All trademarks referenced in this manual that are not the property of Dash Associates are acknowledged.

All companies, products, names and data contained within this user guide are completely fictitious and are used solely to illustrate the use of Xpress-MP. Any similarity between these names or data and reality is purely coincidental.

How to Contact Dash
If you have any questions or comments on the use of Xpress-MP, please contact Dash technical support at:

USA, Canada and The Americas
Dash Optimization Inc.
560 Sylvan Avenue
Englewood Cliffs
NJ 07632
USA

Telephone: (201) 567 9445
Fax: (201) 567 9443
email: support-usa@dashoptimization.com

Elsewhere
Dash Optimization Ltd.
Quinton Lodge, Binswood Avenue
Leamington Spa
Warwickshire CV32 5RX
UK

Telephone: +44 1926 315862
Fax: +44 1926 315854
email: support@dashoptimization.com

If you have any sales questions or wish to order Xpress-MP software, please contact your local sales office, or Dash sales at:

USA, Canada and The Americas
Dash Optimization Inc.
560 Sylvan Avenue
Englewood Cliffs
NJ 07632
USA

Telephone: (201) 567 9445
Fax: (201) 567 9443
email: sales@dashoptimization.com

Elsewhere
Dash Optimization Ltd.
Blisworth House, Church Lane
Blisworth
Northants NN7 3BX
UK

Telephone: +44 1604 858993
Fax: +44 1604 858147
email: sales@dashoptimization.com

For the latest news and Xpress-MP software and documentation updates, please visit the Xpress-MP website at http://www.dashoptimization.com/
# Contents

1. **Introduction** ........................................................................................................................................... 1

2. **The problem** ........................................................................................................................................ 2
   2.1 Problem definition ............................................................................................................................... 2
   2.2 Problem formulation ............................................................................................................................. 2

3. **Modeling in Mosel** .......................................................................................................................... 4
   3.1 Basic formulation ............................................................................................................................... 4
   3.2 Setting up and solving the problem ................................................................................................... 5
   3.3 Looking at the results ......................................................................................................................... 6
   3.4 Mosel user functions .......................................................................................................................... 6
   3.5 Using Extended Variable Arrays ....................................................................................................... 8
   3.6 Mosel multi-valued user functions .................................................................................................... 8
   3.7 A user function in an Excel spreadsheet .......................................................................................... 9

4. **Modeling in extended MPS format** .................................................................................................. 11
   4.1 Basic formulation ............................................................................................................................... 11
   4.2 Using the XSLP console-based interface ........................................................................................... 15
   4.3 Coefficients and terms ....................................................................................................................... 16
   4.4 User functions ................................................................................................................................... 16
      4.4.1 A user function in an Excel macro ........................................................................................... 16
      4.4.2 Extending the Polygon model .................................................................................................... 17
   4.5 Using Extended Variable Arrays ....................................................................................................... 18

5. **The Xpress-SLP API functions** ........................................................................................................ 19
   5.1 Header files ....................................................................................................................................... 19
   5.2 Initialization ....................................................................................................................................... 19
   5.3 Callbacks .......................................................................................................................................... 19
   5.4 Creating the linear part of the problem ............................................................................................. 20
   5.5 Adding the non-linear part of the problem ....................................................................................... 22
   5.6 Adding the non-linear part of the problem using character formulae ............................................. 25
   5.7 Checking the data ............................................................................................................................. 26
   5.8 Solving and printing the solution ..................................................................................................... 26
   5.9 Closing the program ......................................................................................................................... 26
   5.10 Adding initial values ....................................................................................................................... 26
   5.11 User functions ................................................................................................................................ 27
      5.11.1 A user function in C ............................................................................................................... 27
      5.11.2 Extending the Polygon model ............................................................................................... 28
      5.11.3 Internal user functions ......................................................................................................... 29
   5.12 Using Extended Variable Arrays ................................................................................................... 29

6. **The XSLP console program** .............................................................................................................. 31
   6.1 The XSLP.CMM file ......................................................................................................................... 31
   6.2 The XSLP program ............................................................................................................................ 32

7. **Xpress-SLP examples** ....................................................................................................................... 34
1. Introduction

This User Guide is intended to provide a general description of the facilities available for modeling with Xpress-SLP. 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 Xpress-SLP Reference Manual and the Xpress-Mosel Reference Manual (Xpress-SLP Section).

Xpress-SLP uses Successive Linear Programming to solve non-linear models. In essence, the technique 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 neighborhood, 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).

Throughout this Guide, we will be working with a model which is small enough to be quick to create and interpret, but which has most of the characteristics (apart from size) of full-scale non-linear models. The original formulation of the problem is due to Francisco J. Prieto of Carlos III University in Madrid and it appears in the library of non-linear test problems.
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.

![Diagram of a pentagon with coordinates](image)

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

We shall use \( r_i \) and \( \theta_i \) as the coordinates of vertex \( V_i \). Then simple geometry and trigonometry gives:

The area of the triangle \( V_NV_iV_j \): \( \frac{1}{2} \times r_i \times r_j \times \sin(\theta_j - \theta_i) \).

The side \( V_iV_j \) is given by: \( (V_iV_j)^2 = r_i^2 + r_j^2 - 2 \times r_i \times r_j \times \cos(\theta_j - \theta_i) \).

The total area of the polygon is \( \sum_{i=2}^{N-1} \text{area}(V_NV_iV_{i+1}) \).

The maximum diameter of 1 requires that all the sides of all the triangles are \( \leq 1 \) – that is:

\[ r_i \leq 1 \quad \text{for} \quad i=1..N-1 \]

and

\[ V_iV_j \leq 1 \quad \text{for} \quad i=1..N-2, j=i+1..N-1 \]

We have assumed in the diagram 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 \( \theta_1 \) is \( \geq 0 \). This is not an additional restriction if we use normal the modeling convention that all variables are non-negative. We also assumed that the last vertex is still “above” the X-axis – that is, \( \theta_{N-1} \) is \( \leq 180^\circ \) (or \( \pi \) radians).
The requirement is therefore:

\[
\text{maximize} \\
\quad \text{(area of the polygon)} \\
\quad \sum_{i=2}^{N-1} (r_i \cdot r_{i-1} \cdot \sin(\theta_i - \theta_{i-1})) \cdot 0.5.
\]

subject to:

\[
\quad \text{(distances between } V_N \text{ and other vertices)} \\
\quad r_i \leq 1 \text{ for } i=1..N-1
\]

\[
\quad \text{(distances between other pairs of vertices)} \\
\quad r_i^2 + r_j^2 - 2 \cdot r_i \cdot r_j \cdot \cos(\theta_j - \theta_i) \leq 1 \text{ for } i=1..N-2, j=i+1..N-1
\]

\[
\quad \text{(first bearing is non-negative)} \\
\quad \theta_1 \geq 0
\]

\[
\quad \text{(bearings are in order)} \\
\quad \theta_{i+1} - \theta_i \geq 0 \text{ for } i=1..N-2
\]

\[
\quad \text{(last vertex is above X-axis)} \\
\quad \theta_{N-1} \leq \pi
\]

Reference:

3. Modeling in Mosel

3.1 Basic formulation

```plaintext
model "Polygon"
uses "mmxslp"

The model uses the Mosel module mmxslp which contains the extensions required for modeling general nonlinear expressions. This automatically loads the mmxprs module, so there is no need to include this explicitly as well.

parameters
N=5
end-parameters

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.

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

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

- 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 genctr (general constraint) and gexp (general expression) are defined by the mmxslp 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 gexp (general expression). gexp s can contain any type of expression involving other gexp s, mpvars, genctrs and constants.

objdef = area
objdef is_free

What we really want to do is to maximize (area). However, although Xpress-SLP is happy in principle with a non-linear objective function, the Xpress-MP optimizer is not, unless it is handled in a special way. Xpress-SLP 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:

```plaintext
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

SLPmaximize(objdef)
maximizing the linear objective
```
forall (i in 1..N-1) do
  rho(i) >= 0.1
  rho(i) <= 1
  SLPDATA("IV", rho(i), 4*i*(N+1-i)/((N+1)^2))
  SLPDATA("IV", theta(i), M_PI*i/N)
end-do

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.

We also give Xpress-SLP initial values by using the SLPDATA procedure with the “IV” parameter. The second argument is the name of the variable, and the third is the initial value to be used. The initial values for theta are divided equally between 0 and pi. 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) do
  forall (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)) <= 1
  end-do
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 genctr 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.

theta(N-1) <= M_PI

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

3.2 Setting up and solving the problem

SLPloadprob(objdef)

This procedure loads the currently-defined non-linear problem into the Xpress-SLP 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.

SLPmaximise

Optimization is carried out with the SLPmaximise or SLPminimise procedures. They can take a string parameter – for example SLPmaximise("b") – as described in the Xpress-SLP and Xpress-MP reference manuals.

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

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

```mosel
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 \(N=5\):

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>(r=0.616416) (\theta=0.703301)</td>
</tr>
<tr>
<td>V2</td>
<td>(r=1) (\theta=1.33111)</td>
</tr>
<tr>
<td>V3</td>
<td>(r=1) (\theta=1.96079)</td>
</tr>
<tr>
<td>V4</td>
<td>(r=0.620439) (\theta=2.58648)</td>
</tr>
</tbody>
</table>

### 3.4 Mosel user functions

In this example, the most complicated function is the area calculation, and it is not a problem to model it explicitly as a formula. However, there are cases when it is not possible to do so, or when it is undesirable to do so – for example, when the formula is very large or contains conditional evaluations, or when it is simply easier to write it as an iterative calculation (in a do-loop) rather than explicitly. This section of the User Guide shows how to extend the Polygon model to calculate the area using a Mosel function.
function MoselArea (A:array(RA:range) of real, N:integer) : real

declarations
n: integer
r1,r2: real ! distances
t1,t2: real ! angles
area: real
end-declarations

n := 4
area := 0
while (n <= N) do
  r1 := A(n-3)
  r2 := A(n-1)
  t1 := A(n-2)
  t2 := A(n)
  n := n+2
  area := area + 0.5*r1*r2*sin(t2-t1)
end-do

returned := area
end-function

This function takes an array A containing pairs of values (rho, theta), so that A(1) is the distance to V1, A(2) is the angle to V1, A(3) is the distance to V2 and so on. N is the number of items in the array. The function works for any number of vertices.

To use this function within Xpress-SLP, we have to tell Xpress-SLP to use this function to calculate the area, and how to set up the array A with the right values.

declarations
....
  BIGARRAY: array(1..100) of real
end-declarations
....
SLPDATA("UF","MoselArea", ",", "DOUBLE,INTEGER", "MOSEL", "Polygon", "BIGARRAY")

This uses the SLPDATA procedure to define the function MoselArea as a user function. The parameters to the SLPDATA function have the following meaning:

"UF" identifies the SLPDATA as defining a user function
"MoselArea" the name of the function as used within Xpress-SLP and the Func() Mosel function
The third argument has been left blank. If the name of the function in argument 2 is different from the name of the function in the Mosel declaration, then the name used in Mosel must be provided as the third argument. If they are the same, the third argument can be left blank.
"DOUBLE,INTEGER" describes a function taking two arguments – an array of double (real) values and an array of integers. A Mosel function must be defined in this way so that Xpress-SLP can communicate with it.
"MOSEL" defines the function type as being MOSEL (as opposed to, for example, a spreadsheet or a C function in a DLL)
"Polygon" defines the name of the model in which the function resides
"BIGARRAY" is the name of an array declared in the model which is large enough to hold the values being passed to the function. The same array can be used for all function calls.

We cannot use this function immediately, because it has a large number of arguments. In Mosel, unless the function has only one or two arguments, a special structure – the Extended Variable Array – has to be used, which allows for argument lists of any length.
3.5 Using Extended Variable Arrays

```plaintext
declarations
    rTheta: array (1..50) of xvitem
....
end-declarations

SLPDATA("XV", rTheta)

forall (i in 1..N) do
    rTheta (i*2-1) := XVitem(rho(i))
    rTheta (i*2) := XVitem(theta(i))
end-do
```

The next thing is to define the arguments to the function. Unless there are only one or two arguments, this must be done in Mosel using an XV. An XV is an Xpress-SLP entity known as an “extended variable array” and it is normally used to communicate with complicated functions, although it does have other applications. In this case, we want to use an XV to set up the variables for our function, and have them in the right order. In Mosel, an XV is modeled as an array of objects of type xvitem.

`rTheta` is declared as an array of 50 xvitems. This is enough to hold the values for polygons of up to 25 sides. If there are too many items declared, it does not matter as long as they are not initialized — only initialized xvitems are actually used.

The `SLPDATA` procedure with the first argument of “XV” defines `rTheta` as an XV in Xpress-SLP. The items of the array are then initialized in the order `rho(1), theta(1), rho(2), theta(2), ...` by using the `XVitem()` function. This is defining the meaning of the xvitems, not assigning them values. Therefore `rTheta(1)` means `rho(1)` and, when the function is called, it will have the value currently assigned to `rho(1)`.

```plaintext
area := Func("MoselArea", rTheta)
```

Finally, the function is used. When we used the `sin` and `cos` functions, we could use them directly, because Mosel knows what they are and how to handle them when used with non-constant arguments. Mosel knows nothing about our new function, even though it has been declared in the `SLPDATA("UF"...)` statement. This is because Mosel needs to know about the function at compilation time, and the `SLPDATA` function only happens at execution time. We therefore use the general-purpose `Func()` structure, which is provided in Mosel for handling arbitrary functions. The `Func()` function takes as its first argument the name of the function as defined in the `SLPDATA` procedure (as the second argument to the `SLPDATA` procedure). Strictly speaking, this is not case-sensitive, although it is easier to read if the same style is used throughout. The remaining arguments of the `Func()` function define the arguments to be used in calling the user function from Xpress-SLP. In this case, there is only one argument — the XV called `rTheta`.

If the model is changed to use the Mosel user function instead of the explicit formula, then the Mosel function will be called from Xpress-SLP at each SLP iteration to calculate a new set of areas. You can confirm this by putting some “write” statements inside the `MoselArea` function.

3.6 Mosel multi-valued user functions

There are many circumstances where several different values have to be calculated from the same set of values, and it is often convenient for them all to be calculated at once by the same function. The function then has to return an array of values in an agreed order which can be accessed by the optimization process. In the Polygon model, we could calculate all the sides of the triangles as well as the total area. For the purposes of this example, we will just calculate the third side of the first triangle (`D(1,2)`).

