

FICO® Xpress Optimization

Last update 6 July, 2020

8.10

USER GUIDE

Getting Started with Xpress

FICO® **Decisions**

©2003–2020 Fair Isaac Corporation. All rights reserved. This documentation is the property of Fair Isaac Corporation ("FICO"). Receipt or possession of this documentation does not convey rights to disclose, reproduce, make derivative works, use, or allow others to use it except solely for internal evaluation purposes to determine whether to purchase a license to the software described in this documentation, or as otherwise set forth in a written software license agreement between you and FICO (or a FICO affiliate). Use of this documentation and the software described in it must conform strictly to the foregoing permitted uses, and no other use is permitted.

The information in this documentation is subject to change without notice. If you find any problems in this documentation, please report them to us in writing. Neither FICO nor its affiliates warrant that this documentation is error-free, nor are there any other warranties with respect to the documentation except as may be provided in the license agreement. FICO and its affiliates specifically disclaim any warranties, express or implied, including, but not limited to, non-infringement, merchantability and fitness for a particular purpose. Portions of this documentation and the software described in it may contain copyright of various authors and may be licensed under certain third-party licenses identified in the software, documentation, or both.

In no event shall FICO or its affiliates be liable to any person for direct, indirect, special, incidental, or consequential damages, including lost profits, arising out of the use of this documentation or the software described in it, even if FICO or its affiliates have been advised of the possibility of such damage. FICO and its affiliates have no obligation to provide maintenance, support, updates, enhancements, or modifications except as required to licensed users under a license agreement.

FICO is a registered trademark of Fair Isaac Corporation in the United States and may be a registered trademark of Fair Isaac Corporation in other countries. Other product and company names herein may be trademarks of their respective owners.

FICO® Xpress Optimization

Deliverable Version: A

Last Revised: 6 July, 2020

Version 8.10

Contents

Preface	1
Whom this book is intended for	1
How to read this book	1
Using the Mosel language with Xpress Workbench	2
Working in a programming language environment	2
1 Introduction	4
1.1 Mathematical Programming	4
1.2 Xpress product suite	5
1.2.1 Note on product versions	7
2 Building models	8
2.1 Example problem	8
 I Getting started with Mosel	 11
3 Inputting and solving a Linear Programming problem	12
3.1 Starting up Xpress Workbench and creating a new model	12
3.2 LP model	14
3.2.1 General structure	15
3.2.2 Solving	15
3.2.3 Output printing	15
3.2.4 Formating	16
3.3 Correcting errors and debugging a model	16
3.3.1 Debugging	17
3.4 Solving and viewing the solution	18
3.4.1 String indices	20
4 Working with data	22
4.1 Data input from file	22
4.2 Formated data output to file	23
4.3 Parameters	24
4.4 Complete example	25
5 Drawing user graphs	27
5.1 Extended problem description	27
5.2 Looping over optimization	27
5.3 Drawing a user graph	28
5.4 Complete example	30
6 Mixed Integer Programming	32
6.1 Extended problem description	32
6.2 MIP model 1: limiting the number of different shares	32
6.2.1 Implementation with Mosel	33
6.2.2 Analyzing the solution	34

6.3	MIP model 2: imposing a minimum investment in each share	36
6.3.1	Implementation with Mosel	36
7	Quadratic Programming	38
7.1	Problem description	38
7.2	QP	39
7.2.1	Implementation with Mosel	39
7.3	MIQP	41
7.3.1	Implementation with Mosel	42
7.3.2	Analyzing the solution	42
8	Heuristics	44
8.1	Binary variable fixing heuristic	44
8.2	Implementation with Mosel	44
8.2.1	Subroutines	46
8.2.2	Optimizer parameters and functions	47
8.2.3	Comparison tolerance	48
9	Embedding a Mosel model in an application	49
9.1	Generating a deployment template	49
9.2	BIM files	50
9.3	Embedding Mosel models into a host application	51
9.3.1	Executing Mosel models	51
9.3.2	Parameters	52
9.3.3	Retrieving solution information	52
9.4	Matrix files	53
9.4.1	Exporting matrices	53
9.5	Deployment to Xpress Insight	53
9.5.1	Preparing the model file	53
9.5.1.1	The app archive	55
9.5.2	Working with the Xpress Insight Web Client	56
9.5.2.1	VDL	60
II	Getting started with BCL	65
10	Inputting and solving a Linear Programming problem	66
10.1	Implementation with BCL	66
10.1.1	Initialization	67
10.1.2	General structure	67
10.1.3	Solving	68
10.1.4	Output printing	68
10.2	Compilation and program execution	68
10.3	Data input from file	69
10.4	Output functions and error handling	71
10.5	Exporting matrices	72
11	Mixed Integer Programming	73
11.1	Extended problem description	73
11.2	MIP model 1: limiting the number of different shares	73
11.2.1	Implementation with BCL	74
11.2.2	Analyzing the solution	75
11.3	MIP model 2: imposing a minimum investment in each share	76
11.3.1	Implementation with BCL	77
12	Quadratic Programming	79

12.1 Problem description	79
12.2 QP	79
12.2.1 Implementation with BCL	80
12.3 MIQP	82
12.3.1 Implementation with BCL	83
13 Heuristics	85
13.1 Binary variable fixing heuristic	85
13.2 Implementation with BCL	86
III Getting started with the Optimizer	91
14 Matrix input	92
14.1 Matrix files	92
14.2 Implementation	92
14.3 Compilation and program execution	93
15 Inputting and solving a Linear Programming problem	95
15.1 Matrix representation	95
15.2 Implementation with Xpress Optimizer	96
15.3 Compilation and program execution	97
16 Mixed Integer Programming	99
16.1 Extended problem description	99
16.2 MIP model 1: limiting the number of different shares	99
16.2.1 Matrix representation	100
16.2.2 Implementation with Xpress Optimizer	100
16.3 MIP model 2: imposing a minimum investment in each share	102
16.3.1 Matrix representation	102
16.3.2 Implementation with Xpress Optimizer	102
17 Quadratic Programming	105
17.1 Problem description	105
17.2 QP model	105
17.3 Matrix representation	106
17.4 Implementation with Xpress Optimizer	106
Appendix	110
A Going further	111
A.1 Installation, licensing, and trouble shooting	111
A.2 User guides, reference manuals, and other publications	111
A.2.1 Modeling	111
A.2.2 Mosel	111
A.2.3 BCL	112
A.2.4 Optimizer	112
A.2.5 Other solvers and solution methods	112
B Glossary	113
C Contacting FICO	117
Product support	117
Product education	117
Product documentation	117

Sales and maintenance 118

Related services 118

FICO Community 118

About FICO 118

Index **119**

Preface

'Getting Started' is a quick and easy-to-understand introduction to modeling and solving different types of optimization problems with FICO Xpress Optimization. It shows how Linear, Mixed-Integer, and Quadratic Programming problems are formulated with the Mosel language and solved by Xpress Optimizer. We work with these Mosel models by means of the graphical user interface Xpress Workbench. Two alternatives to using this high-level language are also discussed: a model may be defined in a programming language environment using the model builder library Xpress BCL or directly input into the Optimizer in the form of a matrix.

Throughout this book we employ variants of a single problem, namely optimal portfolio selection. To readers who are interested in other types of optimization problems we recommend the book 'Applications of Optimization with Xpress-MP' (Dash Optimization, 2002), see also

http://examples.xpress.fico.com/example.pl#mosel_app

This book shows how to formulate and solve a large number of application problems with Xpress.

A short introduction such as the present book highlights certain features but necessarily remains incomplete. The interested reader is directed to various other documents available from the [Xpress online documentation website](#), such as the user guides and reference manuals for the various pieces of software of the FICO Xpress Optimization suite (Xpress Solver, Mosel language, Mosel modules, etc.) and the collection of white papers on modeling topics. A list of the available documentation is given in the appendix.

Whom this book is intended for

This book is an ideal starting point for software *evaluators* as it gives an overview of the various Xpress products and shows how to get up to speed quickly through experimenting with the models discussed via a high-level language used in a graphical environment.

Starting from a simple linear model, every chapter adds new features to it. *First time users* are taken in small steps from the textual description, via the mathematical model to a complete application (Chapter 9) or the implementation of a solution heuristic that involves some more advanced optimization tasks (Chapter 8).

The variety of topics covered may also help *occasional users* to quickly refresh their knowledge of Mosel, Workbench, BCL and the Optimizer.

How to read this book

For a complete overview and introduction to modeling and solving with the FICO Xpress Optimization product suite, we recommend reading the entire document. However, readers who are only interested in certain topics, may well skip certain parts or chapters as shown in the following diagram.

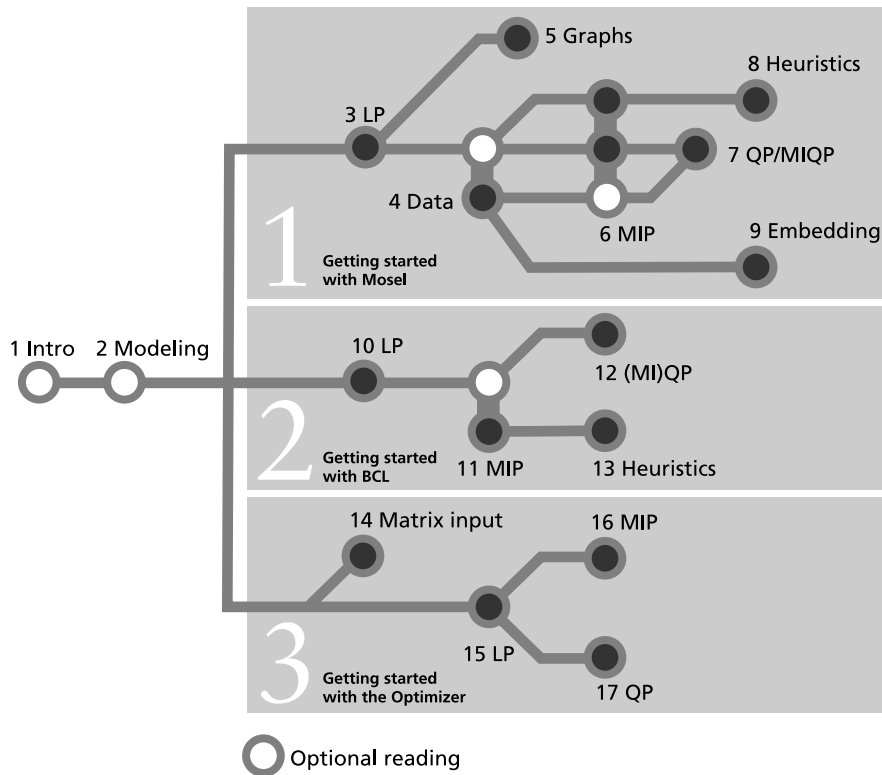


Figure 1: Suggested flow through the book

Using the Mosel language with Xpress Workbench

The approach presented in the first part of this book is recommended for first time users, novices to Mathematical Programming, and users who wish to develop and deploy new models quickly, supported by graphical displays for problem and solution analysis.

For example, if you wish to develop a Linear Programming (LP) model and embed it into some existing application, you should read the first four chapters, followed by Chapter 9 on embedding Mosel models.

To find out how to model and solve Quadratic Programming (QP) problems with Xpress, you should read at least Chapters 1-3, the beginning of Chapter 4 and then Chapter 7; for Mixed Integer Quadratic Programming (MIQP) also include Chapter 6 on Mixed Integer Programming (MIP).

To see how you may implement your own solution algorithms and heuristics in the Mosel language, we suggest reading Chapters 1-3, the beginning of Chapter 4, followed by Chapter 6 on MIP and then Chapter 8 on Heuristics.

Working in a programming language environment

Users who wish to develop their entire application in a programming language environment have two options, using the model builder library BCL or inputting their problem directly into Xpress Optimizer.

Users who are looking for modeling support whilst model execution speed is a decisive factor in their choice of the tool should look at the model builder library BCL. Due to the modeling objects defined by BCL, the resulting code remains relatively close to the algebraic model and is easy to maintain. BCL supports modeling of LP, MIP, and QP problems (Chapters 10-12). Model input with BCL may be combined with direct calls to Xpress Optimizer to define solution algorithms as shown with the example in Chapter 13.

The direct access to the Optimizer (discussed in the last part) is provided mainly for low-level integration with applications that possess their own matrix generation routines (Chapters 15-17 for LP, MIP, and QP problems), or to solve matrices given in standard format (MPS or LP) that were generated externally (Chapter 14). The possibility to directly access very specific features of the Optimizer is also appreciated by advanced users, mostly in the domain of research, who implement their own algorithms involving the solution of LP, MIP, or QP problems.

CHAPTER 1

Introduction

1.1 Mathematical Programming

Mathematical Programming is a technique of mathematical optimization. Many real-world problems in such different areas as industrial production, transport, telecommunications, finance, or personnel planning may be cast into the form of a *Mathematical Programming problem*: a set of decision variables, constraints over these variables and an objective function to be maximized or minimized.

Mathematical Programming problems are usually classified according to the types of the decision variables, constraints, and the objective function.

A well-understood case for which efficient algorithms (Simplex, interior point) are known comprises *Linear Programming (LP)* problems. In this type of problem all constraints and the objective function are linear expressions of the decision variables, and the variables have continuous domains—i.e., they can take on any, usually non-negative, real values. Luckily, many application problems fit into this category. Problems with hundreds of thousands, or even millions of variables and constraints are routinely solved with commercial Mathematical Programming software like Xpress Optimizer.

Researchers and practitioners working on LP quickly found that continuous variables are insufficient to represent decisions of a discrete nature ('yes'/'no' or 1,2,3,...). This observation led to the development of *Mixed Integer Programming (MIP)* where constraints and objective function are linear just as in LP and variables may have either discrete or continuous domains. To solve this type of problems, LP techniques are coupled with an enumeration (known as *Branch-and-Bound*) of the feasible values of the discrete variables. Such enumerative methods may lead to a computational explosion, even for relatively small problem instances, so that it is not always realistic to solve MIP problems to optimality. However, in recent years, continuously increasing computer speed and even more importantly, significant algorithmic improvements (e.g. cutting plane techniques and specialized branching schemes) have made it possible to tackle ever larger problems, modeling ever more exactly the underlying real-world situations.

Another class of problems that is relatively well-handled are *Quadratic Programming (QP)* problems: these differ from LPs in that they have quadratic terms in the objective function (the constraints remain linear). The decision variables may be continuous or discrete, in the latter case we speak of *Mixed Integer Quadratic Programming (MIQP)* problems. In Chapters 7 and 12 of this book we show examples of both cases.

More difficult is the case of non-linear constraints or objective functions, *Non-linear Programming (NLP)* problems. Frequently heuristic or approximation methods are employed to find good (locally optimal) solutions. One method for solving problems of this type is *Successive Linear Programming (SLP)*; such a solver forms part of the FICO Xpress Optimization suite. However, in this book we shall not enlarge on this topic.

Building a model, solving it and then implementing the 'answers' is not generally a linear process. We often make mistakes in our modeling which are usually only detected by the optimization process, where we could get answers that were patently wrong (e.g. unbounded or infeasible) or that do not

accord with our intuition. If this happens we are forced to reflect further about the model and go into an iterative process of model refinement, re-solution and further analyses of the optimum solution. During this process it is quite likely that we will add extra constraints, perhaps remove constraints that we were misled into adding, correct erroneous data or even be forced to collect new data that we had previously not considered necessary.

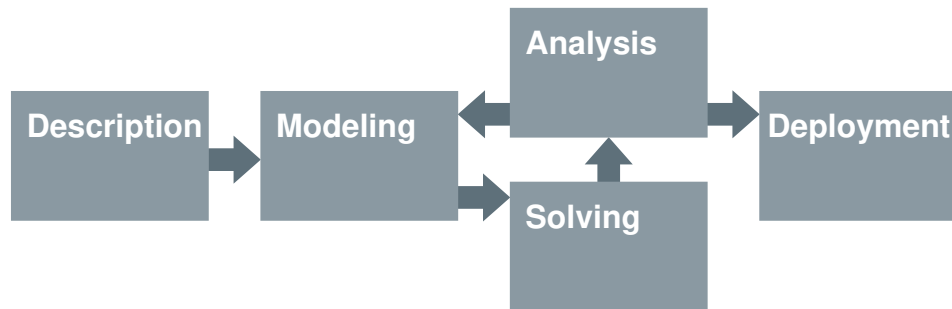


Figure 1.1: Scheme of an optimization project

This book takes the reader through all these steps: from the textual description we develop a mathematical model which is then implemented and solved. Various improvements, additions and reformulations are suggested in the following chapters, including an introduction of the available means to support the analysis of the results. The deployment of a Mathematical Programming application typically includes its embedding into other applications to turn it into a part of a company's information system.

1.2 Xpress product suite

Arising from different users' needs and preferences, there are several ways of working with the modeling and optimization tools that form the FICO Xpress Optimization product suite:

1. *High-level language:* the *Xpress Mosel language* allows the user to define his models in a form that is close to algebraic notation and to solve them in the same environment. Mosel's programming facilities also make it possible to implement solution algorithms directly in this high-level language. Mosel may be used as a standalone program or through the *Xpress Workbench* development environment that provides, amongst many other tools, Mosel syntax and debugging support.
Via the concept of *modules* the Mosel environment is entirely open to additions; modules of the Xpress distribution include access to solvers (Xpress Optimizer for LP, MIP, and convex QP, Xpress Nonlinear, and Xpress Kalis), data handling facilities (e.g. via ODBC), access to system functions, graphing capabilities, distributed and remote computing functionality via the *Mosel Distributed Framework*, and also interfaces to statistics packages such as R or Matlab. In addition, via the *Mosel Native Interface* users may define their own modules to add new features to the Mosel language according to their needs (e.g. to implement problem-specific data handling, or connections to external solvers or solution algorithms).
2. *Deployment as a web app:* Mosel models can be deployed via Xpress Insight as multi-user web apps running locally, on-premises or on a cloud. Insight web apps are configured via a set of XML files that are packaged into an archive along with the Mosel model and its input data.
3. *Libraries for embedding:* two different options are available for embedding mathematical models into host applications. A model developed using the Mosel language may be executed and accessed from a programming language environment (e.g. C, C++, Java, etc.) through the *Mosel*

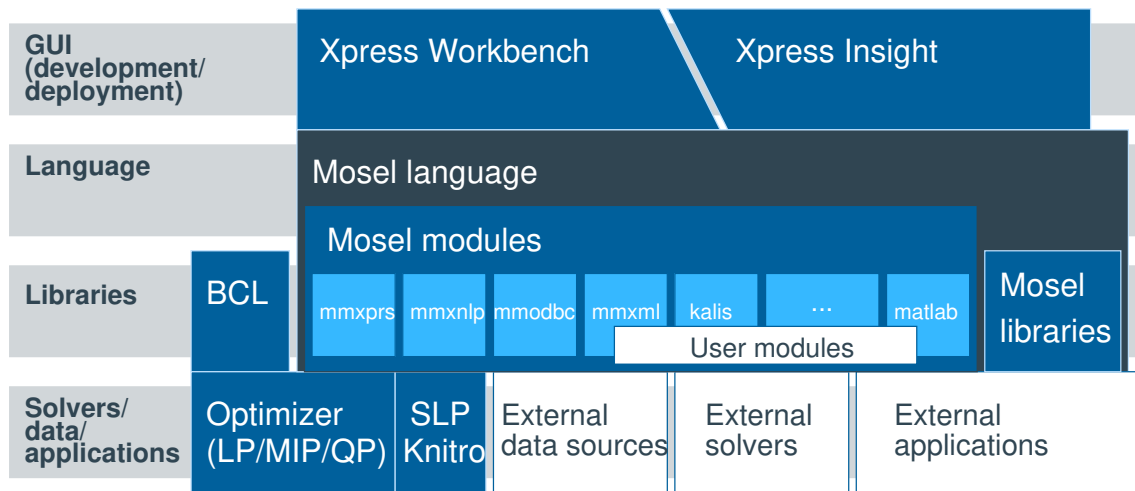


Figure 1.2: Xpress product suite

libraries; certain modules also provide direct access to their functions from a programming language environment.

The second possibility consists of developing a model directly in a programming language with the help of the model builder library *Xpress-BCL*. BCL allows the user to formulate his models with objects (decision variables, constraints, index sets) similar to those of a dedicated modeling language.

All libraries are available for C, C++, Java, C#, and Visual Basic (VBA).

4. *Direct access to solvers*: on the lowest, most immediate level, it is possible to work directly with the Xpress Optimizer or Xpress NonLinear in the form of a library or a standalone program. This facility may be useful for embedding Xpress Optimizer into applications that possess their own, dedicated matrix generation routines.
Advanced Xpress users may wish to employ special features of Xpress Optimizer that are not available through the different interfaces, possibly using a matrix that has previously been generated by Mosel or BCL.

Of the three above mentioned approaches, a high-level language certainly provides the easiest-to-understand access to Mathematical Programming. So in the first and largest part of this book we show how to define and solve problems with the Xpress Mosel language, and also how the resulting models may be embedded into applications using the Mosel libraries or Xpress Insight. We work with Mosel models in the graphical user interface Xpress Workbench, exploiting its facilities for debugging and solution analysis and display.

In the reminder of this book we show how to formulate and solve Mathematical Programming problems directly in a programming language environment. This may be done with modeling support from BCL or directly using the Optimizer library. With BCL, models can be implemented in a form that is relatively close to their algebraic formulation and so are quite easy to understand and to maintain. We discuss BCL implementations of the same example problems as used with Mosel.

The last part of this book explains how problems may be input directly into Xpress Optimizer, either in the form of matrices (possibly generated by another tool such as Mosel or BCL) that are read from file, or by specifying the problem matrix coefficient-wise directly in the application program. The facility of working directly with the Xpress Optimizer library is destined at embedders and advanced Xpress users. It is not recommendable as a starting point for the novice in Mathematical Programming.

1.2.1 Note on product versions

The Mosel examples in this book have been updated to the FICO Xpress Optimization Release 8.7 (Mosel 5.0); Xpress Workbench screenshots have been taken with Release 8.7 (Workbench version 3.1.0). The Xpress Insight examples have been developed with Xpress Release 8.7 (Insight 4.51). The BCL examples are using BCL 4.8.11 that is distributed with Xpress Release 8.3. The Xpress Optimizer examples have equally been updated to the Xpress Release 8.3 (Optimizer 31.01.09). If the examples are run with other product versions the output obtained may look different. In particular, improvements to the algorithms or modifications to the default settings in the Optimizer may influence the behavior of the LP search or the shape of the MIP branching trees. The Xpress Workbench interface may also undergo slight changes in future releases as new features are added, but this will not affect the actions described in this book.

CHAPTER 2

Building models

This chapter shows in detail how the textual description of a real world problem is converted into a mathematical model. We introduce an example problem, optimal portfolio selection, that will be used throughout this book.

