Tutorial: Introduction to Modeling in Xpress MP

This is an introductory tutorial for modeling in Xpress MP using the Xpress IVE environment. Basic modeling practices in Xpress MP are introduced via Model examples.

Xpress student version can be downloaded free of charge at:
http://optimization.fico.com/student-version-of-fico-xpress.html.

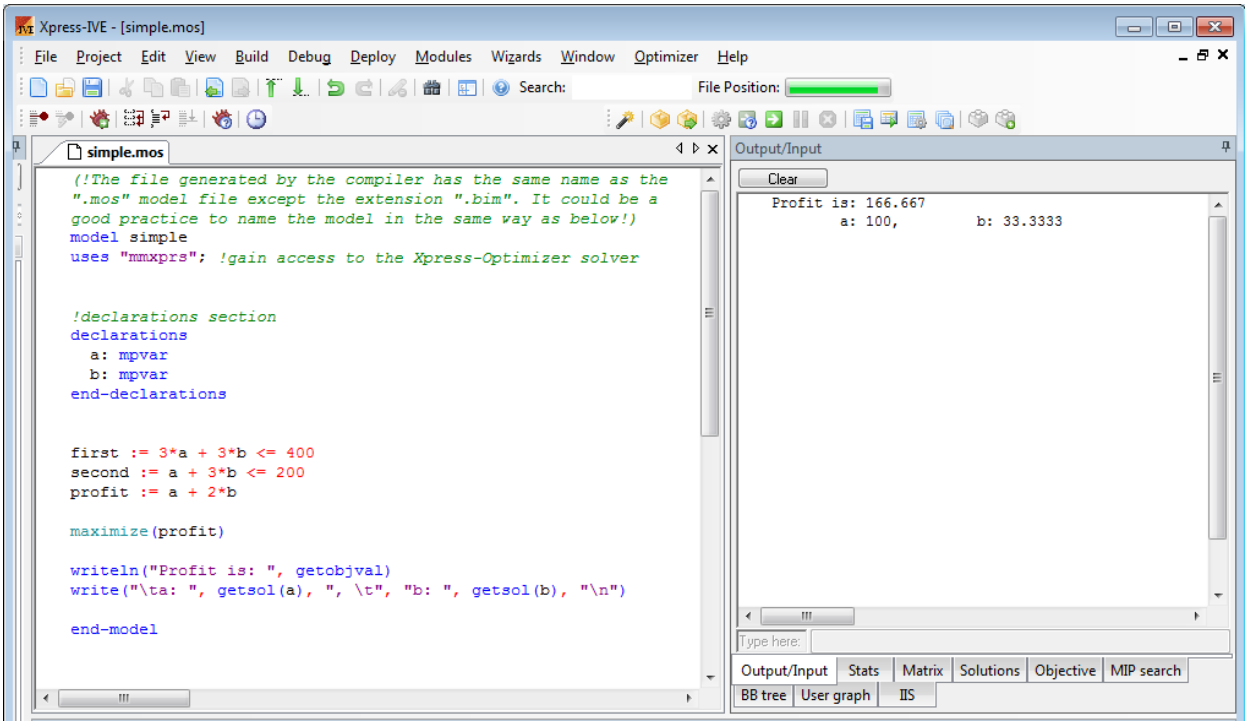
Run the XpressIVE program, the graphical user interface for coding, debugging and running XPress MP files.

From the "Project" menu select "New" and name the new project.

Add a new file named "simple.mos" by selecting "File->New".

Example 1: A Simple Two Decision Variable Linear Programming Model

Enter the following code:



which solves the linear programming problem:

$$\begin{aligned} &\textit{maximize} && a + 2b \\ &\textit{subject to} && 3a + 3b \leq 400 \\ & && a + 3b \leq 200 \\ & && a, b \geq 0. \end{aligned}$$

Compile and run by selecting "Build->Compile" followed by "Build->Run" from the menu bar.

Remarks:

- The exclamation mark "!" is the character reserved for comments. It can be used in the beginning of a line to enter a single line comment, or it can be used after a line of command to append a comment besides the command.

- Between "(!" and "!)" character pairs, a comment spanning multiple lines can be entered.

- Whether the comment marks are used or not, beyond the end-model command is not interpreted by the compiler and this part of the file can be used for comments/notes.

