Methodology, Modelling, and Consulting Skills
School of Mathematics
The University of Edinburgh
Year: 2022/2023

THE UNIVERSITY of EDINBURGH

Computer Session 1: Introduction to Xpress

The software that will be used in this course will be Xpress, a powerful solver for both linear and nonlinear optimization problems. We will use the Mosel language with the Xpress Workbench, a graphical interface to the Mosel language.

You can find much more information on Xpress and the Mosel language on the official documentation website. You can either use the online documentation or download a version for offline reading. The same link to the current documentation can be found in Learn and the documentation is directly accessible from the Xpress Workbench. If you have some programming experience, a very good quick overview is the Mosel Language Quick Reference.

1 Writing our first model

Initially, the screen will look like in Figure 1. Once we create a new Xpress Mosel project, we will see the screen in Figure 2. You should note that every Mosel project will need its own directory.

The first model that we will write is the blending problem that you find in the other handout. We will do it step by step. Although Xpress has a template (`File → New from template`), it is at first, probably easier if you start with an empty file (`File → New`).

Now, we can start with our model. First, there is a line giving a name to the model:

```
model blending
```

We have given it the name `blending`, but we can use any name we like.

Now, we add a line that calls the solver:

```
uses "mmxprs"
```

In Mosel terminology, `mmxprs` is a *module*, which we load with the `uses` command. Without a solver module, we cannot do much, so we will load this in all our models.

Next, we declare the variables of the model:

```
declarations
        x1,x2:mpvar
end-declarations
```

A decision variable is an object of type `mpvar`.

Now, we write the objective function. We have also added a comment line, which begins with the symbol `!`. A comment line is not read at all, but it helps to have the code better organized.

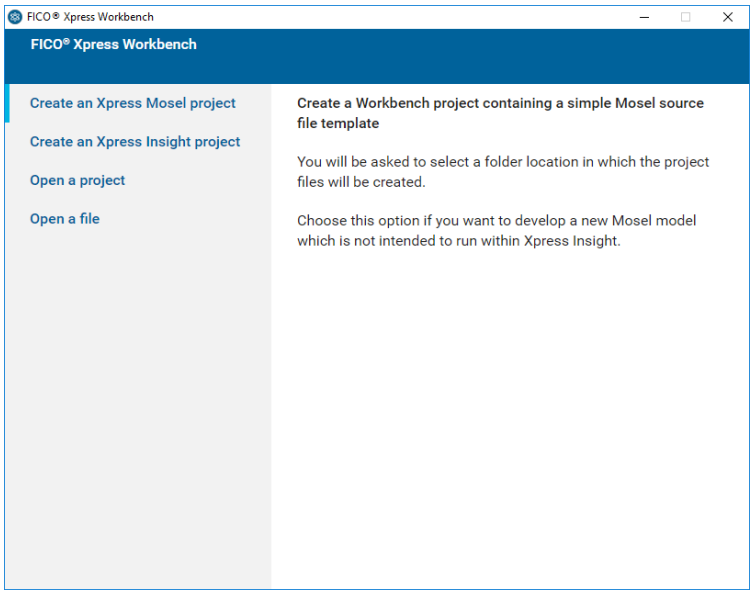

Figure 1: First screen.

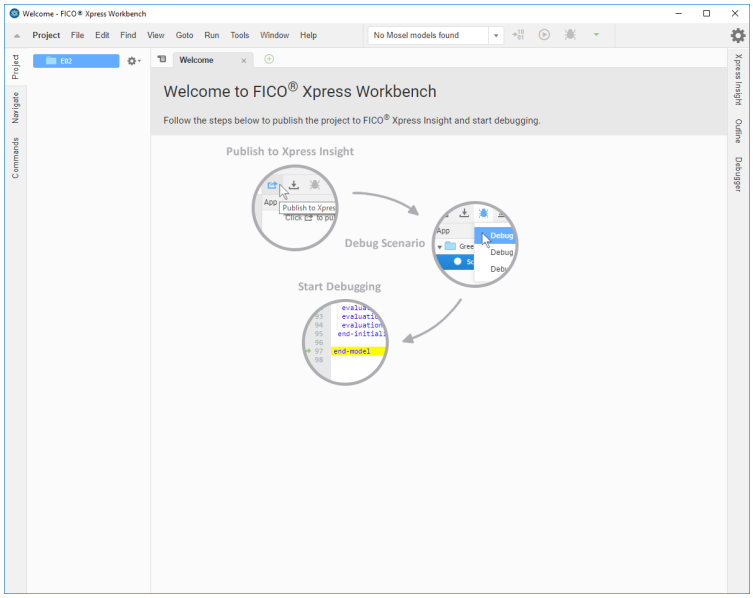

Figure 2: Empty project.