Though not requiring any prior experience of Mathematical Programming, when formulating the mathematical models we assume that the reader is comfortable with the use of symbols such as x or y to represent unknown quantities, and the use of this sort of variable in simple linear equations and inequalities, for example:

$$x + y \leq 6$$

which says that ‘the quantity represented by x plus the quantity represented by y must be less than or equal to six’.

You should also be familiar with the idea of summing over a set of variables. For example, if $produce_i$ is used to represent the quantity produced of product i then the total production of all items in the set $ITEMS$ can be written as:

$$\sum_{i \in ITEMS} produce_i$$

This says ‘sum the produced quantities $produce_i$ over all products i in the set $ITEMS$ ’.

Another common mathematical symbol that is used in the text is the all-quantifier $\forall$ (read ‘for all’): if $ITEMS$ consists in the elements 1, 4, 7, 9 then writing

$$\forall i \in ITEMS : produce_i \leq 100$$

is a shorthand for

$$produce_1 \leq 100$$

$$produce_4 \leq 100$$

$$produce_7 \leq 100$$

$$produce_9 \leq 100$$

Computer based modeling languages, and in particular the language we use, Mosel, closely mimic the mathematical notation an analyst uses to describe a problem. So provided you are happy using the above mathematical notation the step to using a modeling language will be straightforward.

2.1 Example problem

An investor wishes to invest a certain amount of money. He is evaluating ten different securities (‘shares’) for his investment. He estimates the return on investment for a period of one year. The

following table gives for each share its country of origin, the risk category (R: high risk, N: low risk) and the expected return on investment (ROI). The investor specifies certain constraints. To spread the risk he wishes to invest at most 30% of the capital into any share. He further wishes to invest at least half of his capital in North-American shares and at most a third in high-risk shares. How should the capital be divided among the shares to obtain the highest expected return on investment?

Table 2.1: List of shares with countries of origin and estimated return on investment

Number	Description	Origin	Risk	ROI
1	treasury	Canada	N	5
2	hardware	USA	R	17
3	theater	USA	R	26
4	telecom	USA	R	12
5	brewery	UK	N	8
6	highways	France	N	9
7	cars	Germany	N	7
8	bank	Luxemburg	N	6
9	software	India	R	31
10	electronics	Japan	R	21

To construct a mathematical model, we first identify the *decisions* that need to be taken to obtain a solution: in the present case we wish to know how much of every share to take into the portfolio. We therefore define *decision variables* $frac_s$ that denote the fraction of the capital invested in share s . That means, these variables will take fractional values between 0 and 1 (where 1 corresponds to 100% of the total capital). Indeed, every variable is *bounded* by the maximum amount the investor wishes to spend per share: at most 30% of the capital may be invested into every share. The following constraint establishes these bounds on the variables $frac_s$ (read: ‘for all s in SHARES ...’).

$$\forall s \in SHARES : 0 \leq frac_s \leq 0.3$$

In the mathematical formulation, we write $SHARES$ for the set of shares that the investor may wish to invest in and RET_s the expected ROI per share s . NA denotes the subset of the shares that are of North-American origin and $RISK$ the set of high-risk values.

The investor wishes to spend all his capital, that is, the fractions spent on the different shares must add up to 100%. This fact is expressed by the following *equality constraint*:

$$\sum_{s \in SHARES} frac_s = 1$$

We now also need to express the two constraints that the investor has specified: At most one third of the values may be high-risk values—i.e., the sum invested into this category of shares must not exceed 1/3 of the total capital:

$$\sum_{s \in RISK} frac_s \leq 1/3$$

The investor also insists on spending at least 50% on North-American shares:

$$\sum_{s \in NA} frac_s \geq 0.5$$

These two constraints are *inequality constraints*.

The investor’s objective is to maximize the return on investment of all shares, in other terms, to maximize the following sum:

$$\sum_{s \in SHARES} RET_s \cdot frac_s$$

This is the *objective function* of our mathematical model.

After collecting the different parts, we obtain the following complete mathematical model formulation:

$$\begin{aligned} & \text{maximize} && \sum_{s \in SHARES} RET_s \cdot frac_s \\ & && \sum_{s \in RISK} frac_s \leq 1/3 \\ & && \sum_{s \in NA} frac_s \geq 0.5 \\ & && \sum_{s \in SHARES} frac_s = 1 \\ & && \forall s \in SHARES : 0 \leq frac_s \leq 0.3 \end{aligned}$$

In the next chapter we shall see how this mathematical model is transformed into a Mosel model that is then solved with Xpress Optimizer. In Chapter 10 we show how to use BCL for this purpose and Chapter 15 discusses how to input this model directly into the Optimizer without modeling support.

I. Getting started with Mosel

CHAPTER 3

Inputting and solving a Linear Programming problem

In this chapter we take the example formulated in Chapter 2 and show how to transform it into a Mosel model which is solved as an LP using Xpress Workbench. More precisely, this involves the following steps:

- starting up Xpress Workbench,
- creating and saving the Mosel file,
- using the Mosel language to enter the model,
- correcting errors and debugging the model,
- solving the model and understanding the displays in Workbench,
- viewing and verifying the solution and understanding the solution in terms of the real world problem instance.

Chapter 10 shows how to formulate and solve the same example with BCL and in Chapter 15 the problem is input and solved directly with Xpress Optimizer.

3.1 Starting up Xpress Workbench and creating a new model

We shall develop and execute our Mosel model with the graphical environment Xpress Workbench. If you have followed the standard installation procedure for Xpress, start the program;

- In Windows, double click the Workbench icon  on the desktop or select *Start >> Programs >> FICO >> Xpress >> Xpress Workbench*. Otherwise, you can start up Workbench by typing `xpworkbench` at the command prompt.
- On the Mac, you may have created a shortcut during the installation by dragging the Workbench icon  to the Dock. Otherwise, type 'Workbench' into Spotlight, or open *Applications >> FICO Xpress* and double click the *Xpress Workbench* icon.

In either operating system, you can double click an installed model file (file with extension `.mos`) to start Workbench.

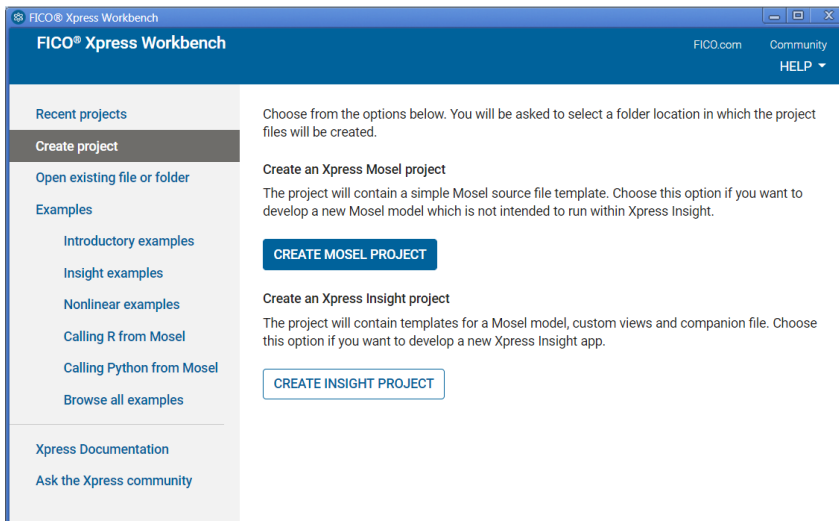


Figure 3.1: Workbench at startup

If you are starting Workbench without having selected any Mosel model you will see the entry screen shown in Figure 3.1 where you need to select the option *Create an Xpress Mosel project*. You will be prompted for location where to create the project. Browse to the desired location and select the folder name. After confirming with *Upload* you will see the welcome page of the Workbench workspace.

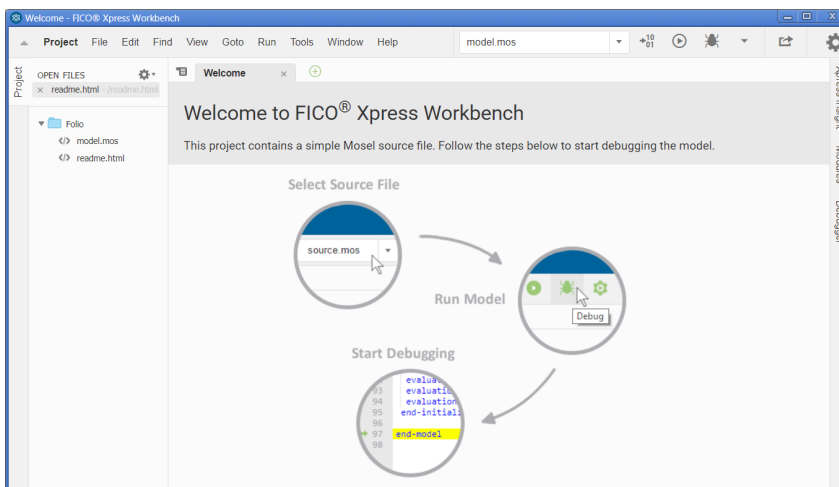


Figure 3.2: Workbench welcome page

Note: If you have started Workbench by selecting a model file this file is opened in the central editor window and the directory listing on its left lists all files in the same location.

The Xpress Workbench workspace window is subdivided into several panes:

At the top, we have the menu and tool bars. The central area is the *editor window* where the working file will be displayed, and at its bottom opens the *logging window* during model execution. The window on the left is the *project navigation and command history*, and the right window contains the *debugger* and *Xpress Insight* panes. You may configure the windows displayed via the *Window* menu.

To create a new model file select *File* » *New* » *Mosel file* or alternatively, double click on the template file `model1.mos` in the directory listing on the left to open it in the central editor window. Select *File* »

Save As... and enter `foliolp.mos` as the name of the new file. Click Save to confirm your choice. The central window of Workbench is now ready for you to enter the model into the displayed model input template.

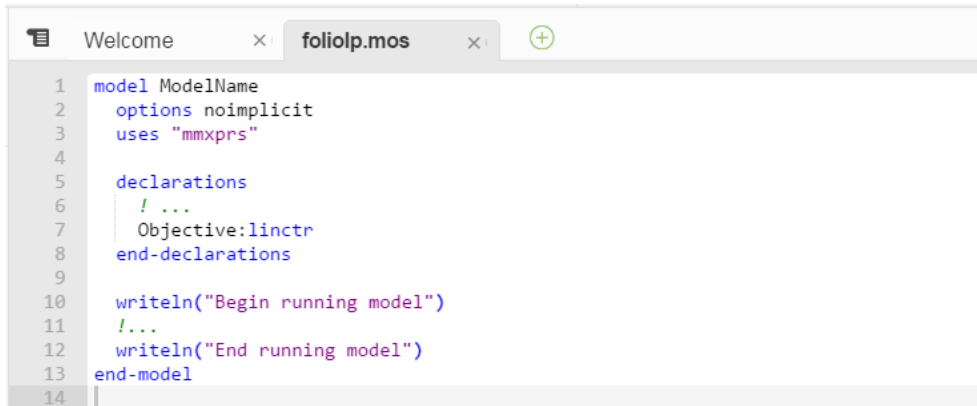


Figure 3.3: Mosel model template

3.2 LP model

The mathematical model in the previous chapter may be transformed into the following Mosel model entered into Workbench:

```

model "Portfolio optimization with LP"
  uses "mmxprs"                                ! Use Xpress Optimizer

  declarations
    SHARES = 1..10                               ! Set of shares
    RISK = {2,3,4,9,10}                           ! Set of high-risk values among shares
    NA = {1,2,3,4}                                ! Set of shares issued in N.-America
    RET: array(SHARES) of real                    ! Estimated return in investment

    frac: array(SHARES) of mpvar                  ! Fraction of capital used per share
  end-declarations

  RET:: [5,17,26,12,8,9,7,6,31,21]

  ! Objective: total return
  Return:= sum(s in SHARES) RET(s)*frac(s)

  ! Limit the percentage of high-risk values
  sum(s in RISK) frac(s) <= 1/3

  ! Minimum amount of North-American values
  sum(s in NA) frac(s) >= 0.5

  ! Spend all the capital
  sum(s in SHARES) frac(s) = 1

  ! Upper bounds on the investment per share
  forall(s in SHARES) frac(s) <= 0.3

  ! Solve the problem
  maximize(Return)

  ! Solution printing
  writeln("Total return: ", getobjval)
  forall(s in SHARES) writeln(s, ": ", getsol(frac(s))*100, "%")

```

```
end-model
```

Let us now try to understand what we have just written.

3.2.1 General structure

Every Mosel program starts with the keyword `model`, followed by a model name chosen by the user. The Mosel program is terminated with the keyword `end-model`.

All objects must be declared in a `declarations` section, unless they are defined unambiguously through an assignment (however, if you have kept the option 'noimplicit' from the Workbench model template then all entities must be declared). For example,

```
Return:= sum(s in SHARES) RET(s)*frac(s)
```

defines `Return` as a linear constraint and assigns to it the expression

```
sum(s in SHARES) RET(s)*frac(s)
```

There may be several such `declarations` sections at different places in a model.

In the present case, we define three sets, and two arrays:

- `SHARES` is a so-called *range set*—i.e., a set of consecutive integers (here: from 1 to 10).
- `RISK` and `NA` are simply *sets of integers*.
- `RET` is an array of real values indexed by the set `SHARES`, its values are assigned after the declarations.
- `frac` is an array of decision variables of type `mpvar`, also indexed by the set `SHARES`. These are the decision variables in our model.

The model then defines the objective function, two linear inequality constraints and one equality constraint and sets upper bounds on the variables.

As in the mathematical model, we use a *forall loop* to enumerate all the indices in the set `SHARES`.

3.2.2 Solving

With the procedure `maximize`, we call Xpress Optimizer to maximize the linear expression `Return`. As Mosel is itself not a solver, we specify that Xpress Optimizer is to be used with the statement

```
uses "mmxprs"
```

at the begin of the model (the module `mmxprs` is documented in the 'Mosel Language Reference Manual').

Instead of defining the objective function `Return` separately, we could just as well have written

```
maximize(sum(s in SHARES) RET(s)*frac(s))
```

3.2.3 Output printing

The last two lines print out the value of the optimal solution and the solution values for all variables.

To print an additional empty line, simply type `writeln` (without arguments). To write several items on a single line use `write` instead of `writeln` for printing the output.

3.2.4 Formatting

Indentation, spaces, and empty lines in our model have been added to increase readability. They are skipped by Mosel.

Line breaks: It is possible to place several statements on a single line, separating them by semicolons, like

```
RISK = {2,3,4,9,10}; NA = {1,2,3,4}
```

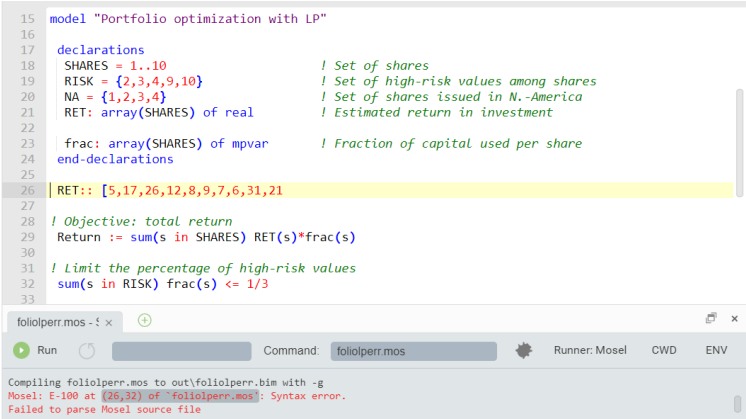
But since there are no special 'line end' or continuation characters, every line of a statement that continues over several lines must end with an operator (+, >=, etc.) or characters like ' , ' that make it obvious that the statement is not terminated.

As shown in the example, single line *comments* in Mosel are preceded by !. Comments over multiple lines start with (! and terminate with !).

3.3 Correcting errors and debugging a model

Having entered the model printed in the previous section, we now wish to execute it, that is, solve the optimization problem and retrieve the results. Choose *Run* >> *Run foliolp.mos* or alternatively, click on the run button:  making sure that the desired model filename is selected in the dropdown box next to it.

At a first attempt to run a model, you are likely to see the message 'Failed to parse Mosel source file'. The bottom window displays the error messages generated by Mosel, for instance as shown in the following figure (Figure 3.4), where clicking on the error message takes you to the indicated location in the Mosel file.



```

15 model "Portfolio optimization with LP"
16
17 declarations
18   SHARES = 1..10                                ! Set of shares
19   RISK = {2,3,4,9,10}                             ! Set of high-risk values among shares
20   NA = {1,2,3,4}                                  ! Set of shares issued in N.-America
21   RET: array(SHARES) of real                     ! Estimated return in investment
22
23   frac: array(SHARES) of mpvar                   ! Fraction of capital used per share
24 end-declarations
25
26 | RET: : [5,17,26,12,8,9,7,6,31,21]
27
28 ! Objective: total return
29 Return := sum(s in SHARES) RET(s)*frac(s)
30
31 ! Limit the percentage of high-risk values
32 sum(s in RISK) frac(s) <= 1/3
33

```

Figure 3.4: Logging output with error messages

When typing in the model from the previous section (there printed in its correct form), we have deliberately introduced some common mistakes that we shall now correct—see the example file `foliolperr.mos` for the erroneous model.

The first message:

```
Mosel: E-100 at (26,32) of `foliolperr.mos': Syntax error.
```

takes us to the line

```
RET: : [5,17,26,12,8,9,7,6,31,21]
```

We need to add the closing bracket to terminate the definition of `RET` (if the definition continues on the next line, we need to add a comma at the end of this line to indicate continuation).

The next messages that appear after re-running the model:

```
Mosel: E-100 at (29,9) of `foliolperr.mos': Syntax error before `='.
Mosel: E-123 at (29,9) of `foliolperr.mos': `Return' is not defined.
Mosel: E-124 at (29,42) of `foliolperr.mos': An expression cannot be used as a statement.
```

take us to the line

```
Return = sum(s in SHARES) RET(s)*frac(s)
```

Finding the error here requires taking a very close look: instead of `:=` we have used `=`. Since `Return` should have been defined by assigning it the sum on the right side, this statement now does not have any meaning.

After correcting this error, we try to run the model again, but we are still left with one error message:

```
Mosel: E-123 at (44,17) of `foliolperr.mos': `maximize' is not defined.
```

located in the line

```
maximize(Return)
```

The procedure `maximize` is defined in the module `mmxprs` but we have forgotten to add the line

```
uses "mmxprs"
```

at the beginning of the Mosel model. After adding this line, the model compiles correctly.

If you do not remember the correct name of a Mosel keyword while typing in a model, then you may use the *code completion* feature of the Workbench editor: while you are typing the editor brings up a list of suggestions with Mosel keywords and subroutines.

3.3.1 Debugging

If a model is correct from Mosel's point of view, this does of course not guarantee that it really does what we would like it to do. For instance, we may have forgotten to initialize data, or variables and constraints are not created correctly (they may be part of complex expressions, including logical tests etc.). To check what has indeed been generated by Mosel, we may pause the execution of the model immediately before it terminates by running the model in debug mode: select button  to run the model. The model will pause on the last statement to allow you to inspect the model entities in the *Debugger* window on the right side of the workspace window. Expand the entries under the heading *Variables* to view the definitions of individual model objects.

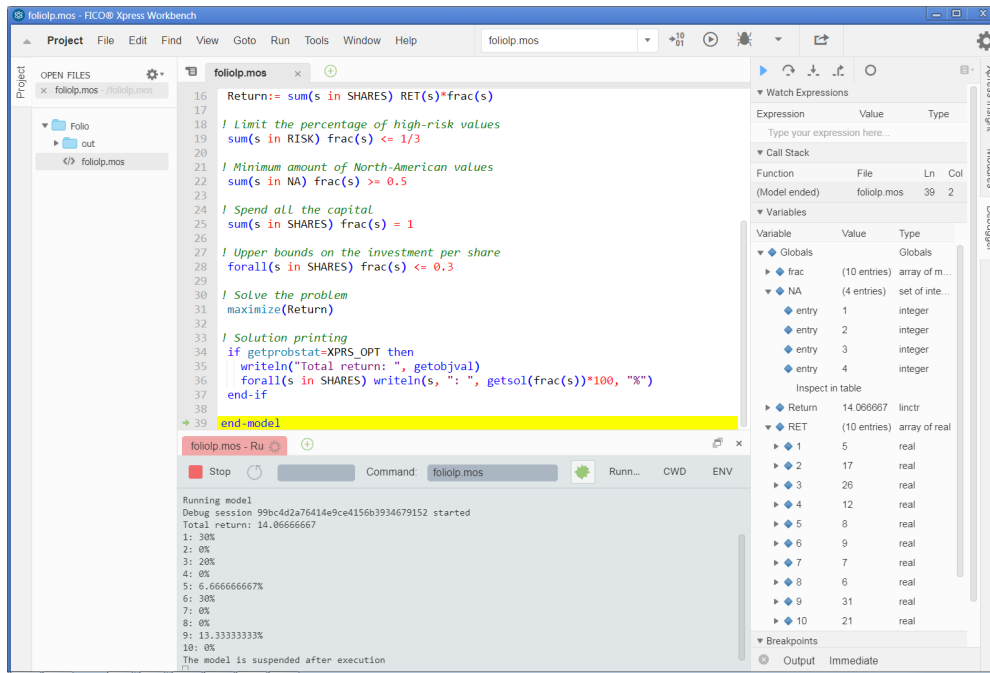


Figure 3.5: Workbench debugger

If you wish to display or trace the values of model entities at other locations you can set breakpoints by clicking onto the grey area in front of the line numbers and re-run the model in debug mode.

