, the number used and the number allocated are shown, together with the size of memory allocated and, where appropriate, the name of the memory control variable to set the array size. Loading and execution of some problems can be speeded up by setting the memory controls immediately after the problem is created. If an array has to be moved to re-allocate it with a larger size, there may be insufficient memory to hold both the old and new versions; pre-setting the memory controls reduces the number of such re-allocations which take place and may allow larger problems to be solved.

XSLPprintevalinfo

Purpose

Print a summary of any evaluation errors that may have occurred during solving a problem

Synopsis

```
int XPRS_CC XSLPprintevalinfo(XSLPprob prob);
```

Argument

`Prob` The current SLP problem.

Related topics

`XSLPsetcbcoefevalerror`

XSLPreadprob

Purpose

Read an Xpress NonLinear extended MPS format matrix from a file into an SLP problem

Synopsis

```
int XPRS_CC XSLPreadprob(XSLPprob Prob, char *Probname, char *Flags);
```

Arguments

`Prob`	The current SLP problem.
`Probname`	Character string containing the name of the file from which the matrix is to be read.
`Flags`	Character string containing any flags needed for the input routine. No flag settings are currently recognized.

Example

The following example reads the problem from file "Matrix.mat".

```
XSLPreadprob(Prob, "Matrix", "");
```

Further information

`XSLPreadprob` tries to open the file with an extension of "mat" or, failing that, an extension of "mps". If both fail, the file name will be tried with no extension.

`XSLPreadprob` is capable to read most Ampl .nl files. To specify that a .nl file is to be read, provide the full filename including the .nl extension.

For details of the format of the file, see the section on Extended MPS format.

Related topics

Extended MPS format, `XSLPwriteprob`

XSLPremaxim

Purpose

Continue the maximization of an SLP problem

Synopsis

```
int XPRS_CC XSLPremaxim(XSLPprob Prob, char *Flags);
```

Arguments

`Prob`	The current SLP problem.
`Flags`	These have the same meaning as for `XSLPmaxim`.

Example

The following example optimizes the SLP problem for up to 10 SLP iterations. If it has not converged, it saves the file and continues for another 10.

```
int Status;

XSLPsetintcontrol(Prob, XSLP_ITERLIMIT, 10);
XSLPmaxim(Prob,"");
XSLPgetintattrib(Prob, XSLP_STATUS, &Status);
if (Status & XSLP_MAXSLPITERATIONS) {
  XSLPsave(Prob);
  XSLPsetintcontrol(Prob, XSLP_ITERLIMIT, 20);
  XSLPremaxim(Prob,"");
}
```

Further information

This allows Xpress NonLinear to continue the maximization of a problem after it has been terminated, without re-initializing any of the parameters. In particular, the iteration count will resume at the point where it previously stopped, and not at 1.

Related topics

`XSLPmaxim`, `XSLPreminim`

XSLPreminim

Purpose

Continue the minimization of an SLP problem

Synopsis

```
int XPRS_CC XSLPreminim(XSLPprob Prob, char *Flags);
```

Arguments

`Prob`	The current SLP problem.
`Flags`	These have the same meaning as for `XSLPminim`.

Example

The following example optimizes the SLP problem for up to 10 SLP iterations. If it has not converged, it saves the file and continues for another 10.

```
int Status;

XSLPsetintcontrol(Prob, XSLP_ITERLIMIT, 10);
XSLPminim(Prob,"");
XSLPgetintattrib(Prob, XSLP_STATUS, &Status);
if (Status & XSLP_MAXSLPITERATIONS) {
  XSLPsave(Prob);
  XSLPsetintcontrol(Prob, XSLP_ITERLIMIT, 20);
  XSLPreminim(Prob,"");
}
```

Further information

This allows Xpress NonLinear to continue the minimization of a problem after it has been terminated, without re-initializing any of the parameters. In particular, the iteration count will resume at the point where it previously stopped, and not at 1.

Related topics

`XSLPminim`, `XSLPremaxim`

XSLPrestore

Purpose

Restore the Xpress NonLinear problem from a file created by XSLPsave

Synopsis

```
int XPRS_CC XSLPrestore(XSLPprob Prob, char *Filename);
```

Arguments

Prob	The current SLP problem.
Filename	Character string containing the name of the problem which is to be restored.

Example

The following example restores a problem originally saved on file "MySave"

```
XSLPrestore(Prob, "MySave");
```

Further information

Normally XSLPrestore restores both the Xpress NonLinear problem and the underlying optimizer problem. If only the Xpress NonLinear problem is required, set the integer control variable XSLP_CONTROL appropriately.

The problem is saved into two files *save.svf* which is the optimizer save file, and *save.svx* which is the SLP save file. Both files are required for a full restore; only the svx file is required when the underlying optimizer problem is not being restored.

Related topics

XSLP_CONTROL, XSLPsave

XSLPreinitialize

Purpose

Reset the SLP problem to match a just augmented system

Synopsis

```
int XPRS_CC XSLPreinitialize(XSLPprob Prob);
```

Argument

`Prob` The current SLP problem.

Further information

Can be used to rerun the SLP optimization process with updated parameters, penalties or initial values, but unchanged augmentation.

Related topics

`XSLPcreateprob`, `XSLPdestroyprob`, `XSLPunconstruct`, `XSLPsetcurrentiv`,

XSLPsave

Purpose

Save the Xpress NonLinear problem to file

Synopsis

```
int XPRS_CC XSLPsave(XSLPprob Prob);
```

Argument

`Prob` The current SLP problem.

Example

The following example saves the current problem to files named *prob1.svf* and *prob1.svx*.

```
XPRSprob xprob;
XSLPgetptrattrib(Prob, XSLP_XPRSPROBLEM, &xprob);
XPRSsetprobname(xprob, "prob1");
XSLPsave(Prob);
```

Further information

The problem is saved into two files *prob.svf* which is the optimizer save file, and *prob.svx* which is the SLP save file, where *prob* is the name of the problem. Both files are used in a full save; only the svx file is required when the underlying optimizer problem is not being saved.

Normally `XSLPsave` saves both the Xpress NonLinear problem and the underlying optimizer problem. If only the Xpress NonLinear problem is required, set the integer control variable `XSLP_CONTROL` appropriately.

Related topics

`XSLP_CONTROL`, `XSLPrestore` `XSLPsaveas`

XSLPsaveas

Purpose

Save the Xpress NonLinear problem to a named file

Synopsis

```
int XPRS_CC XSLPsaveas(XSLPprob Prob, const char *Filename);
```

Arguments

`Prob`	The current SLP problem.
`Filename`	The name of the file (without extension) in which the problem is to be saved.

Example

The following example saves the current problem to files named *MyProb.svf* and *MyProb.svx*.

```
XSLPsaveas(Prob,"MyProb");
```

Further information

The problem is saved into two files *filename.svf* which is the optimizer save file, and *filename.svx* which is the SLP save file, where *filename* is the second argument to the function. Both files are used in a full save; only the svx file is required when the underlying optimizer problem is not being saved.

Normally `XSLPsaveas` saves both the Xpress NonLinear problem and the underlying optimizer problem. If only the Xpress NonLinear problem is required, set the integer control variable `XSLP_CONTROL` appropriately.

Related topics

`XSLP_CONTROL`, `XSLPrestore` `XSLPsave`

XSLPscaling

Purpose

Analyze the current matrix for largest/smallest coefficients and ratios

Synopsis

```
int XPRS_CC XSLPscaling(XSLPprob Prob);
```

Argument

`Prob` The current SLP problem.

Example

The following example analyzes the matrix

```
XSLPscaling(Prob);
```

Further information

The current matrix (including augmentation if it has been carried out) is scanned for the absolute and relative sizes of elements. The following information is reported:

- Largest and smallest elements in the matrix;
- Counts of the ranges of row ratios in powers of 10 (e.g. number of rows with ratio between 1.0E+01 and 1.0E+02);
- List of the rows (with largest and smallest elements) which appear in the highest range;
- Counts of the ranges of column ratios in powers of 10 (e.g. number of columns with ratio between 1.0E+01 and 1.0E+02);
- List of the columns (with largest and smallest elements) which appear in the highest range;
- Element ranges in powers of 10 (e.g. number of elements between 1.0E+01 and 1.0E+02).

Where any of the reported items (largest or smallest element in the matrix or any reported row or column element) is in a penalty error vector, the results are repeated, excluding all penalty error vectors.

XSLPsetcbcascadeend

Purpose

Set a user callback to be called at the end of the cascading process, after the last variable has been cascaded

Synopsis

```
int XPRS_CC XSLPsetcbcascadeend(XSLPprob Prob, int (XPRS_CC *UserFunc)
       (XSLPprob myProb, void *myObject), void *Object);
```

Arguments

`Prob`	The current SLP problem.
`UserFunc`	The function to be called at the end of the cascading process. `UserFunc` returns an integer value. The return value is noted by Xpress-SLP but it has no effect on the optimization.
`myProb`	The problem passed to the callback function.
`myObject`	The user-defined object passed as `Object` to `XSLPsetcbcascadeend`.
`Object`	Address of a user-defined object, which can be used for any purpose by the function. `Object` is passed to `UserFunc` as `myObject`.

Example

The following example sets up a callback to be executed at the end of the cascading process which checks if any of the values have been changed significantly:

```
double *cSol;
XSLPsetcbcascadeend(Prob, CBCascEnd, &cSol);
```

A suitable callback function might resemble this:

```
int XPRS_CC CBCascEnd(XSLPprob MyProb, void *Obj) {
  int iCol, nCol;
  double *cSol, Value;
  cSol = * (double **) Obj;
  XSLPgetintcontrol(MyProb, XPRS_COLS, &nCol);
  for (iCol=0;iCol<nCol;iCol++) {
    XSLPgetvar(MyProb, iCol, NULL, NULL, NULL,
               NULL, NULL, NULL, &Value,
               NULL, NULL, NULL, NULL,
               NULL, NULL, NULL, NULL);
    if (fabs(Value-cSol[iCol]) > .01)
      printf("\nCol %d changed from %lg to %lg",
             iCol, cSol[iCol], Value);
  }
  return 0;
}
```

The `Object` argument is used here to hold the address of the array `cSol` which we assume has been populated with the original solution values.

Further information

This callback can be used at the end of the cascading, when all the solution values have been recalculated.

Related topics

`XSLPcascade`, `XSLPsetcbcascadestart`, `XSLPsetcbcascadevar`, `XSLPsetcbcascadevarfail`

XSLPsetcbcascadestart

Purpose

Set a user callback to be called at the start of the cascading process, before any variables have been cascaded

Synopsis

```
int XPRS_CC XSLPsetcbcascadestart(XSLPprob Prob, int (XPRS_CC *UserFunc)
       (XSLPprob myProb, void *myObject), void *Object);
```

Arguments

`Prob`	The current SLP problem.
`UserFunc`	The function to be called at the start of the cascading process. `UserFunc` returns an integer value. If the return value is nonzero, the cascading process will be omitted for the current SLP iteration, but the optimization will continue.
`myProb`	The problem passed to the callback function.
`myObject`	The user-defined object passed as `Object` to `XSLPsetcbcascadestart`.
`Object`	Address of a user-defined object, which can be used for any purpose by the function. `Object` is passed to `UserFunc` as `myObject`.

Example

The following example sets up a callback to be executed at the start of the cascading process to save the current values of the variables:

```
double *cSol;
XSLPsetcbcascadestart(Prob, CBCascStart, &cSol);
```

A suitable callback function might resemble this:

```
int XPRS_CC CBCascStart(XSLPprob MyProb, void *Obj) {
  int iCol, nCol;
  double *cSol;
  cSol = * (double **) Obj;
  XSLPgetintcontrol(MyProb, XPRS_COLS, &nCol);
  for (iCol=0;iCol<nCol;iCol++) {
    XSLPgetvar(MyProb, iCol, NULL, NULL, NULL,
               NULL, NULL, NULL, &cSol[iCol],
               NULL, NULL, NULL, NULL,
               NULL, NULL, NULL, NULL);
  }
  return 0;
}
```

The `Object` argument is used here to hold the address of the array `cSol` which we populate with the solution values.

Further information

This callback can be used at the start of the cascading, before any of the solution values have been recalculated.

Related topics

`XSLPcascade`, `XSLPsetcbcascadeend`, `XSLPsetcbcascadevar`, `XSLPsetcbcascadevarfail`

XSLPsetcbcascadevar

Purpose

Set a user callback to be called after each column has been cascaded

Synopsis

```
int XPRS_CC XSLPsetcbcascadevar(XSLPprob Prob, int (XPRS_CC *UserFunc)
       (XSLPprob myProb, void *myObject, int ColIndex), void *Object);
```

Arguments

`Prob`	The current SLP problem.
`UserFunc`	The function to be called after each column has been cascaded. `UserFunc` returns an integer value. If the return value is nonzero, the cascading process will be omitted for the remaining variables during the current SLP iteration, but the optimization will continue.
`myProb`	The problem passed to the callback function.
`myObject`	The user-defined object passed as `Object` to `XSLPsetcbcascadevar`.
`ColIndex`	The number of the column which has been cascaded.
`Object`	Address of a user-defined object, which can be used for any purpose by the function. `Object` is passed to `UserFunc` as `myObject`.

Example

The following example sets up a callback to be executed after each variable has been cascaded:

```
double *cSol;
XSLPsetcbcascadevar(Prob, CBCascVar, &cSol);
```

The following sample callback function resets the value of the variable if the cascaded value is of the opposite sign to the original value:

```
int XPRS_CC CBCascVar(XSLPprob MyProb, void *Obj, int iCol) {
  double *cSol, Value;
  cSol = * (double **) Obj;
  XSLPgetvar(MyProb, iCol, NULL, NULL, NULL,
             NULL, NULL, NULL, &Value,
             NULL, NULL, NULL, NULL,
             NULL, NULL, NULL, NULL);
  if (Value * cSol[iCol] < 0) {
    Value = cSol[iCol];
    XSLPchgvar(MyProb, ColNum, NULL, NULL, NULL, NULL,
               NULL, NULL, &Value, NULL, NULL, NULL,
               NULL);
  }
  return 0;
}
```

The `Object` argument is used here to hold the address of the array `cSol` which we assume has been populated with the original solution values.

Further information

This callback can be used after each variable has been cascaded and its new value has been calculated.