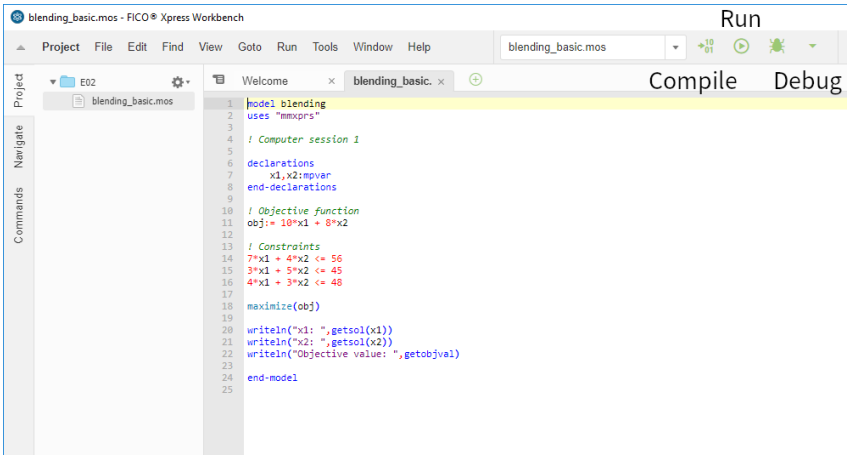

Figure 3: Blending model.

```
! Objective function
revenue:= 10*x1 + 8*x2
```

Then, we define the constraints, one per line.

```
! Constraints
7*x1 + 4*x2 <= 56
3*x1 + 5*x2 <= 45
4*x1 + 3*x2 <= 48
```

Note that we have to write the multiplication symbol "`*`" between the coefficient and the variable (unless it is 0 or 1, in which case nothing is written or only the variable is written, respectively). Remember also that the constraints that we will use are only of the type "$\leq$", "$\geq$", and "=". We will *never* use strict inequalities with "$<$" or "$>$".

An important remark is that we do not need to add explicitly that the variables are non-negative. Unless stated otherwise, Xpress assumes that all the variables are nonnegative and continuous.

Finally, we tell the solver that we are going to maximize the objective function:

```
maximize(revenue)
```

The last line says that the model is completed:

```
end-model
```

This model can be compiled (which automatically saves the file) with `Ctrl-B`. Or we can run it with `Ctrl-F5` (which saves and compiles before that). Alternatively, you can use the buttons shown in Figure 3.

Once we have run the model, we see – hopefully – that the model ran successfully as in Figure 4. This is, however, not particularly useful. After all, we want to know the result!

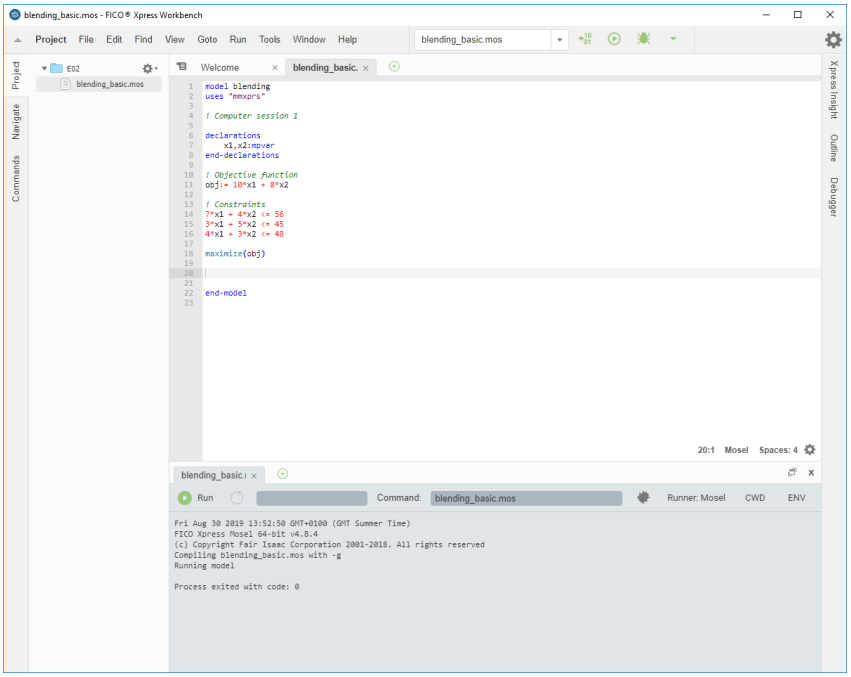

Figure 4: Output for the blending model.