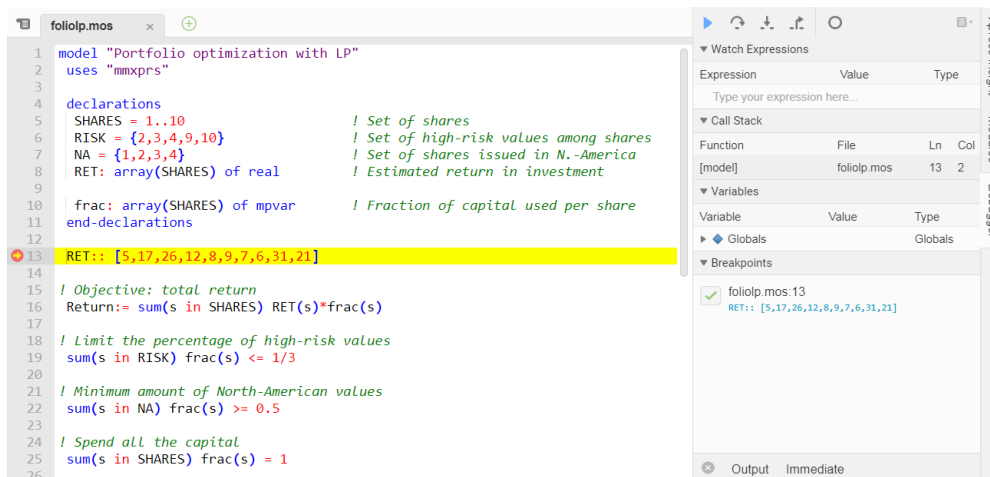


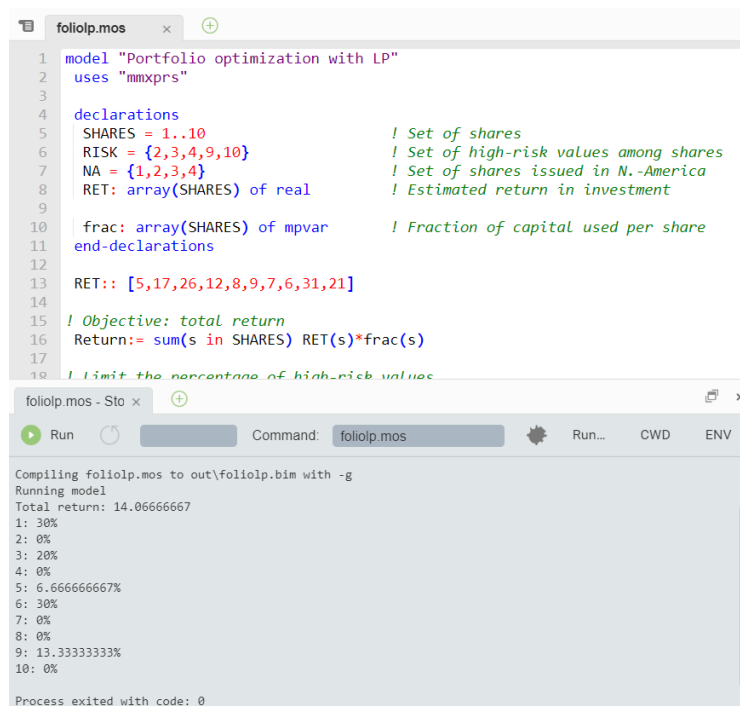
Figure 3.6: Debug run with breakpoint

The debugger controls at the top of the *Debugger* window (step over: , step into: , step out: ) allow you to step through the model line-by-line or resume/pause its execution ().

3.4 Solving and viewing the solution

As mentioned in the previous section, to execute our model we have to select *Run* $\gg$ *Run foliopl.mos* or alternatively, click on the run button:  After the successful execution of our model the screen display

changes to the following (Figure 3.7).



```

1  model "Portfolio optimization with LP"
2  uses "mmxprs"
3
4  declarations
5    SHARES = 1..10           ! Set of shares
6    RISK = {2,3,4,9,10}      ! Set of high-risk values among shares
7    NA = {1,2,3,4}           ! Set of shares issued in N.-America
8    RET: array(SHARES) of real ! Estimated return in investment
9
10   frac: array(SHARES) of mpvar ! Fraction of capital used per share
11 end-declarations
12
13 RET := [5,17,26,12,8,9,7,6,31,21]
14
15 ! Objective: total return
16 Return := sum(s in SHARES) RET(s)*frac(s)
17
18 ! Limit the percentage of high-risk values

```

foliopl.mos - Sto x

Run Command: foliopl.mos Run... CWD ENV

Compiling foliopl.mos to out\foliopl.bim with -g
Running model
Total return: 14.0666667
1: 30%
2: 0%
3: 20%
4: 0%
5: 6.66666667%
6: 30%
7: 0%
8: 0%
9: 13.33333333%
10: 0%
Process exited with code: 0

Figure 3.7: Display after model execution

The bottom window contains the log of the Mosel execution and if running in debug mode the left window displays all model entities. Choose the icon  window to toggle full-screen display of the *output* printed by our program:

```

Total return: 14.0667
1: 30%
2: 0%
3: 20%
4: 0%
5: 6.66667%
6: 30%
7: 0%
8: 0%
9: 13.3333%
10: 0%

```

This means, that the maximum return of 14.0667 is obtained with a portfolio consisting of shares 1, 3, 5, 6, and 9. 30% of the total amount are spent in shares 1 and 6 each, 20% in 3, 13.3333% in 9 and 6.6667% in 5. It is easily verified that all constraints are indeed satisfied: we have 50% of North-American shares (1 and 3) and 33.33% of high-risk shares (3 and 9).

Now add the line

```
setparam("XPRS_VERBOSE", true)
```

into you model before the call to `maximize` and re-run it. You will now see more *detailed solution information* than what is printed by our model (Figure 3.8). The upper part of the log contains some statistics about the matrix, in its original and in presolved form (presolving a problem means applying some numerical methods to simplify or transform it). The center part tells us which LP algorithm has been used (Simplex), and the number of iterations and total time needed by the algorithm. Since this

problem is very small, it is solved almost instantaneously. After the solver log you see as before the output produced by your model.

```

foliopl.mos
31 | setparam("XPRS_VERBOSE", true) ! Enable solver logging output
32
33 | maximize(Return) ! Solve the problem

foliopl.mos - Sto x
Run Command: foliopl.mos Run... CWD ENV

Compiling foliopl.mos to out\foliopl.bim with -g
Running model

Reading Problem \xprs_2d94d38
Problem Statistics
    3 (    0 spare) rows
   10 (    0 spare) structural columns
   19 (    0 spare) non-zero elements
Global Statistics
    0 entities    0 sets    0 set members
FICO Xpress v8.7.0, Hyper, solve started 16:53:39, Oct 15, 2019
Maximizing LP \xprs_2d94d38
Original problem has:
    3 rows    10 cols    19 elements
Presolved problem has:
    3 rows    10 cols    19 elements
Presolve finished in 0 seconds
Heap usage: 87KB (peak 103KB, 486KB system)

Coefficient range
Coefficients [min,max] : [ 1.00e+00, 1.00e+00] / [ 1.00e+00, 1.00e+00]
RHS and bounds [min,max] : [ 3.00e-01, 1.00e+00] / [ 3.00e-01, 1.00e+00]
Objective [min,max] : [ 5.00e+00, 3.10e+01] / [ 5.00e+00, 3.10e+01]

Its    Obj Value    S    Ninf    Nneg    Sum Dual Inf    Time
0      42.600000    D    2      0      .000000    0
5      14.066667    D    0      0      .000000    0

Uncrunching matrix
Optimal solution found
Dual solved problem
5 simplex iterations in 0s

Final objective          : 1.406666666666667e+01
Max primal violation (abs/rel) : 0.0 / 0.0
Max dual violation (abs/rel) : 0.0 / 0.0
Max complementarity viol. (abs/rel) : 0.0 / 0.0
All values within tolerances
Total return: 14.0666667

```

Figure 3.8: Solver log display

3.4.1 String indices

To make the output of the model more easily understandable, it may be a good idea to replace the numerical indices by *string indices*.

In our model, we replace the three declaration lines

```

SHARES = 1..10
RISK = {2,3,4,9,10}
NA = {1,2,3,4}

```

by the following lines:

```

SHARES = {"treasury", "hardware", "theater", "telecom", "brewery",
          "highways", "cars", "bank", "software", "electronics"}
RISK = {"hardware", "theater", "telecom", "software", "electronics"}
NA = {"treasury", "hardware", "theater", "telecom"}

```

And in the initialization of the array RET we now need to use the indices:

```

RET:([{"treasury", "hardware", "theater", "telecom", "brewery",
       "highways", "cars", "bank", "software", "electronics"}][
5,17,26,12,8,9,7,6,31,21]

```

No other changes in the model are required. We save the modified model as `folioplps.mos`.

The solution output then prints as follows which certainly makes the interpretation of the result easier and more immediate:

```
Total return: 14.0667
treasury: 30%
hardware: 0%
theater: 20%
telecom: 0%
brewery: 6.66667%
highways: 30%
cars: 0%
bank: 0%
software: 13.3333%
electronics: 0%
```

Of course, the entity display also works with these string names:

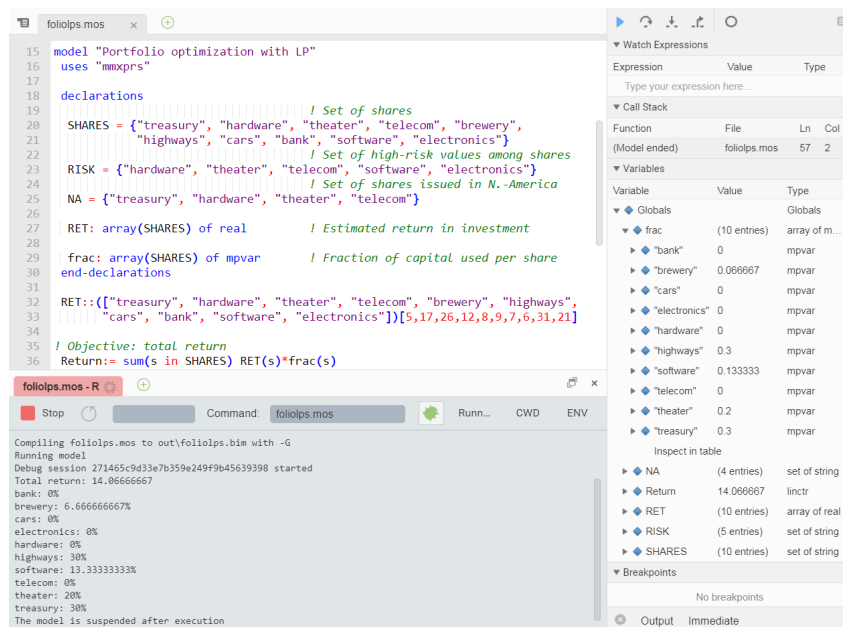


Figure 3.9: Entity display

CHAPTER 4

Working with data

In this chapter we introduce some basic data handling facilities of Mosel:

- the `initializations` block for reading and writing data in Mosel-specific format,
- data output to a file in free format,
- parameterization of files names and numerical constants, and
- some output formatting.

4.1 Data input from file

With Mosel, there are several different ways of reading and writing data from and to external files. For simplicity's sake we shall limit the discussion here to files in text format. Mosel also provides specific modules to exchange data with spreadsheets and databases, for instance using an ODBC connection, but this is beyond the scope of this book and the interested reader is referred to the documentation of these modules (see the *Mosel Language Reference Manual* and the whitepaper *Using ODBC and other database interfaces with Mosel*).

The data file `folio.dat` that we are going to work with has the following contents:

```
! Data file for `folio*.mos`

RET: [("treasury") 5 ("hardware") 17 ("theater") 26 ("telecom") 12
      ("brewery") 8 ("highways") 9 ("cars") 7 ("bank") 6
      ("software") 31 ("electronics") 21 ]

RISK: ["hardware" "theater" "telecom" "software" "electronics"]

NA: ["treasury" "hardware" "theater" "telecom"]
```

Just as in model files, single-line comments preceded by `!` may be used in data files. Every data entry is labeled with the name given to the corresponding entity in the model. Data items may be separated by blanks, tabulations, line breaks, or commas.

We modify the Mosel model from Chapter 3 as follows:

```
declarations
  SHARES: set of string           ! Set of shares
  RISK: set of string             ! Set of high-risk values among shares
  NA: set of string              ! Set of shares issued in N.-America
  RET: array(SHARES) of real     ! Estimated return in investment
end-declarations

initializations from "folio.dat"
```

```

    RISK RET NA
end-initializations

declarations
    frac: array(SHARES) of mpvar      ! Fraction of capital used per share
end-declarations

```

As opposed to the previous model `foliolp.mos`, all index sets and the data array are now declared without fixing their contents: their size is not known at their creation and they are initialized later with data from the file `folio.dat`. Optionally, after the initialization from file, we may *finalize* the sets to make them static. This will make more efficient the handling of any arrays indexed by these sets, and more importantly, this allows Mosel to check for 'out of range' errors that cannot be detected if the sets are allowed to grow dynamically. (Note that sets and arrays in Mosel can also be explicitly marker as *dynamic* in order to prevent them from being finalized/fixed.)

```
finalize(SHARES); finalize(RISK); finalize(NA)
```

Notice that we do not initialize explicitly the set `SHARES`, it is filled automatically when the array `RET` is read. Notice further that we only declare the decision variables *after* initializing the data, and hence when their index set is known.

4.2 Formated data output to file

Just like `initializations` from in the previous section, `initializations to` also exists in Mosel to write out data in a standardized format. However, if we wish to redirect to a file exactly the text that is currently displayed in the logging window of Workbench, then we simply need to surround the printing of this text by calls to the procedures `fopen` and `fclose`:

```

fopen("result.dat", F_OUTPUT)
writeln("Total return: ", getobjval)
forall(s in SHARES) writeln(s, ": ", getsol(frac(s))*100, "%")
fclose(F_OUTPUT)

```

The first argument of `fopen` is the name of the output file, the second indicates in which mode to open it: with the settings shown above, at every re-execution of the model the contents of the result file will be replaced. To append the new output to the existing file contents use:

```
fopen("result.dat", F_OUTPUT+F_APPEND)
```

We may now also wish to format the output more nicely, for instance:

```

forall(s in SHARES)
    writeln(strfmt(s,-12), ": \t", strfmt(getsol(frac(s))*100,5,2), "%")

```

The function `strfmt` indicates the minimum space reserved for printing a string or a number. A negative value for its second argument means left-justified printing. The optional third argument denotes the number of digits after the decimal point. With this formatted way of printing the result file has the following contents:

```

Total return: 14.0667
treasury      : 30.00%
hardware      : 0.00%
theater       : 20.00%
telecom       : 0.00%
brewery       : 6.67%
highways      : 30.00%
cars          : 0.00%
bank          : 0.00%

```

```
software      : 13.33%
electronics  : 0.00%
```

4.3 Parameters

It is commonly considered a good modeling style to hard-code as little information as possible directly in a model. Instead, parameters and data should be specified and read from external sources during the execution of a model to make it more versatile and easily re-usable. With Mosel it is therefore possible to define, for example, file names and numerical constants in the form of *parameters* the values of which may be modified at an execution without changing the model itself.

In our example, we may define the input and output file as parameters and also the constant terms ('right hand side' values) of the constraints and bounds. These parameter definitions must be added to the beginning of the model file, immediately after the `uses` statement:

```
parameters
  DATAFILE= "folio.dat"           ! File with problem data
  OUTFILE= "result.dat"            ! Output file
  MAXRISK = 1/3                    ! Max. investment into high-risk values
  MAXVAL = 0.3                    ! Max. investment per share
  MINAM = 0.5                      ! Min. investment into N.-American values
end-parameters
```

and in the rest of the model the actual file names and data values are replaced by the parameters.

To modify the settings of these parameters when executing a model with Workbench, enter the new values for the parameters after the filename in the *Command* input box of the output pane. For instance to change the value of MINAM:

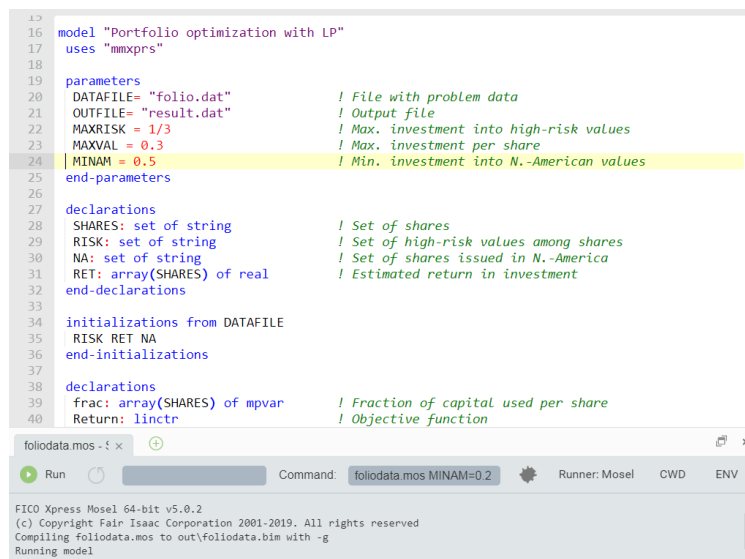


Figure 4.1: Changing model parameter settings

Notice that parameters really become important when the model is not just run in the development environment Workbench but rather used for testing and experimentation (batch mode, scripts using the command line interface) and for final deployment (see Chapter 8). For example, we may wish to write a batch file that runs our model `foliodata.mos` repeatedly with different parameter settings, and writes out the results each time to a different file. To do so, we simply need to add the following lines to a batch file (we then use the standalone version of Mosel to execute the model, which is invoked with

the command `mosel`):

```
mosel exec foliodata MAXRISK=0.1 OUTFILE='result1.dat'
mosel exec foliodata MAXRISK=0.2 OUTFILE='result2.dat'
mosel exec foliodata MAXRISK=0.3 OUTFILE='result3.dat'
mosel exec foliodata MAXRISK=0.4 OUTFILE='result4.dat'
```

Another advantage of the use of parameters is that if models are distributed as *BIM files* (portable, compiled **B**inary **M**odel files), then they remain parameterizable, without having to disclose the model itself and hence protecting your intellectual property.

4.4 Complete example

The complete model file `foliodata.mos` with all the features discussed in this chapter looks as follows:

```
model "Portfolio optimization with LP"
uses "mmxprs"                                ! Use Xpress Optimizer

parameters
  DATAFILE= "folio.dat"                      ! File with problem data
  OUTFILE= "result.dat"                      ! Output file
  MAXRISK = 1/3                               ! Max. investment into high-risk values
  MAXVAL = 0.3                               ! Max. investment per share
  MINAM = 0.5                                ! Min. investment into N.-American values
end-parameters

declarations
  SHARES: set of string                      ! Set of shares
  RISK: set of string                        ! Set of high-risk values among shares
  NA: set of string                          ! Set of shares issued in N.-America
  RET: array(SHARES) of real                ! Estimated return in investment
end-declarations

initializations from DATAFILE
  RISK RET NA
end-initializations

declarations
  frac: array(SHARES) of mpvar              ! Fraction of capital used per share
end-declarations

! Objective: total return
Return:= sum(s in SHARES) RET(s)*frac(s)

! Limit the percentage of high-risk values
sum(s in RISK) frac(s) <= MAXRISK

! Minimum amount of North-American values
sum(s in NA) frac(s) >= MINAM

! Spend all the capital
sum(s in SHARES) frac(s) = 1

! Upper bounds on the investment per share
forall(s in SHARES) frac(s) <= MAXVAL

! Solve the problem
maximize(Return)

! Solution printing to a file
fopen(OUTFILE, F_OUTPUT)
writeln("Total return: ", getobjval)
forall(s in SHARES)
```

```
      writeln(strfmt(s,-12), ": \t", strfmt(getsol(frac(s))*100,2,3), "%")  
    fclose(F_OUTPUT)  
  
end-model
```

CHAPTER 5

Drawing user graphs

In this chapter we show how to draw a user-defined SVG graph. The graph we wish to display is generated as a result of repeated executions of a model with different parameter settings. So we shall first see an example of writing a simple algorithm in the Mosel language involving

- re-definition of constraints,
- repeated re-optimization,
- saving solution information,
- definition of a user graph: drawing points, lines, and texts, and
- simple programming tasks (loops and selections).

5.1 Extended problem description

In addition to the data considered so far (see table in Chapter 2), the investor now also has at hand the estimations of the deviations from the expected return per share (Table 5.1). With this additional information, he decides to run the LP model with different limits on the portion of high-risk shares and to represent the results as a graph, plotting the resulting total return against the deviation as a measure of risk.

Table 5.1: Estimated deviations

Number	Description	Deviation
1	treasury	0.1
2	hardware	19
3	theater	28
4	telecom	22
5	brewery	4
6	highways	3.5
7	cars	5
8	bank	0.5
9	software	25
10	electronics	16

5.2 Looping over optimization

We are going to modify the model `foliodata.mos` from the previous chapter in such a way that the problem is re-optimized repeatedly with different limits on the percentage of high-risk values.

In detail, the model will be transformed to implement the following algorithm:

1. Definition of the part of the model that remains unchanged by the parameter changes.
2. For every parameter value:
 - Re-define the constraint limiting the percentage of high-risk values.
 - Solve the resulting problem.
 - If the problem is feasible: store the solution values.
3. Draw the result graph.

To store the solution value and the total estimated deviation of the result after each optimization run, we declare the following two arrays:

```

declarations
  SOLRET: array(range) of real      ! Solution values (total return)
  SOLDEV: array(range) of real      ! Solution values (average deviation)
end-declarations

```

The following code fragment introduces a loop around the definition of the constraint limiting the portion of high-risk shares and the solution procedure. To be able to override its previous definition at every iteration, we now give this constraint a name, *Risk*. If the constraint did not have a name, then each time the loop was executed, a new constraint would be added, and the existing constraint would not be replaced.

```

ct:=0
forall(r in 0..20) do
  ! Limit the percentage of high-risk values
  Risk:= sum(s in RISK) frac(s) <= r/20

  maximize(Return)                ! Solve the problem
  if (getprobat = XPRS_OPT) then    ! Save the optimal solution value
    ct+=1
    SOLRET(ct) := getobjval
    SOLDEV(ct) := getsol(sum(s in SHARES) DEV(s)*frac(s))
  else
    writeln("No solution for high-risk values <= ", 100*r/20, "%")
  end-if
end-do

```

Above we have used the second form of the *forall* loop, namely *forall/do*. This form must be used when several statements are included in the loop. The loop is terminated by *end-do*.

Another new feature in this code extract is the *if/then/else/end-if* statement. We only want to save the values for a problem instance if the optimal solution has been found—the solution status is obtained with function *getprobat* and tested whether it is ‘solved to optimality’, represented by the constant *XPRS_OPT*.

The selection statement has two other forms, *if/then/end-if* and *if/then/elif/then/else/end-if* where *elif/then* may be repeated several times.

For further examples and a complete description of all loops and selection statements available in Mosel the reader is referred to the ‘Mosel User Guide’.