Related topics

`XSLPcascade`, `XSLPsetcbcascadeend`, `XSLPsetcbcascadestart`, `XSLPsetcbcascadevarfail`

XSLPsetcbcascadevarfail

Purpose

Set a user callback to be called after cascading a column was not successful

Synopsis

```
int XPRS_CC XSLPsetcbcascadevarfail(XSLPprob Prob, int (XPRS_CC *UserFunc)
       (XSLPprob myProb, void *myObject, int ColIndex), void *Object);
```

Arguments

`Prob`	The current SLP problem.
`UserFunc`	The function to be called after cascading a column was not successful. `UserFunc` returns an integer value. If the return value is nonzero, the cascading process will be omitted for the remaining variables during the current SLP iteration, but the optimization will continue.
`myProb`	The problem passed to the callback function.
`myObject`	The user-defined object passed as `Object` to `XSLPsetcbcascadevarfail`.
`ColIndex`	The number of the column which has been cascaded.
`Object`	Address of a user-defined object, which can be used for any purpose by the function. `Object` is passed to `UserFunc` as `myObject`.

Further information

This callback can be used to provide user defined updates for SLP variables having a determining row that were not successfully cascaded due to the determining row being close to singular around the current values. This callback will always be called in place of the cascadevar callback in such cases, and in no situation will both the cascadevar and the cascadevarfail callback be called in the same iteration for the same variable.

Related topics

`XSLPcascade`, `XSLPsetcbcascadeend`, `XSLPsetcbcascadestart`, `XSLPsetcbcascadevar`

XSLPsetcbcoefevalerror

Purpose

Set a user callback to be called when an evaluation of a coefficient fails during the solve

Synopsis

```
int XPRS_CC XSLPsetcbcoefevalerror(XSLPprob Prob, int (XPRS_CC *UserFunc)
      (XSLPprob myProb, void *myObject, int RowIndex, int ColIndex), void
      *Object);
```

Arguments

`Prob`	The current SLP problem.
`UserFunc`	The function to be called when an evaluation fails.
`myProb`	The problem passed to the callback function.
`myObject`	The user-defined object passed as `Object` to `XSLPsetcbcoefevalerror`.
`RowIndex`	The row position of the coefficient.
`ColIndex`	The column position of the coefficient.
`Object`	Address of a user-defined object, which can be used for any purpose by the function. `Object` is passed to `UserFunc` as `myObject`.

Further information

This callback can be used to capture when an evaluation of a coefficient fails. The callback is called only once for each coefficient.

Related topics

`XSLPprintevalinfo`

XSLPsetcbconstruct

Purpose

Set a user callback to be called during the Xpress-SLP augmentation process

Synopsis

```
int XPRS_CC XSLPsetcbconstruct(XSLPprob Prob, int (XPRS_CC *UserFunc)
       (XSLPprob myProb, void *myObject), void *Object);
```

Arguments

`Prob`	The current SLP problem.
`UserFunc`	The function to be called during problem augmentation. `UserFunc` returns an integer value. See below for an explanation of the values.
`myProb`	The problem passed to the callback function.
`myObject`	The user-defined object passed as `Object` to `XSLPsetcbconstruct`.
`Object`	Address of a user-defined object, which can be used for any purpose by the function. `Object` is passed to `UserFunc` as `myObject`.

Example

The following example sets up a callback to be executed during the Xpress-SLP problem augmentation:

```
double *cValue;
cValue = NULL;
XSLPsetcbconstruct(Prob, CBConstruct, &cValue);
```

The following sample callback function sets values for the variables the first time the function is called and returns to `XSLPconstruct` to recalculate the initial matrix. The second time it is called it frees the allocated memory and returns to `XSLPconstruct` to proceed with the rest of the augmentation.

```
int XPRS_CC CBConstruct(XSLPprob MyProb, void *Obj) {
  double *cValue;
  int i, n;
/* if Object is NULL, this is first-time entry */
  if (*(void**)Obj == NULL) {
    XSLPgetintattrib(MyProb,XPRS_COLS,&n);
    cValue = malloc(n*sizeof(double));
/* ... initialize with values (not shown here) and then ... */
    for (i=0;i<n;i++)
/* store into SLP structures */
      XSLPchgvar(MyProb, i, NULL, NULL, NULL, NULL,
                 NULL, NULL, &cValue[i], NULL, NULL, NULL,
                 NULL);
/* set Object non-null to indicate we have processed data */
    *(void**)Obj = cValue;
    return -1;
  }
  else {
/* free memory, clear marker and continue */
    free(*(void**)Obj);
    *(void**)Obj = NULL;
  }
  return 0;
}
```

Further information

This callback can be used during the problem augmentation, generally (although not exclusively) to change the initial values for the variables.

The following return codes are accepted:

`0`	Normal return: augmentation continues
`-1`	Return to recalculate matrix values
`-2`	Return to recalculate row weights and matrix entries
`other`	Error return: augmentation terminates, `XSLPconstruct` terminates with a nonzero error code.

The return values -1 and -2 will cause the callback to be called a second time after the matrix has been recalculated. It is the responsibility of the callback to ensure that it does ultimately exit with a return value of zero.

Related topics

`XSLPconstruct`

XSLPsetcbdestroy

Purpose

Set a user callback to be called when an SLP problem is about to be destroyed

Synopsis

```
int XPRS_CC XSLPsetcbdestroy(XSLPprob Prob, int (XPRS_CC *UserFunc)
       (XSLPprob myProb, void *myObject), void *Object);
```

Arguments

`Prob`	The current SLP problem.
`UserFunc`	The function to be called when the SLP problem is about to be destroyed. `UserFunc` returns an integer value. At present the return value is ignored.
`myProb`	The problem passed to the callback function.
`myObject`	The user-defined object passed as `Object` to `XSLPsetcbdestroy`.
`Object`	Address of a user-defined object, which can be used for any purpose by the function. `Object` is passed to `UserFunc` as `myObject`.

Example

The following example sets up a callback to be executed before the SLP problem is destroyed:

```
double *cSol;
XSLPsetcbdestroy(Prob, CBDestroy, &cSol);
```

The following sample callback function frees the memory associated with the user-defined object:

```
int XPRS_CC CBDestroy(XSLPprob MyProb, void *Obj) {
  if (*(void**)Obj) free(*(void**)Obj);
  return 0;
}
```

The `Object` argument is used here to hold the address of the array `cSol` which we assume was assigned using one of the `malloc` functions.

Further information

This callback can be used when the problem is about to be destroyed to free any user-defined resources which were allocated during the life of the problem.

Related topics

`XSLPdestroyprob`

XSLPsetcbdrcol

Purpose

Set a user callback used to override the update of variables with small determining column

Synopsis

```
int XPRS_CC XSLPsetcbdrcol(XSLPprob Prob, int (XPRS_CC *UserFunc) (XSLPprob
      myProb, void *myObject, int ColIndex, int DrColIndex, double
      DrColValue, double * NewValue, double VLB, double VUB), void
      *Object);
```

Arguments

`Prob`	The current SLP problem.
`UserFunc`	The function to be called after each column has been cascaded. `UserFunc` returns an integer value. If the return value is positive, it will indicate that the value has been fixed, and cascading should be omitted for the variable. A negative value indicates that a previously fixed value has been relaxed. If no action is taken, a 0 return value should be used.
`myProb`	The problem passed to the callback function.
`myObject`	The user-defined object passed as `Object` to `XSLPsetcbcascadevar`.
`ColIndex`	The index of the column for which the determining columns is checked.
`DrColIndex`	The index of the determining column for the column that is being updated.
`DrColValue`	The value of the determining column in the current SLP iteration.
`NewValue`	Used to return the new value for column `ColIndex`, should it need to be updated, in which case the callback must return a positive value to indicate that this value should be used.
`VLB`	The original lower bound of column `ColIndex`. The callback provides this value as a reference, should the bound be updated or changed during the solution process.
`VUB`	The original upper bound of column `ColIndex`. The callback provides this value as a reference, should the bound be updated or changed during the solution process.
`Object`	Address of a user-defined object, which can be used for any purpose. by the function. `Object` is passed to `UserFunc` as `myObject`.

Further information

If set, this callback is called as part of the cascading procedure. Please see Chapter *Cascading* for more information.

Related topics

`XSLP_DRCOLTOL`, `XSLPcascade`, `XSLPsetcbcascadeend`, `XSLPsetcbcascadestart`

XSLPsetcbintsol

Purpose

Set a user callback to be called during MISLP when an integer solution is obtained

Synopsis

```
int XPRS_CC XSLPsetcbintsol(XSLPprob Prob, int (XPRS_CC *UserFunc)
      (XSLPprob myProb, void *myObject), void *Object);
```

Arguments

`Prob`	The current SLP problem.
`UserFunc`	The function to be called when an integer solution is obtained. `UserFunc` returns an integer value. At present, the return value is ignored.
`myProb`	The problem passed to the callback function.
`myObject`	The user-defined object passed as `Object` to `XSLPsetcbintsol`.
`Object`	Address of a user-defined object, which can be used for any purpose by the function. `Object` is passed to `UserFunc` as `myObject`.

Example

The following example sets up a callback to be executed whenever an integer solution is found during MISLP:

```
double *cSol;
XSLPsetcbintsol(Prob, CBIntSol, &cSol);
```

The following sample callback function saves the solution values for the integer solution just found:

```
int XPRS_CC CBIntSol(XSLPprob MyProb, void *Obj) {
  XPRSprob xprob;
  double *cSol;
  cSol = * (double **) Obj;
  XSLPgetptrattrib(MyProb, XSLP_XPRSPROBLEM, &xprob);
  XPRSgetsol(xprob, cSol, NULL, NULL, NULL);
  return 0;
}
```

The `Object` argument is used here to hold the address of the array `cSol` which we assume was assigned using one of the `malloc` functions.

Further information

This callback must be used during MISLP instead of the `XPRSsetcbintsol` callback which is used for MIP problems.

Related topics

`XSLPsetcboptnode`, `XSLPsetcbprenode`

XSLPsetcbiterend

Purpose

Set a user callback to be called at the end of each SLP iteration

Synopsis

```
int XPRS_CC XSLPsetcbiterend(XSLPprob Prob, int (XPRS_CC *UserFunc)
       (XSLPprob myProb, void *myObject), void *Object);
```

Arguments

`Prob`	The current SLP problem.
`UserFunc`	The function to be called at the end of each SLP iteration. `UserFunc` returns an integer value. If the return value is nonzero, the SLP iterations will stop.
`myProb`	The problem passed to the callback function.
`myObject`	The user-defined object passed as `Object` to `XSLPsetcbiterend`.
`Object`	Address of a user-defined object, which can be used for any purpose by the function. `Object` is passed to `UserFunc` as `myObject`.

Example

The following example sets up a callback to be executed at the end of each SLP iteration. It records the number of LP iterations in the latest optimization and stops if there were fewer than 10:

```
XSLPsetcbiterend(Prob, CBIterEnd, NULL);
```

A suitable callback function might resemble this:

```
int XPRS_CC CBIterEnd(XSLPprob MyProb, void *Obj) {
  int nIter;
  XPRSprob xprob;
  XSLPgetptrattrib(MyProb, XSLP_XPRSPROBLEM, &xprob);
  XSLPgetintattrib(xprob, XPRS_SIMPLEXITER, &nIter);
  if (nIter < 10) return 1;
  return 0;
}
```

The `Object` argument is not used here, and so is passed as `NULL`.

Further information

This callback can be used at the end of each SLP iteration to carry out any further processing and/or stop any further SLP iterations.

Related topics

`XSLPsetcbiterstart`, `XSLPsetcbitervar`

XSLPsetcbiterstart

Purpose

Set a user callback to be called at the start of each SLP iteration

Synopsis

```
int XPRS_CC XSLPsetcbiterstart(XSLPprob Prob, int (XPRS_CC *UserFunc)
       (XSLPprob myProb, void *myObject), void *Object);
```

Arguments

`Prob`	The current SLP problem.
`UserFunc`	The function to be called at the start of each SLP iteration. `UserFunc` returns an integer value. If the return value is nonzero, the SLP iterations will stop.
`myProb`	The problem passed to the callback function.
`myObject`	The user-defined object passed as `Object` to `XSLPsetcbiterstart`.
`Object`	Address of a user-defined object, which can be used for any purpose by the function. `Object` is passed to `UserFunc` as `myObject`.

Example

The following example sets up a callback to be executed at the start of the optimization to save to save the values of the variables from the previous iteration:

```
double *cSol;
XSLPsetcbiterstart(Prob, CBIterStart, &cSol);
```

A suitable callback function might resemble this:

```
int XPRS_CC CBIterStart(XSLPprob MyProb, void *Obj) {
  XPRSprob xprob;
  double *cSol;
  int nIter;
  cSol = * (double **) Obj;
  XSLPgetintattrib(MyProb, XSLP_ITER, &nIter);
  if (nIter == 0) return 0; /* no previous solution */
  XSLPgetptrattrib(MyProb, XSLP_XPRSPROBLEM, &xprob);
  XPRSgetsol(xprob, cSol, NULL, NULL, NULL);
  return 0;
}
```

The `Object` argument is used here to hold the address of the array `cSol` which we populate with the solution values.

Further information

This callback can be used at the start of each SLP iteration before the optimization begins.

Related topics

`XSLPsetcbiterend`, `XSLPsetcbitervar`

XSLPsetcbitervar

Purpose

Set a user callback to be called after each column has been tested for convergence

Synopsis

```
int XPRS_CC XSLPsetcbitervar(XSLPprob Prob, int (XPRS_CC *UserFunc)
       (XSLPprob myProb, void *myObject, int ColIndex), void *Object);
```

Arguments

`Prob` The current SLP problem.

`UserFunc` The function to be called after each column has been tested for convergence. `UserFunc` returns an integer value. The return value is interpreted as a convergence status. The possible values are:

`< 0`	The variable has not converged;
`0`	The convergence status of the variable is unchanged;
`1 to 10`	The column has converged on a system-defined convergence criterion (these values should not normally be returned);
`> 10`	The variable has converged on user criteria.

`myProb` The problem passed to the callback function.

`myObject` The user-defined object passed as `Object` to `XSLPsetcbitervar`.

`ColIndex` The number of the column which has been tested for convergence.

`Object` Address of a user-defined object, which can be used for any purpose by the function. `Object` is passed to `UserFunc` as `myObject`.

Example

The following example sets up a callback to be executed after each variable has been tested for convergence. The user object `Important` is an integer array which has already been set up and holds a flag for each variable indicating whether it is important that it converges.

```
int *Important;
XSLPsetcbitervar(Prob, CBIterVar, &Important);
```

The following sample callback function tests if the variable is already converged. If not, then it checks if the variable is important. If it is not important, the function returns a convergence status of 99.

```
int XPRS_CC CBIterVar(XSLPprob MyProb, void *Obj, int iCol) {
  int *Important, Converged;
  Important = *(int **) Obj;
  XSLPgetvar(MyProb, iCol, NULL, NULL, NULL,
             NULL, NULL, NULL, NULL,
             NULL, NULL, &Converged, NULL,
             NULL, NULL, NULL, NULL);
  if (Converged) return 0;
  if (!Important[iCol]) return 99;
  return -1;
}
```

The `Object` argument is used here to hold the address of the array `Important`.