Because Mosel does not have the facility to return an array as the result of a function, the array is passed to the function by the optimizer and the Mosel function fills it in. This involves the use of a second array — rather like `BIGARRAY`, but holding the return values rather than the input values.
declarations
    RETURNARRAY: array(1..100) of real
end-declarations
....
SLPDATA("UF","MoselAreaA","","DOUBLE,INTEGER,,,,DOUBLE","MOSEL",
    "Polygon","BIGARRAY","RETURNARRAY")

Notice also the changes in the function definition: it is now defined as having three arguments, the new
one being a DOUBLE (real) array use for the return values; the new argument to the SLPDATA
function is the name of the array to hold the return values.

function MoselAreaA (A:array(RA:range) of real, N:integer,
    B:array(RB:range) of real) : real
    B(1) := area
    B(2) := A(1)^2 + A(3)^2 - 2*A(1)*A(3)*cos(A(4)-A(2))
    returned := 0
end-function

We are not showing the whole function here – most of it is the same as the earlier version.

The declaration includes the new return array (B). When the function is called, B will contain three
values: B(1) will have the number of arguments (this is the same as N); B(2) will have the number of
return items; B(3) will have the number of partial derivatives required (this is for advanced use only –
beyond the scope of this guide – but will be zero for normal purposes). B must be dimensioned at least
B(2)*(B(3)+1)

The new bits of the function are in the last few lines: the first item in the array B is the area. The
second item is the length of D(1,2). The return value from the function is zero, indicating that it has
been successfully executed. A non-zero value means it has failed for some reason. Use a value of 1.0 to
indicate an unrecoverable error which will stop the program.

Because this is now a multi-valued function, all the function references must include something to say
which item is required out of the return array. This is done with the last argument to Func().

area := Func("MoselAreaA",rTheta,"1")
D(1,2) := Func("MoselAreaA",rTheta,"2") <= 1

3.7 A user function in an Excel spreadsheet

A spreadsheet can be regarded as a special case of a function, which calculates one or more output
values (cells) from a set of one or more input values (cells).

Xpress-SLP can use an Excel workbook or macro as a user function. It works by using a specified sheet
of the workbook as an input/output area, filling in data on one side and extracting the results on the
other. Between the input and output there can be any sort of processing including cell formulae,
macros, calls to DLL functions or calls to external systems.

SLPDATA("UF","ExcelArea","","VARIANT,VARIANT","XLS",
    "C:\XpressMP\examples\slp\spreadsheet\Polygon.xls","Sheet1")

The function is defined within Mosel using the same type of SLPDATA procedure as for a Mosel user
function. However, some of the parameters are different:

"UF" identifies the SLPDATA as defining a user function
"ExcelArea" the name of the function as used within Xpress-SLP and the Func() Mosel function
The third argument has been left blank. For functions which are simply spreadsheets and cell
formulae, there is no other function name.
“VARIANT, VARIANT” describes a function taking two arguments – both arrays of type VARIANT, which is the standard general-purpose data type within Excel. Depending on the type of function that is being modeled, there may be additional items in this list.

“XLS” defines the function type as being an Excel spreadsheet (as opposed to, for example, a macro or a Mosel user function).

“C:\XpressMP..” defines the location of the Excel workbook. Notice the use of the double “\” because “\” is a special character in Mosel strings.

“Sheet1” is the name of the sheet within the workbook which is used for input and output. Xpress-SLP will put the input arguments into columns A, B, ... and expect to receive the results in columns I, J, ...

\[
\text{area} := \text{Func}("ExcelArea", \text{rTheta})
\]

The usage of the Mosel \( \text{Func}() \) function is similar to the preceding example. Spreadsheets can, of course, calculate more than one cell and so can return more than one value. If your spreadsheet does return more than one value, the results must be in cells I1, I2, etc, and you should then use an additional string argument to the Mosel \( \text{Func}() \) function, giving the number of the item required. Therefore, in our present case, we could ask for the first item, by using

\[
\text{area} := \text{Func}("ExcelArea", \text{rTheta}, "1")
\]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>N</td>
<td>RHO</td>
<td>THETA</td>
<td>AREA</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>=A2+1</td>
<td>=IF($A3&lt;=$Sheet1!B$1/2,OFFSET(Sheet1!$A$1,$A3*2-0,0))</td>
<td>=IF($A3&lt;=$Sheet1!B$1/2,OFFSET(Sheet1!$A$1,$A3*2-0,0),C2)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>=A3+1</td>
<td>=IF($A4&lt;=$Sheet1!B$1/2,OFFSET(Sheet1!$A$1,$A4*2-0,0))</td>
<td>=IF($A4&lt;=$Sheet1!B$1/2,OFFSET(Sheet1!$A$1,$A4*2-0,0),C3)</td>
<td>=0.5<em>B4</em>B3*SIN(C4-C3)</td>
</tr>
<tr>
<td>5</td>
<td>=A4+1</td>
<td>=IF($A5&lt;=$Sheet1!B$1/2,OFFSET(Sheet1!$A$1,$A5*2-0,0))</td>
<td>=IF($A5&lt;=$Sheet1!B$1/2,OFFSET(Sheet1!$A$1,$A5*2-0,0),C4)</td>
<td>=0.5<em>B5</em>B4*SIN(C5-C4)</td>
</tr>
</tbody>
</table>

The spreadsheet itself can be set up in any way. It will receive the input values in column A, in the order defined in \( \text{rTheta} \). The number of values will be in cell B1, and the result is expected in column I1. How the value is obtained from the input cells and placed in the output cells is entirely up to you. The spreadsheet in the examples uses a second sheet, referring back to Sheet1, and picking up the values in pairs into columns B (rho) and C (theta). The area of each triangle is calculated, and the total is put in cell I1 of Sheet1.
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-SLP.

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

<table>
<thead>
<tr>
<th>NAME</th>
<th>POLYGON</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>N</th>
<th>OBJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>OBJEQ</td>
</tr>
<tr>
<td>G</td>
<td>T2T1</td>
</tr>
<tr>
<td>G</td>
<td>T3T2</td>
</tr>
<tr>
<td>G</td>
<td>T4T3</td>
</tr>
<tr>
<td>L</td>
<td>V1V2</td>
</tr>
<tr>
<td>L</td>
<td>V1V3</td>
</tr>
<tr>
<td>L</td>
<td>V2V3</td>
</tr>
<tr>
<td>L</td>
<td>V2V4</td>
</tr>
<tr>
<td>L</td>
<td>V3V4</td>
</tr>
</tbody>
</table>

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 Vi and Vj is \( \leq 1 \).

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.

<table>
<thead>
<tr>
<th>OBJX</th>
<th>OBJ</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJX</td>
<td>OBJEQ</td>
<td>-1.0</td>
</tr>
</tbody>
</table>

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 the \( \theta \) \( i \) in order are all linear – that is, the coefficients are all constant.

<table>
<thead>
<tr>
<th>THETA1</th>
<th>T2T1</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>THETA2</td>
<td>T2T1</td>
<td>1</td>
</tr>
<tr>
<td>THETA3</td>
<td>T3T2</td>
<td>1</td>
</tr>
<tr>
<td>THETA4</td>
<td>T4T3</td>
<td>1</td>
</tr>
</tbody>
</table>

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 maximize 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.