5.3 Drawing a user graph

We now have gathered all the data required to draw the graph. Graphing functions are provided by the module *mmsvg* (documented in the ‘Mosel Language Reference Manual’), so it needs to be loaded at the beginning of the model by adding the following line:

```
uses "mmsvg"
```

Then the following lines draw the graph (note the use of the `sum` operator to create a list of points):

```
svgaddgroup("GrS", "Solution values", SVG_GREY)
forall(r in 1..ct) svgaddpoint("GrS", SOLRET(r), SOLDEV(r))
svgaddline("GrS", sum(r in 1..ct) [SOLRET(r), SOLDEV(r)])
```

The user graph will be displayed in the editor window of the Workbench workspace. Select the tab *SVG drawing* to move it to the foreground. With the above we obtain the following output (due to the interplay of the various constraints the resulting graph is not a straight line as one might have expected at first thought):

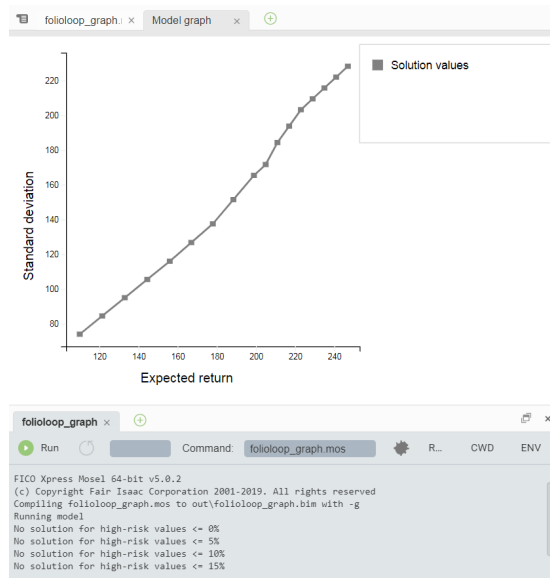


Figure 5.1: Plot of the result graph

In addition to this graph, we may also display labeled points representing the input data ('GrL' for low risk shares and 'GrH' for high risk shares):

```
svgaddgroup("GrL", "Low risk", SVG_GREEN)
svgaddgroup("GrH", "High risk", SVG_RED)

forall(s in SHARES - RISK) do
  svgaddpoint("GrL", RET(s), DEV(s))
  svgaddtext("GrL", RET(s)+1, 1.3*(DEV(s)-1), s)
end-do

forall(s in RISK) do
  svgaddpoint("GrH", RET(s), DEV(s))
  svgaddtext("GrH", RET(s)-2.5, DEV(s)-1, s)
end-do
```

Notice the set notation: `SHARES - RISK` means 'all elements of `SHARES` that are not contained in `RISK`'.

The complete output now is:

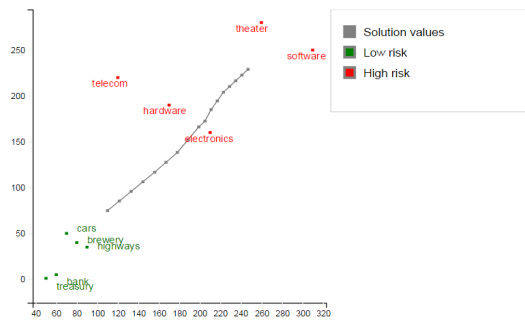


Figure 5.2: Plot of result graph and data

5.4 Complete example

The complete model file `folioloop_graph.mos` with all the features discussed in this chapter looks as follows. Notice that the two modules `mmxprs` and `mmive` may be loaded with a single `uses` statement. The deviation data may either be added to the original data file or, as shown here, read from a second file.

```
model "Portfolio optimization with LP"
  uses "mmxprs", "mmsvg"           ! Use Xpress Optimizer with SVG graphing

  parameters
    DATAFILE= "folio.dat"          ! File with problem data
    DEVDATA= "foliodev.dat"         ! File with deviation data
    MAXVAL = 0.3                    ! Max. investment per share
    MINAM = 0.5                     ! Min. investment into N.-American values
  end-parameters

  declarations
    SHARES: set of string           ! Set of shares
    RISK: set of string              ! Set of high-risk values among shares
    NA: set of string               ! Set of shares issued in N.-America
    RET: array(SHARES) of real      ! Estimated return in investment
    DEV: array(SHARES) of real      ! Standard deviation
    SOLRET: array(range) of real    ! Solution values (total return)
    SOLDEV: array(range) of real    ! Solution values (average deviation)
  end-declarations

  initializations from DATAFILE
    RISK RET NA
  end-initializations

  initializations from DEVDATA
    DEV
  end-initializations

  declarations
    frac: array(SHARES) of mpvar    ! Fraction of capital used per share
    Return, Risk: lincstr           ! Constraint declaration (optional)
  end-declarations

  ! Objective: total return
  Return:= sum(s in SHARES) RET(s)*frac(s)

  ! Minimum amount of North-American values
  sum(s in NA) frac(s) >= MINAM

  ! Spend all the capital
  sum(s in SHARES) frac(s) = 1
```

```

! Upper bounds on the investment per share
forall(s in SHARES) frac(s) <= MAXVAL

! Solve the problem for different limits on high-risk shares
ct:=0
forall(r in 0..20) do
  ! Limit the percentage of high-risk values
  Risk:= sum(s in RISK) frac(s) <= r/20

  maximize(Return)          ! Solve the problem

  if (getprobat = XPRS_OPT) then ! Save the optimal solution value
    ct+=1
    SOLRET(ct):= getobjval
    SOLDEV(ct):= getsol(sum(s in SHARES) DEV(s)*frac(s))
  else
    writeln("No solution for high-risk values <= ", 100*r/20, "%")
  end-if
end-do

! Drawing a graph to represent results (`GrS`) and data (`GrL` & `GrH`)
svgaddgroup("GrS", "Solution values", SVG_GREY)
svgaddgroup("GrL", "Low risk", SVG_GREEN)
svgaddgroup("GrH", "High risk", SVG_RED)

forall(r in 1..ct) svgaddpoint("GrS", SOLRET(r), SOLDEV(r))
svgaddline("GrS", sum(r in 1..ct) [SOLRET(r), SOLDEV(r)])

forall(s in SHARES - RISK) do
  svgaddpoint("GrL", RET(s), DEV(s))
  svgaddtext("GrL", RET(s)+1, 1.3*(DEV(s)-1), s)
end-do

forall(s in RISK) do
  svgaddpoint("GrH", RET(s), DEV(s))
  svgaddtext("GrH", RET(s)-2.5, DEV(s)-1, s)
end-do

! Scale the size of the displayed graph
svgsetgraphscale(10)
svgsetgraphpointsize(2)

! Optionally save graphic to file
svgsave("foliograph.svg")

! Display the graph and wait for window to be closed by the user
svgrefresh
svgwaitclose

end-model

```

The problem is not feasible for small limit values on the constraint `Risk`. Besides the graphs we therefore obtain the following text output:

```

No solution for high-risk values <= 0%
No solution for high-risk values <= 5%
No solution for high-risk values <= 10%
No solution for high-risk values <= 15%

```

CHAPTER 6

Mixed Integer Programming

This chapter extends the model developed in Chapter 3 to a Mixed Integer Programming (MIP) problem. It describes how to

- define different types of discrete variables,
- understand and exploit the MIP optimization displays.

Chapter 11 shows how to formulate and solve the same example with BCL and in Chapter 16 the problem is input and solved directly with Xpress Optimizer.

6.1 Extended problem description

The investor is unwilling to have small share holdings. He looks at the following two possibilities to formulate this constraint:

1. Limiting the number of different shares taken into the portfolio.
2. If a share is bought, at least a certain minimum amount $MINVAL = 10\%$ of the budget is spent on the share.

We are going to deal with these two constraints in two separate models.

6.2 MIP model 1: limiting the number of different shares

To be able to count the number of different values we are investing in, we introduce a second set of variables buy_s in the LP model developed in Chapter 2. These variables are *indicator variables* or *binary variables*. A variable buy_s takes the value 1 if the share s is taken into the portfolio and 0 otherwise.

We introduce the following constraint to limit the total number of assets to a maximum of $MAXNUM$. It expresses the constraint that at most $MAXNUM$ of the variables buy_s may take the value 1 at the same time.

$$\sum_{s \in SHARES} buy_s \leq MAXNUM$$

We now still need to link the new binary variables buy_s with the variables $frac_s$, the quantity of every share selected into the portfolio. The relation that we wish to express is 'if a share is selected into the portfolio, then it is counted in the total number of values' or 'if $frac_s > 0$ then $buy_s = 1$ '. The following inequality formulates this implication:

$$\forall s \in SHARES : frac_s \leq buy_s$$

If, for some s , $frac_s$ is non-zero, then buy_s must be greater than 0 and hence 1. Conversely, if buy_s is at 0, then $frac_s$ is also 0, meaning that no fraction of share s is taken into the portfolio. Notice that these constraints do not prevent the possibility that buy_s is at 1 and $frac_s$ at 0. However, this does not matter in our case, since any solution in which this is the case is also valid with both variables, buy_s and $frac_s$, at 0.

6.2.1 Implementation with Mosel

We extend the LP model developed in Chapter 3 (using the initialization of data from file introduced in Chapter 4) with the new variables and constraints. The fact that the new variables are *binary variables* (i.e. they only take the values 0 and 1) is expressed through the `is_binary` constraint.

Another common type of discrete variable is an *integer variable*, that is, a variable that can only take on integer values between given lower and upper bounds. This variable type is defined in Mosel with an `is_integer` constraint. In the following section (MIP model 2) we shall see yet another example of discrete variables, namely semi-continuous variables.

```

model "Portfolio optimization with MIP"
  uses "mmxprs"                                ! Use Xpress Optimizer

  parameters
    MAXRISK = 1/3                                ! Max. investment into high-risk values
    MAXVAL = 0.3                                ! Max. investment per share
    MINAM = 0.5                                ! Min. investment into N.-American values
    MAXNUM = 4                                  ! Max. number of different assets
  end-parameters

  declarations
    SHARES: set of string                      ! Set of shares
    RISK: set of string                        ! Set of high-risk values among shares
    NA: set of string                         ! Set of shares issued in N.-America
    RET: array(SHARES) of real                ! Estimated return in investment
  end-declarations

  initializations from "folio.dat"
    RISK RET NA
  end-initializations

  declarations
    frac: array(SHARES) of mpvar              ! Fraction of capital used per share
    buy: array(SHARES) of mpvar               ! 1 if asset is in portfolio, 0 otherwise
  end-declarations

  ! Objective: total return
  Return:= sum(s in SHARES) RET(s)*frac(s)

  ! Limit the percentage of high-risk values
  sum(s in RISK) frac(s) <= MAXRISK

  ! Minimum amount of North-American values
  sum(s in NA) frac(s) >= MINAM

  ! Spend all the capital
  sum(s in SHARES) frac(s) = 1

  ! Upper bounds on the investment per share
  forall(s in SHARES) frac(s) <= MAXVAL

  ! Limit the total number of assets
  sum(s in SHARES) buy(s) <= MAXNUM

  forall(s in SHARES) do
    buy(s) is_binary                          ! Turn variables into binaries
    frac(s) <= buy(s)                        ! Linking the variables
  end-do

```

```

! Solve the problem
maximize(Return)

! Solution printing
writeln("Total return: ", getobjval)
forall(s in SHARES)
  writeln(s, ": ", getsol(frac(s))*100, "% (", getsol(buy(s)), ")")

end-model

```

In the model `foliomip1.mos` above we have used the second form of the *forall* loop, namely *forall/do*, that needs to be used if the loop encompasses several statements. Equivalently we could have written

```

forall(s in SHARES) buy(s) is_binary
forall(s in SHARES) frac(s) <= buy(s)

```

6.2.2 Analyzing the solution

As the result of our model execution we obtain the following output:

```

Total return: 13.1
treasury: 20% (1)
hardware: 0% (0)
theater: 30% (1)
telecom: 0% (0)
brewery: 20% (1)
highways: 30% (1)
cars: 0% (0)
bank: 0% (0)
software: 0% (0)
electronics: 0% (0)

```

The maximum return is now lower than in the original LP problem due to the additional constraint. As required, only four different shares are selected to form the portfolio.

Let us now have a look at the detailed solver information:

Enable the Optimizer logging output by adding the line

```
setparam("XPRS_VERBOSE", true)
```

into the model before the call to `maximize` and re-run the model. There are now more rows (constraints) and columns (variables) than in the LP matrix of the previous chapters.

```

Run Command: foliomip1.mos CWD ENV

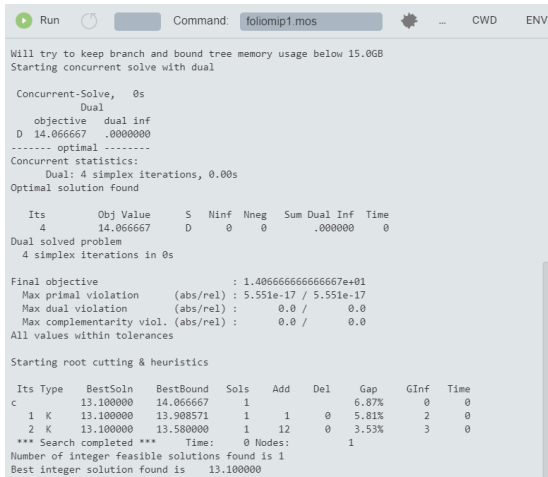
Compiling foliomip1.mos to out\foliomip1.bim with -g
Running model

Reading Problem \xprs_2645da8
Problem Statistics
  14 ( 0 spare) rows
  20 ( 0 spare) structural columns
  49 ( 0 spare) non-zero elements
Global Statistics
  10 entities 0 sets 0 set members
FICO Xpress v8.7.0, Hyper, solve started 18:29:46, Oct 15, 2019
Maximizing MILP \xprs_2645da8
Original problem has:
  14 rows 20 cols 49 elements 10 globals
Presolved problem has:
  13 rows 19 cols 46 elements 9 globals
LP relaxation tightened
Presolve finished in 0 seconds
Heap usage: 119KB (peak 136KB, 486KB system)

Coefficient range original solved
Coefficients [min,max] : [ 1.00e+00, 1.00e+00] / [ 1.33e-01, 1.00e+00]
RHS and bounds [min,max] : [ 3.00e-01, 4.00e+00] / [ 1.33e-01, 3.00e+00]
Objective [min,max] : [ 5.00e+00, 3.10e+01] / [ 5.00e+00, 3.10e+01]

```

Figure 6.1: Solver log for MIP problem - part 1: statistics



```

Run Command: foliomp1.mos CWD ENV

Will try to keep branch and bound tree memory usage below 15.0GB
Starting concurrent solve with dual

Concurrent-Solve, 0s
Dual
  objective dual inf
  D 14.066667 .0000000
  ..... optimal .....
Concurrent statistics:
  Dual: 4 simplex iterations, 0.00s
Optimal solution found

Its Obj Value S Ninf Nneg Sum Dual Inf Time
4 14.066667 0 0 0 .000000 0
Dual solved problem
  4 simplex iterations in 0s

Final objective : 1.406666666666667e+01
Max primal violation (abs/rel) : 5.551e-17 / 5.551e-17
Max dual violation (abs/rel) : 0.0 / 0.0
Max complementarity viol. (abs/rel) : 0.0 / 0.0
All values within tolerances

Starting root cutting & heuristics

Its Type BestSoln BestBound Sols Add Del Gap GInf Time
c 1 13.100000 14.066667 1 1 0 6.87% 0 0
1 K 13.100000 13.908571 1 1 0 5.81% 2 0
2 K 13.100000 13.580000 1 12 0 3.53% 3 0
*** Search completed *** Time: 0 Nodes: 1
Number of integer feasible solutions found is 1
Best integer solution found is 13.100000

```

Figure 6.2: Solver log for MIP problem - part 2: algorithm

As we have seen, it is relatively easy to turn an LP model into a MIP model by adding an integrality condition on some (or all) variables. However, the same does not hold for the solution algorithms: MIP problems are solved by repeatedly solving LP problems. Initially, the problem is solved without any integrality constraints (the *LP relaxation*). Then, one at a time, a discrete variable is chosen that does not satisfy the integrality condition in the current solution and new upper or lower bounds are added for this variable to bring it to an integer value. If we represent every LP solution as a node and connect these nodes by the bound changes or added constraints, then we obtain a tree-like structure, the *Branch-and-Bound tree*.

In particular, the branching information tells us how many Branch-and-Bound nodes have been needed to solve the problem: here it is just one, the enumeration did not even start. By default, Xpress Optimizer enables certain MIP pre-treatment algorithms (see the 'Optimizer Reference Manual' for further detail on algorithmic settings), among others the automated generation of cuts—i.e., additional constraints that cut off parts of the LP solution space, but no solution of the MIP. This problem is of very small size and becomes so easy through the pre-treatment that it is solved immediately.

Add the lines

```

setparam("XPRS_CUTSTRATEGY", 0)
setparam("XPRS_HEURSTRATEGY", 0)
setparam("XPRS_PRESOLVE", 0)

```

to your model before the call to `maximize` and re-execute it. You have now switched off the MIP pre-treatment routines for automated cut generation and MIP heuristics, and also the presolve mechanism (a treatment to the matrix that tries to reduce its size and improve its numerical properties).

It now takes several nodes to solve the problem:

```

Run Command: foliomip1.mos CWD ENV

Its Obj Value S Ninf Nneg Sum Dual Inf Time
5 14.066667 0 0 0 .000000 0

Dual solved problem
5 simplex iterations in 0s

Final objective : 1.406666666666667e+01
Max primal violation (abs/rel) : 0.0 / 0.0
Max dual violation (abs/rel) : 0.0 / 0.0
Max complementarity viol. (abs/rel) : 0.0 / 0.0
All values within tolerances

Starting root cutting & heuristics

Its Type BestSoln BestBound Sols Add Del Gap GInf Time

Starting tree search.
Deterministic mode with up to 8 running threads and up to 16 tasks.
Heap usage: 2885KB (peak 5091KB, 603KB system)

Node BestSoln BestBound Sols Active Depth Gap GInf Time
1 14.066667 0 2 1 4 0
2 14.066667 0 2 3 2 0
3 14.066667 0 2 3 3 0
4 14.066667 0 2 4 2 0
6 14.066667 0 2 4 2 0
8 13.700000 0 3 5 2 0
9 13.700000 0 5 4 2 0
* 10 13.100000 13.700000 1 5 5 4.38% 0 0
12 13.100000 13.700000 1 5 5 4.38% 1 0

STOPPING - MIPRELSTOP target reached (MIPRELSTOP=0.0001 gap=1.07379e-006).
*** Search completed *** Time: 0 Nodes: 20
Number of integer feasible solutions found is 1
Best integer solution found is 13.100000

```

Figure 6.3: Solver log for MIP problem with Branch-and-Bound

6.3 MIP model 2: imposing a minimum investment in each share

To formulate the second MIP model, we start again with the LP model from Chapters 2 and 3. The new constraint we wish to formulate is ‘if a share is bought, at least a certain minimum amount $MINVAL = 10\%$ of the budget is spent on the share.’ Instead of simply constraining every variable $frac_s$ to take a value between 0 and $MAXVAL$, it now must either lie in the interval between $MINVAL$ and $MAXVAL$ or take the value 0. This type of variable is known as *semi-continuous variable*. In the new model, we replace the bounds on the variables $frac_s$ by the following constraint:

$$\forall s \in SHARES : frac_s = 0 \text{ or } MINVAL \leq frac_s \leq MAXVAL$$

6.3.1 Implementation with Mosel

The following model `foliomip2.mos` implements the MIP model 2, again starting with the LP model from Chapter 3 augmented by the data initialization from file explained in Chapter 4. The semi-continuous variables are defined with the `is_semcont` constraint.

A similar type is available for integer variables that take either the value 0 or an integer value between a given limit and their upper bound (so-called *semi-continuous integers*): `is_semint`. A third composite type is a *partial integer* which takes integer values from its lower bound to a given limit value and is continuous beyond this value (marked by `is_partint`).

```

model "Portfolio optimization with MIP"
uses "mmxprs" ! Use Xpress Optimizer

parameters
  MAXRISK = 1/3 ! Max. investment into high-risk values
  MINAM = 0.5 ! Min. investment into N.-American values
  MAXVAL = 0.3 ! Max. investment per share
  MINVAL = 0.1 ! Min. investment per share
end-parameters

declarations
  SHARES: set of string ! Set of shares
  RISK: set of string ! Set of high-risk values among shares
  NA: set of string ! Set of shares issued in N.-America
  RET: array(SHARES) of real ! Estimated return in investment
end-declarations

```

```

initializations from "folio.dat"
  RISK RET NA
end-initializations

declarations
  frac: array(SHARES) of mpvar      ! Fraction of capital used per share
end-declarations

! Objective: total return
Return:= sum(s in SHARES) RET(s)*frac(s)

! Limit the percentage of high-risk values
sum(s in RISK) frac(s) <= MAXRISK

! Minimum amount of North-American values
sum(s in NA) frac(s) >= MINAM

! Spend all the capital
sum(s in SHARES) frac(s) = 1

! Upper and lower bounds on the investment per share
forall(s in SHARES) do
  frac(s) <= MAXVAL
  frac(s) is_semcont MINVAL
end-do

! Solve the problem
maximize(Return)

! Solution printing
writeln("Total return: ", getobjval)
forall(s in SHARES) writeln(s, ": ", getsol(frac(s))*100, "%")

end-model

```

When executing this model of the solution information window) we obtain the following output:

```

Total return: 14.0333
treasury: 30%
hardware: 0%
theater: 20%
telecom: 0%
brewery: 10%
highways: 26.6667%
cars: 0%
bank: 0%
software: 13.3333%
electronics: 0%

```

Now five securities are chosen for the portfolio, each forming at least 10% and at most 30% of the total investment. Due to the additional constraint, the optimal MIP solution value is again lower than the initial LP solution value.

CHAPTER 7

Quadratic Programming

In this chapter we turn the LP problem from Chapter 3 into a Quadratic Programming (QP) problem, and the first MIP model from Chapter 6 into a Mixed Integer Quadratic Programming (MIQP) problem. The chapter shows how to

- define quadratic objective functions,
- incrementally define and solve problems,
- understand and exploit the MIP optimization displays.

Chapter 12 shows how to formulate and solve the same examples with BCL and in Chapter 17 the QP problem is input and solved directly with Xpress Optimizer.

7.1 Problem description

The investor may also look at his portfolio selection problem from a different angle: instead of maximizing the estimated return and limiting the portion of high-risk investments he now wishes to minimize the risk whilst obtaining a certain target yield. He adopts the Markowitz idea of getting estimates of the variance/covariance matrix of estimated returns on the securities. (For example, hardware and software company worths tend to move together, but are oppositely correlated with the success of theatrical production, as people go to the theater more when they have become bored with playing with their new computers and computer games.) The return on theatrical productions are highly variable, whereas the treasury bill yield is certain. The estimated returns and the variance/covariance matrix are given in the following table:

Table 7.1: Variance/covariance matrix

	treasury	hardw.	theater	telecom	brewery	highways	cars	bank	softw.	electr.
treasury	0.1	0	0	0	0	0	0	0	0	0
hardware	0	19	-2	4	1	1	1	0.5	10	5
theater	0	-2	28	1	2	1	1	0	-2	-1
telecom	0	4	1	22	0	1	2	0	3	4
brewery	0	1	2	0	4	-1.5	-2	-1	1	1
highways	0	1	1	1	-1.5	3.5	2	0.5	1	1.5
cars	0	1	1	2	-2	2	5	0.5	1	2.5
bank	0	0.5	0	0	-1	0.5	0.5	1	0.5	0.5
software	0	10	-2	3	1	1	1	0.5	25	8
electronics	0	5	-1	4	1	1.5	2.5	0.5	8	16

Question 1: Which investment strategy should the investor adopt to minimize the variance subject to getting some specified minimum target yield?