Further information

This callback can be used after each variable has been checked for convergence, and allows the convergence status to be reset if required.

Related topics

`XSLPsetcbiterend`, `XSLPsetcbiterstart`

XSLPsetcbmessage

Purpose

Set a user callback to be called whenever Xpress NonLinear outputs a line of text

Synopsis

```
int XPRS_CC XSLPsetcbmessage(XSLPprob Prob, void (XPRS_CC *UserFunc)
       (XSLPprob myProb, void *myObject, char *msg, int len, int msgtype),
       void *Object);
```

Arguments

`Prob`	The current SLP problem.
`UserFunc`	The function to be called whenever Xpress NonLinear outputs a line of text. `UserFunc` does not return a value.
`myProb`	The problem passed to the callback function.
`myObject`	The user-defined object passed as `Object` to `XSLPsetcbmessage`.
`msg`	Character buffer holding the string to be output.
`len`	Length in characters of `msg` excluding the null terminator.
`msgtype`	Type of message. The following are system-defined: `1` Information message `3` Warning message `4` Error message A negative value indicates that the Optimizer is about to finish and any buffers should be flushed at this time.
`Object`	Address of a user-defined object, which can be used for any purpose by the function. `Object` is passed to `UserFunc` as `myObject`.

Example

The following example creates a log file into which all messages are placed. System messages are also printed on standard output:

```
FILE *logfile;
logfile = fopen("myLog","w");
XSLPsetcbmessage(Prob, CBMessage, logfile);
```

A suitable callback function could resemble the following:

```
void XPRS_CC CBMessage(XSLPprob Prob, void *Obj,
                       char *msg, int len, int msgtype) {
  FILE *logfile;
  logfile = (FILE *) Obj;
  if (msgtype < 0) {
    fflush(stdout);
    if (logfile) fflush(logfile);
    return;
  }
  switch (msgtype) {
    case 1: /* information */
    case 3: /* warning */
    case 4: /* error */
      printf("%s\n",msg);
    default: /* user */
      if (logfile)
        fprintf(logfile,"%s\n",msg);
      break;
```

```
    }
    return;
}
```

Further information

If a user message callback is defined then screen output is automatically disabled.

Output can be directed into a log file by using `XSLPsetlogfile`.

Related topics

`XSLPsetlogfile`,

XSLPsetcbmsjobend

Purpose

Set a user callback to be called every time a new multistart job finishes. Can be used to overwrite the default solution ranking function

Synopsis

```
int XSLP_CC XSLPsetcbmsjobend(XSLPprob Prob, int (XSLP_CC
      *UserFunc)(XSLPprob myProb, void *myObject,void *pJobObject,const
      char *JobDescription,int *Status), void *Object);
```

Arguments

`Prob`	The current SLP problem.
`UserFunc`	The function to be called when a new multistart job is created
`myProb`	The problem passed to the callback function.
`myObject`	The user-defined object passed as `Object` to `XSLPsetcbmsjobend`.
`pJobObject`	Job specific user-defined object, as specified in by the multistart job creating API functions.
`JobDescription`	The description of the problem as specified in by the multistart job creating API functions.
`Status`	User return status variable: 0 - use the default evaluation of the finished job 1 - disregard the result and continue 2 - stop the multistart search

Further information

The multistart pool is dynamic, and this callback can be used to load new multistart jobs using the normal API functions.

Related topics

`XSLPsetcbmsjobstart`, `XSLPsetcbmswinner`

XSLPsetcbmsjobstart

Purpose

Set a user callback to be called every time a new multistart job is created, and the pre-loaded settings are applied

Synopsis

```
int XSLP_CC XSLPsetcbmsjobstart(XSLPprob Prob, int (XSLP_CC
      *UserFunc)(XSLPprob myProb, void *myObject,void *pJobObject,const
      char *JobDescription,int *Status), void *Object);
```

Arguments

`Prob`	The current SLP problem.
`UserFunc`	The function to be called when a new multistart job is created
`myProb`	The problem passed to the callback function.
`myObject`	The user-defined object passed as `Object` to `XSLPsetcbmsjobstart`.
`pJobObject`	Job specific user-defined object, as specified in by the multistart job creating API functions.
`JobDescription`	The description of the problem as specified in by the multistart job creating API functions.
`Status`	User return status variable: 0 - normal return, solve the job, 1 - disregard this job and continue, 2 - Stop multistart.

Further information

All mulit-start jobs operation on an independent copy of the original problem, and any modification to the problem is allowed, including structural changes. Please note however, that any modification will be carried over to the base problem, should a modified problem be declared the winner prob.

Related topics

`XSLPsetcbmsjobend`, `XSLPsetcbmswinner`

XSLPsetcbmswinner

Purpose

Set a user callback to be called every time a new multistart job is created, and the pre-loaded settings are applied

Synopsis

```
int XSLP_CC XSLPsetcbmswinner(XSLPprob Prob, int (XSLP_CC
      *UserFunc)(XSLPprob myProb, void *myObject,void *pJobObject,const
      char *JobDescription), void *Object);
```

Arguments

`Prob`	The current SLP problem.
`UserFunc`	The function to be called when a new multistart job is created
`myProb`	The problem passed to the callback function.
`myObject`	The user-defined object passed as `Object` to `XSLPsetcbmswinner`.
`pJobObject`	Job specific user-defined object, as specified in by the multistart job creating API functions.
`JobDescription`	The description of the problem as specified in by the multistart job creating API functions.

Further information

The multistart pool is dynamic, and this callback can be used to load new multistart jobs using the normal API functions.

Related topics

`XSLPsetcbmsjobstart`, `XSLPsetcbmsjobend`

XSLPsetcboptnode

Purpose

Set a user callback to be called during MISLP when an optimal SLP solution is obtained at a node

Synopsis

```
int XPRS_CC XSLPsetcboptnode(XSLPprob Prob, int (XPRS_CC *UserFunc)
       (XSLPprob myProb, void *myObject, int *feas), void *Object);
```

Arguments

`Prob`	The current SLP problem.
`UserFunc`	The function to be called when an optimal SLP solution is obtained at a node. `UserFunc` returns an integer value. If the return value is nonzero, or if the feasibility flag is set nonzero, then further processing of the node will be terminated (it is declared infeasible).
`myProb`	The problem passed to the callback function.
`myObject`	The user-defined object passed as `Object` to `XSLPsetcboptnode`.
`feas`	Address of an integer containing the feasibility flag. If `UserFunc` sets the flag nonzero, the node is declared infeasible.
`Object`	Address of a user-defined object, which can be used for any purpose by the function. `Object` is passed to `UserFunc` as `myObject`.

Example

The following example defines a callback function to be executed at each node when an SLP optimal solution is found. If there are significant penalty errors in the solution, the node is declared infeasible.

```
XSLPsetcboptnode(Prob, CBOptNode, NULL);
```

A suitable callback function might resemble the following:

```
int XPRS_CC CBOptNode(XSLPprob myProb, void *Obj, int *feas) {
  double Total, ObjVal;
  XSLPgetdblattrib(myProb, XSLP_ERRORCOSTS, &Total);
  XSLPgetdblattrib(myProb, XSLP_OBJVAL, &ObjVal);
  if (fabs(Total) > fabs(ObjVal) * 0.001 &&
    fabs(Total) > 1) *feas = 1;
  return 0;
```

Further information

If a node is declared infeasible from the callback function, the cost of exploring the node further will be avoided.

This callback must be used in place of `XPRSsetcboptnode` when optimizing with MISLP.

Related topics

`XSLPsetcbprenode`, `XSLPsetcbslpnode`

XSLPsetcbprenode

Purpose

Set a user callback to be called during MISLP after the set-up of the SLP problem to be solved at a node, but before SLP optimization

Synopsis

```
int XPRS_CC XSLPsetcbprenode(XSLPprob Prob, int (XPRS_CC *UserFunc)
       (XSLPprob myProb, void *myObject, int *feas), void *Object);
```

Arguments

`Prob`	The current SLP problem.
`UserFunc`	The function to be called after the set-up of the SLP problem to be solved at a node. `UserFunc` returns an integer value. If the return value is nonzero, or if the feasibility flag is set nonzero, then further processing of the node will be terminated (it is declared infeasible).
`myProb`	The problem passed to the callback function.
`myObject`	The user-defined object passed as `Object` to `XSLPsetcbprenode`.
`feas`	Address of an integer containing the feasibility flag. If `UserFunc` sets the flag nonzero, the node is declared infeasible.
`Object`	Address of a user-defined object, which can be used for any purpose by the function. `Object` is passed to `UserFunc` as `myObject`.

Example

The following example sets up a callback function to be executed at each node before the SLP optimization starts. The array `IntList` contains a list of integer variables, and the function prints the bounds on these variables.

```
int *IntList;
XSLPsetcbprenode(Prob, CBPreNode, IntList);
```

A suitable callback function might resemble the following:

```
int XPRS_CC CBPreNode(XSLPprob myProb, void *Obj, int *feas) {
  XPRSprob xprob;
  int i, *IntList;
  double LO, UP;
  IntList = (int *) Obj;
  XSLPgetptrattrib(myProb, XSLP_XPRSPROBLEM, &xprob);
  for (i=0; IntList[i]>=0; i++) {
    XPRSgetlb(xprob,&LO,IntList[i],IntList[i]);
    XPRSgetub(xprob,&UP,IntList[i],IntList[i]);
    if (LO > 0 || UP < XPRS_PLUSINFINITY)
      printf("\nCol %d: %lg <= %lg",LO,UP);
  }
  return 0;
}
```

Further information

If a node can be identified as infeasible by the callback function, then the initial optimization at the current node is avoided, as well as further exploration of the node.

This callback must be used in place of `XPRSsetcbprenode` when optimizing with MISLP.

Related topics

`XSLPsetcboptnode`, `XSLPsetcbslpnode`

XSLPsetcbpreupdatelinearization

Purpose

Set a user callback to be called before the linearization is updated

Synopsis

```
int XPRS_CC XSLPsetcbpreupdatelinearization(XSLPprob Prob, int (XPRS_CC
      *UserFunc) (XSLPprob myProb, void *myObject, int *ifRepeat), void
      *Object);
```

Arguments

`Prob`	The current SLP problem.
`UserFunc`	The function to be called at the end of the SLP optimization. `UserFunc` returns an integer value. If the return value is nonzero, the optimization will return an error code and the "User Return Code" error will be set.
`myProb`	The problem passed to the callback function.
`myObject`	The user-defined object passed as `Object` to
`ifRepeat`	If returned nonzero, SLP restart the lienarization update. `XSLPsetcbslpend`.
`Object`	Address of a user-defined object, which can be used for any purpose by the function. `Object` is passed to `UserFunc` as `myObject`.

Further information

This callback is intended to be used with user functions, allowing to peak where the functions will be evaluated, and then asked to redo the linearization. This is usefull for user functions returning their own partial derivatives implemented in a parallel setup. The callback is called again after the linearization is complete with ifRepeat being initialzied to -1, to indicate that any further evaluations are no longer part of updating the linearization.

XSLPsetcbslpend

Purpose

Set a user callback to be called at the end of the SLP optimization

Synopsis

```
int XPRS_CC XSLPsetcbslpend(XSLPprob Prob, int (XPRS_CC *UserFunc)
       (XSLPprob myProb, void *myObject), void *Object);
```

Arguments

`Prob`	The current SLP problem.
`UserFunc`	The function to be called at the end of the SLP optimization. `UserFunc` returns an integer value. If the return value is nonzero, the optimization will return an error code and the "User Return Code" error will be set.
`myProb`	The problem passed to the callback function.
`myObject`	The user-defined object passed as `Object` to `XSLPsetcbslpend`.
`Object`	Address of a user-defined object, which can be used for any purpose by the function. `Object` is passed to `UserFunc` as `myObject`.

Example

The following example sets up a callback to be executed at the end of the SLP optimization. It frees the memory allocated to the object created when the optimization began:

```
void *ObjData;
ObjData = NULL;
XSLPsetcbslpend(Prob, CBSlpEnd, &ObjData);
```

A suitable callback function might resemble this:

```
int XPRS_CC CBSlpEnd(XSLPprob MyProb, void *Obj) {
  void *ObjData;
  ObjData = * (void **) Obj;
  if (ObjData) free(ObjData);
  * (void **) Obj = NULL;
  return 0;
}
```

Further information

This callback can be used at the end of the SLP optimization to carry out any further processing or housekeeping before the optimization function returns.

Related topics

`XSLPsetcbslpstart`

XSLPsetcbslpnode

Purpose

Set a user callback to be called during MISLP after the SLP optimization at each node.

Synopsis

```
int XPRS_CC XSLPsetcbslpnode(XSLPprob Prob, int (XPRS_CC *UserFunc)
       (XSLPprob myProb, void *myObject, int *feas), void *Object);
```

Arguments

`Prob`	The current SLP problem.
`UserFunc`	The function to be called after the set-up of the SLP problem to be solved at a node. `UserFunc` returns an integer value. If the return value is nonzero, or if the feasibility flag is set nonzero, then further processing of the node will be terminated (it is declared infeasible).
`myProb`	The problem passed to the callback function.
`myObject`	The user-defined object passed as `Object` to `XSLPsetcbslpnode`.
`feas`	Address of an integer containing the feasibility flag. If `UserFunc` sets the flag nonzero, the node is declared infeasible.
`Object`	Address of a user-defined object, which can be used for any purpose by the function. `Object` is passed to `UserFunc` as `myObject`.

Example

The following example sets up a callback function to be executed at each node after the SLP optimization finishes. If the solution value is worse than a target value (referenced through the user object), the node is cut off (it is declared infeasible).

```
double OBJtarget;
XSLPsetcbslpnode(Prob, CBSLPNode, &OBJtarget);
```

A suitable callback function might resemble the following:

```
int XPRS_CC CBSLPNode(XSLPprob myProb, void *Obj, int *feas) {
  double TargetValue, LPValue;
  XSLPgetdblattrib(prob, XPRS_LPOBJVAL, &LPValue);
  TargetValue = * (double *) Obj;
  if (LPValue < TargetValue) *feas = 1;
  return 0;
}
```

Further information

If a node can be cut off by the callback function, then further exploration of the node is avoided.