- In Xpress MP, there are no specific marks at the end of command lines, the end of a command line is decided by the completion of the statement. A statement is assumed to be complete at the end of a line, unless the line ends at a point such that the compiler expects further input. For instance, if a statement consists of many terms added to each other, and there is a "+" sign at the end of the line, then the compiler expects the next term to be added in the beginning of the following line.

- Modeling in Xpress is made in blocks. The outermost block is the model block, between the "model" and "end-model" commands. Similar to grouping a block of statements between a pair of curly brackets in Java or C++, a block of commands in Xpress MP is formed by grouping in between the pair of statements "blockname"/"end-blockname", such as "model"/"end-model". One such block that is especially important is the "do"/"end-do" block. Unless a group of command lines following a loop statement (e.g. "forall", "while") is inserted between the pair "do"/"end-do", they are not executed as a block. In this case only the first command following the loop statement is executed inside the loop. Execution continues with the remaining lines after exiting the loop.

- Xpress libraries that are necessary for the compilation and solution of the model file are recalled by the "uses" command. For optimization of mathematical programming problems, it is necessary to recall the "mmxprs" library.

- Linear/mathematical programming decision variables are declared in the declarations block between the "declarations" and "end-declarations" commands. Decision variables are declared to be of the type "mpvar".

- It is a good practice to name constraints, equations/inequalities and other expressions. Observe that the model remains unchanged removing "first :=" and "second :=" assignments from the constraint definitions. However, when investigating the solution of the model, such as retrieving values of the expressions or doing sensitivity analysis, these names are necessary for for referring to the expressions.

- The example has upperbound inequality type ("≤") constraints. Lower bound inequality ("≥") and equality ("=") type constraints can be modeled with a similar syntax, ">=" and "=", respectively.

- The expression for the objective function is named as "profit". The direction of optimization ("the optimization sense") is to "maximize" the objective function. Similarly, there is a "minimize" function.

Compile and run the file again minimizing profit to observe that the decision variables are nonnegative by default.

- “write” and “writeln” functions are used for displaying output in the standard output window of XpressIVE. “write” and “writeln” functions can receive multiple arguments separated by commas. The arguments can be of various types such as string (characters between double quotes), integer, real, etc.

- The “write” function accepts newline (“\n”) and tab (“\t”) characters for indenting and organizing the output. If it is necessary to output backslash (\) or double quote (“) characters, a backslash character is placed before these characters: “\\text between backslash characters\\”, “\”text in double quotes\””. “writeln” works in the same manner, except that it skips to a new line after outputting the function arguments.

- The objective value and values of the decision variables are retrieved by “getobjval” and “getsol()” functions, respectively.

Example 2: Modeling Using Arrays, Separating Model and Data Files

In this section there are six Xpress models that solve the same linear programming problem formulated as below:

$$
\begin{aligned}
&\textit{minimize} && -10x_1 - 12x_2 - 12x_3 \\
&\textit{subject to} && x_1 + 2x_2 + 2x_3 \leq 20 \\
& && 2x_1 + x_2 + 2x_3 \leq 20 \\
& && 2x_1 + 2x_2 + x_3 \leq 20 \\
& && x_1, x_2, x_3 \geq 0.
\end{aligned}
$$

With each new example, different capabilities of the Xpress MP modeling environment is introduced.

Example 2.1 Basic Types and Mathematical Programming Types, Assigning Values

```
model example2_1
uses "mmxprs"; !gain access to the Xpress-Optimizer solver
declarations
    x1, x2, x3: mpvar
    a11, a12, a13, a21, a22, a23, a31, a32, a33: real
    b1, b2, b3: integer
    c1, c2, c3: integer
    constraint1, constraint2, constraint3: linctr
end-declarations

a11 := 1
a12 := 2
a13 := 2
a21 := 2
a22 := 1
a23 := 1
a31 := 2
a32 := 2
a33 := 1
b1 := 20
b2 := 20
b3 := 20
c1 := -10
c2 := -12
c3 := -12

constraint1 := a11*x1 + a12*x2 + a13*x3 <= b1
constraint2 := a21*x1 + a22*x2 + a23*x3 <= b2
constraint3 := a31*x1 + a32*x2 + a33*x3 <= b3

obj_fnc := c1*x1 + c2*x2 + c3*x3

minimize(obj_fnc)
writeln("Optimal value is: ", getobjval)
writeln("x1: ", getsol(x1), " x2: ", getsol(x2), " x3: ", getsol(x3))
end-model
```

- In the declarations block of the model example2_1, the types of the declarted entities are defined. Four basic entity types are:

  • integer: an integer value between -214783648 and 2147483647,
  • real: a rational number with value between -1.7e+308 and 1.7e+308,
  • string: text in double quotes,
  • boolean: the result of a Boolean (logical) expression. takes values *false* or *true*.