Question 2: Which is the least variance investment strategy if the investor wants to choose at most four different securities (again subject to getting some specified minimum target yield)?

The first question leads us to a *Quadratic Programming* problem, that is, a Mathematical Programming problem with a quadratic objective function and linear constraints. The second question necessitates the introduction of discrete variables to count the number of securities, and so we obtain a *Mixed Integer Quadratic Programming* problem. The two cases will be discussed separately in the following two sections.

7.2 QP

To adapt the model developed in Chapter 2 to the new way of looking at the problem, we need to make the following changes:

- New objective function: mean variance instead of total return.
- The risk-related constraint disappears.
- Addition of a new constraint: target yield.

The new objective function is the mean variance of the portfolio, namely:

$$\sum_{s,t \in \text{SHARES}} \text{VAR}_{st} \cdot \text{frac}_s \cdot \text{frac}_t$$

where VAR_{st} is the variance/covariance matrix of all shares. This is a *quadratic objective function* (an objective function becomes quadratic either when a variable is squared, e.g., frac_1^2 , or when two variables are multiplied together, e.g., $\text{frac}_1 \cdot \text{frac}_2$).

The target yield constraint can be written as follows:

$$\sum_{s \in \text{SHARES}} \text{RET}_s \cdot \text{frac}_s \geq \text{TARGET}$$

The limit on the North-American shares as well as the requirement to spend all the money, and the upper bounds on the fraction invested into every share are retained. We therefore obtain the following complete mathematical model formulation:

$$\begin{aligned} & \text{minimize} && \sum_{s,t \in \text{SHARES}} \text{VAR}_{st} \cdot \text{frac}_s \cdot \text{frac}_t \\ & && \sum_{s \in \text{NA}} \text{frac}_s \geq \text{MINAM} \\ & && \sum_{s \in \text{SHARES}} \text{frac}_s = 1 \\ & && \sum_{s \in \text{SHARES}} \text{RET}_s \cdot \text{frac}_s \geq \text{TARGET} \\ & && \forall s \in \text{SHARES} : 0 \leq \text{frac}_s \leq \text{MAXVAL} \end{aligned}$$

7.2.1 Implementation with Mosel

In addition to the Xpress Optimizer module *mmxprs* we now also need to load the module *mmnl* that adds to the Mosel language the facilities required for the definition of quadratic expressions (*mmnl* is

documented in the 'Mosel Language Reference Manual'). We can then use the optimization function `maximize` (or alternatively `minimize`) for quadratic objective functions to start the solution process.

This model uses a different data file (`folioqp.dat`) than the previous models:

```

      ! trs  haw  thr  tel  brw  hgw  car  bnk  sof  elc
RET: [ (1)  5   17   26   12    8    9    7    6   31   21]

VAR: [ (1 1) 0.1    0    0    0    0    0    0    0    0    0 ! treasury
      (2 1)  0   19   -2    4    1    1    1  0.5   10    5 ! hardware
      (3 1)  0   -2   28    1    2    1    1    0   -2   -1 ! theater
      (4 1)  0    4    1   22    0    1    2    0    3    4 ! telecom
      (5 1)  0    1    2    0    4  -1.5   -2   -1    1    1 ! brewery
      (6 1)  0    1    1    1  -1.5   3.5    2  0.5    1  1.5 ! highways
      (7 1)  0    1    1    2    -2    2    5  0.5    1  2.5 ! cars
      (8 1)  0  0.5    0    0   -1  0.5  0.5    1  0.5  0.5 ! bank
      (9 1)  0   10   -2    3    1    1    1  0.5   25    8 ! software
      (10 1) 0    5   -1    4    1  1.5  2.5  0.5    8   16 ! electronics
      ]

RISK: [2 3 4 9 10]
NA: [1 2 3 4]
```

Note that we have chosen to use numerical instead of string indices. Since the set `SHARES` is defined in the model, we do not have to list the index-tuple for every data entry in the file—those tuples given are for clarity's sake only.

```

model "Portfolio optimization with QP/MIQP"
uses "mmxprs", "mmnl"           ! Use Xpress Optimizer with QP solver

parameters
  MAXVAL = 0.3                   ! Max. investment per share
  MINAM = 0.5                    ! Min. investment into N.-American values
  MAXNUM = 4                    ! Max. number of different assets
  TARGET = 9.0                  ! Minimum target yield
end-parameters

declarations
  SHARES = 1..10                ! Set of shares
  RISK: set of integer           ! Set of high-risk values among shares
  NA: set of integer             ! Set of shares issued in N.-America
  RET: array(SHARES) of real    ! Estimated return in investment
  VAR: array(SHARES,SHARES) of real ! Variance/covariance matrix of
                                   ! estimated returns
end-declarations

initializations from "folioqp.dat"
  RISK RET NA VAR
end-initializations

declarations
  frac: array(SHARES) of mpvar  ! Fraction of capital used per share
end-declarations

! Objective: mean variance
Variance:= sum(s,t in SHARES) VAR(s,t)*frac(s)*frac(t)

! Minimum amount of North-American values
sum(s in NA) frac(s) >= MINAM

! Spend all the capital
sum(s in SHARES) frac(s) = 1

! Target yield
sum(s in SHARES) RET(s)*frac(s) >= TARGET

! Upper bounds on the investment per share
```

```

forall(s in SHARES) frac(s) <= MAXVAL

! Solve the problem
minimize(Variance)

! Solution printing
writeln("With a target of ", TARGET, " minimum variance is ", getobjval)
forall(s in SHARES) writeln(s, ": ", getsol(frac(s))*100, "%")

end-model

```

This model (file `folioqp.mos`) produces the following solution output (tab *Output/input* of the solution information window):

```

With a target of 9 minimum variance is 0.557393
1: 30%
2: 7.15391%
3: 7.38246%
4: 5.46363%
5: 12.6554%
6: 5.91228%
7: 0.332458%
8: 30%
9: 1.09983%
10: 0%

```

Similarly to the algorithm shown in Chapter 5, we may re-solve this problem with different values of `TARGET` and plot the results in a target return/standard deviation graph, know as the 'efficient frontier' (model file `folioqp_graph.mos`):

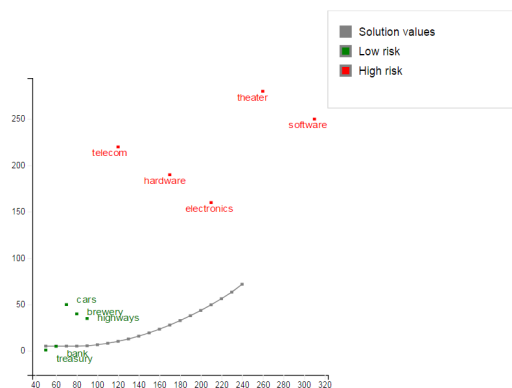


Figure 7.1: Graph of the efficient frontier

7.3 MIQP

We now wish to express the fact that at most a given number *MAXNUM* of different assets may be selected into the portfolio, subject to all other constraints of the previous QP model. In Chapter 6 we have already seen how this can be done, namely by introducing an additional set of binary decision variables buy_s that are linked logically to the continuous variables:

$$\forall s \in SHARES : frac_s \leq buy_s$$

Through this relation, a variable buy_s will be at 1 if a fraction $frac_s$ greater than 0 is selected into the portfolio. If, however, buy_s equals 0, then $frac_s$ must also be 0.

To limit the number of different shares in the portfolio, we then define the following constraint:

$$\sum_{s \in \text{SHARES}} \text{buy}_s \leq \text{MAXNUM}$$

7.3.1 Implementation with Mosel

We may modify the previous QP model or simply add the following lines to the end of the QP model in the previous section: the problem is then solved once as a QP and once as a MIQP in a single model run.

```

declarations
  buy: array(SHARES) of mpvar      ! 1 if asset is in portfolio, 0 otherwise
end-declarations

! Limit the total number of assets
sum(s in SHARES) buy(s) <= MAXNUM

forall(s in SHARES) do
  buy(s) is_binary
  frac(s) <= buy(s)
end-do

! Solve the problem
minimize(Variance)

writeln("With a target of ", TARGET," and at most ", MAXNUM,
        " assets, minimum variance is ", getobjval)
forall(s in SHARES) writeln(s, ": ", getsol(frac(s))*100, "%")

```

When executing the MIQP model, we obtain the following solution output:

```

With a target of 9 and at most 4 assets,
  minimum variance is 1.24876
1: 30%
2: 20%
3: 0%
4: 0%
5: 23.8095%
6: 26.1905%
7: 0%
8: 0%
9: 0%
10: 0%

```

With the additional constraint on the number of different assets the minimum variance is more than twice as large as in the QP problem.

7.3.2 Analyzing the solution

If we enable the Optimizer logging output display by setting `XPRS_VERBOSE` to 'true' we see the following information:

```

Run Command: foliomqc.mos CWD ENV

Compiling foliomqc.mos to out\foliomqc.bim with -g
Running model

Reading Problem \xprs_2e2c2c8
Problem Statistics
  13 ( 0 spare) rows
  20 ( 0 spare) structural columns
  44 ( 0 spare) non-zero elements

Global Statistics
  10 entities      0 sets      0 set members
FICO Xpress v8.7.0, Hyper, solve started 18:45:54, Oct 15, 2019
Maximizing MIQCP \xprs_2e2c2c8
Original problem has:
  14 rows      20 cols      44 elements      10 globals
  1 group      43 growelem
Converted 1 quadratic matrices to their separable equivalent
Converted 1 separable quadratic matrices to rotated cones
Presolved problem has:
  35 rows      43 cols      103 elements      10 globals
  11 cones      22 celems
LP relaxation tightened
Presolve finished in 0 seconds
Heap usage: 433KB (peak 461KB, 510KB system)

Coefficient range      original      solved
Coefficients [min,max] : [ 1.00e+00, 1.00e+00] / [ 2.34e-02, 1.80e+00]
RHS and bounds [min,max] : [ 3.00e-01, 4.00e+00] / [ 2.82e-02, 4.00e+00]
Objective [min,max] : [ 5.00e+00, 3.10e+01] / [ 5.00e+00, 3.10e+01]
Quadratic [min,max] : [ 2.00e-01, 5.60e+01] / [ 2.00e+00, 2.00e+00]

Will try to keep branch and bound tree memory usage below 15.0GB
Barrier cache sizes : L1=32K L2=8192K
Using AVX support
Cores per CPU (CORESPERCPU): 8
Barrier starts after 0 seconds, using up to 8 threads, 4 cores
Matrix ordering - Dense cols.: 30 NZ(L): 580 Flops: 5016

Its P.inf D.inf U.inf Primal obj. Dual obj. Compl.
0 1.05e+001 3.10e+001 1.91e+000 1.4200000e+002 3.9056252e+001 1.2e+002
1 4.36e+000 1.29e+001 7.92e-001 7.1010230e+001 2.8228674e+001 7.0e+001
2 1.09e+000 3.22e+000 1.98e-001 3.0367038e+001 1.9649337e+001 1.9e+001

```

Figure 7.2: Detailed MIQP solution information

This is quite similar to the MIP statistics, perhaps with the exception of the LP solution algorithm: the initial LP relaxation has been solved by the Newton-Barrier algorithm.

Just as with linear problems, the root solving as continuous problem is followed by a root cutting and heuristics phase:

Starting root cutting & heuristics								
Its	Type	BestSoln	BestBound	Sols	Add	Del	Gap	GInf Time
1	0		12.835517	0	17	1		6 1
2	0		12.364884	0	22	13		6 1
3	0		11.850442	0	28	20		8 1

Figure 7.3: MIQP root cutting and heuristics

A single integer fesaible solution is found during the Branch-and-Bound search. The search has been completed, this means that optimality of this solution has been proven (we may have chosen to stop the search, for example, after a given number of nodes, in which case it may not be possible to prove optimality or even to find the best solution).

Cuts in the matrix : 23								
Cut elements in the matrix : 184								
Starting tree search.								
Deterministic mode with up to 8 running threads and up to 16 tasks.								
Heap usage: 3353KB (peak 7201KB, 598KB system)								
Node	BestSoln	BestBound	Sols Active	Depth	Gap	GInf	Time	
1		11.055248	0	2	1		7	1
2		11.055248	0	2	2		5	1
3		11.038710	0	1	3		3	1
4		11.038710	0	1	4		2	1
c 5	9.172764	11.038710	1	1	5	16.90%	0	1
6	9.172764	11.038710	1	1	5	16.90%	2	1
8	9.172764	9.178014	1	3	1	0.06%	1	1
9	9.172764	9.178014	1	3	5	0.06%	2	1
10	9.172764	9.178014	1	3	6	0.06%	2	1
STOPPING - MIPRELSTOP target reached (MIPRELSTOP=0.0001 gap=1.96355e-005).								
*** Search completed *** Time: 1 Nodes: 10								
Number of integer feasible solutions found is 1								
Best integer solution found is 9.172764								
Best bound is 9.172944								

Figure 7.4: MIQP Branch-and-Bound search

CHAPTER 8

Heuristics

In this chapter we show a simple binary variable fixing solution heuristic that involves

- structuring a Mosel model via the definition of subroutines, and
- a heuristic solution procedure interacting with Xpress Optimizer through parameter settings, saving and recovering bases, and modifications of variable bounds.

Chapter 13 shows how to implement the same heuristic with BCL.

8.1 Binary variable fixing heuristic

The heuristic we wish to implement should perform the following steps:

1. Solve the LP relaxation and save the basis of the optimal solution
2. *Rounding heuristic*: Fix all variables 'buy' to 0 if the corresponding fraction bought is close to 0, and to 1 if it has a relatively large value.
3. Solve the resulting MIP problem.
4. If an integer feasible solution was found, save the value of the best solution.
5. Restore the original problem by resetting all variables to their original bounds, and load the saved basis.
6. Solve the original MIP problem, using the heuristic solution as cutoff value.

Step 2: Since the fraction variables *frac* have an upper bound of 0.3, as a 'relatively large value' in this case we may choose 0.2. In other applications, for binary variables a more suitable choice may be $1 - \varepsilon$, where ε is a very small value such as 10^{-5} .

Step 6: Setting a *cutoff value* means that we only search for solutions that are better than this value. If the LP relaxation of a node is worse than this value it gets cut off, because this node and its descendants can only lead to integer feasible solutions that are even worse than the LP relaxation.

8.2 Implementation with Mosel

For the implementation (file `folioheur.mos`) of the variable fixing solution heuristic we work with the MIP 1 model from Chapter 6. Through the definition of the heuristic in the form of a subroutine (more precisely, a *procedure*) we only make minimal changes to the model itself: at the beginning we declare

the procedure using the keyword `forward`, and before solving our problem with the standard call to the maximization function we execute our own solution heuristic. The solution printing also has been adapted.

```

model "Portfolio optimization solved heuristically"
uses "mmxprs"                                ! Use Xpress Optimizer

parameters
  MAXRISK = 1/3                                ! Max. investment into high-risk values
  MAXVAL = 0.3                                ! Max. investment per share
  MINAM = 0.5                                ! Min. investment into N.-American values
  MAXNUM = 4                                ! Max. number of assets
end-parameters

forward procedure solve_heur                  ! Heuristic solution procedure

declarations
  SHARES: set of string                        ! Set of shares
  RISK: set of string                          ! Set of high-risk values among shares
  NA: set of string                            ! Set of shares issued in N.-America
  RET: array(SHARES) of real                  ! Estimated return in investment
end-declarations

initializations from "folio.dat"
  RISK RET NA
end-initializations

declarations
  frac: array(SHARES) of mpvar                ! Fraction of capital used per share
  buy: array(SHARES) of mpvar                 ! 1 if asset is in portfolio, 0 otherwise
end-declarations

! Objective: total return
Return:= sum(s in SHARES) RET(s)*frac(s)

! Limit the percentage of high-risk values
sum(s in RISK) frac(s) <= MAXRISK

! Minimum amount of North-American values
sum(s in NA) frac(s) >= MINAM

! Spend all the capital
sum(s in SHARES) frac(s) = 1

! Upper bounds on the investment per share
forall(s in SHARES) frac(s) <= MAXVAL

! Limit the total number of assets
sum(s in SHARES) buy(s) <= MAXNUM

forall(s in SHARES) do
  buy(s) is_binary
  frac(s) <= buy(s)
end-do

! Solve problem heuristically
solve_heur

! Solve the problem
maximize(Return)

! Solution printing
if getprobatat=XPRS_OPT then
  writeln("Exact solution: Total return: ", getobjval)
  forall(s in SHARES) writeln(s, ": ", getsol(frac(s))*100, "%")
else
  writeln("Heuristic solution is optimal.")
end-if

```

```

!-----
procedure solve_heur
  declarations
    TOL: real                ! Solution feasibility tolerance
    fsol: array(SHARES) of real ! Solution values for `frac' variables
    bas: basis                ! LP basis
  end-declarations

  setparam("XPRS_VERBOSE",true) ! Enable message printing in mmxprs
  setparam("XPRS_CUTSTRATEGY",0) ! Disable automatic cuts
  setparam("XPRS_HEURSTRATEGY",0) ! Disable automatic MIP heuristics
  setparam("XPRS_PRESOLVE",0) ! Switch off presolve
  TOL:=getparam("XPRS_FEASTOL") ! Get feasibility tolerance
  setparam("ZEROTOL",TOL) ! Set comparison tolerance

  maximize(XPRS_LPSTOP,Return) ! Solve the LP problem
  savebasis(bas) ! Save the current basis

  ! Fix all variables `buy' for which `frac' is at 0 or at a relatively
  ! large value
  forall(s in SHARES) do
    fsol(s) := getsol(frac(s)) ! Get the solution values of `frac'
    if (fsol(s) = 0) then
      setub(buy(s), 0)
    elif (fsol(s) >= 0.2) then
      setlb(buy(s), 1)
    end-if
  end-do

  maximize(XPRS_CONT,Return) ! Solve the MIP problem
  ifgsol:=false
  if getprobat=XPRS_OPT then ! If an integer feas. solution was found
    ifgsol:=true
    solval:=getobjval ! Get the value of the best solution
    writeln("Heuristic solution: Total return: ", solval)
    forall(s in SHARES) writeln(s, ": ", getsol(frac(s))*100, "%")
  end-if

  ! Reset variables to their original bounds
  forall(s in SHARES)
    if ((fsol(s) = 0) or (fsol(s) >= 0.2)) then
      setlb(buy(s), 0)
      setub(buy(s), 1)
    end-if

  loadbasis(bas) ! Load the saved basis

  if ifgsol then ! Set cutoff to the best known solution
    setparam("XPRS_MIPABSCUTOFF", solval+TOL)
  end-if
end-procedure
end-model

```

This model certainly requires some more detailed explanations.

8.2.1 Subroutines

A *subroutine* in Mosel has a similar structure as the model itself: a procedure starts with the keyword `procedure`, followed by the name of the procedure, and terminates with `end-procedure`. Similarly, a function starts with the keyword `function`, followed by its name, and terminates with `end-function`. Both types of subroutines may take a list of arguments and for functions in addition the return type must be indicated, for example:

```
function myfunc(myint: integer, myarray: array(range) of string): real
```

for a function that returns a real and takes as input arguments an integer and an array of string.

As shown in our example, a subroutine may contain one (or several) `declarations` blocks. The objects defined in a subroutine are only valid locally and are deleted at the end of the subroutine.

Subroutine definitions may be *overloaded*, that is, a single subroutine may take different combinations of arguments. It is possible to overload any subroutines defined by Mosel and its modules, provided that the new definition differs from the existing one(s) in at least one argument.

For more detail and further examples of subroutine definition see the 'Mosel User Guide'.

8.2.2 Optimizer parameters and functions

Parameters: The solution heuristic starts with parameter settings for Xpress Optimizer. For a detailed explanation of all Optimizer parameters the reader is referred to the 'Optimizer Reference Manual'. All parameters are accessed through the Mosel subroutines `setparam` and `getparam`. In the example, we first enable the output printing by the module `mmxprs`. As a result, more information than what is printed by our model will be displayed in the logging pane:

```

Run Command: folioheur.mos CWD ENV

Its Obj Value S Minf Nneg Sum Dual Inf Time
5 14.066667 0 0 0 .000000 0
Dual solved problem
5 simplex iterations in 0s

Final objective : 1.406666666666667e+01
Max primal violation (abs/rel) : 0.0 / 0.0
Max dual violation (abs/rel) : 0.0 / 0.0
Max complementarity viol. (abs/rel) : 0.0 / 0.0
All values within tolerances
*** Search unfinished *** Time: 0 Nodes: 0
Number of integer feasible solutions found is 0
Best bound is 14.066667

Starting root cutting & heuristics

Its Type BestSoln BestBound Sols Add Del Gap GInf Time
* 13.100000 13.100000 1 -0.00% 0 0
*** Search completed *** Time: 0 Nodes: 1
Number of integer feasible solutions found is 1
Best integer solution found is 13.100000
Best bound is 13.100000
Heuristic solution: Total return: 13.1
treasury: 20%
hardware: 0%
theater: 30%
telecom: 0%
brewery: 20%
highways: 30%
cars: 0%
bank: 0%
software: 0%
electronics: 0%
FICO Xpress v8.7.0, Hyper, solve started 19:04:38, Oct 15, 2019
Maximizing MILP \xprs_2b51478
Original problem has:
14 rows 20 cols 49 elements 10 globals

```

Figure 8.1: Optimizer output display

Switching off the automated cut generation (parameter `XPRS_CUTSTRATEGY`) and the MIP heuristics (parameter `XPRS_HEURSTRATEGY`) is optional, whereas it is required in our case to disable the presolve mechanism (a treatment of the matrix that tries to reduce its size and improve its numerical properties, set with parameter `XPRS_PRESOLVE`), because we interact with the problem in the Optimizer in the course of its solution and this is only possible correctly if the matrix has not been modified by the Optimizer.

In addition to the parameter settings we also retrieve the feasibility tolerance used by Xpress Optimizer: the Optimizer works with tolerance values for integer feasibility and solution feasibility that are typically of the order of 10^{-6} by default. When evaluating a solution, for instance by performing comparisons, it is important to take into account these tolerances.

Optimization statement: We use a new version of the maximization procedure with an additional argument, `XPRS_LPSTOP`, indicating that we only want to solve the top node LP relaxation (and not yet the entire MIP problem). To continue with MIP solving from the point where we have stopped the algorithm we use the argument `XPRS_CONT`. This is an example of an overloaded subroutine definition.

Saving and loading bases: To speed up the solution process, we save (in memory) the current basis of the Simplex algorithm after solving the initial LP relaxation, before making any changes to the problem.

This basis is loaded again at the end, once we have restored the original problem. The MIP solution algorithm then does not have to re-solve the LP problem from scratch, it resumes the state where it was 'interrupted' by our heuristic.

Bound changes: When a problem has already been loaded into the Optimizer (e.g. after executing an optimization statement or following an explicit call to `loadprob`) bound changes via `setlb` and `setub` are passed on directly to the Optimizer. Any other changes (addition or deletion of constraints or variables) always lead to a complete reloading of the problem.