Related topics

`XSLPsetcboptnode`, `XSLPsetcbprenode`

XSLPsetcbslpstart

Purpose

Set a user callback to be called at the start of the SLP optimization

Synopsis

```
int XPRS_CC XSLPsetcbslpstart(XSLPprob Prob, int (XPRS_CC *UserFunc)
       (XSLPprob myProb, void *myObject), void *Object);
```

Arguments

`Prob`	The current SLP problem.
`UserFunc`	The function to be called at the start of the SLP optimization. `UserFunc` returns an integer value. If the return value is nonzero, the optimization will not be carried out.
`myProb`	The problem passed to the callback function.
`myObject`	The user-defined object passed as `Object` to `XSLPsetcbslpstart`.
`Object`	Address of a user-defined object, which can be used for any purpose by the function. `Object` is passed to `UserFunc` as `myObject`.

Example

The following example sets up a callback to be executed at the start of the SLP optimization. It allocates memory to a user-defined object to be used during the optimization:

```
void *ObjData;
ObjData = NULL;
XSLPsetcbslpstart(Prob, CBSlpStart, &ObjData);
```

A suitable callback function might resemble this:

```
int XPRS_CC CBSlpStart(XSLPprob MyProb, void *Obj) {
  void *ObjData;
  ObjData = * (void **) Obj;
  if (ObjData) free(ObjData);
  * (void **) Obj = malloc(99*sizeof(double));
  return 0;
}
```

Further information

This callback can be used at the start of the SLP optimization to carry out any housekeeping before the optimization actually starts. Note that a nonzero return code from the callback will terminate the optimization immediately.

Related topics

`XSLPsetcbslpend`

XSLPsetcurrentiv

Purpose

Transfer the current solution to initial values

Synopsis

```
int XPRS_CC XSLPsetcurrentiv(XSLPprob Prob);
```

Argument

`Prob` The current SLP problem.

Further information

Provides a way to set the current iterates solution as initial values, make changes to parameters or to the underlying nonlinear problem and then rerun the SLP optimization process.

Related topics

`XSLPreinitialize`, `XSLPunconstruct`

XSLPsetdblcontrol

Purpose

Set the value of a double precision problem control

Synopsis

```
int XPRS_CC XSLPsetdblcontrol(XSLPprob Prob, int Param, double dValue);
```

Arguments

`Prob`	The current SLP problem.
`Param`	control (SLP or optimizer) whose value is to be returned.
`dValue`	Double precision value to be set.

Example

The following example sets the value of the Xpress NonLinear control `XSLP_CTOL` and of the optimizer control `XPRS_FEASTOL`:

```
XSLPsetdblcontrol(Prob, XSLP_CTOL, 0.001);
XSLPgetdblcontrol(Prob, XPRS_FEASTOL, 0.005);
```

Further information

Both SLP and optimizer controls can be set using this function. If an optimizer control is set, the return value will be the same as that from `XPRSsetdblcontrol`.

Related topics

`XSLPgetdblcontrol`, `XSLPsetintcontrol`, `XSLPsetstrcontrol`

XSLPsetdefaultcontrol

Purpose

Set the values of one SLP control to its default value

Synopsis

```
int XPRS_CC XSLPsetdefaultcontrol(XSLPprob Prob, int Param);
```

Arguments

`Prob`	The current SLP problem.
`Param`	The number of the control to be reset to its default.

Example

The following example reads a problem from file, sets the XSLP_LOG control, optimizes the problem and then reads and optimizes another problem using the default setting.

```
XSLPreadprob(Prob, "Matrix1", "");
XSLPsetintcontrol(Prob, XSLP_LOG, 4);
XSLPmaxim(Prob, "");
XSLPsetdefaultcontrol(Prob,XSLP_LOG);
XSLPreadprob(Prob, "Matrix2", "");
XSLPmaxim(Prob, "");
```

Further information

This function cannot reset the optimizer controls. Use `XPRSsetdefaults` or `XPRSsetdefaultcontrol`as well to reset optimizer controls to their default values.

Related topics

`XSLPsetdblcontrol`, `XSLPsetdefaults`, `XSLPsetintcontrol`, `XSLPsetstrcontrol`

XSLPsetdefaults

Purpose

Set the values of all SLP controls to their default values

Synopsis

```
int XPRS_CC XSLPsetdefaults(XSLPprob Prob);
```

Argument

`Prob` The current SLP problem.

Example

The following example reads a problem from file, sets some controls, optimizes the problem and then reads and optimizes another problem using the default settings.

```
XSLPreadprob(Prob, "Matrix1", "");
XSLPsetintcontrol(Prob, XSLP_LOG, 4);
XSLPsetdblcontrol(Prob, XSLP_CTOL, 0.001);
XSLPsetdblcontrol(Prob, XSLP_ATOL_A, 0.005);
XSLPmaxim(Prob, "");
XSLPsetdefaults(Prob);
XSLPreadprob(Prob, "Matrix2", "");
XSLPmaxim(Prob, "");
```

Further information

This function does not reset the optimizer controls. Use `XPRSsetdefaults` as well to reset all the controls to their default values.

Related topics

`XSLPsetdblcontrol`, `XSLPsetintcontrol`, `XSLPsetstrcontrol`

XSLPsetfunctionerror

Purpose

Set the function error flag for the problem

Synopsis

```
int XPRS_CC XSLPsetfunctionerror(XSLPprob Prob);
```

Argument

`Prob` The current SLP problem.

Further information

Once the function error has been set, calculations generally stop and the routines will return to their caller with a nonzero return code.

XSLPsetintcontrol

Purpose

Set the value of an integer problem control

Synopsis

```
int XPRS_CC XSLPsetintcontrol(XSLPprob Prob, int Param, int iValue);
```

Arguments

`Prob`	The current SLP problem.
`Param`	control (SLP or optimizer) whose value is to be returned.
`iValue`	The value to be set.

Example

The following example sets the value of the Xpress NonLinear control `XSLP_ALGORITHM` and of the optimizer control `XPRS_DEFAULTALG`:

```
XSLPsetintcontrol(Prob, XSLP_ALGORITHM, 934);
XSLPsetintcontrol(Prob, XPRS_DEFAULTALG, 3);
```

Further information

Both SLP and optimizer controls can be set using this function. If an optimizer control is requested, the return value will be the same as that from `XPRSsetintcontrol`.

Related topics

`XSLPgetintcontrol`, `XSLPsetdblcontrol`, `XSLPsetintcontrol`, `XSLPsetstrcontrol`

XSLPsetlogfile

Purpose

Define an output file to be used to receive messages from Xpress NonLinear

Synopsis

```
int XPRS_CC XSLPsetlogfile(XSLPprob Prob, char *Filename, int Option);
```

Arguments

`Prob`	The current SLP problem.
`FileName`	Character string containing the name of the file to be used for output.
`Option`	Option to indicate whether the output is directed to the file only (`Option`=0) or (in console mode) to the console as well (`Option`=1).

Example

The following example defines a log file "MyLog1" and directs output to the file and to the console:

```
XSLPsetlogfile(Prob, "MyLog1", 1);
```

Further information

If `Filename` is `NULL`, the current log file (if any) will be closed, and message handling will revert to the default mechanism.

Related topics

`XSLPsetcbmessage`

XSLPsetparam

Purpose

Set the value of a control parameter by name

Synopsis

```
int XPRS_CC XSLPsetparam(XSLPprob Prob, const char *Param, const char
      *cValue);
```

Arguments

`Prob`	The current SLP problem.
`Param`	Name of the control or attribute whose value is to be returned.
`cValue`	Character buffer containing the value.

Example

The following example sets the value of XSLP_ALGORITHM:

```
XSLPprob Prob;
int Algorithm;
char Buffer[32];
Algorithm = 934;
sprintf(Buffer,"%d",Algorithm);
XSLPsetparam(Prob, "XSLP_ALGORITHM", Buffer);
```

Further information

This function can be used to set any Xpress NonLinear or Optimizer control. The value is always passed as a character string. It is the user's responsibility to create the character string in an appropriate format.

Related topics

`XSLPsetdblcontrol`, `XSLPsetintcontrol`, `XSLPsetparam`, `XSLPsetstrcontrol`

XSLPsetstrcontrol

Purpose

Set the value of a string problem control

Synopsis

```
int XPRS_CC XSLPsetstrcontrol(XSLPprob Prob, int Param, const char
      *cValue);
```

Arguments

`Prob`	The current SLP problem.
`Param`	control (SLP or optimizer) whose value is to be returned.
`cValue`	Character buffer containing the value.

Example

The following example sets the value of the Xpress NonLinear control `XSLP_CVNAME` and of the optimizer control `XPRS_MPSOBJNAME`:

```
XSLPsetstrcontrol(Prob, XSLP_CVNAME, "CharVars");
XSLPsetstrcontrol(Prob, XPRS_MPSOBJNAME, "_OBJ_");
```

Further information

Both SLP and optimizer controls can be set using this function. If an optimizer control is requested, the return value will be the same as that from `XPRSsetstrcontrol`.

Related topics

`XSLPgetstrcontrol`, `XSLPsetdblcontrol`, `XSLPsetintcontrol`, `XSLPsetstrcontrol`

XSLPunconstruct

Purpose

Removes the augmentation and returns the problem to its pre-linearization state

Synopsis

```
int XPRS_CC XSLPunconstruct(XSLPprob Prob);
```

Argument

`Prob` The current SLP problem.

Further information

Only limited changes are allowed to an augmented problem.

Related topics

`XSLPconstruct`

XSLPupdatelinearization

Purpose

Updates the current linearization

Synopsis

```
int XPRS_CC XSLPupdatelinearization(XSLPprob Prob);
```

Argument

`Prob`	The current SLP problem.

Further information

Updates the augmented probem (the linearization) to match the current base point. The base point is the current SLP solution. The values of the SLP variables can be changed using `XSLPchgvar`.

The linearization must be present, and this function can only be called after the problem has been augmented by `XSLPconstruct`.

Related topics

`XSLPconstruct`

XSLPvalidate

Purpose

Validate the feasibility of constraints in a converged solution

Synopsis

```
int XPRS_CC XSLPvalidate(XSLPprob Prob);
```

Argument

`Prob` The current SLP problem.

Example

The following example sets the validation tolerance parameters, validates the converged solution and retrieves the validation indices.

```
double IndexA, IndexR;
XSLPsetdblcontrol(Prob, XSLP_VALIDATIONTOL_A, 0.001);
XSLPsetdblcontrol(Prob, XSLP_VALIDATIONTOL_R, 0.001);
XSLPvalidate(Prob);
XSLPgetdblattrib(Prob, XSLP_VALIDATIONINDEX_A, &IndexA);
XSLPgetdblattrib(Prob, XSLP_VALIDATIONINDEX_R, &IndexR);
```

Further information

`XSLPvalidate` checks the feasibility of a converged solution against relative and absolute tolerances for each constraint. The left hand side and the right hand side of the constraint are calculated using the converged solution values. If the calculated values imply that the constraint is infeasible, then the difference (*D*) is tested against the absolute and relative validation tolerances.
If $D <$ *XSLP_VALIDATIONTOL_A*
then the constraint is within the absolute validation tolerance. The total positive (*TPos*) and negative contributions (*TNeg*) to the left hand side are also calculated.
If $D <$ *MAX*(*ABS*(*TPos*),*ABS*(*TNeg*)) $*$ *XSLP_VALIDATIONTOL_R*
then the constraint is within the relative validation tolerance. For each constraint which is outside both the absolute and relative validation tolerances, validation factors are calculated which are the factors by which the infeasibility exceeds the corresponding validation tolerance; the smallest factor is printed in the validation report.
The validation index `XSLP_VALIDATIONINDEX_A` is the largest absolute validation factor multiplied by the absolute validation tolerance; the validation index `XSLP_VALIDATIONINDEX_R` is the largest relative validation factor multiplied by the relative validation tolerance.

Related topics

`XSLP_VALIDATIONINDEX_A`, `XSLP_VALIDATIONINDEX_R`, `XSLP_VALIDATIONTOL_A`, `XSLP_VALIDATIONTOL_R`

XSLPvalidatekkt

Purpose

Validates the first order optimality conditions also known as the Karush-Kuhn-Tucker (KKT) conditions versus the currect solution

Synopsis

```
int XPRS_CC XSLPvalidatekkt(XSLPprob Prob, int iCalculationMode, int
      iRespectBasisStatus, int iUpdateMultipliers, double
      dKKTViolationTarget);
```

Arguments

`Prob` The current SLP problem.

`iCalculationMode` The calculation mode can be:

0	recalculate the reduced costs at the current solution using the current dual solution.
1	minimize the sum of KKT violations by adjusting the dual solution.
2	perform both.

`iRespectBasisStatus` The following ways are defined to assess if a constraint is active:

0	evaluate the recalculated slack activity versus `XSLP_ECFTOL_R`.
1	use the basis status of the slack in the linearized problem if available.
2	use both.

`iUpdateMultipliers` The calculated values can be:

0	only used to calculate the `XSLP_VALIDATIONINDEX_K` measure.
1	used to update the current dual solution and reduced costs.

`dKKTViolationTarget` When calculating the best KKT multipliers, it is possible to enforce an even distribution of reduced costs violations by enforcing a bound on them.

Further information

The bounds enforced by dKKTViolationTarget are automatically relaxed if the desired accuracy cannot be achieved.

XSLPvalidateprob

Purpose

Validates the current problem formulation and statement

Synopsis

```
int XPRS_CC XSLPvalidateprob(XSLPprob Prob, int *nErrors, int *nWarnings);
```

Arguments

`Prob`	The current SLP problem.
`nErrors`	Returns the number of errors found in the problem. Errors are expected to make the problem not solve.
`nWarnings`	Returns the number of potential issues found in the problem. The solver may be able to automatically recover during the solve.

Further information

This function is expected to be used in the development stage of a model.

XSLPvalidaterow

Purpose

Prints an extensive analysis on a given constraint of the SLP problem

Synopsis

```
int XPRS_CC XSLPvalidate(XSLPprob Prob, int Row);
```

Arguments

`Prob`	The current SLP problem.
`Row`	The index of the row to be analyzed

Further information

The analysis will include the readable format of the original constraint and the augmented constraint. For infeasible constraints, the absolute and relative infeasibility is calculated. Variables in the constraints are listed including their value in the solution of the last linearization, the internal value (e.g. cascaded), reduced cost, step bound and convergence status. Scaling analysis is also provided.

XSLPvalidatevector

Purpose

Validate the feasibility of constraints for a given solution

Synopsis

```
int XPRS_CC XSLPvalidate(XSLPprob Prob, double *Vector, double *SumInf,
      double *SumScaledInf, double *Objective);
```

Arguments

`Prob`	The current SLP problem.
`Vector`	A vector of length `XPRS_COLS` containing the solution vector to be checked.
`SumInf`	Pointer to double in which the sum of infeasibility will be returned. May be NULL if not required.
`SumScaledInf`	Pointer to double in which the sum of scaled (relative) infeasibility will be returned. May be NULL if not required.
`Objective`	Pointer to double in which the net objective will be returned. May be NULL if not required.