After declaration, basic entities receive default values 0, empty string, or false depending on type.

-Two mathematical programming types are:

  • mpvar: a decision variable,
  • linctr: a linear constraint of the model.

The linear constraint type linctr stores expressions of linear functions of the variables ($c_1*x_1 + c_2*x_2 + c_3*x_3$, as in the objective function obj_fnc) besides entire inequalities and equalities that form the constraints (such as $c_1*x_1+c_2*x_2+c_3*x_3 \leq d$).

- Basic types and mathematical programming types (mp types) together are called *elementary types*.

- ":" is used for declarations and ":=" is used for assigning value for a single entity of basic type.

Example 2.2 Arrays, Summing Over an Index

```
model example2_2
uses "mmxprs";

declarations
    x: array(1..3) of mpvar

    a1: array(1..3) of real
    a2: array(1..3) of real
    a3: array(1..3) of real
    b: array(1..3) of integer
    c: array(1..3) of real

    constr: array(1..3) of linctr
end-declarations

a1 :: [ 1, 2, 2 ]
a2 :: [ 2, 1, 1 ]
a3 :: [ 2, 2, 1 ]

b :: [20, 20, 20]

c :: [-10, -12, -12]

constr :: [sum(i in 1..3) a1(i)*x(i) <= b(1),
           sum(i in 1..3) a2(i)*x(i) <= b(2),
           sum(i in 1..3) a3(i)*x(i) <= b(3)]

obj_fnc := sum(i in 1..3) c(i)*x(i)

minimize(obj_fnc)

writeln("Optimal value is: ", getobjval)
forall(i in 1..3) write("x(", i, "): ", getsol(x(i)), "\t")


end-model
```

- Elementary type entities can be used to buid more complex data structures such as array and sets. Sets will be defined in Example 2.5.

- Note the syntax “arrayname: **array**(*range/set*) **of** entitytype” for array declaration.

- “::” is used for assignment of the values for the entire array. The values assigned are formed into a list inside square brackets, separated by commas. Note that the syntax works in the same manner when assigning into the array of linctr, linear constraint type.

- Arrays are defined using ranges or sets. *1..3* is a range, a structure that consists of numbers 1, 2, and 3 in order. The array defined by the range *1..3* has size 3, and its elements are referred by the elements of the range. a2(2) returns the second element of the array a2, which is 1. If another array was declared as:

    a4: array(3..7) of real,

and defined as:

    a4 :: [13, 14, 15, 16, 17],

then a4(5) would return 15.

- When an index that is not in the defining range or set is referred in an array, such as calling a3(4) or a4(1), this results in a run time error, although the model compiles successfully.

- The “sum” operator is useful for making iterative calculations, or forming expressions iteratively. The operator iterates over ranges or sets, by defining an index iterator inside the range or set.

- The scope of the sum operator covers the first expression following the operator. This expression may contain multiplication and division operators. If the expression contains an additon or subtraction operator, the expression has to be placed inside parantheses.

Example 2.3 Multi-dimensional Arrays, The Forall Loop

```
model example2_3
uses "mmxprs";

n := 3
declarations
    x: array(1..n) of mpvar
    A: array(1..n,1..n) of real
    b: array(1..n) of real
    c: array(1..n) of real
    constr: array(1..n) of linctr
end-declarations

A :: [ 1, 2, 2, 2, 1, 1, 2, 2, 1 ]
b :: [20, 20, 20]
c :: [10, 12, 12]

forall(i in 1..n)   constr(i) := sum(j in 1..n) A(i,j)*x(j) <= b(i)
obj_fnc := sum(i in 1..n) c(i)*x(i)

maximize(obj_fnc)

writeln("Optimal value is: ", getobjval)
forall(i in 1..3) write("x(", i, "): ", getsol(x(i)), "\t")
writeln

end-model
```

- The Xpress MP modeling environment allows definition of multi-dimensional arrays. An array is defined by a list of ranges and sets, each defining the size of the array and the iterators for indexing in the corresponding dimension. For instance, a 5 dimensional array of size 3 x |set1| x 5 x |set2| x |set3| can be declared as:

multi_dim_array_1: array(1..3,set1,1..5,set2,set3) of real.