For more detail on the Optimizer functionality used in this example see the documentation of the module *mmxprs* in the 'Mosel Language Reference Manual'.

8.2.3 Comparison tolerance

After retrieving the feasibility tolerance of the Optimizer we set the comparison tolerance of Mosel (`ZEROTOL`) to this value `TOL`. This means that the test `fso1(s) = 0` evaluates to true if `fso1(s)` lies between `-TOL` and `TOL`, and `fso1(s) >= 0.2` is satisfied if the value of `fso1(s)` is at least `0.2-TOL`.

Comparisons in Mosel always use a tolerance, with a very small default value. By resetting this parameter to the Optimizer feasibility tolerance Mosel evaluates solution values just like the Optimizer.

CHAPTER 9

Embedding a Mosel model in an application

Mosel models frequently need to be embedded in applications so they can be deployed easily. In this chapter we discuss

- how to generate a deployment template,
- the meaning and use of BIM files,
- embedding Mosel models into a host application,
- the use of parameterized model and BIM files,
- how to export matrix files with Mosel, and
- how to create an Xpress Insight application from a model file.

9.1 Generating a deployment template

Open menu *File* » *New* and select the entry *Mosel deployment - Java*. (For deployment with C, C#, or any other supported language the procedure is similar.)

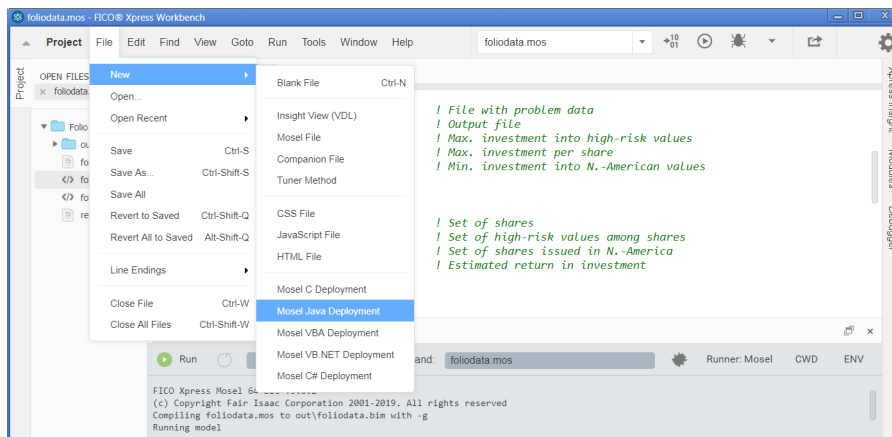


Figure 9.1: Choosing the deployment type

This will open a new file in the editor window with the resulting code:



```

1 // To compile: javac -classpath .;%XPRESSDIR%\Lib\xprm.jar RunModel.java
2 // To run: java -classpath .;%XPRESSDIR%\Lib\xprm.jar RunModel
3
4 import java.io.*;
5 import com.dashoptimization.*;
6
7 public class RunModel
8 {
9     // Replace this with the path to your compiled bim file
10    private static final String BIM_FILE = "test.bim";
11
12    public static void main(String[] args)
13    {
14        int result;
15
16        try {
17            result=runMoselModel();
18            System.out.println("Model execution returned: "+result);
19        }
20        catch(XPRMLicenseError e)
21        {
22            System.out.println("Failed to licence Mosel : " + e.getMessage());
23        }
24        catch(java.lang.Exception e)
25        {
26            System.out.println("Model loading or execution error : " + e.getMessage());
27        }
28    }
29
30    static int runMoselModel() throws XPRMLicenseError, IOException
31    {
32        XPRMModel model;
33        XPRM xprm;
34        int result;
35
36        // Initialize Mosel
37        xprm = new XPRM();
38
39        // Load compiled model (.BIM file)
40        model=xprm.loadModel(BIM_FILE);

```

Figure 9.2: Code preview

Find the constant with the value `test.bim` near the top of the file and change its value to the name of your BIM file (e.g. `foliodata.bim`). Use the menu *File* » *Save As...* to set the name (`folio.java`) and location of the new file. At the top of the code window a standard compilation line for Java under Windows is shown. To use it with the file we have just generated, replace `RunModel.java` by the name of our file, `folio.java`.

The Java program may be run on all systems for which Mosel is available. To compile under Linux or Solaris use:

```
javac -cp .:${XPRESSDIR}/lib/xprm.jar folio.java
```

For other systems please refer to the examples `makefile` of the corresponding Mosel distribution.

9.2 BIM files

Mosel models are typically distributed in the form of a *BIM file* (**B**inary **M**odel file). A BIM file is a compiled version of the `.mos` model file that is portable across all platforms for which Mosel is available. It does *not* include any data read from external files. These must still be provided in separate files, thus making it possible to run the same BIM file with different data sets (see section *Parameters* below).

To generate a BIM file with Workbench you may use *Run* » *Compile* or equivalently, click on the button

 The BIM file will then be created in the subdirectory `out` of the Workbench project by appending the extension `.bim` to the file name (instead of `.mos`). You may also use the *Compiler Options* dialog (opened either from the *Run* menu or by clicking on the tools button ) to configure, for example, various debugging settings for the compilation.

It is also possible to execute Mosel source files (`.mos`) directly from an application (see the following section). In this case the BIM file does not need to be generated.

9.3 Embedding Mosel models into a host application

9.3.1 Executing Mosel models

The following simple Java program can be used to run a Mosel model that is provided in the form of a BIM file (for simplicity's sake we are leaving out any kind of error handling):

```
import com.dashoptimization.*;

public class folio
{
    public static void main(String[] args) throws Exception
    {
        XPRM mosel;
        XPRMModel model;

        mosel = new XPRM(); // Initialize Mosel
        model = mosel.loadModel("foliodata.bim"); // Load compiled model
        model.run(); // Run the model

        System.out.println("Model execution returned: " + model.getResult());
    }
}
```

This Java program may be run on all systems for which Mosel is available. Under Windows use these commands to compile and run the program:

```
javac -classpath .;%XPRESSDIR%\lib\xprm.jar folio.java
java -classpath .;%XPRESSDIR%\lib\xprm.jar folio
```

To compile under Linux or Solaris use:

```
javac -cp .:${XPRESSDIR}/lib/xprm.jar folio.java
```

If we also wish to create the BIM file from the Java application, we may compile, load, and run the Mosel model `foliodata.mos` directly from the Java program, for instance as shown in the following code fragment. The compilation functionality is equally contained in the JAR file `xprm.jar` so that we can use the same compilation command as before.

```
import com.dashoptimization.*;

public class folio
{
    public static void main(String[] args) throws Exception
    {
        XPRM mosel;
        XPRMModel model;

        mosel = new XPRM(); // Initialize Mosel
        mosel.compile("foliodata.mos"); // Compile the model
        model = mosel.loadModel("foliodata.bim"); // Load compiled model
    }
}
```

```

        model.run(); // Run the model

        System.out.println("Model execution returned: " + model.getResult());
    }
}

```

9.3.2 Parameters

In Chapter 4 we have shown how to modify parameter settings with Workbench or when running the Mosel standalone version (for instance in batch files or scripts). The model parameters may also be reset when a Mosel model or BIM file is embedded in an application, making it possible to solve many different problem instances without having to change the model source.

In this example we modify the name of the result file and the settings for two numerical parameters of our model `foliodata.mos`. All other model parameters will take the default values specified at their definition in the model.

```

import com.dashoptimization.*;

public class folioparam
{
    public static void main(String[] args) throws Exception
    {
        XPRM mosel;
        XPRMModel model;

        mosel = new XPRM(); // Initialize Mosel
        mosel.compile("foliodata.mos"); // Compile the model
        model = mosel.loadModel("foliodata.bim"); // Load compiled model
        // Set the run-time parameters
        model.execParams = "OUTFILE=result2.dat,MAXRISK=0.4,MAXVAL=0.25";
        model.run(); // Run the model

        System.out.println("`foliodata' returned: " + model.getResult());
    }
}

```

9.3.3 Retrieving solution information

After running a model, it is possible to retrieve information about the model objects and the solution of the (last) optimization run. The following example shows how to test the problem status and retrieve the objective function value.

```

import com.dashoptimization.*;

public class folioobj
{
    public static void main(String[] args) throws Exception
    {
        XPRM mosel;
        XPRMModel model;

        mosel = new XPRM(); // Initialize Mosel
        mosel.compile("foliodata.mos"); // Compile the model
        model = mosel.loadModel("foliodata.bim"); // Load compiled model
        model.run(); // Run the model

        // Test whether a solution is found and print the objective value
        if(model.getProblemStatus()==XPRMModel.PB_OPTIMAL)
            System.out.println("Objective value: " + model.getObjectiveValue());
    }
}

```

9.4 Matrix files

9.4.1 Exporting matrices

If the optimization process with Xpress Optimizer is started from within a Mosel program, or if the solving procedure is part of the application into which a Mosel model has been embedded, then the problem matrix is loaded in memory into the solver without writing it out to a file (which would be expensive in terms of running time). However, in certain cases it may still be required to be able to produce a matrix. With Xpress, the user has the choice between two matrix formats: extended MPS and extended LP format, the latter being in general more easily human-readable since constraints are printed in algebraic form.

With Mosel, there are several possibilities for generating a matrix:

1. *With a matrix generation statement in the model file:*
to create an MPS matrix for our problem add the line

```
exportprob(EP_MPS, "folio", Return)
```

for an LP format matrix (which we intend to maximize at some point) add the line

```
exportprob(EP_MAX, "folio", Return)
```

immediately before or instead of the optimization statement.

2. *From a Java application after having executed the model file:*

```
XPRMModel model;  
model.exportProblem("m", "folio");
```

This will output the matrix in MPS format. To print with LP format change the first argument of `exportProblem`:

```
model.exportProblem("p", "folio");
```

9.5 Deployment to Xpress Insight

Xpress Insight embeds Mosel models into a multi-user application for deploying optimization models in a distributed client-server architecture. Through the Xpress Insight GUI, business users interact with Mosel models to evaluate different scenarios and model configurations without directly accessing to the model itself.

9.5.1 Preparing the model file

For embedding a Mosel model into Xpress Insight, we need to make a few edits to the Mosel model in order to establish the connection between Mosel and Xpress Insight.

Firstly, we need to load the package *mminstight* that provides the required additional functionality. Since Insight manages the data scenarios, we only need to read in data from the original sources when *loading* the scenario (also referred to as *baseline run*) into Insight (triggered by the test of the run mode with `insightgetmode` in the model below). Scenario data will otherwise be input directly from Xpress Insight at the insertion point marked with `insightpopulate`. All model entities that are to be managed by Xpress Insight need to be declared as `public`. Furthermore, the solver call to start the optimization is replaced by `insightminimize` / `insightmaximize`.

The resulting model file `folioinsight.mos` (based on `foliodata.mos`) has the following contents—this model can also simply be run standalone, e.g. from Workbench or the Mosel command line, this is the case handled by `INSIGHT_MODE_NONE`.

```

model "Portfolio optimization with LP"
  uses "mmxprs"                ! Use Xpress Optimizer
  uses "mminsight"             ! Use Xpress Insight

  parameters
    DATAFILE= "folio.dat"      ! File with problem data
    MAXRISK = 1/3                ! Max. investment into high-risk values
    MAXVAL = 0.3                 ! Max. investment per share
    MINAM = 0.5                  ! Min. investment into N.-American values
  end-parameters

  public declarations
    SHARES: set of string       ! Set of shares
    RISK: set of string          ! Set of high-risk values among shares
    NA: set of string            ! Set of shares issued in N.-America
    RET: array(SHARES) of real  ! Estimated return in investment
  end-declarations

  case insightgetmode of
    INSIGHT_MODE_LOAD: do      ! 'Load data' mode: Read data, then stop
      initializations from DATAFILE
      RISK RET NA
    end-initializations
    exit(0)
  end-do
  INSIGHT_MODE_RUN:           ! 'Run' mode: Inject scen. data, continue
    insightpopulate
  INSIGHT_MODE_NONE:          ! Standalone run: Read data and continue
    initializations from DATAFILE
    RISK RET NA
  end-initializations
  else
    writeln("Unknown execution mode")
    exit(1)
  end-case

  public declarations
    frac: array(SHARES) of mpvar ! Fraction of capital used per share
    Return, LimitRisk, LimitAM, TotalOne: lincpr ! Constraints
  end-declarations

  ! Objective: total return
  Return:= sum(s in SHARES) RET(s)*frac(s)

  ! Limit the percentage of high-risk values
  LimitRisk:= sum(s in RISK) frac(s) <= MAXRISK

  ! Minimum amount of North-American values
  LimitAM:= sum(s in NA) frac(s) >= MINAM

  ! Spend all the capital
  TotalOne:= sum(s in SHARES) frac(s) = 1

  ! Upper bounds on the investment per share
  forall(s in SHARES) frac(s) <= MAXVAL

  ! Solve the problem through Xpress Insight
  insightmaximize(Return)

end-model

```

Note that we have removed all solution output from this model: we are going to use Xpress Insight for representing the results.

9.5.1.1 The app archive

Xpress Insight expects models to be provided in compiled form, that is, as BIM files—see Section 9.2 on how to generate BIM files from the model source. Since Xpress Insight executes Mosel models in a distributed architecture (so, possibly not on the same machine from where the model file is input) we recommend to include any input data files used by the model in the Xpress Insight *app archive*. The app archive is a ZIP archive that contains the BIM file and the optional subdirectories `model_resources` (data files), `client_resources` (custom view definitions), and `source` (Mosel model source files). For our example, we create a ZIP archive `folioinsight.zip` with the file `folioinsight.bim` and the data file `folio.dat` in the subdirectory `model_resources`.

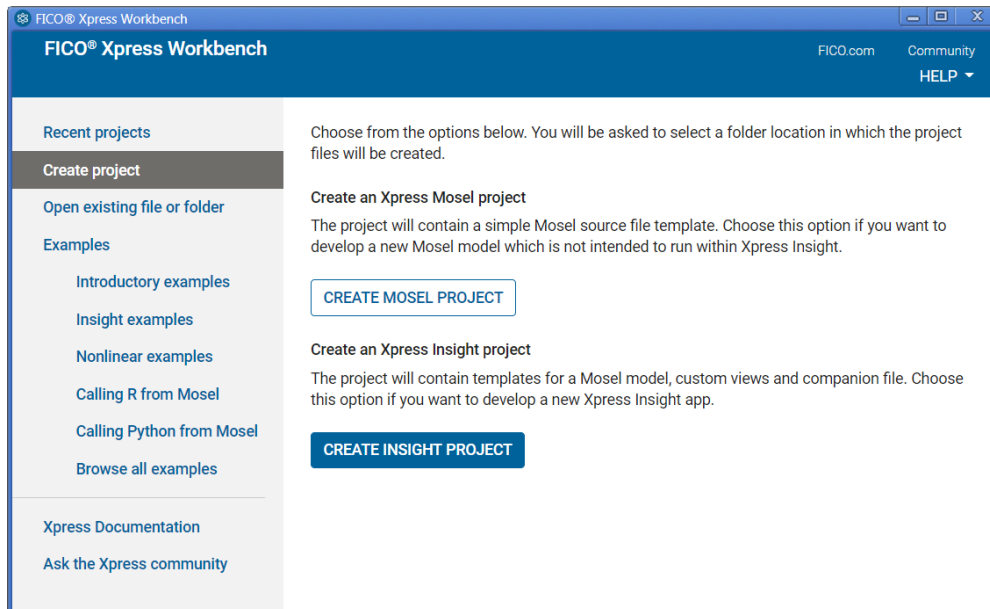


Figure 9.3: Creating a new Insight project

With Xpress Workbench, select the option 'Create project' followed by 'Create Insight project' at startup to create the directory structure expected by Xpress Insight and replace the template model (in subdirectory `source`), configuration (`application.xml` and subdirectory `client_resources`), and data files (in subdirectory `model_resources`) by the files of your Mosel project. In order to work with an existing app, select *Open existing file or folder* followed by *Open project* when starting up Workbench and browse to the desired folder or double click on a Mosel file in the `source` subdirectory and select 'Open Insight app' in the dialog box.

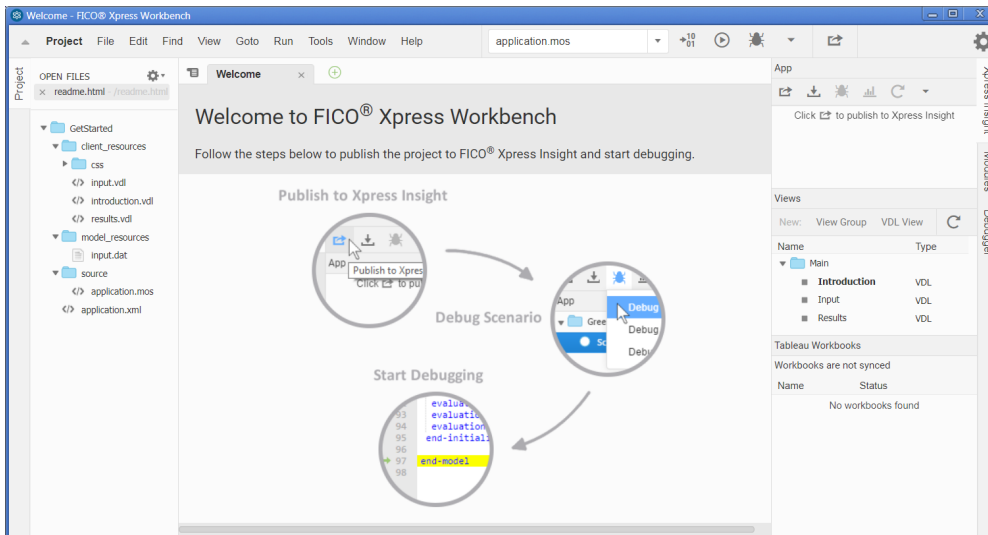


Figure 9.4: Default Xpress Insight app template

Select the button  to create the app archive or  to publish the app directly to Insight. If the app has been published successfully the link 'Open in Xpress Insight' in the green message box will take you to the app loaded in the Insight web client opened with your default web browser.

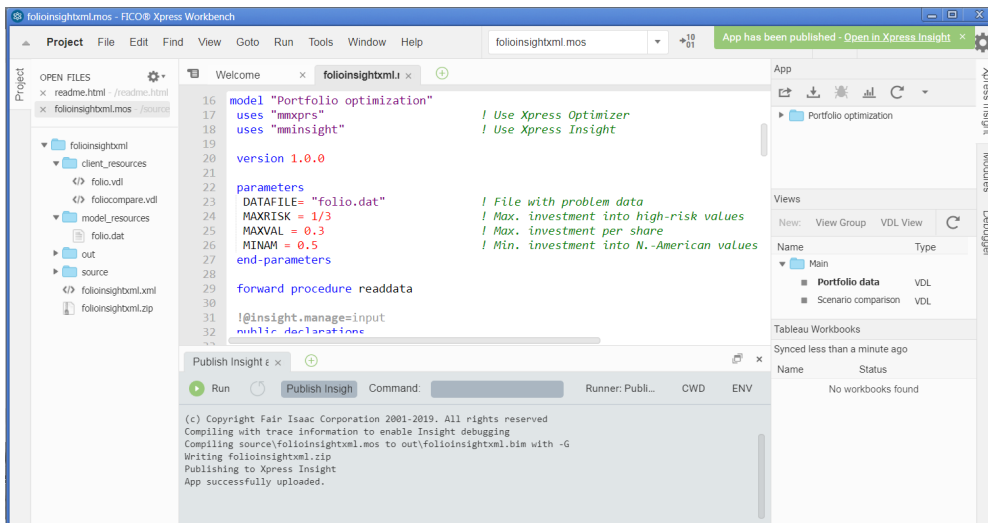


Figure 9.5: Deploying an app to Xpress Insight

9.5.2 Working with the Xpress Insight Web Client

Open the Xpress Insight Web Client by directing your web browser to the Web Client entry page: with a default desktop installation of Xpress Insight this will be the page <http://localhost:8860/insight>

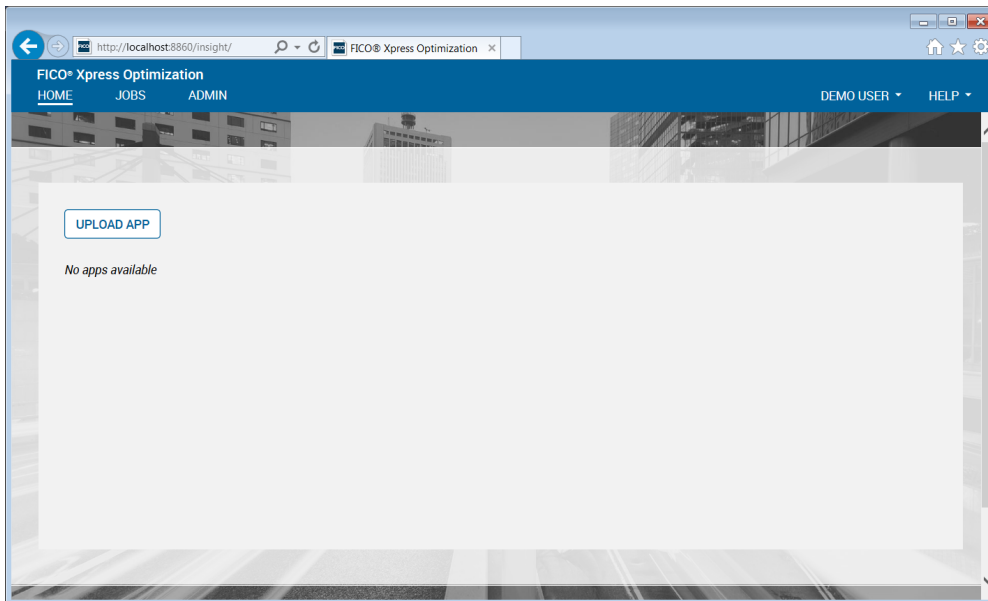


Figure 9.6: Xpress Insight web client entry page

If you have currently loaded any apps in Insight these will show up on the Web Client entry page, otherwise this page only displays the 'Upload app' icon. We now upload the app archive `folioinsightxml.zip` that adds a *VDL view definition* file and an XML configuration file to the archive `folioinsight.zip`. The Mosel model has been extended with the array `CtrlSol` to store some result and data values in a convenient format for display (note the use of annotation marker `!@insight.manage` that is required to inform Xpress Insight that these data are not input but result values):

```
!@insight.manage=result
public declarations
  CTRS: set of string                ! Constraint names
  CTRINFO: set of string            ! Constraint info type
  CtrlSol: dynamic array(CTRS,CTRINFO) of real ! Solution values
end-declarations

! Save solution values for GUI display
CtrlSol::("Limit high risk shares", ["Activity", "Lower limit", "Upper limit"])
  [LimitRisk.act, 0, MAXRISK]
CtrlSol::("Limit North-American", ["Activity", "Lower limit", "Upper limit"])
  [LimitAM.act, MINAM, 1]
forall(s in SHARES | frac(s).sol > 0) do
  CtrlSol("Limit per value: "+s, "Activity") := frac(s).sol
  CtrlSol("Limit per value: "+s, "Upper limit") := MAXVAL
  CtrlSol("Limit per value: "+s, "Lower limit") := 0
end-do
```

Optionally, we can also add annotations to individual declarations in order to configure the GUI display of model entities:

```
public declarations
  SHARES: set of string                !@insight.alias Shares
  RET: array(SHARES) of real           !@insight.alias Estimated return in investment
  frac: array(SHARES) of mpvar         !@insight.alias Fraction used
  Return: lincstr                      !@insight.alias Total return
  TotalOne: lincstr                   !@insight.hidden true
end-declarations
```

Once you have successfully loaded the app archive, the app 'Portfolio Optimization' will show up as a new icon:

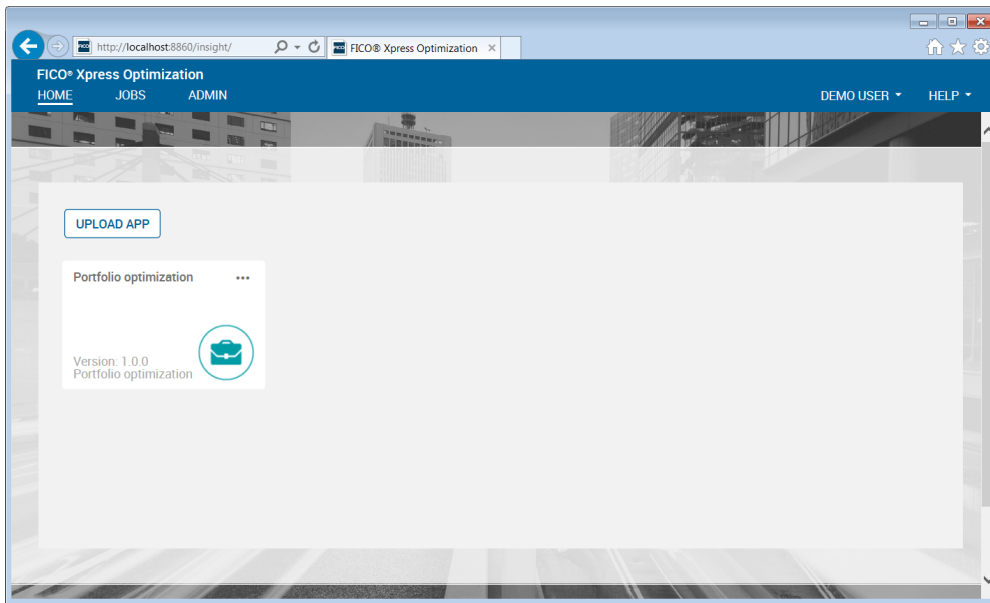


Figure 9.7: Xpress Insight web client after loading the Portfolio app

Select the 'Portfolio Optimization' app icon to open the app. Note that if you have deployed an app from Workbench and followed the link 'Open in Xpress Insight' you will immediately be taken to this page.