Further information

`XSLPvalidatevector` works the same way as `XSLPvalidate`, and will update `XSLP_VALIDATIONINDEX_A` and `XSLP_VALIDATIONINDEX_R`.

Related topics

`XSLP_VALIDATIONINDEX_A`, `XSLP_VALIDATIONINDEX_R`, `XSLP_VALIDATIONTOL_A`, `XSLP_VALIDATIONTOL_R`

XSLPwriteprob

Purpose

Write the current problem to a file in extended MPS or text format

Synopsis

```
int XPRS_CC XSLPwriteprob(XSLPprob Prob, char *Filename, char *Flags);
```

Arguments

`Prob` The current SLP problem.

`Filename` Character string holding the name of the file to receive the output. The extension ".mat" will automatically be appended to the file name, except for "text" format when ".txt" will be appended.

`Flags` The following flags can be used:

- `a` write the current approximation (linearized) matrix (the default is to write the non-linear matrix including formulae);
- `o` one coefficient per line (the default is up to two numbers or one formula per line);
- `l` write the matrix in the tradition LP like format. Similar to the "text" format, with more SLP specific information
- `s` "scrambled" names (the default is to use the names provided on input);
- `t` write the matrix in "text" (the default is to write extended MPS format).

Example

The following example reads a problem from file, augments it and writes the augmented (linearized) matrix in text form to file "output.txt":

```
XSLPreadprob(Prob, "Matrix", "");
XSLPconstruct(Prob);
XSLPwriteprob(Prob, "output", "lt");
```

Further information

The `t` flag is used to produce a "human-readable" form of the problem. It is similar to the `lp` format of `XPRSwriteprob`, but does not contain all the potential complexities of the Extended MPS Format, so the resulting file cannot be used for input. A quadratic objective is written with its true coefficients (not scaled by 2 as in the equivalent `lp` format).

Related topics

`XSLPreadprob`

XSLPwriteslxsol

Purpose

Write the current solution to an MPS like file format

Synopsis

```
int XPRS_CC XSLPwriteslxsol(XSLPprob Prob, char *Filename, char *Flags);
```

Arguments

`Prob`	The current SLP problem.
`Filename`	Character string holding the name of the file to receive the output. The extension ".slx" will automatically be appended to the file name, unless an extension is already specified in the filename.
`Flags`	The following flags can be used: `p` Use double precision numbers

CHAPTER 22

Internal Functions

Xpress NonLinear provides a set of standard functions for use in formulae. Many are standard mathematical functions; there are a few which are intended for specialized applications.

The following is a list of all the Xpress NonLinear internal functions:

22.1 Trigonometric functions

The trigonometric functions `SIN`, `COS` and `TAN` return the value corresponding to their argument in radians. `SIN` and `COS` are well-defined, continuous and differentiable for all values of their arguments; care must be exercised when using `TAN` because it is discontinuous.

The inverse trigonometric functions `ARCSIN` and `ARCCOS` are undefined for arguments outside the range -1 to +1 and special care is required to ensure that no attempt is made to evaluate them outside this range. Derivatives for the inverse trigonometric functions are always calculated numerically.

ARCCOS

Purpose

Arc cosine trigonometric function

Synopsis

```
ARCSIN(value)
```

Argument

`value`	One of the following: a constant; a variable; a formula evaluating to a single value

Return value

A value in the range 0 to +π.

Further information

`value` must be in the range -1 to +1. Values outside the range will return zero and produce an appropriate error message. If `XSLP_STOPOUTOFRANGE` is set then the function error flag will be set.

ARCSIN

Purpose

Arc sine trigonometric function

Synopsis

```
ARCSIN(value)
```

Argument

`value`	One of the following: a constant; a variable; a formula evaluating to a single value

Return value

A value in the range $-\pi/2$ to $+\pi/2$.

Further information

`value` must be in the range -1 to +1. Values outside the range will return zero and produce an appropriate error message. If `XSLP_STOPOUTOFRANGE` is set then the function error flag will be set.

ARCTAN

Purpose

Arc tangent trigonometric function

Synopsis

```
ARCTAN(value)
```

Argument

`value`	One of the following: a constant; a variable; a formula evaluating to a single value

Return value

A value in the range $-\pi/2$ to $+\pi/2$.

COS

Purpose

Cosine trigonometric function

Synopsis

```
COS(value)
```

Argument

`value`	One of the following: a constant; a variable; a formula evaluating to a single value

SIN

Purpose

Sine trigonometric function

Synopsis

```
SIN(value)
```

Argument

`value`	One of the following: a constant; a variable; a formula evaluating to a single value

TAN

Purpose

Tangent trigonometric function

Synopsis

```
TAN(value)
```

Argument

`value`	One of the following: a constant; a variable; a formula evaluating to a single value

22.2 Other mathematical functions

Most of the mathematical functions are differentiable, although care should be taken in using analytic derivatives where the derivative is changing rapidly.

ABS

Purpose

Absolute value

Synopsis

```
ABS(value)
```

Argument

`value`	One of the following: a constant; a variable; a formula evaluating to a single value

Further information

`ABS` is not always differentiable and so alternative modeling approaches should be used where possible.

ERF

Purpose

The error function

Synopsis

```
ERF(value)
```

Argument

`value`	One of the following: a constant; a variable; a formula evaluating to a single value

ERFC

Purpose

The complementary error function

Synopsis

```
ERFC(value)
```

Argument

`value`	One of the following: a constant; a variable; a formula evaluating to a single value

EXP

Purpose

Exponential function (e raised to the power)

Synopsis

```
EXP(value)
```

Argument

`value`	One of the following: a constant; a variable; a formula evaluating to a single value

LN

Purpose

Natural logarithm

Synopsis

```
LN(value)
```

Argument

`value`	One of the following: a constant; a variable; a formula evaluating to a single value

Further information

`value` must be strictly positive (greater than 1.0E-300).

LOG, LOG10

Purpose

Logarithm to base 10

Synopsis

```
LOG(value)
LOG10(value)
```

Argument

`value`	One of the following: a constant; a variable; a formula evaluating to a single value

Further information

`value` must be strictly positive (greater than 1.0E-300).

MAX

Purpose

Maximum value of two or more expressions

Synopsis

```
MAX(value1, value2)
```

Argument

`value1, value2` Each argument is one of the following: a constant; a variable; a formula evaluating to a single value

Further information

`MAX` is not always differentiable and so alternative modeling approaches should be used where possible.

In mmxnlp, `fmax` is used to represent the max function.

MIN

Purpose

Minimum value of two or more expressions

Synopsis

```
MIN(value1, value2)
```

Argument

`value1, value2` Each argument is one of the following: a constant; a variable; a formula evaluating to a single value

Further information

`MIN` is not always differentiable and so alternative modeling approaches should be used where possible.

In mmxnlp, `fmin` is used to represent the min function.

PWL

Purpose

The piecewise linear function

Synopsis

```
PWL( variable, x1, y1, ..., xk, yk )
```

Arguments

`variable` is a single variable that describes where the pwl shold be evaluated.

`x1,y2, ..., xk, yk` are the k breakpoints of the piecewise linear function. The pwl is extended to minus and plus infinity using the first and last 2 breakpoints respectively.

SIGN

Purpose

The sign function

Synopsis

```
SIGN(value)
```

Argument

`value`	One of the following: a constant; a variable; a formula evaluating to a single value

SQRT

Purpose

Square root

Synopsis

```
SQRT(value)
```

Argument

`value`	One of the following: a constant; a variable; a formula evaluating to a single value

Further information

`value` must be non-negative.

CHAPTER 23

Error Messages

If the optimization procedure or some other library function encounters an error, then the procedure normally terminates with a nonzero return code and sets an error code. For most functions, the return code is 32 for an error; those functions which can return Optimizer return codes (such as the functions for accessing attributes and controls) will return the Optimizer code in such circumstances.

If an error message is produced, it will normally be output to the message handler; for console-based output, it will appear on the console. The error message and the error code can also be obtained using the function `XSLPgetlasterror`. This allows the user to retrieve the message number and/or the message text. The format is:

```
XSLPgetlasterror(Prob, &ErrorCode, &ErrorMessage);
```

The following is a list of the error codes and an explanation of the message. In the list, error numbers are prefixed by *E-* and warnings by *W-*. The printed messages are generally prefixed by *Xpress NonLinear error* and *Xpress NonLinear warning* respectively.

E-12001 ***invalid parameter number num***
This message is produced by the functions which access SLP or Optimizer controls and attributes. The parameter numbers for SLP are given in the header file `xslp.h`. The parameter is of the wrong type for the function, or cannot be changed by the user.

E-12002 ***internal hash error***
This is a non-recoverable program error. If this error is encountered, please contact your local Xpress support office.

E-12003 ***XSLPprob problem pointer is NULL***
The problem pointer has not been initialized and contains a zero address. Initialize the problem using `XSLPcreateprob`.

E-12004 ***XSLPprob is corrupted or is not a valid problem***
The problem pointer is not the address of a valid problem. The problem pointer has been corrupted, and no longer contains the correct address; or the problem has not been initialized correctly; or the problem has been corrupted in memory. Check that your program is using the correct pointer and is not overwriting part of the memory area.

E-12005 ***memory manager error - allocation error***
This message normally means that the system has run out of memory when trying to allocate or reallocate arrays. Use `XSLPprintmemory` to obtain a list of the arrays and amounts of memory allocated by the system. Ensure that any memory allocated by user programs is freed at the appropriate time.

E-12006 ***memory manager error - Array expansion size (num) $\leq$ 0***
This may be caused by incorrect setting of the `XSLP_EXTRA*` control parameters to negative numbers. Use `XSLPprintmemory` to obtain a list of the arrays and amounts of memory

allocated by the system for the specified array. If the problem persists, please contact your local Xpress support office.

E-12007 ***memory manager error - object `Obj` size not defined***
This is a non-recoverable program error. If this error is encountered, please contact your local Xpress support office.

E-12008 ***cannot open file `name`***
This message appears when Xpress NonLinear is required to open a file of any type and encounters an error while doing so. Check that the file name is spelt correctly (including the path, directory or folder) and that it is accessible (for example, not locked by another application).

E-12009 ***cannot open problem file `name`***
This message is produced by `XSLPreadprob` if it cannot find `name.mat`, `name.mps` or `name`. Note that "lp" format files are not accepted for SLP input.

E-12010 ***internal I/O error***
This error is produced by `XSLPreadprob` if it is unable to read or write intermediate files required for input.

E-12011 ***XSLPreadprob unknown record type `name`***
This error is produced by `XSLPreadprob` if it encounters a record in the file which is not identifiable. It may be out of place (for example, a matrix entry in the *BOUNDS* section), or it may be a completely invalid record type.

E-12012 ***XSLPreadprob invalid function argument type `name`***
This error is produced by `XSLPreadprob` if it encounters a user function definition with an argument type that is not one of `NULL`, `DOUBLE`, `INTEGER`, `CHAR` or `VARIANT`.

E-12013 ***XSLPreadprob invalid function linkage type `name`***
This error is produced by `XSLPreadprob` if it encounters a user function with a linkage type that is not one of `DLL`, `XLS`, `XLF`, `MOSEL` or `COM`.

E-12014 ***XSLPreadprob unrecognized function `name`***
This error is produced by `XSLPreadprob` if it encounters a function reference in a formula which is not a pre-defined internal function nor a defined user function. Check the formula and the function name, and define the function if required.

E-12015 ***`func`: `item` `num` out of range***
This message is produced by the Xpress NonLinear function `func` which is referencing the SLP `item` (row, column variable, etc). The index provided is out of range (less than 1 unless zero is explicitly allowed, or greater than the current number of items of that type). Remember that most Xpress NonLinear items count from 1.

E-12016 ***missing left bracket in formula***
This message is produced during parsing of formulae provided in character or unparsed internal format. A right bracket is not correctly paired with a corresponding left bracket. Check the formulae.

E-12017 ***missing left operand in formula***
This message is produced during parsing of formulae provided in character or unparsed internal format. An operator which takes two operands is missing the left hand one (and so immediately follows another operator or a bracket). Check the formulae.

E-12018 ***missing right operand in formula***
This message is produced during parsing of formulae provided in character or unparsed internal format. An operator is missing the right hand (following) operand (and so is immediately followed by another operator or a bracket). Check the formulae.

E-12019 ***missing right bracket in formula***
This message is produced during parsing of formulae provided in character or unparsed internal format. A left bracket is not correctly paired with a corresponding right bracket. Check the formulae.

E-12020 ***column #n is defined more than once as an SLP variable***
This message is produced by XSLPaddvars or XSLPloadvars if the same column appears more than once in the list, or has already been defined as an SLP variable. Although XSLPchgvar is less efficient, it can be used to set the properties of an SLP variable whether or not it has already been declared.

E-12021 ***row #num is defined more than once as an SLP delayed constraint***
This message is produced by the deprecated XSLPadddcs or XSLPloaddcs if the same row appears more than once in the list, or has already been defined as a delayed constraint. Although XSLPchgdc is less efficient, it can be used to set the properties of an SLP delayed constraint whether or not it has already been declared.

E-12022 ***undefined tolerance type name***
This error is produced by XSLPreadprob if it encounters a tolerance which is not one of the 9 defined types (TC, TA, TM, TI, TS, RA, RM, RI, RS). Check the two-character code for the tolerance.

W-12023 ***name has been given a tolerance but is not an SLP variable***
This error is produced by XSLPreadprob if it encounters a tolerance for a variable which is not an SLP variable (it is not in a coefficient, it does not have a non-constant coefficient and it has not been given an initial value). If the tolerance is required (that is, if the variable is to be monitored for convergence) then give it an initial value so that it becomes an SLP variable. Otherwise, the tolerance will be ignored.

W-12024 ***name has been given SLP data of type ty but is not an SLP variable***
This error is produced by XSLPreadprob if it encounters *SLPDATA* for a variable which has not been defined as an SLP variable. Typically, this is because the variable would only appear in coefficients, and the relevant coefficients are missing. The data item will be ignored.

E-12025 ***func has the same source and destination problems***
This message is produced by XSLPcopycallbacks, XSLPcopycontrols and XSLPcopyprob if the source and destination problems are the same. If they are the same, then there is no point in copying them.

E-12026 ***invalid or corrupt SAVE file***
This message is produced by XSLPrestore if the SAVE file header is not valid, or if internal consistency checks fail. Check that the file exists and was created by XSLPsave.