The elements of the arrays are referred as multi_dim_array1(i1,i2,i3,i4,i5), where i1, ..., i5 have to be elements of the corresponding sets and ranges.

- When assigning onto arrays, a linear list is acceptable, as a single line list is assigned into the 2 dimensional array A in the above model. A is filled row by row in this case. This can be done similarly for arrays of higher dimension. When assigning a linear list onto an array of higher dimension, the rightmost index iterator is the fastest changing and the leftmost iterator is the slowest changing.

- **"**forall" is the for loop syntax, that iterates over indexes in a range or set. If multiple lines are to be executed inside the for loop, these lines are grouped inside the "do/end-do" block.

- Similar to the way multi-dimensional arrays are defined, nested forall blocks and sums are possible. For instance, forall(i1 in 1..3, i2 in set1, i3 in 1..5, i4 in set2, i5 in set3) makes a nested forall loop of depth 5, where i1 is the iterator of the outermost forall loop i5 is that of the innermost.

- Note the definition of the problem dimension n. A single entity of basic type can be assigned a value without declaration. The regular assignment syntax ":=" is used.

- For assignment to a single entry of an array, again the ":=" syntax is used. Assignment to the entries of the constr array is made one by one in this way.

Example 2.4 Separating Model and Data: Data Input from File, Multiple Blocks of One Type

```
model example2_4
uses "mmxprs";

declarations
    n: integer
end-declarations
initializations from 'example2_4.dat'
    n
end-initializations

declarations
    x: array(1..n) of mpvar
    A: array(1..n,1..n) of real
    b: array(1..n) of real
    c: array(1..n) of real
    constr: array(1..n) of linctr
end-declarations

initializations from 'example2_4.dat'
    A   b   c
end-initializations

forall(i in 1..n)   constr(i) := sum(j in 1..n) A(i,j)*x(j) <= b(i)

obj_fnc := sum(i in 1..n) c(i)*x(i)

maximize(obj_fnc)

writeln("Optimal value is: ", getobjval)
forall(i in 1..n) write("x(", i, "): ", getsol(x(i)), "\t")
writeln

end-model
```

- A file named "example2_4.dat" in the same folder with the model file should contain data for A, b and c arrays as follows:

```
!Data File for example2_4 and example2_5
A: [1 2 2 2 1 1 2 2 1]
b: [20 20 20 ]
c: [10 12 12]
n: 3
```

Note that only single colons (":") are used for the definition of data instead of the assignment syntax in model files (":=" or "::"), and there are no commas in the lists between data entries.

- The above model file examples that a block type can occur multiple times in the model. In this model there are two declaration and two initialization blocks. First, n is declared and initialized from the data file, based on which the declarations of the other data structures are made.
- An entity has to be declared to be initialized from a file, thus n was first declared in the model file before initialization. This way, the size of the data sought in the data file is known by the environment.

Example 2.5 Sets for Indexing Arrays, Model Solution Time, Number of Simplex Iterations

```
model example2_5
uses "mmxprs", "mmsystem";

declarations
    sweets = {"mosaic cake", "chocolate muffin", "chocolate cookie"}
    ingredients = {"flour","sugar","cocoa"}
    x: array(sweets) of mpvar

    A: array(ingredients,sweets) of real
    b: array(ingredients) of real
    c: array(sweets) of real

    constr: array(ingredients) of linctr
end-declarations

initializations from 'example2_4.dat'
    A   b   c
end-initializations

forall(i in ingredients)    constr(i) := sum(j in sweets) A(i,j)*x(j) <= b(i)

obj_fnc := sum(i in sweets) c(i)*x(i)
start_soln := gettime
maximize(obj_fnc)
end_soln := gettime
writeln("Optimal value is: ", getobjval)
forall(i in sweets) write("x(", i, "): ", getsol(x(i)), "\t")
writeln
writeln("Number of simplex iterations made: ", getparam("XPRS_SIMPLEXITER"), ".")
writeln("Model solutiton time: ", end_soln - start_soln, " seconds.")
end-model
```

- This model is an example for indexing arrays over sets. A set is defined by a list of entities inside curly brackets, with entities separated by commas. Sets sweets and ingredients are defined in the declarations block in the above model, where the assignments are done by equality signs without colons (“=”). Alternatively, it is possible to define the same sets before, outside the declarations block, in that case using the “:=” syntax.