- which is a non-linear function. Xpress-SLP does not itself have a problem with non-linear objective functions, but Xpress-MP 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-SLP also requires that the objective function as passed to Xpress-MP 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

\[
\text{OBJX} = \text{AREA} \quad \text{(or, because the RHS must be constant, \( \text{AREA} - \text{OBJX} = 0 \))}
\]

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:

\[
\begin{align*}
\text{RH01} \times (0.5 \times \text{RH02} \times \sin(\text{THETA2} - \text{THETA1})) + \\
\text{RH02} \times (0.5 \times \text{RH03} \times \sin(\text{THETA3} - \text{THETA2})) + \\
\text{RH03} \times (0.5 \times \text{RH04} \times \sin(\text{THETA4} - \text{THETA3}))
\end{align*}
\]

which is of the right form and can be written in the COLUMNS section as follows:
\begin{align*}
\text{RHO1 OBJEQ} &= 0.5 \times \text{RHO2} \times \sin ( \text{THETA2} - \text{THETA1} ) \\
\text{RHO2 OBJEQ} &= 0.5 \times \text{RHO3} \times \sin ( \text{THETA3} - \text{THETA2} ) \\
\text{RHO3 OBJEQ} &= 0.5 \times \text{RHO4} \times \sin ( \text{THETA4} - \text{THETA3} )
\end{align*}

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
\[ \text{RHO1} \times \text{RHO1} + \text{RHO2} \times \text{RHO2} - 2 \times \text{RHO1} \times \text{RHO2} \times \cos ( \text{THETA2} - \text{THETA1} ) \leq 1 \]

These can again be split into coefficients, for example:
\[ \text{RHO1} \times (\text{RHO1} - 2 \times \text{RHO2} \times \cos (\text{THETA2} - \text{THETA1})) + \text{RHO2} \times (\text{RHO2}) \]

This looks a little strange, because \text{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 \text{ViVj} constraints) like this:

\begin{align*}
\text{RHO1 V1V2} &= \text{RHO1} - 2 \times \text{RHO2} \times \cos (\text{THETA2} - \text{THETA1}) \\
\text{RHO2 V1V2} &= \text{RHO2}
\end{align*}

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

\section*{RHS}

The RHS record introduces the right hand sides 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.

\begin{align*}
\text{RHS1 T2T1} &= .001 \\
\text{RHS1 T4T3} &= .001 \\
\text{RHS1 V1V2} &= 1 \\
\text{RHS1 V1V3} &= 1 \\
\text{RHS1 V2V3} &= 1 \\
\text{RHS1 V3V4} &= 1
\end{align*}

\text{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.

\section*{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 \( -\infty \) 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:

- \text{LO} a lower bound
- \text{UP} an upper bound
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.

<table>
<thead>
<tr>
<th>FR</th>
<th>BOUND1 OBJX</th>
</tr>
</thead>
<tbody>
<tr>
<td>LO</td>
<td>BOUND1 RHO1 0.01</td>
</tr>
<tr>
<td>UP</td>
<td>BOUND1 RHO1 1</td>
</tr>
<tr>
<td>LO</td>
<td>BOUND1 RHO2 0.01</td>
</tr>
<tr>
<td>UP</td>
<td>BOUND1 RHO2 1</td>
</tr>
<tr>
<td>LO</td>
<td>BOUND1 RHO3 0.01</td>
</tr>
<tr>
<td>UP</td>
<td>BOUND1 RHO3 1</td>
</tr>
<tr>
<td>LO</td>
<td>BOUND1 RHO4 0.01</td>
</tr>
<tr>
<td>UP</td>
<td>BOUND1 RHO4 1</td>
</tr>
<tr>
<td>UP</td>
<td>BOUND1 THETA4 3.1415926</td>
</tr>
</tbody>
</table>

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 positive 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-SLP 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-SLP. 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.

<table>
<thead>
<tr>
<th>IV</th>
<th>IVSET1 RHO1 0.555</th>
</tr>
</thead>
<tbody>
<tr>
<td>IV</td>
<td>IVSET1 RHO2 0.888</td>
</tr>
<tr>
<td>IV</td>
<td>IVSET1 RHO3 1</td>
</tr>
<tr>
<td>IV</td>
<td>IVSET1 RHO4 0.888</td>
</tr>
</tbody>
</table>

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 XSLP console-based interface

XSLP is a data-driven console-based interface for operating Xpress-SLP. It allows you full access to the Xpress-SLP API functions, although not all of them can sensibly be used in this way.

XSLP is a two-part system. Behind the program is a text file which defines the commands that the program can execute, and what happens when it does. Once a command has been defined in this way, it can be entered as input to the XSLP program – together with any necessary parameters – and will be executed. We shall use XSLP in a fairly simplistic way to read in, solve and print out a solution for the polygon problem. More details about the XSLP system are given in a separate chapter.

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

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.

XSLP
This starts the XSLP program. It first reads the file XSLP.CMM which must be in the same directory, in the Xpress bin directory or on the path, and checks for the existence of the Xpress-MP and Xpress-SLP DLLs. If you are using an OEM version of the Xpress-MP DLL, you may need a special password or license file from your usual supplier.

LOGFILE = polylog.log 0
This command is optional, but is useful especially in batch commands, or when there is a lot of output which you may want to see later. All output from the program is diverted to the specified file. The number at the end is 0 (for output only to the file) or 1 (for output to both the file and the screen).

SLPINPUT polygon
This reads a non-linear problem from the file polygon.mat. You can put the file anywhere you like, but the extension must always be “.mat”. If there is more than one N-type row in the model, you will need to specify the objective function before SLPINPUT, using the MPSOBJNAME = name command.

SLPMAXIM
This form of the maximize command does a non-linear optimization with the default settings of all the parameters.

WRITEPRTSOL
This will use the normal Xpress-MP 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 XSLP command program.
4.3 Coefficients and terms

So far we have managed to express the formulae as coefficients. However, there are constraints — for example \( \sin(A) \leq 0.5 \) — 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:

\[
\begin{align*}
= & \: V1V2 \: RHO1 \: * \: RHO1 + \: RH02 \: * \: RH02 - \: 2 \: * \: RH01 \: * \: RH02 \: * \: \cos(\: THETA2 - \: THETA1 \: ) \\
= & \: V1V3 \: RHO1 \: * \: RHO1 + \: RH03 \: * \: RH03 - \: 2 \: * \: RH01 \: * \: RH03 \: * \: \cos(\: THETA3 - \: THETA1 \: ) 
\end{align*}
\]

4.4 User functions

In this example, the most complicated function is the area calculation, and it is not a problem to model it explicitly as a formula. However, there are cases when it is not possible to do so, or when it is undesirable to do so — for example, when the formula is very large or contains conditional evaluations, or when it is simply easier to write it as an iterative calculation (in a do-loop) rather than explicitly. This section of the User Guide shows how to extend the Polygon model to calculate the area using a “user function”.

A user function is essentially a function which is not built in to Xpress-SLP. It can be written in a language such as C or Fortran, and compiled into a DLL; it can be written as a set of formulae in an Excel spreadsheet (with or without a macro as well); it can be written entirely within an Excel macro. This example shows the area function written as an Excel macro.

4.4.1 A user function in an Excel macro

```vbnet
Function Area(Values() As Variant, nArgs() As Variant) As Double
    n = nArgs(0)
i = 3
Total = 0
For Count = 1 To n
    Rho1 = Values(i - 3)
    Theta1 = Values(i - 2)
    Rho2 = Values(i - 1)
    Theta2 = Values(i)
    Total = Total + 0.5 * Rho1 * Rho2 * Sin(Theta2 - Theta1)
i = i + 2
Next Count
Area = Total
End Function
```

This is a function written as an Excel macro, in the sheet Sheet1 of the Excel workbook C:\xpressmp\examples\slp\spreadsheet\Polygon.xls.