E-12027 ***SAVE file version is too old***
This message is produced by XSLPrestore if the SAVE file was produced by an earlier version of Xpress NonLinear. In general, it is not possible to restore a file except with the same version of the program as the one which SAVEd it.

W-12028 ***problem already has augmented SLP structure***
This message is produced by XSLPconstruct if it is called for a second time for the same problem. The problem can only be augmented once, which must be done after all the variables and coefficients have been loaded. XSLPconstruct is called automatically by XSLPmaxim and XSLPminim if it has not been called earlier.

E-12029 ***zero divisor***
This message is produced by the formula evaluation routines if an attempt is made to divide by a value less than XSLP_ZERO. A value of +/-XSLP_INFINITY is returned as the result and the calculation continues.

E-12030 ***negative number, fractional exponent - truncated to integer***
This message is produced by the formula evaluation routines if an attempt is made to raise a negative number to a non-integer exponent. The exponent is truncated to an integer value and the calculation continues.

E-12031 ***binary search failed***
This is a non-recoverable program error. If this error is encountered, please contact your local Xpress support office.

E-12032 ***wrong number (`num`) of arguments to function `func`***
This message is produced by the formula evaluation routines if a formula contains the wrong number of arguments for an internal function (for example, *SIN*(*A*, *B*)). Correct the formula.

E-12033 ***argument `value` out of range in function `func`***
This message is produced by the formula evaluation routines if an internal function is called with an argument outside the allowable range (for example, `LOG` of a negative number). The function will normally return zero as the result and, if `XSLP_STOPOUTOFRANGE` is set, will set the function error flag.

W-12034 ***terminated following user return code `num`***
This message is produced by `XSLPmaxim` and `XSLPminim` if a nonzero value is returned by the callback defined by `XSLPsetcbiterend` or `XSLPsetcbslpend`.

E-12037 ***failed to load library/file/program "`name`" containing function "`func`"***
This message is produced if a user function is defined to be in a file, but Xpress NonLinear cannot the specified file. Check that the correct file name is specified (also check the search paths such as $PATH and %path% if necessary).
This message may also be produced if the specified library exists but is dependent on another library which is missing.

E-12038 ***function "`func`" is not correctly defined or is not in the specified location***
This message is produced if a user function is defined to be in a file, but Xpress NonLinear cannot find it in the file. Check that the number and type of the arguments is correct, and that the (external) name of the user function matches the name by which it is known in the file.

E-12084 ***Xpress NonLinear has not been initialized***
An attempt has been made to use Xpress NonLinear functions without a previous call to `XSLPinit`. Only a very few functions can be called before initialization. Check the sequence of calls to ensure that `XSLPinit` is called first, and that it completed successfully. *This error message normally produces return code 279*.

E-12085 ***Xpress NonLinear has not been licensed for use here***
Either Xpress NonLinear is not licensed at all (although the Xpress Optimizer may be licensed), or the particular feature (such as MISLP) is not licensed. Check the license and contact the local Fair Isaac sales office if necessary. *This error message normally produces return code 352*.

E-12105 ***Xpress NonLinear error: I/O error on `file`***
The message is produced by `XSLPsave` or `XSLPwriteprob` if there is an I/O error when writing the output file (usually because there is insufficient space to write the file).

E-12107 ***Xpress NonLinear error: user function type `name` not supported on this platform***
This message is produced if a user function defined as being of type *XLS*, *XLF* or *COM* and is run on a non-Windows platform.

E-12110 ***Xpress NonLinear error: unidentified section in REVISE: `name`***
The file provided to XSLPrevise contains an unsupported MPS section.

E-12111 ***Xpress NonLinear error: unidentified row type in REVISE: `name`***
The file provided to XSLPrevise contains an unsupported row type.

E-12112 ***Xpress NonLinear error: unidentified row in REVISE: name***
The file provided to XSLPrevise contains a row name not found in the current problem.

E-12113 ***Xpress NonLinear error: unidentified bound in REVISE: name***
The file provided to XSLPrevise contains an unsupported bound type.

E-12114 ***Xpress NonLinear error: unidentified column in REVISE: name***
The file provided to XSLPrevise contains a column name not found in the current problem.

E-12121 ***Xpress NonLinear error: bad return code num from user function func***
This message is produced during evaluation of a complicated user function if it returns a value (-1) indicating that the system should estimate the result from a previous function call, but there has been no previous function call.

E-12124 ***Xpress NonLinear error: augmented problem not set up***
The message is produced by `XSLPvalidate` if an attempt is made to validate the problem without a preceding call to `XSLPconstruct`. In fact, unless a solution to the linearized problem is available, `XSLPvalidate` will not be able to give useful results.

E-12125 ***Xpress NonLinear error: user function func terminated with errors***
This message is produced during evaluation of a user function if it sets the function error flag (see `XSLPsetfunctionerror`).

W-12142 ***Xpress NonLinear warning: invalid record: text***
This error is produced by `XSLPreadprob` if it encounters a record in the file which is identifiable but invalid (for example, a *BOUNDS* record without a bound set name). The record is ignored.

E-12147 ***Xpress NonLinear error: incompatible arguments in user function func***
This message is produced if a user function is called by `XSLPcalluserfunc` but the function call does not provide the arguments required by the function.

E-12158 ***Xpress NonLinear error: unknown parameter name name***
This message is produced if an attempt is made to set or retrieve a value for a control parameter or attribute given by name where the name is incorrect.

E-12159 ***Xpress NonLinear error: unknown parameter type name***
A parameter has an unexpected type and cannot be retrieved. This is an internal error, please contanct FICO support.

E-12159 ***Xpress NonLinear error: parameter number is not writable***
This message is produced if an attempt is made to set a value for an attribute.

E-12160 ***Xpress NonLinear error: parameter num is not available***
This message is produced if an attempt is made to retrieve a value for a control or attribute which is not readable

E-12161 ***Xpress NonLinear error: parameter num is not available***
The parameter corresponding to the provided ID is aninternal, not readable parameter.

E-12192 ***Xpress NonLinear error: no problem or solution read***
No problem or solution has been read. If a problem read fails, it is not valid to continue with any problem building or solving functions.

E-12193 ***Xpress NonLinear error: this version of SLP requires XPRS version num or newer***
Altough not recommended, Xpress SLP can work with different xprs library versions. This error is issued when a tool old xpres library is found.

E-12194 ***Xpress NonLinear error: provided buffer is too short***
The provided buffer is too short. This error may occur if a formula is retrieved from Xpress, into a buffer that is not large enough.

E-12195 ***Xpress NonLinear error: `type` index `value` is invalid***
The index provided is not valid for the this type.

E-12196 ***Xpress NonLinear error: error in problem transformation***
An error occurred while the problem was attempted to be reformulated as part of the nonlinear presolver. Please contact FICO support.

E-12197 ***Xpress NonLinear error: `request index invalid`***
The requested information cannot be retrieved as it is not valid or not availanble.

E-12198 ***Xpress NonLinear error: error while cascading. Cannot evaluate coefficient at row* rowname *column.***
Evaluating an expression in cascading has returned an error. There is likely a user function in the expression returning an error.

E-12199 ***Xpress NonLinear error: nonlinear coefficient in neutral objective row '*rowname*'. Please use an objective transfer row instead.***
A nonlinear objective function in SLP needs to be modelled using an objective transfer row.

E-12200 ***Xpress NonLinear error: problem is not augmented.***
The operation is only valid for augmented problems. Please call the construct method first, or solve using SLP.

E-12201 ***Xpress NonLinear error: an internal error has occured.***
An internal error has occured that is not expected to have been caused by incorrect input. Please contact FICO support.

E-12202 ***Xpress NonLinear error: attribute `i` cannot be changed.***
Attributes normally cannot be changed, as they are set up by the solver. There are a few exceptions to this rule, the requested attribute is not among the exceptions.

E-12203 ***Xpress NonLinear error: no problem or solution written.***
No problem or solution was written to disk due to an error processing the data.

CHAPTER 24

Xpress Knitro Control Parameters

This chapter provides a full list of the controls accepted by Xpress for setting Knitro parameters. Knitro has a great number and variety of user option settings and although it tries to choose the best settings by default, often significant performance improvements can be realized by choosing some non-default option settings.

24.1 Double control parameters

These double control parameters can be set using XSLPsetdblcontrol using the Xpress NonLinear API, XNLPsetsolverdoublecontrol using the XNLP API and setparam in Mosel using module mmxnlp.

XKTR_PARAM_BAR_FEASMODETOL

Description Specifies the tolerance in equation that determines whether Knitro will force subsequent iterates to remain feasible.

Type Double

Note The tolerance applies to all inequality constraints in the problem. This option only has an effect if option `XKTR_PARAM_BAR_FEASIBLE` = stay or `XKTR_PARAM_BAR_FEASIBLE` = get_stay.

Default value 1.0e-4

XKTR_PARAM_BAR_INITMU

Description Specifies the initial value for the barrier parameter : μ used with the barrier algorithms. This option has no effect on the Active Set algorithm.

Type Double

Default value 1.0e-1

XKTR_PARAM_DELTA

Description Specifies the initial trust region radius scaling factor used to determine the initial trust region size.

Type Double

Default value 1.0e0

XKTR_PARAM_FEASTOL

Description Specifies the final relative stopping tolerance for the feasibility error.

Type Double

Note Smaller values of feastol result in a higher degree of accuracy in the solution with respect to feasibility.

Default value 1.0e-6

XKTR_PARAM_FEASTOLABS

Description Specifies the final absolute stopping tolerance for the feasibility error.

Type Double

Note Smaller values of feastol_abs result in a higher degree of accuracy in the solution with respect to feasibility.

Default value 0.0e0

XKTR_PARAM_INFEASTOL

Description Specifies the (relative) tolerance used for declaring infeasibility of a model.

Type Double

Note Smaller values of infeastol make it more difficult to satisfy the conditions Knitro uses for detecting infeasible models. If you believe Knitro incorrectly declares a model to be infeasible, then you should try a smaller value for infeastol.

Default value 1.0e-8

XKTR_PARAM_MIP_INTEGERTOL

Description This value specifies the threshold for deciding whether or not a variable is determined to be an integer.

Type Double

Default value 1.0e-8

XKTR_PARAM_MIP_INTGAPABS

Description The absolute integrality gap stop tolerance for MIP.

Type Double

Default value 1.0e-6

XKTR_PARAM_MIP_INTGAPREL

Description The relative integrality gap stop tolerance for MIP.

Type Double

Default value 1.0e-6

XKTR_PARAM_OBJRANGE

Description Specifies the extreme limits of the objective function for purposes of determining unboundedness.

Type Double

Note If the magnitude of the objective function becomes greater than objrange for a feasible iterate, then the problem is determined to be unbounded and Knitro proceeds no further.

Default value 1.0e20

XKTR_PARAM_OPTTOL

Description Specifies the final relative stopping tolerance for the KKT (optimality) error.

Type Double

Note Smaller values of opttol result in a higher degree of accuracy in the solution with respect to optimality.

Default value 1.0e-6

XKTR_PARAM_OPTTOLABS

Description Specifies the final absolute stopping tolerance for the KKT (optimality) error.

Type Double

Note Smaller values of opttol_abs result in a higher degree of accuracy in the solution with respect to optimality.

Default value 0.0e0

XKTR_PARAM_PRESOLVE_TOL

Description Determines the tolerance used by the Knitro presolver to remove variables and constraints from the model.

Type Double

Note If you believe the Knitro presolver is incorrectly modifying the model, use a smaller value for this tolerance (or turn the presolver off).

Default value 1.0e-6

XKTR_PARAM_XTOL

Description The optimization process will terminate if the relative change in all components of the solution point estimate is less than xtol.

Type Double

Note If using the Interior/Direct or Interior/CG algorithm and the barrier parameter is still large, Knitro will first try decreasing the barrier parameter before terminating.

Default value 1.0e-15

24.2 Integer control parameters

These integer control parameters can be set using XSLPsetintcontrol using the Xpress NonLinear API, XNLPsetsolverintcontrol using the XNLP API and setparam in Mosel using module mmxnlp.

XKTR_PARAM_ALGORITHM

Description Indicates which algorithm to use to solve the problem

Type Integer

Values

0	(auto) let Knitro automatically choose an algorithm, based on the problem characteristics.
1	(direct) use the Interior/Direct algorithm.
2	(cg) use the Interior/CG algorithm.
3	(active) use the Active Set algorithm.
4	(sqp) use the SQP algorithm.
5	(multi) run all algorithms, perhaps in parallel.

Default value 0

XKTR_PARAM_BAR_DIRECTINTERVAL

Description Controls the maximum number of consecutive conjugate gradient (CG) steps before Knitro will try to enforce that a step is taken using direct linear algebra.

Type Integer

Note This option is only valid for the Interior/Direct algorithm and may be useful on problems where Knitro appears to be taking lots of conjugate gradient steps. Setting bar_directinterval to 0 will try to enforce that only direct steps are taken which may produce better results on some problems.

Default value 10

XKTR_PARAM_BAR_FEASIBLE

Description Specifies whether special emphasis is placed on getting and staying feasible in the interior-point algorithms.

Type Integer

Values

0	(no) No special emphasis on feasibility.
1	(stay) Iterates must satisfy inequality constraints once they become sufficiently feasible.
2	(get) Special emphasis is placed on getting feasible before trying to optimize.
3	(get_stay) Implement both options 1 and 2 above.

Note This option can only be used with the Interior/Direct and Interior/CG algorithms. If bar_feasible = stay or bar_feasible = get_stay, this will activate the feasible version of Knitro. The feasible version of Knitro will force iterates to strictly satisfy inequalities, but does not require satisfaction of equality constraints at intermediate iterates. This option and the honorbnds option may be useful in applications where functions are undefined outside the region defined by inequalities. The initial point must satisfy inequalities to a sufficient degree; if not, Knitro may generate infeasible iterates and does not switch to the feasible version until a sufficiently feasible point is found. Sufficient satisfaction occurs at a point x if it is true for all inequalities that cl + tol $\leq$ c(x) $\leq$ cu - tol The constant tol is determined by the option bar_feasmodetol. If bar_feasible = get or bar_feasible = get_stay, Knitro will place special emphasis on first trying to get feasible before trying to optimize.

Default value 0

XKTR_PARAM_BAR_INITPT

Description Indicates whether an initial point strategy is used with barrier algorithms.

Type Integer

Values

0	(auto) Let Knitro automatically choose the strategy.
1	(yes) Shift the initial slacks and multipliers to improve barrier algorithm performance.
2	(no) Do no alter the initial slacks and multipliers.

Note This option has no effect on the Active Set algorithm.