- Iterating over sets, and assignments to arrays defined by sets is done in the same manner they are done in the case ranges are used.

- If an index that is not inside the defining set is used for reference to an array element, there is no compilation error, but a run time error is encountered. b(“salt”) would result in a run time error. Although

sweets and ingredients sets have the same size, iterating over the sweets set would again cause a run time error if reference to the b array is made using the iterator.

- Notice that there is no indexing in the data file, therefore it is possible to use the same data file for two models that do the indexing by ranges and sets, or that have sets with different elements for indexing.

- Although the data file contains additional information such as that for n, in initializaiton it is possible to pick and retrieve only the data necessary for the model.

- The *getparam* function with argument "SIMPLEX_ITER" returns the number of simplex iterations made to solve the LP model.

- It is possible to calculate the time it takes for Xpress to solve the problem by recording the time before and after the minimize/maximize command, which starts the optimization process. The function *gettime* is used for recording the time, which returns the time that has passed after the execution of the compiled model. For using the *gettime* function, it is necessary to recall the "mmsystem" library.

Example 2.6 Initializing Index Sets from File, Sensitivity Analysis

```
model example2_6
uses "mmxprs";
declarations
    sweets: set of string
    ingredients: set of string
    A: array(ingredients,sweets) of real
    b: array(ingredients) of real
    c: array(sweets) of real
end-declarations
initializations from 'example2_6.dat'
    sweets ingredients A b c
end-initializations
declarations
    x: array(sweets) of mpvar
    obj_fnc: linctr
    constr: array(ingredients) of linctr
end-declarations

forall(i in ingredients) constr(i) := sum(j in sweets) A(i,j)*x(j) <= b(i)
obj_fnc := sum(i in sweets) c(i)*x(i)
maximize(obj_fnc)
writeln("Optimal value is: ", getobjval)
forall(i in sweets) write("x(", i, "): ", getsol(x(i)), "\t")
!Sensitivity Analysis
writeln("Constraint Lower and Upper Activities")
forall(i in ingredients) do writeln("\t", getname(constr(i)), ": ",
        getrange(XPRS_LOACT,constr(i)), "/ ", getrange(XPRS_UPACT,constr(i)))
end-do
writeln("Unit Costs for Down and Up")
forall(i in ingredients) do writeln("\t", getname(constr(i)), ": ",
        getrange(XPRS_UDN,constr(i)), "/ ", getrange(XPRS_UUP,constr(i)))
end-do
writeln("Unit Costs for Down and Up")
forall(i in ingredients) do writeln("\t", getname(constr(i)), ": ",
        getsensrng(XPRS_DN,constr(i)), "/ ", getrange(XPRS_UP,constr(i)))
end-do
writeln("Variable Lower/Upper Activities")
forall(i in sweets)do writeln("\t", "x(", i,"): ",
        getrange(XPRS_LOACT,x(i)), "/ ", getrange(XPRS_UPACT,x(i)))
end-do
writeln("Variable Unit Down/Up Costs")
forall(i in sweets)do writeln("\t", "x(", i,"): ",
        getrange(XPRS_UDN,x(i)), "/ ", getrange(XPRS_UUP,x(i)))
end-do
writeln("Variable Lower/Upper Cost")
forall(i in sweets)do writeln("\t", "x(", i,"): ",
        getrange(XPRS_LCOST,x(i)), "/ ", getrange(XPRS_UCOST,x(i)))
end-do
end-model
```

- In this model, there is another example to the usage of multiple declaration and initializaiton blocks, to read the data for the index sets sweets and ingredients before declaring the arrays based on these sets.