It takes two arguments, both arrays of type Variant (a general-purpose type which can contain any type of data). It returns a single value of type Double.

This calculates the area for a polygon with any number of sides, by iterating through all the adjacent triangles. The array Values contains pairs of items in the order RHO1, THETA1, RHO2, THETA2, etc. The first loop calculates the area between (RHO2,THETA2) and (RHO1,THETA1). Subsequent loops then add the area of the next triangle.

Notice that all the arrays which communicate with Xpress-SLP count from zero.
In this example, we are calculating only one value, and so there is only one item to return. A more complicated function might calculate and return more than one value (for example, the circumference and the area). In such a case, the function must return an array of type Double, as in the abbreviated example below:

```
Dim DArray(1) As Double
Function ArrayArea(Values() As Variant, nArgs() As Variant) As Double()
...
DArray(0) = Total
DArray[1] = Circum
Area = DArray
End Function
```

### 4.4.2 Extending the Polygon model

The model needs to be modified slightly in order to use the new function. There are two parts – using the function in the model; and declaring the function and explaining how the interface works.

To use the function in the model, we give it a name – say “PolyArea”. We can then use it like any other function.

```
PolyArea ( RHO1 , THETA1 , RHO2 , THETA2 , RHO3 , THETA3 , RHO4 , THETA4 )
```

The arguments RHO1 up to THETA4 are in the order that the function expects.

If the function returns an array, then we have to specify which item in the array is the one we want. In our case, there is only one value, and it is the first. The formula for the area would then become:

```
PolyArea ( RHO1 , THETA1 , RHO2 , THETA2 , RHO3 , THETA3 , RHO4 , THETA4 : 1 )
```

The colon (":") indicates that the next item specifies which array value is required. The number "1" indicates the first item.

The OBJEQ constraint will now have only two items – the OBJX entry and the new PolyArea function, which will be a coefficient of the special equals column. The relevant piece of the MPS file is:

```
OBJX OBJEQ –1
   =  OBJEQ = PolyArea ( RHO1 , THETA1 , RHO2 , THETA2 , RHO3 , THETA3 , RHO4 , THETA4 )
```

The function declaration is made in the SLPDATA section, using a record of type UF. There are several fields which can be used, but not all of them are necessary in this case.

```
UF PolyArea = Area ( VARIANT , VARIANT ) XLF = C:\XpressMP...\Polygon.xls = Sheet1
```

The fields we have used are as follows:
- **UF** indicates this is a user function declaration
- **PolyArea** the name of the function as used within the model
- **Area** the name of the function as used in the spreadsheet. If it is the same as that used in the model, it can be omitted (in which case the “=” sign is omitted as well)
- **VARIANT** the arguments in brackets indicate the number and type of the arguments. For Excel macros, the type is always VARIANT, and the first two arguments are the array of values and the number of items in the array.
- **XLF** indicates an Excel macro function (as opposed to spreadsheet formulae or a DLL)
- **C:\Xpress..** the name of the spreadsheet containing the macro (we’ve had to abbreviate the full path to fit on the page – the full name is in the file in the examples)
- **Sheet1** the name of the sheet containing the macro
Notice that the declaration does not itself say whether the function returns an array or a single item. Xpress-SLP deduces this from the form of the function reference itself (whether or not there is a return item number).

The model can now be run using the Excel macro to calculate the values instead of using a formula inside the model itself.

4.5 Using Extended Variable Arrays

The extended variable array (XV) is a special type of entity in Xpress-SLP which can be used to simplify the calling of complicated functions. The complete XV structure is really beyond the scope of this guide, and we shall be using it here just to declare an array of variables for use in the function. However, the full functionality of XV s allows them to be used with functions that can take a variable number of arguments and to simplify the setting up of complicated formulae.

An XV is declared in the SLPDATA section as a list of items, one per record, which are taken as the members of the XV in the order in which they are provided.

```
XV rTheta RHO1
XV rTheta THETA1
XV rTheta RHO2
XV rTheta THETA2
XV rTheta RHO3
XV rTheta THETA3
XV rTheta RHO4
XV rTheta THETA4
```

The first field on the record is XV, which indicates that this defines an item in an XV array.
The second field is the name of the XV. This can be anything you like, but it must be different from the name of any variable.
The third item is the name of the variable which occupies this position in the array.

It is possible to use constants within an XV. In such a case, the field containing the name of the variable is blank, as is the next field (which contains the name of the argument as it is known to the function) and the value goes in the next field – for example:

```
XV AnotherXV = = 42
```

Notice the use of the equals sign as the delimiter.

Once the XV has been declared, it can be used as an argument to a function. It will be replaced by its list of members. The OBJEQ constraint therefore becomes just

```
OBJX OBJEQ -1
= OBJEQ = PolyArea ( rTheta : 1 )
```
5. The Xpress-SLP API functions

Instead of writing an extended MPS file and reading in the model from the file, it is possible to embed Xpress-SLP directly into your application, and to create the problem, solve it and analyze the solution entirely by using the Xpress-SLP API functions. This example uses the C header files and API calls. We shall assume you have some familiarity with the Xpress-MP 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 on “Modeling In Extended MPS Format” first.

5.1 Header files

The header file containing the Xpress-SLP definitions is xslp.h. This must be included together with the Xpress-MP header xprs.h. xprs.h must come first.

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

5.2 Initialization

Xpress-SLP and Xpress-MP both need to be initialized, and an empty problem created. All Xpress-SLP 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.

```c
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-SLP and Xpress-MP callbacks, so that all messages appear in the same place.

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

```c
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 messages 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-MP functions.

```c
#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.

```c
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.

```c
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.

```c
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 printf function). printf 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;
Element = 0;
ColStart[nCol] = nElement;
OBJ[nCol] = 1.0;
Lower[nCol++] = XPRS_MINUSINFINITY; /* free column */
Element[nElement] = -1.0;
RowIndex[nElement++] = 0;
ColStart[nCol] = nElement;
(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-MP can handle both the names you have created and the names that will be created by Xpress-SLP. Typically, Xpress-SLP 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-MP 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-SLP “construct” phase.

XPRSloadlp(mprob,"Polygon",nCol,nRow,RowType,RHS,NULL,OBJ,
ColStart,NULL,RowIndex,Element,Lower,Upper);
This actually loads the model into Xpress-MP. 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 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;
int ColIndex[MAXCOL];
int FormulaStart[MAXCOEF];
int 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-SLP internal functions SIN and COS. The XSLPgetindex function finds the index of an Xpress-SLP entity (XV, character variable, internal or user function).
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.

```c
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:

\[
RHO2 \times RHO1 \times \sin (THETA2 - THETA1) + \ldots
\]

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

\[
RHO2 \times RHO1 \times \sin (THETA2 - THETA1) + \ldots
\]

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.