Default value 0

XKTR_PARAM_BAR_MAXBACKTRACK

Description Indicates the maximum allowable number of backtracks during the linesearch of the Interior/Direct algorithm before reverting to a CG step.

Type Integer

Note Increasing this value will make the Interior/Direct algorithm less likely to take CG steps. If the Interior/Direct algorithm is taking a large number of CG steps (as indicated by a positive value for 'Gits' in the output), this may improve performance. This option has no effect on the Active Set algorithm.

Default value 3

XKTR_PARAM_BAR_MAXCROSSIT

Description Specifies the maximum number of crossover iterations before termination.

Type Integer

Note If the value is positive and the algorithm in operation is Interior/Direct or Interior/CG, then Knitro will crossover to the Active Set algorithm near the solution. The Active Set algorithm will then perform at most bar_maxcrossit iterations to get a more exact solution. If the value is 0, no Active Set crossover occurs and the interior-point solution is the final result. If Active Set crossover is unable to improve the approximate interior-point solution, then Knitro will restore the interior-point solution. In some cases (especially on large-scale problems or difficult degenerate problems) the cost of the crossover procedure may be significant - for this reason, crossover is disabled by default. Enabling crossover generally provides a more accurate solution than Interior/Direct or Interior/CG.

Default value 0

XKTR_PARAM_BAR_MAXREFACTOR

Description Indicates the maximum number of refactorizations of the KKT system per iteration of the Interior/Direct algorithm before reverting to a CG step.

Type Integer

Note These refactorizations are performed if negative curvature is detected in the model. Rather than reverting to a CG step, the Hessian matrix is modified in an attempt to make the subproblem convex and then the KKT system is refactorized. Increasing this value will make the Interior/Direct algorithm less likely to take CG steps. If the Interior/Direct algorithm is taking a large number of CG steps (as indicated by a positive value for "CGits" in the output), this may improve performance. This option has no effect on the Active Set algorithm.

Default value -1

XKTR_PARAM_BAR_MURULE

Description Indicates which strategy to use for modifying the barrier parameter mu in the barrier algorithms.

Type Integer

Values

0	(auto) Let Knitro automatically choose the strategy.
1	(monotone) Monotonically decrease the barrier parameter. Available for both barrier algorithms.
2	(adaptive) Use an adaptive rule based on the complementarity gap to determine the value of the barrier parameter. Available for both barrier algorithms.
3	(probing) Use a probing (affine-scaling) step to dynamically determine the barrier parameter. Available only for the Interior/Direct algorithm.
4	(dampmpc) Use a Mehrotra predictor-corrector type rule to determine the barrier parameter, with safeguards on the corrector step. Available only for the Interior/Direct algorithm.
5	(fullmpc) Use a Mehrotra predictor-corrector type rule to determine the barrier parameter, without safeguards on the corrector step. Available only for the Interior/Direct algorithm.
6	(quality) Minimize a quality function at each iteration to determine the barrier parameter. Available only for the Interior/Direct algorithm.

Note Not all strategies are available for both barrier algorithms. This option has no effect on the Active Set algorithm.

Default value 0

XKTR_PARAM_BAR_PENCONS

Description Indicates whether a penalty approach is applied to the constraints.

Type Integer

Values

0	(auto) Let Knitro automatically choose the strategy.
1	(none) No constraints are penalized.
2	(all) A penalty approach is applied to all general constraints.

Note Using a penalty approach may be helpful when the problem has degenerate or difficult constraints. It may also help to more quickly identify infeasible problems, or achieve feasibility in problems with difficult constraints. This option has no effect on the Active Set algorithm.

Default value 0

XKTR_PARAM_BAR_PENRULE

Description Indicates which penalty parameter strategy to use for determining whether or not to accept a trial iterate.

Type Integer

Values

0	(auto) Let Knitro automatically choose the strategy.
1	(single) Use a single penalty parameter in the merit function to weight feasibility versus optimality.
2	(flex) Use a more tolerant and flexible step acceptance procedure based on a range of penalty parameter values.

Note This option has no effect on the Active Set algorithm.

Default value 0

XKTR_PARAM_BAR_SWITCHRULE

Description Indicates whether or not the barrier algorithms will allow switching from an optimality phase to a pure feasibility phase.

Type Integer

Values

0	(auto) Let Knitro determine the switching procedure.
1	(never) Never switch to feasibility phase.
2	(level1) Allow switches to feasibility phase.
3	(level2) Use a more aggressive switching rule.

Note This option has no effect on the Active Set algorithm.

Default value 0

XKTR_PARAM_GRADOPT

Description Specifies how to compute the gradients of the objective and constraint functions.

Type Integer

Values

1	(exact) User provides a routine for computing the exact gradients.
2	(forward) Knitro computes gradients by forward finite-differences.
3	(central) Knitro computes gradients by central finite differences.

Note It is highly recommended to provide exact gradients if at all possible as this greatly impacts the performance of the code.

Default value 1

XKTR_PARAM_HESSOPT

Description Specifies how to compute the (approximate) Hessian of the Lagrangian.

Type Integer

Values

1	(exact) User provides a routine for computing the exact Hessian.
2	(bfgs) Knitro computes a (dense) quasi-Newton BFGS Hessian.
3	(sr1) Knitro computes a (dense) quasi-Newton SR1 Hessian.
4	(finite_diff) Knitro computes Hessian-vector products using finite-differences.
5	(product) User provides a routine to compute the Hessian-vector products.
6	(lbfgs) Knitro computes a limited-memory quasi-Newton BFGS Hessian (its size is determined by the option lmsize).

Note Options hessopt = 4 and hessopt = 5 are not available with the Interior/Direct algorithm. Knitro usually performs best when the user provides exact Hessians (hessopt = 1) or exact Hessian-vector products (hessopt = 5). If neither can be provided but exact gradients are available (i.e., gradopt = 1), then hessopt = 4 is recommended. This option is comparable in terms of robustness to the exact Hessian option and typically not much slower in terms of time, provided that gradient evaluations are not a dominant cost. If exact gradients cannot be provided, then one of the quasi-Newton options is preferred. Options hessopt = 2 and hessopt = 3 are only recommended for small problems ($n \leq 1000$) since they require working with a dense Hessian approximation. Option hessopt = 6 should be used for large problems.

Default value 1

XKTR_PARAM_HONORBNDS

Description Indicates whether or not to enforce satisfaction of simple variable bounds throughout the optimization.

Type Integer

Values

0	(no) Knitro does not require that the bounds on the variables be satisfied at intermediate iterates.
1	(always) Knitro enforces that the initial point and all subsequent solution estimates satisfy the bounds on the variables.
2	(initpt) Knitro enforces that the initial point satisfies the bounds on the variables.

Note This option and the bar_feasible option may be useful in applications where functions are undefined outside the region defined by inequalities.

Default value 2

XKTR_PARAM_LMSIZE

Description Specifies the number of limited memory pairs stored when approximating the Hessian using the limited-memory quasi-Newton BFGS option.

Type Integer

Note The value must be between 1 and 100 and is only used with XKTR_PARAM_HESSOPT = 6. Larger values may give a more accurate, but more expensive, Hessian approximation. Smaller values may give a less accurate, but faster, Hessian approximation. When using the limited memory BFGS approach it is recommended to experiment with different values of this parameter.

Default value 10

XKTR_PARAM_MAXCGIT

Description Specifies the number of limited memory pairs stored when approximating the Hessian using the limited-memory quasi-Newton BFGS option.

Type Integer

Values

0	Let Knitro automatically choose a value based on the problem size.
n	At most n>0 CG iterations may be performed during one minor iteration of Knitro.

Default value 0

XKTR_PARAM_MAXIT

Description Specifies the maximum number of iterations before termination.

Type Integer

Values

0	Let Knitro automatically choose a value based on the problem type. Currently Knitro sets this value to 10000 for LPs/NLPs and 3000 for MIP problems.
n	At most n>0 iterations may be performed before terminating.

Default value 0

XKTR_PARAM_MIP_BRANCHRULE

Description Specifies which branching rule to use for MIP branch and bound procedure.

Type Integer

Values

0	(auto) Let Knitro automatically choose the branching rule.
1	(most_frac) Use most fractional (most infeasible) branching.
2	(pseudcost) Use pseudo-cost branching.
3	(strong) Use strong branching (see options XKTR_PARAM_MIP_STRONG_CANDLIM, XKTR_PARAM_MIP_STRONG_LEVEL, XKTR_PARAM_MIP_STRONG_MAXIT for further control of strong branching procedure).

Default value 0

XKTR_PARAM_MIP_GUB_BRANCH

Description Specifies whether or not to branch on generalized upper bounds (GUBs).

Type Integer

Values

0	(no) Do not branch on GUBs.
1	(yes) Allow branching on GUBs.

Default value 0

XKTR_PARAM_MIP_HEURISTIC

Description Specifies which MIP heuristic search approach to apply to try to find an initial integer feasible point.

Type Integer

Values

0	(auto) Let Knitro choose the heuristic to apply (if any).
1	(none) No heuristic search applied.
2	(feaspump) Apply feasibility pump heuristic.
3	(mpec) Apply heuristic based on MPEC formulation.

Note If a heuristic search procedure is enabled, it will run for at most mip_heuristic_maxit iterations, before starting the branch and bound procedure.

Default value 0

XKTR_PARAM_MIP_HEURISTIC_MAXIT

Description Specifies the maximum number of iterations to allow for MIP heuristic, if one is enabled.

Type Integer

Default value 100

XKTR_PARAM_MIP_IMPLICATNS

Description Specifies whether or not to add constraints to the MIP derived from logical implications.

Type Integer

Values

0	(no) Do not add constraints from logical implications.
1	(yes) Knitro adds constraints from logical implications.

Default value 1

XKTR_PARAM_MIP_KNAPSACK

Description Specifies rules for adding MIP knapsack cuts.

Type Integer

Values

0	(none) Do not add knapsack cuts.
1	(ineqs) Add cuts derived from inequalities only.
2	(ineqs_eqs) Add cuts derived from both inequalities and equalities.

Default value 1

XKTR_PARAM_MIP_LPALG

Description Specifies which algorithm to use for any linear programming (LP) subproblem solves that may occur in the MIP branch and bound procedure.

Type Integer

Values

0	(auto) Let Knitro automatically choose an algorithm, based on the problem characteristics.
1	(direct) Use the Interior/Direct (barrier) algorithm.
2	(cg) Use the Interior/CG (barrier) algorithm.
3	(active) Use the Active Set (simplex) algorithm.

Note LP subproblems may arise if the problem is a mixed integer linear program (MILP), or if using `XKTR_PARAM_MIP_METHOD` = HQG. (Nonlinear programming subproblems use the algorithm specified by the algorithm option.)

Default value 0

XKTR_PARAM_MIP_MAXNODES

Description Specifies the maximum number of nodes explored.

Type Integer

Note Zero vealue means no limit.

Default value 100000

XKTR_PARAM_MIP_MAXSOLVES

Description Specifies the maximum number of subproblem solves allowed (0 means no limit).

Type Integer

Default value 200000

XKTR_PARAM_MIP_METHOD

Description Specifies which MIP method to use.

Type Integer

Values

`0`	(auto) Let Knitro automatically choose the method.
`1`	(BB) Use the standard branch and bound method.
`2`	(HQG) Use the hybrid Quesada-Grossman method (for convex, nonlinear problems only).

Default value 0

XKTR_PARAM_MIP_OUTINTERVAL

Description Specifies node printing interval for `XKTR_PARAM_MIP_OUTLEVEL` when `XKTR_PARAM_MIP_OUTLEVEL` > 0.

Type Integer

Values

`0`	Print output every node.
`2`	Print output every 2nd node.
`N`	Print output every Nth node.

Default value 10

XKTR_PARAM_MIP_OUTLEVEL

Description Specifies how much MIP information to print.

Type Integer

Values

0	(none) Do not print any MIP node information.
1	(iters) Print one line of output for every node.

Default value 1

XKTR_PARAM_MIP_PSEUDOINIT

Description Specifies the method used to initialize pseudo-costs corresponding to variables that have not yet been branched on in the MIP method.

Type Integer

Values

0	Let Knitro automatically choose the method.
1	Initialize using the average value of computed pseudo-costs.
2	Initialize using strong branching.

Default value 0

XKTR_PARAM_MIP_ROOTALG

Description Specifies which algorithm to use for the root node solve in MIP (same options as `XKTR_PARAM_ALGORITHM` user option).

Type Integer

Default value 0

XKTR_PARAM_MIP_ROUNDING

Description Specifies the MIP rounding rule to apply.

Type Integer

Values

0	(auto) Let Knitro choose the rounding rule.
1	(none) Do not round if a node is infeasible.
2	(heur_only) Round using a fast heuristic only.
3	(nlp_sometimes) Round and solve a subproblem if likely to succeed.
4	(nlp_always) Always round and solve a subproblem.

Default value 0

XKTR_PARAM_MIP_SELECTRULE

Description Specifies the MIP select rule for choosing the next node in the branch and bound tree.

Type Integer

Values

0	(auto) Let Knitro choose the node selection rule.
1	(depth_first) Search the tree using a depth first procedure.
2	(best_bound) Select the node with the best relaxation bound.
3	(combo_1) Use depth first unless pruned, then best bound.

Default value 0

XKTR_PARAM_MIP_STRONG_CANDLIM

Description Specifies the maximum number of candidates to explore for MIP strong branching.

Type Integer

Default value 10

XKTR_PARAM_MIP_STRONG_LEVEL

Description Specifies the maximum number of tree levels on which to perform MIP strong branching.

Type Integer

Default value 10

XKTR_PARAM_MIP_STRONG_MAXIT

Description Specifies the maximum number of iterations to allow for MIP strong branching solves.

Type Integer

Default value 1000

XKTR_PARAM_MIP_TERMINATE

Description Specifies conditions for terminating the MIP algorithm.

Type Integer

Values

0	(optimal) Terminate at optimum.
1	(feasible) Terminate at first integer feasible point.

Default value 0

XKTR_PARAM_OUTLEV

Description Controls the level of output produced by Knitro.

Type Integer

Values

0	(none) Printing of all output is suppressed.
1	(summary) Print only summary information.
2	(iter_10) Print basic information every 10 iterations.
3	(iter) Print basic information at each iteration.
4	(iter_verbose) Print basic information and the function count at each iteration.
5	(iter_x) Print all the above, and the values of the solution vector x.
6	(all) Print all the above, and the values of the constraints c at x and the Lagrange multipliers lambda.

Default value 2

XKTR_PARAM_PRESOLVE

Description Determine whether or not to use the Knitro presolver to try to simplify the model by removing variables or constraints. Specifies conditions for terminating the MIP algorithm.