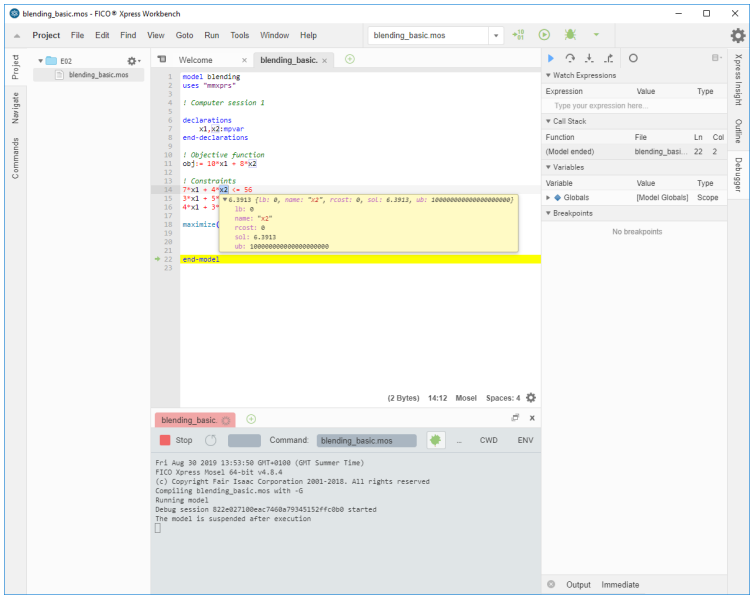

Figure 5: Variables in Debug mode.

There are two ways of getting that information: One is to use *Debug mode* and the other is to print the relevant information to the main output screen or into a file.

Using *Debug mode* is good during the developement and testing of a model. To see what it does, we can run our model in *Debug mode* by pressing F5 or using the button shown in Figure 3. In this mode, the model is run and stopped right before it finishes. Then we can access data about the solution, eg the value of a variable (Figure 5) or, on the right hand side, the values of all named expressions (Figure 6).

But this is not a good solution, when we want to implement the solution later. In a typical model, we have much information that is not interesting for the practioner. So, we need to output the parts of the solution that are of interest to us. For the beginning, we just output them to the main output screen. Add the following lines before the last line of the model and run the model again:

```
writeln("x1: ",getsol(x1))
writeln("x2: ",getsol(x2))
writeln("Objective value: ",getobjval)
```

Now, the results are shown. We will later discuss how to write output to a file.

2 The blending problem with general data

The model we have written has a serious deficiency: if we change the data, we will have to modify individually each entry, which can be highly time consuming for problems with

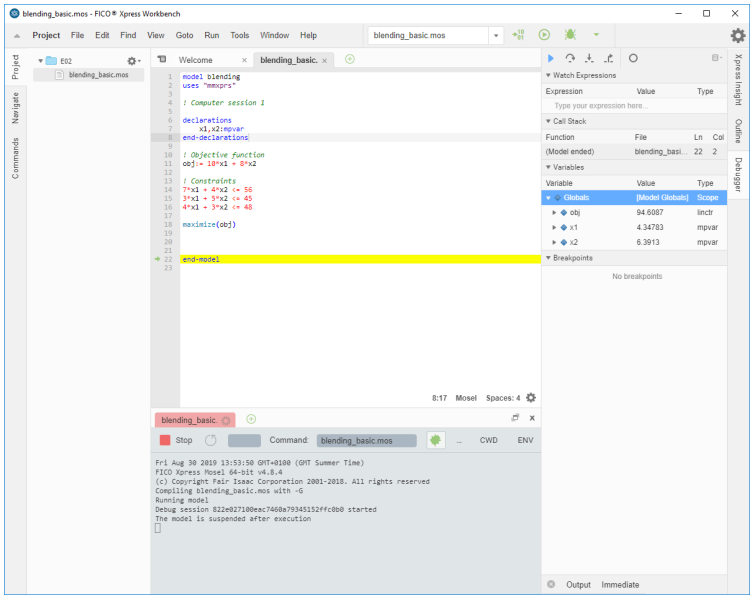

Figure 6: Data in Debug mode.

many variables and constraints. Therefore, we are going to use arrays and read the data from files so that we can solve *any problem with the same structure*.

2.1 Using sets

A first approach is to define sets of indices. We replace the previous declarations block with:

```
declarations
        products = 1..2
        resources = 1..3
        benefit: array(products) of real
        limit: array(resources) of real
        use:array(resources,products) of real
        make: array(products) of mpvar
end-declarations
```

We have defined two sets of indices: `products`, which has two elements, and `resources`, which has three. Then, we define several unidimensional arrays. We also define a bidimensional matrix (`use`). In this case, the first argument is for rows and the second is for columns. Be careful with not reversing the order of the parameters or you will end up with the transposed matrix of the one that you actually want.