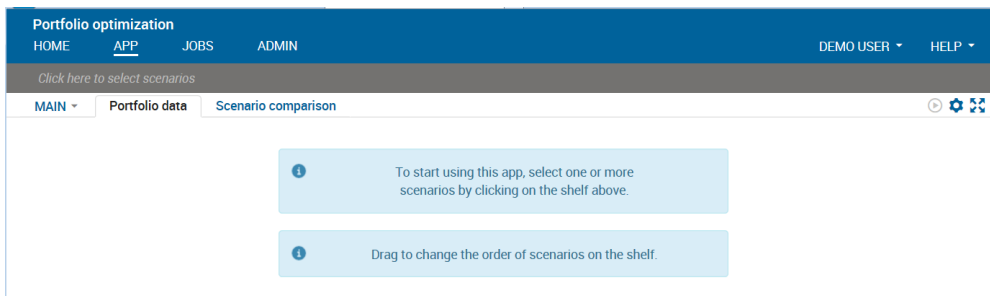


Figure 9.8: App entry page

Now click on the text *Click here to select scenarios* in the shelf to create a scenario. In the 'Scenario Explorer' window, select 'Scenario 1' and confirm with *Close*.

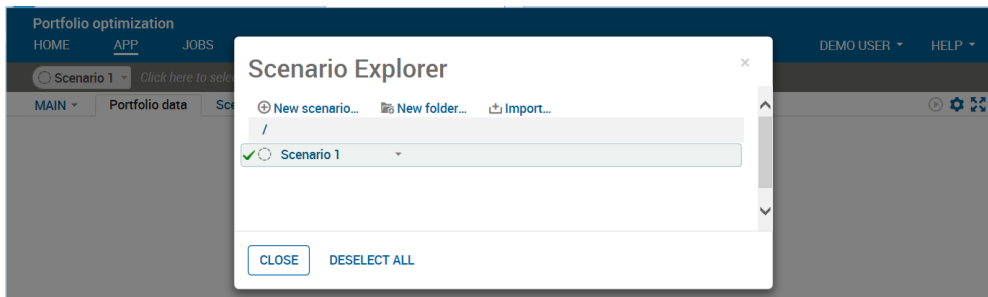


Figure 9.9: Scenario creation in the Xpress Insight web client

Use the *Load* entry from the drop-down menu on the scenario name in the shelf to load the baseline data.

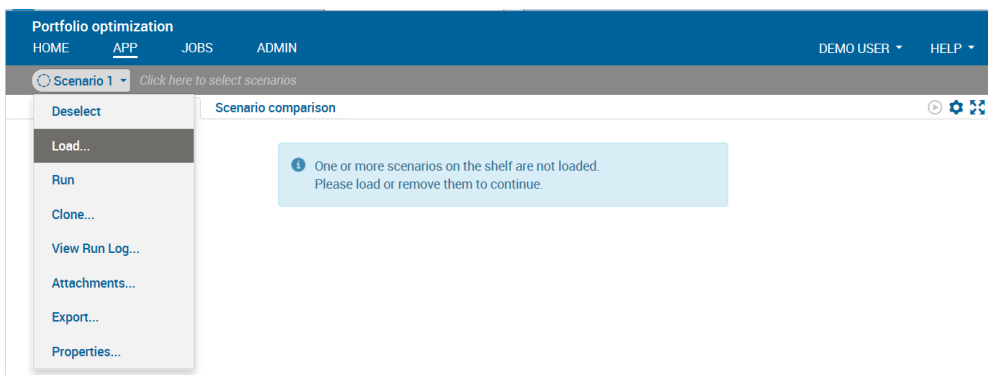


Figure 9.10: Scenario menu in the Xpress Insight web client

After loading the scenario the view display changes, showing the input data of our optimization model. You can edit these data by entering new values into the input fields or table cells. Use the *Run* button on the view or the corresponding entry in the scenario menu to run the model with the data shown on screen.

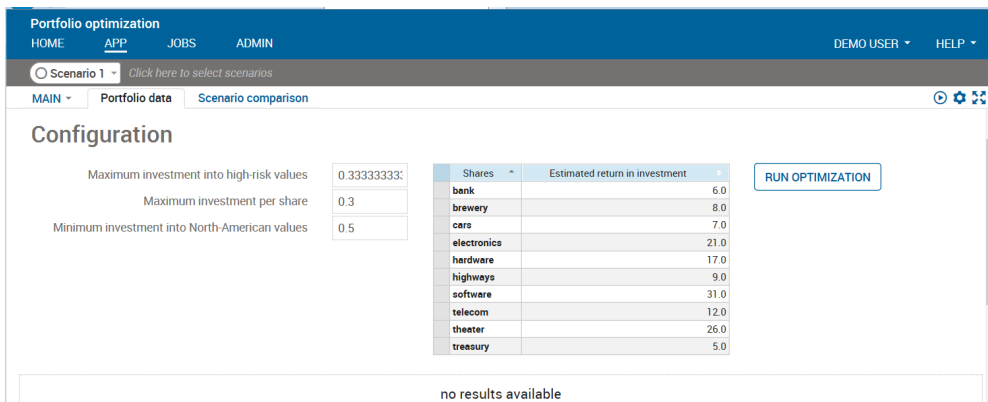


Figure 9.11: Display after scenario loading

After a successful model run the placeholder message *no results available* in the lower half of our view is replaced by the results display, as shown below.

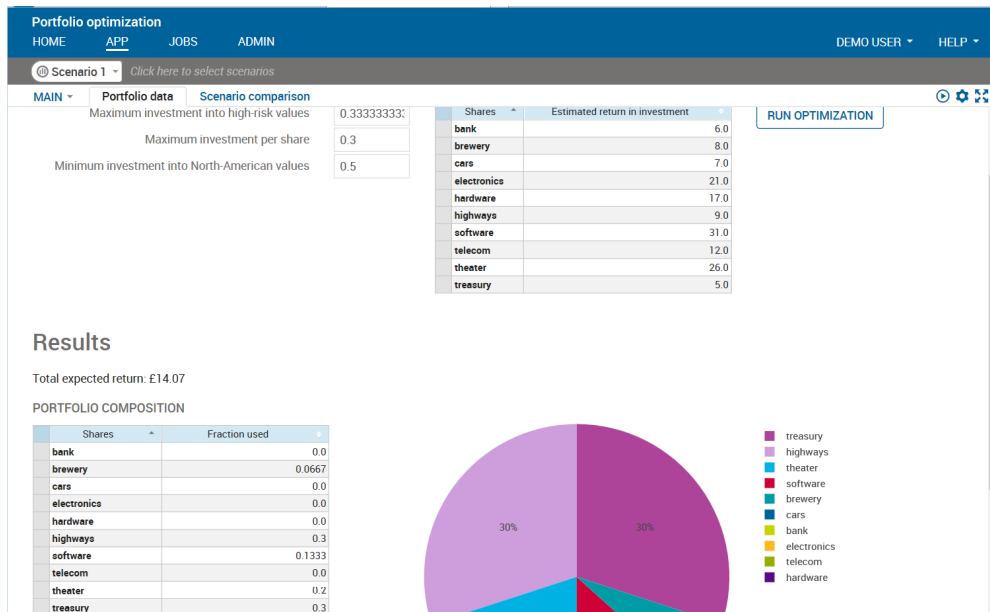


Figure 9.12: VDL view with input and result data elements

You can create new scenarios from existing ones (selecting 'Clone' in the scenario menu) or with the original input data by selecting 'New scenario' in the *Scenario Explorer* window. The results of multiple scenarios can be displayed in a single view for comparison.

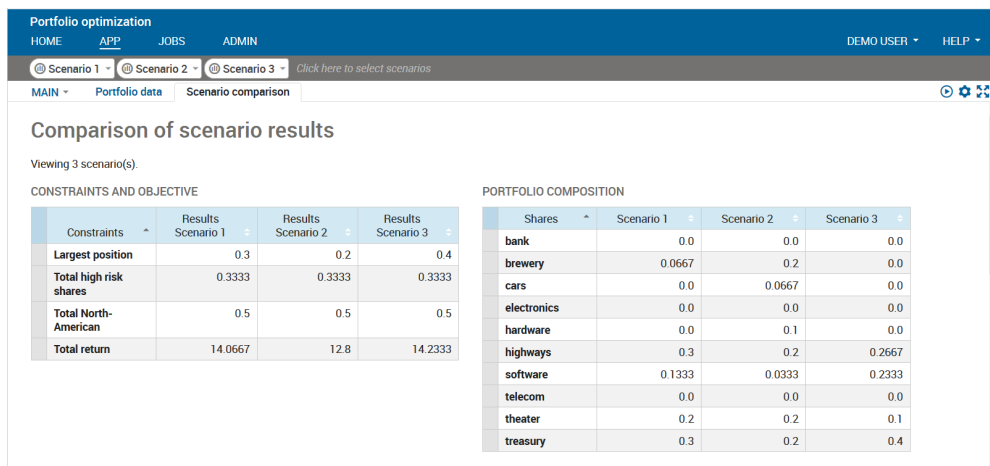


Figure 9.13: VDL view comparing several scenarios

9.5.2.1 VDL

VDL (View Definition Language) is a markup language for the creation of views for Xpress Insight apps from a set of predefined components and built-in styling options. Optionally, VDL view definitions can be extended with HTML tags and Javascript code for further customization.

Xpress Workbench includes a drag-and-drop editor for the creation and editing of VDL views. Within Xpress Workbench, select menu *File* » *New* » *Insight View (VDL)* to launch the view creation dialog. Enter 'Portfolio data' as the view title and `f01io.vdl` as the filename for the view and in the following

screen select 'Basic view' layout before terminating the dialog with 'Finish'. In the drag-and-drop editor that now shows, drag objects from the palette on the left onto the central artboard area—when doing so the editor will provide guidance regarding which combinations of objects are permitted (for example, a 'row' needs to contain 'columns' into which you can then add objects like 'table', 'chart' or 'text').

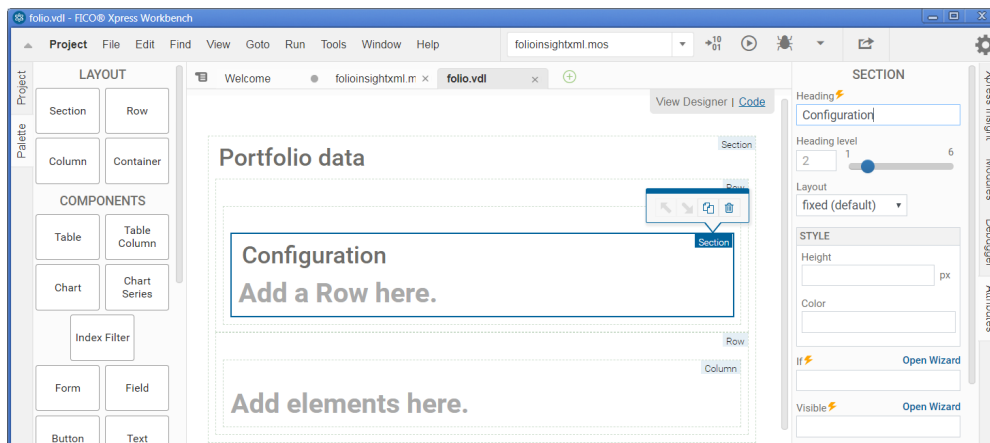


Figure 9.14: VDL view designer in Xpress Workbench

The attributes for the currently selected element in the editor can be edited in the pane on the right hand side.

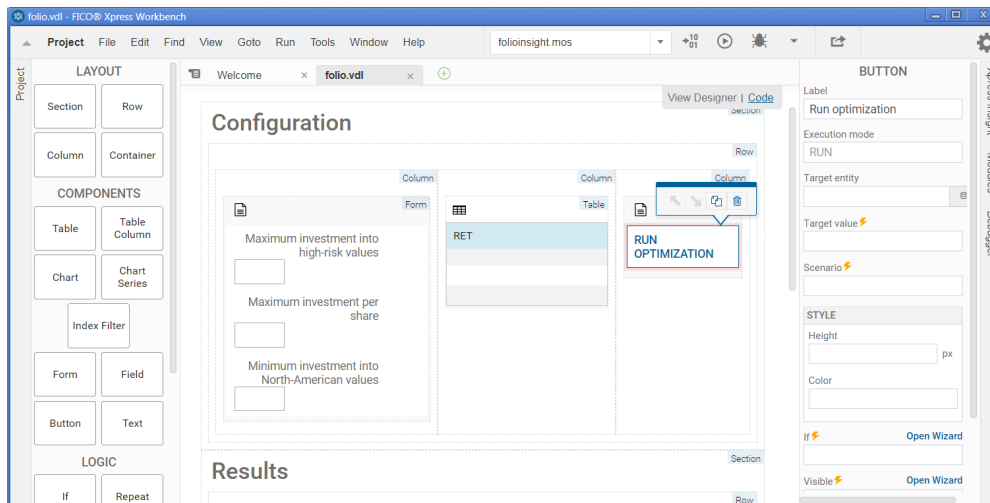


Figure 9.15: VDL view designer: editing view elements

For certain elements (table, chart) specific dialog windows will open to guide the user through their configuration.

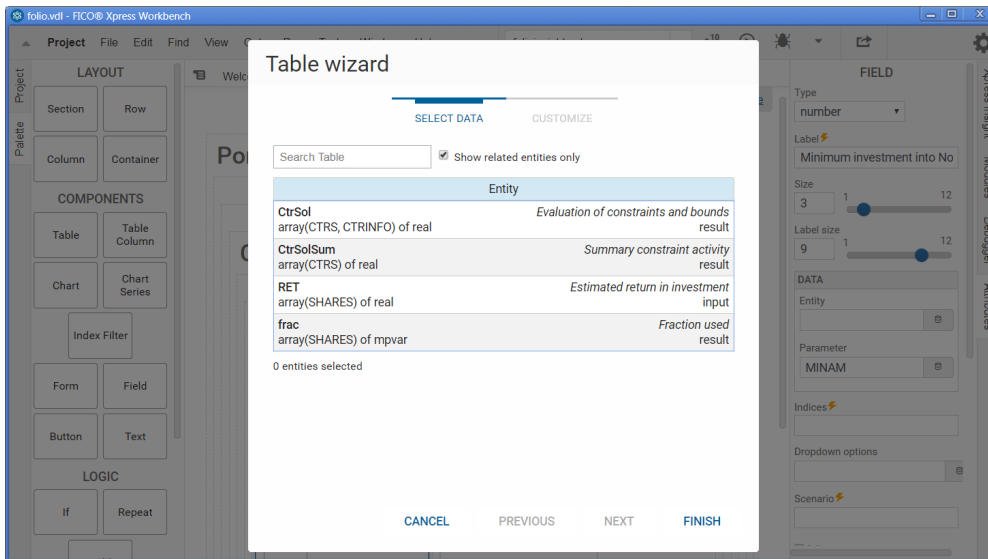


Figure 9.16: VDL view designer: table definition wizard

Note that at any time during the editing of VDL views in Workbench the app can be published to Insight by selecting the button  in order to inspect the actual appearance of the web views when they are populated with scenario data.

The view 'Portfolio data' shown as web view in Figure 9.12 and in the VDL designer in Figure 9.15 is created entirely from the following VDL view definition (file `folio.vdl` in the subdirectory `client_resources` of the app archive). All data entities marked as 'editable' can be modified by the UI user.

```
<vdl version="4.7">
  <vdl-page>
    <!-- 'vdl' and 'vdl-page' tags must always be present -->

    <!-- 'header' element: container for any vdl elements that are not part
      of the page layout -->
    <vdl-header>
      <vdl-action-group name="runModel">
        <vdl-action-execute mode="RUN"></vdl-action-execute>
      </vdl-action-group>
    </vdl-header>

    <!-- Structural element 'section': print header text for a section -->
    <vdl-section heading="Configuration">

      <!-- Structural element 'row': arrange contents in rows -->
      <vdl-row>
        <!-- Several columns within a 'row' for display side-by-side,
          dividing up the total row width of 12 via 'size' setting
          on each column. -->
        <vdl-column size="5">
          <!-- A form groups several input elements -->
          <vdl-form>
            <!-- Input fields for constraint limits -->
            <vdl-field parameter="MAXRISK" size="3" label-size="9"
              label="Maximum investment into high-risk values"/>
            <vdl-field parameter="MAXVAL" size="3" label-size="9"
              label="Maximum investment per share"/>
            <vdl-field parameter="MINAM" size="3" label-size="9"
              label="Minimum investment into North-American values" />
            <!-- default sizes: 2 units each -->
```

```

        </vdl-form>
    </vdl-column>
    <vdl-column size="4">
        <!-- Display editable input values, default table format -->
        <vdl-table>
            <vdl-table-column entity="RET" editable="true"/>
        </autotable>
    </vdl-column>
    <vdl-column size="3">
        <vdl-form>
            <!-- 'Run' button to launch optimization -->
            <vdl-button vdl-event="click:actions.runModel"
                label="Run optimization"></vdl-button>
        </vdl-form>
    </vdl-column>
</vdl-row>
</vdl-section>

<!-- Placeholder message for 'Results' section -->
<vdl-container vdl-if="=!scenario.summaryData.hasResultData">
    <span vdl-text="no results available"></span></vdl-container>

<!-- Structural element 'section';
display: with option 'none' nothing gets displayed by default
if hasResultData: display section once result values become available
(after scenario execution) -->
<vdl-section heading="Results"
    vdl-if="=scenario.summaryData.hasResultData" style="display: none">

    <vdl-row>
        <vdl-column>
            <!-- Display text element with the objective value -->
            <span vdl-text="='Total expected return: &#163;' +
                insight.Formatter.formatNumber(scenario.entities.Return.value,
                    '##.00') "></span>
        </vdl-column>
    </vdl-row>

    <vdl-row>
        <vdl-column size="4" heading="Portfolio composition">
            <!-- Display the 'frac' solution values, default table format -->
            <vdl-table>
                <vdl-table-column entity="frac"></vdl-table-column>
            </vdl-table>
        </vdl-column>
        <vdl-column size="8">
            <!-- Display the 'frac' solution values as a pie chart -->
            <vdl-chart style="width:400px;">
                <vdl-chart-series entity="frac" type="pie"></vdl-chart-series>
            </vdl-chart>
        </vdl-column>
    </vdl-row>
</vdl-section>
</vdl-page>
</vdl>

```

VDL views need to be declared in an app archive via an XML configuration file (the so-called *companion file*). When VDL views are created via the view designer in Workbench then the required entry is added automatically to this file. Companion files can also be created and edited using the Workbench editor (select *File* » *New* » *Companion file*). The following companion file definition integrates the VDL view `folio.vdl` and a second view 'Scenario comparison' into our example app `folioinsight.xml.zip`.

```

<?xml version="1.0" encoding="iso-8859-1"?>
<model-companion version="3.0"
    xmlns="http://www.fico.com/xpress/optimization-modeler/model-companion" >
    <client>
        <view-group title="Main">

```

```
        <vdl-view title="Portfolio data" default="true" path="folio.vdl" />
        <vdl-view title="Scenario comparison" default="false" path="foliocompare.vdl"/>
    </view-group>
</client>
</model-companion>
```

II. Getting started with BCL

CHAPTER 10

Inputting and solving a Linear Programming problem

In this chapter we take the example formulated in Chapter 2 and show how to implement the model with BCL. With some extensions to the initial formulation we also introduce input and output functionalities of BCL:

- writing an LP model with BCL,
- data input from file using index sets,
- output facilities of BCL,
- exporting a problem to a matrix file.

Chapter 3 shows how to formulate and solve the same example with Mosel and in Chapter 15 the problem is input and solved directly with Xpress Optimizer.

10.1 Implementation with BCL

All BCL examples in this book are written with C++. Due to the possibility of overloading arithmetic operators in the C++ programming language, this interface provides the most convenient way of stating models in a close to algebraic form. The same models can also be implemented using the C, Java, or C# interfaces of BCL.