```c
for (i=1;i<side-1;i++) {
  for (j=i+1;j<side;j++) {
   RowIndex[nCoef] = iRow++;
    ColIndex[nCoef] = side;
    Factor[nCoef] = 1.0;
    FormulaStart[nCoef++] = nToken;
    Type[nToken] = XSLP_COL;
    Value[nToken++] = side+i;
    Type[nToken] = XSLP_CON;
    Value[nToken++] = 2;
    Type[nToken] = XSLP_OP;
    Value[nToken++] = XSLP_EXPONENT;
    Type[nToken] = XSLP_COL;
    Value[nToken++] = side+j;
    Type[nToken] = XSLP_CON;
    Value[nToken++] = 2;
    Type[nToken] = XSLP_OP;
    Value[nToken++] = XSLP_EXPONENT;
    Type[nToken] = XSLP_OP;
    Value[nToken++] = XSLP_PLUS;
    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_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_OP;
    Value[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:

\[
RHO_i \ 2 \ ^ \ RHO_j \ 2 \ ^ {+ \ 2 \ RHO_i \ * \ RHO_j \ * \ } \ THETA_j \  THETA_i \ - \ COS \ * \ -
\]

The XSLP_loadcoefs is the most efficient way of loading non-linear coefficients into a problem. There is an XSLP_addcoefs 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, XVs 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.

```c
/* 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.

```c
/* 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).

```c
/* 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 \texttt{SIN} and \texttt{COS} have been used above.

5.7 Checking the data
Xpress-SLP includes the function \texttt{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.

\begin{verbatim}
XSLPwriteprob (sprob,"testmat",""");
\end{verbatim}

The first argument is the Xpress-SL 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
\begin{verbatim}
XSLPmaxim(sprob,"");
\end{verbatim}

The \texttt{XSLPmaxim} and \texttt{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-SLP Reference Manual.

\begin{verbatim}
XPRSwriteprtsol(mprob);
\end{verbatim}

The standard Xpress-MP solution print can be obtained by using the \texttt{XPRSwriteprtsol} function. The row and column activities and dual values can be obtained using the \texttt{XPRSgetsol} function.

In addition, you can use the \texttt{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-SLP reference manual for more details) so that Xpress-SLP recalculates the values of the dependent SLP variables at each SLP iteration, then the value from \texttt{XSLPgetvar} will be the recalculated value, whereas the value from \texttt{XPRSgetsol} will be the value from the LP solution (before recalculation).

5.9 Closing the program
\begin{verbatim}
XSLPdestroyprob(sprob);
XPRSdestroyprob(mprob);
XSLPfree();
XPRSfree();
\end{verbatim}

The \texttt{XSLPdestroyprob} function frees any system resources allocated by Xpress-SLP for the specific problem. The problem pointer is then no longer valid. \texttt{XPRSdestroyprob} performs a similar function for the underlying linear problem \texttt{mprob}.

The \texttt{XSLPfree} function frees any system resources allocated by Xpress-SLP. You must then call \texttt{XPRSfree} to perform a similar operation for the optimizer.

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 itself in poorer performance and ultimately a system crash.

5.10 Adding initial values
So far, Xpress-SLP 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
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.

```c
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.

```c
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-SLP Reference Manual) Bit 2 of the type indicates that the initial value is being set.

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

XSLPloadvars can take several other arguments apart from the initial value. It is a general principle in Xpress-SLP 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.

### 5.11 User functions

The most complicated formula in this model is the area calculation. With only 5 sides, it is still possible to write it out explicitly, but it becomes large (and perhaps inefficient) if the number of sides increases. The alternative is to calculate the formula in a function and then use the function within the model.

A user function is essentially a function which is not built in to Xpress-SLP. It can be written in a language such as C or Fortran, and compiled into a DLL; it can be written as a set of formulae in an Excel spreadsheet (with or without a macro as well); it can be written entirely within an Excel macro. This example shows the area function written as a compiled C function.

#### 5.11.1 A user function in C
double XPRS_CC MyFunc(double *Values, int *nArg) {
    int i;
    double Area;
    Area = 0;
    for (i=3;i<nArg[0];i=i+2) {
        Area = Area +
            Values[i-3]*Values[i-1]*sin(Values[i]-Values[i-2]);
    }
    return Area*0.5;
}

This function calculates the area from an array of values, ordered as (RHO1, THETA1, RHO2, THETA2, ...). The number of items in the Values array is given as the first item in nArg.

This is the standard interface for a user function in Xpress-SLP. The first argument is an array of double precision values holding the values of the arguments for the Xpress-SLP function in order; the second argument is an array of integers, the first of which contains the size of the first array.

The function must be declared using XPRS_CC as shown, to ensure that the correct function linkage is created.

This function can be compiled into a DLL. To make use of it, we also need to be able to access the formula from outside, so you may need to add suitable externalization definitions. In Visual C++ under Microsoft Windows®, you can use a Definition File, containing an EXPORTS section, such as

```
EXPORTS
MyFunc=?MyFunc@@YGNPANPAH@Z
```

### 5.11.2 Extending the Polygon model

We can now declare this function in the model and use it instead of the explicit area formula.

```
nToken = 0;
XSLPsetstring(sprob,&i,"MyFunc");
Type[nToken] = XSLP_STRING;
Value[nToken++] = (double) i;
Type[nToken] = XSLP_UFEXETYPE;
Value[nToken++] = (double) 0x01;
Type[nToken] = XSLP_UFARGTYPE;
Value[nToken++] = (double) 023;
XSLPsetstring(sprob,&i,"MyDLL.DLL");
Type[nToken] = XSLP_STRING;
Value[nToken++] = (double) i;
Type[nToken] = XSLP_EOF;
XSLPloaduserfuncs(sprob,1,Type,Value);
XSLPaddnames(sprob,XSLP_USERFUNCNAMES,"MyArea",1,1);
```

User functions are declared using XSLPloaduserfuncs. The definition of the function is stored in parsed arrays similar to the ones used for defining formulae. There are two special token types used here; see the Xpress-SLP Reference Manual for full details about the corresponding values.

- **XSLP_UFEXETYPE** is the type of function. We are defining this to be a DLL function.
- **XSLP_UFVARTYPE** is the type and number of the arguments to the function. Each 3 bits (octal digit) represents one argument. The least significant digit is the first argument and so on. In this case, “3” means a double array, “2” means an integer array, and the rest are all zero, which means they do not exist.

We must also define the name of the function. This is a character string, and it is the first item in the array of tokens. To pass a character string to Xpress-SLP, use the XSLPsetstring function to store the string and return an index to the string. Then use the index with the XSLP_STRING token type.
Because this is a DLL function, we must also define the name of the DLL. This is the first string after the tokens defining the function and argument types. For other types of function (for example, Excel spreadsheets or macros), other string parameters may be needed as well.

The XSLPaddnames function creates a name for the function to be used inside Xpress-SLP when the function is referenced. It is what you will see if you write the problem out using XSLPwriteprob. It can be the same name as the function name in the DLL, but it does not have to be. If you are not writing the problem out, then you do not need to set a name at all.

```c
Type[nToken] = XSLP_RB;
Value[nToken++] = 0;
for (i=nSide-1;i>0;i--) {
    Type[nToken] = XSLP_COL;
    Value[nToken++] = i;
    Type[nToken] = XSLP_COL;
    Value[nToken++] = nSide+i;
}
Type[nToken] = XSLP_FUN;
Value[nToken++] = 1;
Type[nToken] = XSLP_EOF;
Value[nToken++] = 0;
```

In reverse Polish, the arguments to the function must appear in reverse order, so the items start with THETA4 and work down to RHO1. The arguments are preceded by a right bracket token and followed by the user function token for function number 1.

### 5.11.3 Internal user functions

The example above used a function written in a DLL. If the function is compiled into something else – for example, the main executable program – or is not externalized, then you will need to define its address explicitly.