Type Integer

Values

0	(none) Do not use Knitro presolver.
1	(basic) Use the Knitro basic presolver.

Default value 1

XKTR_PARAM_SCALE

Description Performs a scaling of the objective and constraint functions based on their values at the initial point.

Type Integer

Values

0	(no) No scaling is performed.
1	(yes) Knitro is allowed to scale the objective function and constraints.

Note If scaling is performed, all internal computations, including the stopping tests, are based on the scaled values.

Default value 1

XKTR_PARAM_SOC

Description Specifies whether or not to try second order corrections (SOC).

Type Integer

Values

0	(no) No second order correction steps are attempted.
1	(maybe) Second order correction steps may be attempted on some iterations.
2	(yes) Second order correction steps are always attempted if the original step is rejected and there are nonlinear constraints.

Note A second order correction may be beneficial for problems with highly nonlinear constraints.

Default value 1

Appendix

APPENDIX A
The Xpress-SLP Log

The Xpress-SLP log consists of log lines of two different types: the output of the underlying XPRS optimizer, and the log of XSLP itself.

Output is sent to the screen (stdout) by default, but may be intercepted by a user function using the user output callback; see `XSLPsetcbmessage`. 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.

A.0.1 Logging controls

General SLP logging

`XPRS_OUTPUTLOG`	Logging level of the underlying XPRS problem
`XPRS_LPLOG`	Logging frequency for solving the linearization
`XPRS_MIPLOG`	Logging frequency for the MIP solver

Logging for the underlying XPRS problem

`XSLP_LOG`	Level of SLP logging (iteration, penalty, convergence)
`XSLP_SLPLOG`	Logging frequency for SLP iterations
`XSLP_MIPLOG`	MI-SLP specific logging

Special logging settings

`XPRS_DCLOG`	Logging of delayed constraint activation
`XSLP_ERRORTOL_P`	Absolute tolerance for printing error vectors

A.0.2 The structure of the log

The typical log with the default settings starts with statistics about the problem sizes. On the polygon1.mps example, using the XSLP console program this looks like

```
[xpress mps] readprob Polygon1.mat
Reading Problem Polygon
Problem Statistics
          11 (      0 spare) rows
          10 (      4 spare) structural columns
           8 (      0 spare) non-zero elements
Global Statistics
           0 entities        0 sets        0 set members
    PV:       0     DC:       0     DR:       0     EC:       0
    IV:       0     RX:       0     TX:       0     SB:       0
    UF:       0     WT:       0     XV:       0     Total:       0
Xpress-SLP Statistics:
           7 coefficients
           9 SLP variables
```

The standard XPRS optimizer problem loading statistics is extended with a report about the special structures possibly present in the problem, including DC (delayed constraints), DR (determining rows), EC (enforced constraints), IV (initial values), RX/TX (relative and absolute tolerances), SB (initial step bounds), UF (user functions), WT (initial row weights), followed by a statistics about the number of SLP coefficients and variables.

```
------------------------------------------------------------------
 SLP iteration 1, 0s
Minimizing LP Polygon
Original problem has:
        20 rows           27 cols           68 elements
Presolved problem has:
         0 rows            0 cols            0 elements

   Its         Obj Value      S   Ninf  Nneg        Sum Inf  Time
     0       828864.7136      D      0     0        .000000     0
Uncrunching matrix
     0       828864.7136      D      0     0        .000000     0
Optimal solution found
8 unconverged values (at least 1 in active constraints)
Total feasibility error costs 829100.765742

Penalty Error Vectors - Penalties scaled by  200
Variable             Activity       Penalty
BE-V1V4             1381.836001       1.000000
BE-V2V4             1381.834610       1.000000
BE-V3V4             1381.833218       1.000000
Total:              4145.503829
Error Costs: 829100.765742  Penalty Delta Costs: 0.000000  Net Objective: -236.052107

------------------------------------------------------------------
 SLP iteration 2, 0s
Minimizing LP Polygon
Original problem has:
        20 rows           27 cols           73 elements
Presolved problem has:
         0 rows            0 cols            0 elements

   Its         Obj Value      S   Ninf  Nneg        Sum Inf  Time
     0      -3.13860E-05      D      0     0        .000000     0
Uncrunching matrix
     0      -3.13860E-05      D      0     0        .000000     0
Optimal solution found
4 unconverged values (at least 1 in active constraints)

------------------------------------------------------------------
 SLP iteration 3, 0s
Minimizing LP Polygon
Original problem has:
        20 rows           27 cols           72 elements
Presolved problem has:
         0 rows            0 cols            0 elements

   Its         Obj Value      S   Ninf  Nneg        Sum Inf  Time
     0      -1.56933E-05      D      0     0        .000000     0
Uncrunching matrix
     0      -1.56933E-05      D      0     0        .000000     0
Optimal solution found
```

The default solution log consists of the optimizer output of solving the linearizations, followed by statistics of the nonlinear infeasibilities, the penalty and the objective, and the convergence status.

```
------------------------------------------------------------------
 Iteration summary
 Itr. LPS    NetObj        ErrorSum     ErrorCost    Unconv. Extended  Action
    1  O   -236.052107   4145.503829   829100.7657        8        0
```

```
    2  O  -3.13860E-05       .000000       .000000        4        0
    3  O  -1.56932E-05       .000000       .000000        0        0

 Xpress-SLP stopped after 3 iterations. 0 unconverged items
 No unconverged values in active constraints
```

The final iteration summary contains the following fields:

Itr: The iteration number.

LPS: The LP status of the linearization, which can take the following values:
- `O` Linearization is optimal
- `I` Linearization is infeasible
- `U` Linearization is unbounded
- `X` Solving the linearization was interupted

NetObj: The net objective of the SLP iteration.

ErrorSum: Sum of the error delta variables. A measure of infeasibility.

ErrorCost: The value of the weighted error delta variables in the objective. A measure of the effort needed to push the model towards feasiblity.

Unconv: The number of SLP variables that are not converged.

Extended: The number of SLP variables that are converged, but only by extended criteria

Action: The special actions that happened in the iteration. These can be
- `0` Failed line search (non-improving)
- `B` Enforcing step bounds
- `E` Some infeasible rows were enforced
- `G` Global variables were fixed
- `P` The solution needed polishing, postsolve instability
- `P!` Solution polishing failed
- `R` Penalty error vectors were removed
- `V` Feasiblity validation induces further iterations
- `K` Optimality validation induces further iterations

The presence of a `P!` suggests that the problem is particularly hard to solve without postsolve, and the model might benefit from setting `XSLP_NOLPPOLISHING` on `XSLP_ALGORITHM` (please note, that this should only be considered if the solution polishing features is very slow or fails, as the numerical inaccuracies it aims to remove can cause other problems to the solution process).

APPENDIX B

Selecting the right algorithm for a nonlinear problem - when to use the XPRS library instead of XSLP

This chapter focuses on the nonlinear capabilities of the Xpress XPRS optimizer. As a general rule of thumb, problems that can be handled by the XPRS library do not require the use of XSLP; while Xpress XSLP is able to efficiently solve most nonlinear problems, there are subclasses of nonlinear problems for which the Xpress optimizer features specialized algorithms that are able to solve those problems more efficiently and in larger sizes. These are notably the convex quadratic programming and the convex quadratically constrained problems and their mixed integer counterparts.

It is also possible to separate the convex quadratic information from the rest of XSLP, and let the Xpress XPRS optimizer handle those directly. Doing so is good modelling practice, but emphasis must be placed on that the optimizer can only handle convex quadratic constraints.

B.0.1 Convex Quadratic Programs (QPs)

Convex Quadratic Programming (QP) problems are an extension of Linear Programming (LP) problems where the objective function may include a second order polynomial. The FICO Xpress Optimizer can be used directly for solving QP problems (and the Mixed Integer version MIQP).

If there are no other nonlinearities in the problem, the XPRS library povides specialized algorithms for the solution of convex QP (MIQP) problems, that are much more efficient than solving the problem as a general nonlinear problem with XSLP.

B.0.2 Convex Quadratically Constrained Quadratic Programs (QCQPs)

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

A QCQP problem may be written as:

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

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

If there are no other nonlinearities in the problem, the XPRS library povides specialized algorithms for the solution of convex QCQP (and the integer counterpart MIQCQP) problems, that are much more efficient than solving the problem as a general nonlinear problem with XSLP.

B.0.3 Convexity

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

The lack of convexity may give rise to several unfavorable model properties. Lack of convexity in the objective may introduce the phenomenon of locally optimal solutions that are not global ones (a local optimal solution is one for which a neighborhood in the feasible region exists in which that solution is the best). While the lack of convexity in constraints can also give rise to local optimums, they may even introduce non–connected feasible regions as shown in Figure B.1.

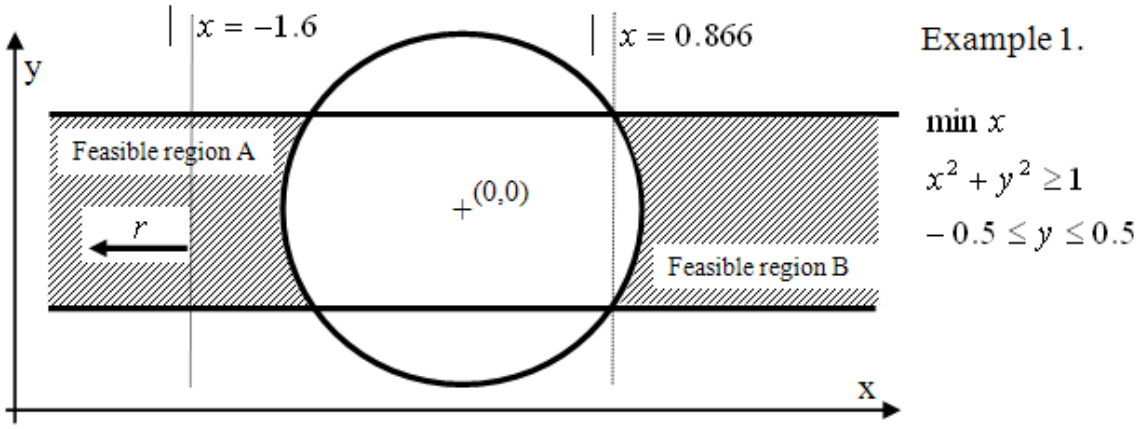

Figure B.1: Non-connected feasible regions

In this example, the feasible region is divided into two parts. Over feasible region B, the objective function has two alterative local optimal solutions, while over feasible region A the objective is not even bounded.

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

B.0.4 Characterizing Convexity in Quadratic Constraints

A quadratic constraint of form

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

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

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

It follows that for greater or equal constraints

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

the negative of Q shall be PSD.

A nontrivial quadratic equality constraint (one for which not every coefficient is zero) always defines a nonconvex region, therefore those must be modelled as XSLP structures.

There is no straightforward way of checking if a matrix is PSD or not. An intuitive way of checking this property, is that the quadratic part shall always only make a constraint harder to satisfy (i.e. taking the quadratic part away shall always be a relaxation of the original problem).

There are certain constructs however, that can easily be recognized as being non convex:

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

As a general rule, a convex quadratic objective and convex quadratic constraints are best handled by the XPRS library; while all nonconvex counterparts should be modelled as XSLP structures.

APPENDIX C

Files used by Xpress NonLinear

Most of the data used by Xpress NonLinear is held in memory. However, there are a few files which are written, either automatically or on demand, in addition to those created by the Xpress Optimizer.

LOGFILE	Created by: `XSLPsetlogfile` The file name and location are user-defined.
NAME.mat	Created by: `XSLPwriteprob` This is the matrix file in extended MPS format. The name is user-defined. The extension *.mat* is appended automatically.
NAME.txt	Created by: `XSLPwriteprob` This is the matrix file in human-readable "text". The name is user-defined. The extension *.txt* is appended automatically.
PROBNAME.svx	Created by: `XSLPsave` This is the SLP part of the save file (the linear part is in *probname*.svf). Used by `XSLPrestore`.

APPENDIX D

Contacting FICO

FICO provides clients with support and services for all our products. Refer to the following sections for more information.

Product support

FICO offers technical support and services ranging from self-help tools to direct assistance with a FICO technical support engineer. Support is available to all clients who have purchased a FICO product and have an active support or maintenance contract. You can find support contact information and a link to the Customer Self Service Portal (online support) on the Product Support home page (www.fico.com/en/product-support).

The FICO Customer Self Service Portal is a secure web portal that is available 24 hours a day, 7 days a week from the Product Support home page. The portal allows you to open, review, update, and close cases, as well as find solutions to common problems in the FICO Knowledge Base.

Please include *'Xpress'* in the subject line of your support queries.

Product education

FICO Product Education is the principal provider of product training for our clients and partners. Product Education offers instructor-led classroom courses, web-based training, seminars, and training tools for both new user enablement and ongoing performance support. For additional information, visit the Product Education homepage at www.fico.com/en/product-training or email producteducation@fico.com.

Product documentation

FICO continually looks for new ways to improve and enhance the value of the products and services we provide. If you have comments or suggestions regarding how we can improve this documentation, let us know by sending your suggestions to techpubs@fico.com.

Please include your contact information (name, company, email address, and optionally, your phone number) so we may reach you if we have questions.

Sales and maintenance

If you need information on other Xpress Optimization products, or you need to discuss maintenance contracts or other sales-related items, contact FICO by:

- Phone: +1 (408) 535-1500 or +44 207 940 8718
- Web: www.fico.com/optimization and use the available contact forms

Related services

Strategy Consulting: Included in your contract with FICO may be a specified amount of consulting time to assist you in using FICO Optimization Modeler to meet your business needs. Additional consulting time can be arranged by contract.

Conferences and Seminars: FICO offers conferences and seminars on our products and services. For announcements concerning these events, go to www.fico.com or contact your FICO account representative.

FICO Community

The FICO Community is a great resource to find the experts and information you need to collaborate, support your business, and solve common business challenges. You can get informal technical support, build relationships with local and remote professionals, and improve your business practices. For additional information, visit the FICO Community (community.fico.com/welcome).

About FICO

FICO (NYSE:FICO) powers decisions that help people and businesses around the world prosper. Founded in 1956 and based in Silicon Valley, the company is a pioneer in the use of predictive analytics and data science to improve operational decisions. FICO holds more than 165 US and foreign patents on technologies that increase profitability, customer satisfaction, and growth for businesses in financial services, telecommunications, health care, retail, and many other industries. Using FICO solutions, businesses in more than 100 countries do everything from protecting 2.6 billion payment cards from fraud, to helping people get credit, to ensuring that millions of airplanes and rental cars are in the right place at the right time. Learn more at www.fico.com.

Index