This only provides information about the dimensions. Next, we fill the arrays with data:

```
benefit:: [10,8]
limit:: [56,45,48]
```

```
use:: [7,4,3,5,4,3]
```

Note that it is possible to write the data in a format which is visually closer to how we write the table in our notes:

```
use:: [7,4,
       3,5,
       4,3]
```

Now, we have to rewrite the model taking into account that all the elements are indexed with the previous sets. The objective function is:

```
total_benefit:= sum(p in products) benefit(p)*make(p)
```

And the constraints are written as follows:

```
forall(r in resources) do
        sum(p in products) use(r,p)*make(p) <= limit(r)
end-do
```

Finally, the code displaying the solution needs to be modified too:

```
forall(p in products) do
        writeln("Amount produced of product ",p,": ",getsol(make(p)),".")
end-do
writeln
writeln("The net benefit is ",getobjval,".")
```

Figure 7 shows the full code.

2.2 Initializing from files

Instead of including the data explicitly in the code file, there is the option of having external files and reading this information from them. In order to so, we need first to have a file `blending.dat` where all the information is stored. The file must be included in the same folder where your Mosel file is and it will look like this:

```
benefit: [10 8]
limit: [56 45 48]
use: [7 4 3 5 4 3]
```

Note two differences with respect to how we wrote the arrays in the Mosel file:

- Now we use a single colon (:) instead of double (::).
- We do no write commas to separate the data.

Next we need to replace the lines defining the values of the arrays with the following code:

```
initializations from "blending.dat"
        benefit limit use
end-initializations
```

As an alternative, we can use sets of strings to give names to the indices:

```
model blending
uses "mmxprs"

! Computer session 1

declarations
        products = 1..2
        resources = 1..3
        benefit: array(products) of real
        limit: array(resources) of real
        use:array(resources,products) of real
        make: array(products) of mpvar
end-declarations

! Initialize values
benefit:: [10,8]
limit:: [56,45,48]
use:: [7,4,3,5,4,3]

! Objective function
total_benefit:= sum(p in products) benefit(p)*make(p)

! Constraints
forall(r in resources) do
        sum(p in products) use(r,p)*make(p) <= limit(r)
end-do

maximize(total_benefit)

forall(p in products) do
        writeln("Amount produced of product ",p,": ",getsol(make(p)),".")
end-do
writeln
writeln("The net benefit is ",getobjval,".")

end-model
```

Figure 7: Blending model using sets.

```
declarations
        products, resources: set of string
        benefit: array(products) of real
        limit: array(resources) of real
        use:array(resources,products) of real
        make: array(products) of mpvar
end-declarations

! Initialize values
initializations from "blending2.dat"
         products resources benefit limit use
end-initializations
```

File `blending2.dat` is file `blending.dat` with two new lines:

```
products: ["A" "B"]
resources: ["iron" "lead" "tin"]
benefit: [10 8]
limit: [56 45 48]
use: [7 4 3 5 4 3]
```

You can see the full code in Figure 8.

Mosel has additional modules to initalize data from different file types, notably, you can directly initialize from MS Excel files – the module for that is called `mmsheet`.

3 Final remarks

The purpose of a model written in a modeling language is twofold: The computer needs to be able to understand it and the people working with the model need to understand it. In the beginning, your focus will be on getting the computer to understand the model, that is you will focus on getting the model to compile and give the correct result. But in practice it is very important that your model is understandable to other humans or – also important – to *you* if you have not looked at it for some time.

Some good practices are

- **Divide the model clearly in blocks.**
- **Add comments that give information about the blocks.**
- **Use meaningful names.**
- **Use version control for your files. At least, backup your files often.**

The last point is important: You don't want to lose your model, but you will also want to know which version of the model is current.

When using comments, there is always a trade off between being too short and repeating information that is already given in the model. I personally tend to use longer variable and constraint names and less comments, but this is a matter of taste. The important point is: Can someone who knows nothing about the specific model understand the model? What

```
model blending
uses "mmxprs"

! Computer session 1

declarations
        products = 1..2
        resources = 1..3
        benefit: array(products) of real
        limit: array(resources) of real
        use:array(resources,products) of real
        make: array(products) of mpvar
end-declarations

! Initialize values
initializations from "blending.dat"
        benefit limit use
end-initializations

! Objective function
total_benefit:= sum(p in products) benefit(p)*make(p)

! Constraints
forall(r in resources) do
        sum(p in products) use(r,p)*make(p) <= limit(r)
end-do

maximize(total_benefit)

forall(p in products) do
        writeln("Amount produced of product ",p,": ",getsol(make(p)),".")
end-do
writeln
writeln("The net benefit is ",getobjval,".")

end-model
```

Figure 8: Blending model using sets and file input.

additional information do they need? If you can assume that the model is only distributed together with a written report, then it is very important that names are consistent between the report and the model. It is not so important that names are understandable on their own.

For more detailed Mosel specific recommendations, have a look at the official Mosel model building style recommendations.