```c
void *Func;
Func = MyFunc;
XSLPchguserfuncaddress(sprob,1,&Func);
```

XSLPchguserfuncaddress takes as its arguments the number of the function, and a pointer to its address. As usual, if the pointer is NULL, the data is left unaltered. The main use of the routine is to define the address of a user function directly, without relying on Xpress-SLP to find it.

### 5.12 Using Extended Variable Arrays

The argument list to the function is quite large, but it is only used once. If the same arguments are used for several different functions, then it may become inefficient or difficult to keep writing out the full list. Also, there are functions which can take varying numbers of arguments and which identify the arguments by name rather than position. If any of these circumstances apply, then an extended variable array (XV) may be useful.
nToken = 0;
XVStart[0] = nToken;
for (i=1;i<nSide;i++) {
    Type[nToken] = XSLP_XVVARTYPE;
    Value[nToken++] = XSLP_VAR;
    Type[nToken] = XSLP_XVVARINDEX;
    Value[nToken++] = nSide+i+1;
    Type[nToken] = XSLP_EOF;
    Value[nToken++] = 0;
    Type[nToken] = XSLP_XVVARTYPE;
    Value[nToken++] = XSLP_VAR;
    Type[nToken] = XSLP_XVVARINDEX;
    Value[nToken++] = i+1;
    Type[nToken] = XSLP_EOF;
    Value[nToken++] = 0;
}
XVStart[1] = nToken;
XSLPloadxvs(splob,1,XVStart,1,Type,Value);
XSLPaddnames(splob,XSLP_XVNAMEs,"rTheta",1,1);

An XV can be regarded as an array of items (called XVitems) each of which can be any one of a variety of different entities: variables, constants, formulae or other XVs. Each XVitem can also have a name which would be passed to a function which receives its arguments by name rather than by position. In the example, we shall make a simple XV which is just an array of variables.

The order of the items in the array is significant, because it is the order in which they will be passed to the function. Our function expects the order RHO1, THETA1, RHO2, THETA2, ..., so we define the XVitems in the same order. XVitems are defined using the same sort of token array as formulae or user functions. The full list of possibilities is in the Xpress-SLP Reference Manual. In the example, we are using two new token types:

XSLP_XVVARTYPE describes the type of entity. The corresponding Value is the type number. In the example, we are using XSLP_VAR. This is similar to XSLP_COL but it always counts from 1, whereas XSLP_COL counts from zero or one, depending on the setting of XPRS_CSTYLE. You must always use XSLP_VAR when defining XVs.

XSLP_XVVARINDEX defines the index of the entity – in this case, it is the variable number.

Each XVitem is terminated with an XSLP_EOF token. XV number n is the set of XVitems between XVStart[n] and XVStart[n+1].

XSLPloadxvs loads the XVs. The XSLPaddnames function can be used to give the XVs names, to aid readability if the problem is printed out.

Once the XV has been defined, it can used in functions just like any other argument.

Type[nToken] = XSLP_RB;
Value[nToken++] = 0;
Type[nToken] = XSLP_XV;
Value[nToken++] = 1;
Type[nToken] = XSLP_FUN;
Value[nToken++] = 1;
Type[nToken] = XSLP_EOF;
Value[nToken++] = 0;

The function is now just MyArea(rTheta)
6. The XSLP console program

XSLP is a data-driven console-based interface for operating Xpress-SLP. It allows you full access to the Xpress-SLP API functions, although not all of them can sensibly be used in this way.

XSLP is a two-part system. Behind the program is a text file which defines the commands that the program can execute, and what happens when it does. Once a command has been defined in this way, it can be entered as input to the XSLP program – together with any necessary parameters – and will be executed.

6.1 The XSLP.CMM file

```
! Current max commands is 500
500
! Current max string space is 50000
50000
IV IVSET1 RHO1 0.555
```

The exclamation mark (“!”) at the beginning of a line in the file indicates a comment, and can be used anywhere. Blank lines can be used to improve readability.

The first two lines of the file contain the maximum number of commands (500 in this example) and the amount of space to reserve to hold the command list and other information (50,000 in this example).

```
WRITEPRTSOL := XPRSwriteprtsol ( XPRSprob prob )
```

The rest of the file contains definitions of API commands from the Optimizer or Xpress-SLP Libraries. The general form of each line is:

```
command [parameters] := APIfunction ( [arglist] )
```

- **command**: the command to be used on the input screen
- **parameters**: place holders to describe the number of parameters. In general, make these meaningful so that they are easy to read and understand.
- **APIfunction**: The name of the Optimizer (prefixed XPRS) or Xpress-SLP (prefixed XSLP) API function as given in the relevant documentation
- **arglist**: The types and locations of the arguments in order.

The following types of arguments are provided:

- **XPRSprob**: the optimizer problem pointer
- **XSLPprob**: the XSLP problem pointer
- **int**: integer
- **int***: pointer to integer
- **char***: pointer to character string
- **double**: double precision (real) number
- **double***: pointer to double precision

The following locations are provided:

- **prob**: The optimizer problem pointer – must be used with type XPRSprob
- **xprob**: The XSLP problem pointer – must be used with type XSLPprob
- **arg1, arg2, ...**: The first, second, etc argument in the parameters list
- **int1, int2, int3**: Integers to receive values
- **dbl1, dbl2, dbl3**: Double precision numbers to receive values
- **chr1, chr2, chr3**: Character buffers to receive values
- **NULL**: A NULL pointer
- **BLANK**: A blank character string.

The following examples from the XSLP commands file will show how the system works
WRITEPRTSOL := XPRSwriteprtsol ( XPRSprob prob )
MAXIM := XPRSmaxim ( XPRSprob prob , char* NULL )
MAXIM FLAG := XPRSmaxim ( XPRSprob prob , char* arg1 )
MPSFORMAT = x := XPRSsetintcontrol ( XPRSprob prob , int 8137 , int arg1 )
ITERMAX = x := XSLPsetintcontrol ( XSLPprob xprob , int 12315 , int arg1 )
ITERMAX := XSLPgetintcontrol ( XSLPprob xprob , int 12315 , int* int1 )

**WRITEPRTSOL** This defines a command with no parameters, which calls the **XPRSwriteprtsol** API function. This function takes one argument, which is the **XPRSprob** problem pointer.

**MAXIM** With no parameters, this calls the **XPRSmaxim** function with the **XPRSprob** problem pointer and a **NULL** second argument.

**MAXIM** With one parameter, this calls the **XPRSmaxim** function with the **XPRSprob** problem pointer and the parameter as the second argument, interpreted as characters.

**MPSFORMAT** This takes an equals sign and one parameter. It calls the **XPRSsetintcontrol** function with the **XPRSprob** problem pointer, and integer value 8137 and the parameter, interpreted as an integer.

**ITERMAX** With an equals sign and one parameter, this works in a similar way to the **MPSFORMAT** example

**ITERMAX** With no parameters, this calls the **XSLPgetintcontrol** function with an integer value 12315 and returns an integer value into location int1.

If any of the locations int1-3, dbl1-3 and chr1-3 contain values which are returned by the function, they are printed on the screen.

The first item in the XSLP commands file which matches the command in name and number of arguments will be used.

There are a few built-in commands, which are used for special purposes. The most important one is **Q** or **QUIT** which terminates the program.

Commands are not normally case-sensitive. However, the API function names are case-sensitive, and must be included exactly as they appear in the documentation. It is very important when adding new functions that the argument list is correct in type and order of items. If you are using the “int*”, “char*” or “double*” types, they must appear exactly like that — with the asterisk at the end of the type rather than at the beginning of the name.

### 6.2 The XSLP program