- The *getrange* and *getsensrng* function is used to retrieve the sensitivity analysis information on variables/cost coefficients and constraints/right hand side coefficients. Based on the argument entered to the *getrange* and *getsensrng* functions, different sensitivity information about the variables and constraints is retrieved as indicated in the tables below.

getrange(w, x or c)

w	**Which information to return. Possible values:**
	XPRS_UPACT — **Upper activity**
	XPRS_LOACT — **Lower activity**
	XPRS_UUP — **Upper unit cost**
	XPRS_UDN — **Lower unit cost**
	XPRS_UCOST — **Upper cost (variable only)**
	XPRS_LCOST — **Lower cost (variable only)**
x	**A variable of the problem**
c	**A constraint of the problem**

getsensrng(w, x or c)

w	**Which information to return. Possible values:**
	XPRS_UP — **Upper sensivity range**
	XPRS_DN — **Lower sensivity range**
x	**A variable of the problem**
c	**A constraint of the problem**

Example 3.1 Integer Programming, Number of Nodes Generated for IP/MIP Solution

```
model example3_1
uses "mmxprs";

declarations
    sweets: set of string
    ingredients: set of string

    A: array(ingredients,sweets) of real
    b: array(ingredients) of real
    c: array(sweets) of real

    constr: array(ingredients) of linctr
end-declarations

initializations from 'example3_1.dat'
    sweets ingredients A b c
end-initializations

declarations
    x: array(sweets) of mpvar
end-declarations

forall(i in sweets) x(i) is_binary

forall(i in ingredients)    constr(i) := sum(j in sweets) A(i,j)*x(j) <= b(i)

obj_fnc := sum(i in sweets) c(i)*x(i)

maximize(obj_fnc)

writeln("Optimal value is: ", getobjval)
forall(i in sweets) write("x(", i, "): ", getsol(x(i)), "\t")
writeln
writeln(getparam("XPRS_NODES"))
end-model
```

```
!Data File for example3_1 and example3_2
sweets: ["mosaic cake" "chocolate muffin" "chocolate cookie"]
ingredients: ["flour" "sugar" "cocoa"]
A: [1 2 2 2 1 1 2 2 1]
b: [20 20 20 ]
c: [10 12 12]
```

- After the declaration, an mp variable is continuous by default. This can be changed using the *is_integer* and *is_binary* keywords, defining the variable as an integer or binary variable.

- In this model all variables are defined to be binary, exploting the forall loop.

- The *getparam* function called with the argument “XPRS_NODES” returns the number of nodes generated by the Branch and Bound algorithm for the solution of a (mixed) integer programming problem.

Example 3.2 Mixed Integer Programming, Random Data Generation, Data Output

```
model example3_2
uses "mmxprs";
declarations
    sweets: set of string
    ingredients: set of string
end-declarations
initializations from 'example3_1.dat'
    sweets ingredients
end-initializations
declarations
    A: array(ingredients,sweets) of real
    b: array(ingredients) of real
    c: array(sweets) of real
    constr: array(ingredients) of linctr
    x: array(sweets) of mpvar
    solution: array(sweets) of real
end-declarations
forall(i in ingredients) do
    b(i) := 20 + 10 * random
    forall(j in sweets) A(i,j) := 3 + 7 * random
end-do
forall(i in sweets) c(i) := 10 + 10 * random
x(sweets(2)) is_integer
x(sweets(3)) is_binary
forall(i in ingredients)    constr(i) := sum(j in sweets) A(i,j)*x(j) <= b(i)
obj_fnc := sum(i in sweets) c(i)*x(i)
maximize(obj_fnc)
writeln("Optimal value is: ", getobjval)
forall(i in sweets) write("x(", i, "): ", getsol(x(i)), "\t")
writeln
opt_val := getobjval
forall(i in sweets) solution(i) := getsol(x(i))
initializations to 'example3_2.out'
    A b c opt_val solution
end-initializations
end-model
```

- In this example, a mixed integer program is modeled, where x(1) is a continuous variable, x(2) is an integer variable, and x(3) is a binary variable.

- The *random* function returns a random number from the uniform distribution in [0,1]. Multiplication of the returned number with a (positive) constant r, followed by the addition of another constant m gives a random number from a uniform distribution in [m, m + r].

- If necessary, the seed of the random number generator can be set by the *setrandseed()* function, taking as argument the integer which will be the new seed value.

- Outputting to data files works in a similar way to data initialization from files. Again an initialization block is formed, but it begins with "initializations *to*", instead of "*from*". A good practice is to give the same name to the output file as the model file, with the extension ".out". Note that it is necessary to store variable values in data structures of basic entity types, before outputting to the data file.

For further reading on modeling techniques in Xpress MP, see the Getting Started Guide and Reference Manual in:

http://www.fico.com/en/Products/DMTools/xpress-overview/Pages/Xpress-Documentation.aspx .