```
XSLP [commandfile] [parameters]
```

XSLP normally takes its commands from the console. However, it can be used as a batch command processor by putting the commands into a file and using the file name as an argument to the function, for example:

```
XSLP C:\myCommands.dat
```

Unless the command file includes a command to terminate the program (normally **Q** or **QUIT**), the program will return to the console for input after it reaches the end of the command file.

When the program is used in batch mode, parameters can be included on the command line. These are all of the form `-pVALUE`, where `p` is a single character and `VALUE` is the value of the parameter. A parameter can be referred to in the batch command file as `cmd-p` where `p` is the parameter character.

For example, if XSLP is called with the parameter `-oOBJFN`, then the statement

```
MPSOBJNAME = cmd-o
```

will be interpreted as

```
MPSOBJNAME = OBJFN
```

Command lines which are entirely blank or which start with an asterisk will be ignored.
The XSLP console program is not a direct substitute for the Xpress-MP console program. It does not have all the controls and attributes built in, and it is usually necessary to use the internal number (defined in xprs.h) as one of the arguments to, for example, the XPRSgetintattrib() function.
7. Xpress-SLP examples

On the Dash website there are two small demonstrations for the XSLP console program, as well as sample models for the Polygon problem used in this guide.

The Polygon examples are as follows:

**Xpress-SLP User Guide: Mosel examples**
- Polygon1.mos Basic Polygon model
- Polygon2.mos Polygon with Mosel single-valued user function
- Polygon3.mos Polygon with Mosel multi-valued user function
- Polygon4.mos Polygon, with Excel spreadsheet function
- Polygon5.mos Polygon, with Excel macro function
- Polygon6.mos Polygon, with Excel macro multi-valued function

**Xpress-SLP User Guide: Extended MPS Format examples**
- Polygon0.mat Basic Polygon, using coefficients
- Polygon1.mat Polygon, using "equals column"
- Polygon2.mat Polygon with initial values
- Polygon3.mat Polygon, with Excel macro single-valued user function
- Polygon4.mat Polygon, with Excel macro multi-valued user function and XV
- Polygon5.mat Polygon, with Excel spreadsheet function returning derivatives
- Polygon6.mat Polygon, with DLL user function

- Polygon1.cpp Basic Polygon model
- Polygon2.cpp Polygon with initial values
- Polygon3.cpp Polygon, with internal C user function
- Polygon4.cpp Polygon, with internal C user function and XV
- Polygon5.cpp Polygon, with C user function in a DLL and XV
- Polygon1c.cpp As Polygon1.cpp but using XSLP::coef() to load coefficient structures

For information about using these examples, see the relevant sections in this User Guide.

**Xpress-SLP console examples**
- demo.mat demonstration non-linear matrix (minimisation)
- demo.cm XSLP batch command file for demo.mat
- integer.mat demonstration non-linear integer problem (minimisation)
- integer.cm XSLP batch command file for integer.mat

Note that if you are using console-based input for these examples, they are both MINIMIZATION problems.
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>API functions</td>
<td>19</td>
</tr>
<tr>
<td>Coefficients and terms</td>
<td>16</td>
</tr>
<tr>
<td>Examples</td>
<td>33</td>
</tr>
<tr>
<td>Extended MPS format</td>
<td>11</td>
</tr>
<tr>
<td>Extended Variable Arrays</td>
<td></td>
</tr>
<tr>
<td>in API</td>
<td>28</td>
</tr>
<tr>
<td>in extended MPS</td>
<td>18</td>
</tr>
<tr>
<td>in Mosel</td>
<td>8</td>
</tr>
<tr>
<td>gexp</td>
<td>4</td>
</tr>
<tr>
<td>Local optimum</td>
<td>1</td>
</tr>
<tr>
<td>Mosel</td>
<td>4</td>
</tr>
<tr>
<td>Mosel multi-valued user functions</td>
<td>8</td>
</tr>
<tr>
<td>Mosel user functions</td>
<td></td>
</tr>
<tr>
<td>multi-valued</td>
<td>8</td>
</tr>
<tr>
<td>single-valued</td>
<td>6</td>
</tr>
<tr>
<td>multi-valued functions</td>
<td>8</td>
</tr>
<tr>
<td>Optimum, local</td>
<td>1</td>
</tr>
<tr>
<td>SLPDATA</td>
<td></td>
</tr>
<tr>
<td>in extended MPS</td>
<td>14</td>
</tr>
<tr>
<td>in Mosel</td>
<td>5, 7</td>
</tr>
<tr>
<td>Successive Linear Programming</td>
<td>1</td>
</tr>
<tr>
<td>Terms and coefficients</td>
<td>16</td>
</tr>
<tr>
<td>User functions</td>
<td></td>
</tr>
<tr>
<td>in C (API)</td>
<td>26</td>
</tr>
<tr>
<td>in Excel macro</td>
<td>16</td>
</tr>
<tr>
<td>in Excel spreadsheet (Mosel)</td>
<td>9</td>
</tr>
<tr>
<td>in extended MPS</td>
<td>16</td>
</tr>
<tr>
<td>internal</td>
<td>28</td>
</tr>
<tr>
<td>Mosel</td>
<td>6</td>
</tr>
<tr>
<td>with API</td>
<td>26</td>
</tr>
<tr>
<td>xprs.verbose</td>
<td>6</td>
</tr>
<tr>
<td>XSLP console program</td>
<td>15, 30</td>
</tr>
<tr>
<td>XSLP program</td>
<td>31</td>
</tr>
<tr>
<td>XSLP.CMM file</td>
<td>30</td>
</tr>
<tr>
<td>xslp.log</td>
<td>6</td>
</tr>
<tr>
<td>xslp_slplog</td>
<td>6</td>
</tr>
<tr>
<td>xslp_verbose</td>
<td>6</td>
</tr>
<tr>
<td>XSLPaddnames</td>
<td>27</td>
</tr>
<tr>
<td>XSLPchguserfuncaddress</td>
<td>28</td>
</tr>
<tr>
<td>XSLPdestroyprob</td>
<td>25</td>
</tr>
<tr>
<td>XSLPfree</td>
<td>25</td>
</tr>
<tr>
<td>XSLPgetindex</td>
<td>22</td>
</tr>
<tr>
<td>XSLPgetvar</td>
<td>25</td>
</tr>
<tr>
<td>XSLPloadcoefs</td>
<td>24</td>
</tr>
<tr>
<td>XSLPloadvars</td>
<td>26</td>
</tr>
<tr>
<td>XSLPloadxvs</td>
<td>29</td>
</tr>
<tr>
<td>XSLPmaxim</td>
<td>25</td>
</tr>
<tr>
<td>XSLPminim</td>
<td>25</td>
</tr>
<tr>
<td>XSLPsetcbmessage</td>
<td>19</td>
</tr>
<tr>
<td>XSLPwriteprob</td>
<td>25</td>
</tr>
<tr>
<td>xvitem</td>
<td>8</td>
</tr>
<tr>
<td>XVitem</td>
<td>28</td>
</tr>
</tbody>
</table>