The following BCL program implements the LP example introduced in Chapter 2:

```
#include <iostream>
#include "xprb_cpp.h"

using namespace std;
using namespace ::dashoptimization;

#define NSHARES 10           // Number of shares
#define NRISK 5              // Number of high-risk shares
#define NNA 4                // Number of North-American shares

double RET[] = {5,17,26,12,8,9,7,6,31,21}; // Estimated return in investment
int RISK[] = {1,2,3,8,9}; // High-risk values among shares
int NA[] = {0,1,2,3}; // Shares issued in N.-America

int main(int argc, char **argv)
{
    int s;
```

```

XPRBprob p("FolioLP");           // Initialize a new problem in BCL
XPRBexpr Risk, Na, Return, Cap;
XPRBvar frac[NSHARES];           // Fraction of capital used per share

// Create the decision variables
for(s=0;s<NSHARES;s++) frac[s] = p.newVar("frac");

// Objective: total return
for(s=0;s<NSHARES;s++) Return += RET[s]*frac[s];
p.setObj(Return);                 // Set the objective function

// Limit the percentage of high-risk values
for(s=0;s<NRISK;s++) Risk += frac[RISK[s]];
p.newCtr("Risk", Risk <= 1.0/3);

// Minimum amount of North-American values
for(s=0;s<NNA;s++) Na += frac[NA[s]];
p.newCtr("NA", Na >= 0.5);

// Spend all the capital
for(s=0;s<NSHARES;s++) Cap += frac[s];
p.newCtr("Cap", Cap == 1);

// Upper bounds on the investment per share
for(s=0;s<NSHARES;s++) frac[s].setUB(0.3);

// Solve the problem
p.setSense(XPRB_MAXIM);
p.lpOptimize("");

// Solution printing
cout << "Total return: " << p.getObjVal() << endl;
for(s=0;s<NSHARES;s++)
    cout << s << ": " << frac[s].getSol()*100 << "%" << endl;

return 0;
}

```

Let us now have a closer look at what we have just written.

10.1.1 Initialization

To use the BCL C++ interface you need to include the header file `xprb_cpp.h`. We also define the namespace to which the BCL classes belong.

If the software has not been initialized previously, BCL is initialized automatically when the first problem is created, that is by the line

```
XPRBprob p("FolioLP");
```

which creates a new problem with the name 'FolioLP'.

10.1.2 General structure

The definition of the model itself starts with the creation of the decision variables (method `newVar`), followed by the definition of the objective function and the constraints. In C++ (and Java, C#) constraints may be created starting with linear expressions as shown in the example. Equivalently, they may be constructed termwise, for instance the constraint limiting the percentage of high-risk shares:

```

XPRBctr CRisk;
CRisk = p.newCtr("Risk");
for(s=0;s<NRISK;s++) CRisk.addTerm(frac[RISK[s]], 1);
CRisk.setType(XPRB_L);
CRisk.addTerm(1.0/3);

```

This second type of constraint definition is common to all BCL interfaces and is the only method of defining constraints in C where overloading is not available.

Notice that in the definition of *equality constraints* (here the constraint stating that we wish to spend all the capital) we need to employ a double equality sign `==`.

The method `setUB` is used to set the *upper bounds* on the decision variables `frac`. Alternatively to this separate function call, we may also specify the bounds directly at the creation of the variables, but in this case we need to provide the full information including the name, variable type (`XPRB_PL` for continuous), lower and upper bound values:

```
for(s=0;s<NSHARES;s++) frac[s] = p.newVar("frac", XPRB_PL, 0, 0.3);
```

Giving string *names* to modeling objects (decision variables, constraints, etc.) as shown in our example program is optional. If the user does not specify any name, BCL will generate a default name. However, user-defined names may be helpful for debugging and for the interpretation of output produced by the Optimizer.

10.1.3 Solving

Prior to launching the solver, the optimization direction is set to maximization with a call to `setSense`. With the method `lpOptimize` we then call Xpress Optimizer to maximize the objective function (Return) set with the method `setObj`, subject to all constraints that have been defined. The empty string argument of `lpOptimize` indicates that the default LP algorithm is to be used. Other possible values are "p" for primal, "d" for dual Simplex, and "b" for Newton-Barrier.

10.1.4 Output printing

The last few lines print out the value of the optimal solution and the solution values for all variables.

10.2 Compilation and program execution

If you have followed the standard installation procedure of Xpress Optimizer and BCL, you may compile this file with the following command under Windows (note that it is important to use the flag `/MD`):

```
cl /MD /I%XPRESSDIR%\include %XPRESSDIR%\lib\xprb.lib foliolp.cpp
```

For Linux or Solaris use

```
cc -D_REENTRANT -I${XPRESSDIR}/include -L${XPRESSDIR}/lib foliolp.C -o foliolp -lxprb
```

For other systems please refer to the example makefile provided with the corresponding distribution.

Running the resulting program will generate the following output:

```
Reading Problem FolioLP
Problem Statistics
      3 (      0 spare) rows
     10 (      0 spare) structural columns
     19 (      0 spare) non-zero elements
Global Statistics
      0 entities          0 sets          0 set members
Maximizing LP FolioLP
Original problem has:
      3 rows             10 cols             19 elements
Presolved problem has:
      3 rows             10 cols             19 elements
```

```

      Its      Obj Value      S      Ninf      Nneg      Sum Inf      Time
      0      42.600000      D      2      0      .000000      0
      5      14.066667      D      0      0      .000000      0
Uncrunching matrix
Optimal solution found
Dual solved problem
  5 simplex iterations in 0s

Final objective           : 1.406666666666667e+01
Max primal violation      (abs / rel) :      0.0 /      0.0
Max dual violation        (abs / rel) :      0.0 /      0.0
Max complementarity viol. (abs / rel) :      0.0 /      0.0
All values within tolerances
Problem status: optimal
Total return: 14.0667
0: 30%
1: 0%
2: 20%
3: 0%
4: 6.66667%
5: 30%
6: 0%
7: 0%
8: 13.3333%
9: 0%

```

The upper half of this display is the log of Xpress Optimizer: the size of the matrix, 3 rows (i.e. constraints) and 10 columns (i.e. decision variables), and the log of the LP solution algorithm (here: 'D' for dual Simplex). The lower half is the output produced by our program: the maximum return of 14.0667 is obtained with a portfolio consisting of shares 1, 3, 5, 6, and 9. 30% of the total amount are spent in shares 1 and 6 each, 20% in 3, 13.3333% in 9 and 6.6667% in 5. It is easily verified that all constraints are indeed satisfied: we have 50% of North-American shares (1 and 3) and 33.33% of high-risk shares (3 and 9).

It is possible to modify the amount of output printing by BCL and Xpress Optimizer by adding the following line before the start of the optimization:

```
p.setMsgLevel(1);
```

This setting will disable all output (including warnings) from BCL and Xpress Optimizer, with the exception of error messages. The possible values for the printing level range from 0 to 4. In Chapter 13 we show how to access the Optimizer control parameters directly which, for instance, allows fine tuning the message display.

10.3 Data input from file

Instead of simply numbering the decision variables, we may wish to use more meaningful indices in our model. For instance, the problem data may be given in file(s) using string indices such as the file `foliocpplp.dat` we now wish to read:

```

! Return
"treasury"  5
"hardware"  17
"theater"   26
"telecom"   12
"brewery"   8
"highways"  9
"cars"      7
"bank"      6
"software"  31
"electronics" 21

```

We modify our previous model as follows to work with this data file:

```
#include <iostream>
#include "xprb_cpp.h"

using namespace std;
using namespace ::dashoptimization;

#define DATAFILE "foliocpplp.dat"

#define NSHARES 10           // Number of shares
#define NRISK 5              // Number of high-risk shares
#define NNA 4                // Number of North-American shares

double RET[NSHARES];        // Estimated return in investment
char RISK[][100] = {"hardware", "theater", "telecom", "software",
                   "electronics"}; // High-risk values among shares
char NA[][100] = {"treasury", "hardware", "theater", "telecom"};
                                // Shares issued in N.-America

XPRBindexSet SHARES;        // Set of shares

XPRBprob p("FolioLP");      // Initialize a new problem in BCL

void readData(void)
{
    double value;
    int s;
    FILE *datafile;
    char name[100];

    SHARES=p.newIndexSet("Shares",NSHARES); // Create the `SHARES' index set

    // Read `RET' data from file
    datafile=fopen(DATAFILE,"r");
    for(s=0;s<NSHARES;s++)
    {
        XPRBreadlinecb(XPRB_FGETS, datafile, 200, "T g", name, &value);
        RET[SHARES+=name]=value;
    }
    fclose(datafile);

    SHARES.print();          // Print out the set contents
}

int main(int argc, char **argv)
{
    int s;
    XPRBexpr Risk,Na,Return,Cap;
    XPRBvar frac[NSHARES];   // Fraction of capital used per share

    // Read data from file
    readData();

    // Create the decision variables
    for(s=0;s<NSHARES;s++) frac[s] = p.newVar("frac");

    // Objective: total return
    for(s=0;s<NSHARES;s++) Return += RET[s]*frac[s];
    p.setObj(Return);        // Set the objective function

    // Limit the percentage of high-risk values
    for(s=0;s<NRISK;s++) Risk += frac[SHARES[RISK[s]]];
    p.newCtr("Risk", Risk <= 1.0/3);

    // Minimum amount of North-American values
    for(s=0;s<NNA;s++) Na += frac[SHARES[NA[s]]];
    p.newCtr("NA", Na >= 0.5);
}
```

```

// Spend all the capital
for(s=0;s<NSHARES;s++) Cap += frac[s];
p.newCtr("Cap", Cap == 1);

// Upper bounds on the investment per share
for(s=0;s<NSHARES;s++) frac[s].setUB(0.3);

// Solve the problem
p.setSense(XPRB_MAXIM);
p.lpOptimize("");

// Solution printing
cout << "Total return: " << p.getObjVal() << endl;
for(s=0;s<NSHARES;s++)
    cout << SHARES[s] << ": " << frac[s].getSol()*100 << "%" << endl;

return 0;
}

```

The arrays `RISK` and `NA` now store indices in the form of strings and we have added a new object, the *index set* `SHARES` that is defined while the return-on-investment values `RET` are read from the data file. In this example we have initialized the index set with exactly the right size. This is not really necessary since index sets may grow dynamically if more entries are added to them than the initially allocated space. The actual set size can be obtained with the method `getSize`.

For reading the data we use the function `XPRBreadlinecb`. It will skip comments preceded by `!` and any empty lines in the data file. The format string `"T g"` indicates that we wish to read a text string (surrounded by single or double quotes if it contains blanks) followed by a real number, the two separated by spaces (including tabulation). If the data file used another separator sign such as `,` then the format string could be changed accordingly (e.g. `"T,g"`).

In the model itself, the definition of the linear expressions `Risk` and `Na` has been adapted to the new indices.

Another modification concerns the solution printing: we print the name of every share, not simply its sequence number and hence the solution now gets displayed as follows:

```

Total return: 14.0667
treasury: 30%
hardware: 0%
theater: 20%
telecom: 0%
brewery: 6.66667%
highways: 30%
cars: 0%
bank: 0%
software: 13.3333%
electronics: 0%

```

10.4 Output functions and error handling

Most BCL modeling objects (`XPRBprob`, `XPRBvar`, `XPRBctr`, `XPRBsos`, and `XPRBindexSet`) have a method `print`. For variables, depending on where the method is invoked either their bounds or their solution value is printed: adding the line

```
frac[2].print();
```

before the call to the optimization will print the name of the variable and its bounds,

```
frac2: [0,0.3]
```

whereas after the problem has been solved its solution value gets displayed:

```
frac2: 0.2
```

Whenever BCL detects an error it stops the program execution with an error message so that it will usually not be necessary to test the return value of every operation. If a BCL program is embedded into some larger application, it may be helpful to use the *explicit initialization* to check early on that the software can be accessed correctly, for instance:

```
if(XPRB::init() != 0)
{
    cout << "Initialization failed." << endl;
    return 1;
}
```

The method `getLPStat` may be used to test the *LP problem status*. Only if the LP problem has been solved successfully BCL will return or print out meaningful solution values:

```
char *LPSTATUS[] = {"not loaded", "optimal", "infeasible",
                   "worse than cutoff", "unfinished", "unbounded",
                   "cutoff in dual", "unsolved", "nonconvex"};

cout << "Problem status: " << LPSTATUS[p.getLPStat()] << endl;
```

10.5 Exporting matrices

If the optimization process with Xpress Optimizer is started from within a BCL program (methods `lpOptimize` or `mipOptimize` of `XPRBprob`), then the problem matrix is loaded in memory into the solver without writing it out to a file (which would be expensive in terms of running time). However, in certain cases it may still be required to be able to produce a matrix. With Xpress, the user has the choice between two matrix formats: extended MPS and extended LP format, the latter being in general more easily human-readable since constraints are printed in algebraic form.

To export a matrix in MPS format add the following line to your BCL program, immediately before or instead of the optimization statement; this will create the file `Folio.mat` in your working directory:

```
p.exportProb(XPRB_MPS, "Folio");
```

For an LP format matrix use the following:

```
p.setSense(XPRB_MAXIM);
p.exportProb(XPRB_LP, "Folio");
```

The LP format contains information about the sense of optimization. Since the default is to minimize, for maximization we first need to reset the sense. The resulting matrix file will have the name `Folio.lp`.

CHAPTER 11

Mixed Integer Programming

This chapter extends the model developed in Chapter 10 to a Mixed Integer Programming (MIP) problem. It describes how to

- define different types of discrete variables,
- get the MIP solution status and understand the MIP optimization log produced by Xpress Optimizer.

Chapter 6 shows how to formulate and solve the same example with Mosel and in Chapter 16 the problem is input and solved directly with Xpress Optimizer.

11.1 Extended problem description

The investor is unwilling to have small share holdings. He looks at the following two possibilities to formulate this constraint:

1. Limiting the number of different shares taken into the portfolio.
2. If a share is bought, at least a minimum amount 10% of the budget is spent on the share.

We are going to deal with these two constraints in two separate models.

11.2 MIP model 1: limiting the number of different shares

To be able to count the number of different values we are investing in, we introduce a second set of variables buy_s in the LP model developed in Chapter 2. These variables are *indicator variables* or *binary variables*. A variable buy_s takes the value 1 if the share s is taken into the portfolio and 0 otherwise.

We introduce the following constraint to limit the total number of assets to a maximum of $MAXNUM$. It expresses the constraint that at most $MAXNUM$ of the variables buy_s may take the value 1 at the same time.

$$\sum_{s \in SHARES} buy_s \leq MAXNUM$$

We now still need to link the new binary variables buy_s with the variables $frac_s$, the quantity of every share selected into the portfolio. The relation that we wish to express is 'if a share is selected into the portfolio, then it is counted in the total number of values' or 'if $frac_s > 0$ then $buy_s = 1$ '. The following inequality formulates this implication:

$$\forall s \in SHARES : frac_s \leq buy_s$$

If, for some s , $frac_s$ is non-zero, then buy_s must be greater than 0 and hence 1. Conversely, if buy_s is at 0, then $frac_s$ is also 0, meaning that no fraction of share s is taken into the portfolio. Notice that these constraints do not prevent the possibility that buy_s is at 1 and $frac_s$ at 0. However, this does not matter in our case, since any solution in which this is the case is also valid with both variables, buy_s and $frac_s$, at 0.

11.2.1 Implementation with BCL

We extend the LP model developed in Chapter 10 with the new variables and constraints. The fact that the new variables are *binary variables* (i.e. they only take the values 0 and 1) is expressed through the type `XPRB_BV` at their creation.

Another common type of discrete variable is an *integer variable*, that is, a variable that can only take on integer values between given lower and upper bounds. These variables are defined in BCL with the type `XPRB_UI`. In the following section (MIP model 2) we shall see yet another example of discrete variables, namely semi-continuous variables.

```
#include <iostream>
#include "xprb_cpp.h"

using namespace std;
using namespace ::dashoptimization;

#define MAXNUM 4                // Max. number of shares to be selected

#define NSHARES 10             // Number of shares
#define NRISK 5                // Number of high-risk shares
#define NNA 4                  // Number of North-American shares

double RET[] = {5,17,26,12,8,9,7,6,31,21}; // Estimated return in investment
int RISK[] = {1,2,3,8,9};        // High-risk values among shares
int NA[] = {0,1,2,3};           // Shares issued in N.-America

int main(int argc, char **argv)
{
    int s;
    XPRBprob p("FolioMIP1");      // Initialize a new problem in BCL
    XPRBexpr Risk,Na,Return,Cap,Num;
    XPRBvar frac[NSHARES];        // Fraction of capital used per share
    XPRBvar buy[NSHARES];         // 1 if asset is in portfolio, 0 otherwise

    // Create the decision variables (including upper bounds for `frac')
    for(s=0;s<NSHARES;s++)
    {
        frac[s] = p.newVar("frac", XPRB_PL, 0, 0.3);
        buy[s] = p.newVar("buy", XPRB_BV);
    }

    // Objective: total return
    for(s=0;s<NSHARES;s++) Return += RET[s]*frac[s];
    p.setObj(Return);              // Set the objective function

    // Limit the percentage of high-risk values
    for(s=0;s<NRISK;s++) Risk += frac[RISK[s]];
    p.newCtr(Risk <= 1.0/3);

    // Minimum amount of North-American values
    for(s=0;s<NNA;s++) Na += frac[NA[s]];
    p.newCtr(Na >= 0.5);

    // Spend all the capital
    for(s=0;s<NSHARES;s++) Cap += frac[s];
    p.newCtr(Cap == 1);

    // Limit the total number of assets
```

```

    for(s=0;s<NSHARES;s++) Num += buy[s];
    p.newCtr(Num <= MAXNUM);

// Linking the variables
    for(s=0;s<NSHARES;s++) p.newCtr(frac[s] <= buy[s]);

// Solve the problem
    p.setSense(XPRB_MAXIM);
    p.mipOptimize("");

// Solution printing
    cout << "Total return: " << p.getObjVal() << endl;
    for(s=0;s<NSHARES;s++)
        cout << s << ": " << frac[s].getSol()*100 << "% (" << buy[s].getSol()
            << ")" << endl;

    return 0;
}

```

Besides the additional variables and constraints, the choice of optimization algorithm needs to be adapted to the problem type: we now wish to solve a MIP problem via Branch-and-Bound, and we therefore use the method `mipOptimize`.

Just as with the LP problem in the previous chapter, it is usually helpful to check the solution status before accessing the MIP solution—only if the MIP status is ‘unfinished (solution found)’ or ‘optimal’ will BCL print out a meaningful solution:

```

char *MIPSTATUS[] = {"not loaded", "not optimized", "LP optimized",
                    "unfinished (no solution)",
                    "unfinished (solution found)", "infeasible",
                    "optimal", "unbounded"};

cout << "Problem status: " << MIPSTATUS[p.getMIPStat()] << endl;

```

11.2.2 Analyzing the solution

As the result of the execution of our program we obtain the following output:

```

Reading Problem FolioMIP1
Problem Statistics
    14 (    514 spare) rows
    20 (     0 spare) structural columns
    49 (   5056 spare) non-zero elements
Global Statistics
    10 entities          0 sets          0 set members
Maximizing MILP FolioMIP1
Original problem has:
    14 rows              20 cols              49 elements          10 globals
Presolved problem has:
    13 rows              19 cols              46 elements          9 globals
LP relaxation tightened
Will try to keep branch and bound tree memory usage below 14.8Gb
Starting concurrent solve with dual

Concurrent-Solve,   0s
    Dual
    objective    dual inf
D  14.066667    .0000000
----- optimal -----
Concurrent statistics:
    Dual: 4 simplex iterations, 0.00s
Optimal solution found

    Its      Obj Value      S   Ninf  Nneg   Sum Dual Inf   Time
    4         14.066667    D     0     0       .000000       0
Dual solved problem

```

```

4 simplex iterations in 0s

Final objective           : 1.4066666666666667e+01
  Max primal violation      (abs / rel) : 5.551e-17 / 5.551e-17
  Max dual violation        (abs / rel) : 0.0 / 0.0
  Max complementarity viol. (abs / rel) : 0.0 / 0.0
All values within tolerances

Starting root cutting & heuristics

  Its Type   BestSoln   BestBound   Sols   Add   Del   Gap   GInf   Time
c          13.100000   14.066667    1
  1 K       13.100000   13.908571    1    1    0    5.81    0
  2 K       13.100000   13.580000    1   12    0    3.53    0
*** Search completed ***   Time: 0 Nodes: 1
Number of integer feasible solutions found is 1
Best integer solution found is 13.100000
Best bound is 13.100014
Uncrunching matrix
Problem status: optimal
Total return: 13.1
0: 20% (1)
1: 0% (0)
2: 30% (1)
3: 0% (0)
4: 20% (1)
5: 30% (1)
6: 0% (0)
7: 0% (0)
8: 0% (0)
9: 0% (0)

```

At the beginning we see the log of the execution of Xpress Optimizer: the problem statistics (we now have 14 constraints and 20 variables, out of which 10 are MIP variables, referred to as ‘entities’), the log of the execution of the LP algorithm (concurrently solving with primal and dual simplex on a multi-core processor), the log of the built-in MIP heuristics (a solution with the value 13.1 has been found) and the automated cut generation (a total of 13 cuts of type ‘K’ = knapsack have been generated). Since this problem is very small, it is solved by the MIP heuristics and the addition of cuts (additional constraints that cut off parts of the LP solution space, but no MIP solution) tightens the LP formulation in such a way that the solution to the LP relaxation becomes integer feasible. The Branch-and-Bound search therefore stops at the first node and no log of the Branch-and-Bound search gets displayed.

The output printed by our program tells us that the problem has been solved to optimality (i.e. the MIP search has been completed and at least one integer feasible solution has been found). The maximum return is now lower than in the original LP problem due to the additional constraint. As required, only four different shares are selected to form the portfolio.

11.3 MIP model 2: imposing a minimum investment in each share

To formulate the second MIP model, we start again with the LP model from Chapters 2 and 10. The new constraint we wish to formulate is ‘if a share is bought, at least a minimum amount 10% of the budget is spent on the share.’ Instead of simply constraining every variable $frac_s$ to take a value between 0 and 0.3, it now must either lie in the interval between 0.1 and 0.3 or take the value 0. This type of variable is known as *semi-continuous variable*. In the new model, we replace the bounds on the variables $frac_s$ by the following constraint:

$$\forall s \in SHARES : frac_s = 0 \text{ or } 0.1 \leq frac_s \leq 0.3$$

11.3.1 Implementation with BCL

The following program implements the MIP model 2. The semi-continuous variables are defined by the type `XPRB_SC`. By default, BCL assumes a continuous limit of 1, so we need to set this value to 0.1 with the method `setLim`.

A similar type is available for integer variables that take either the value 0 or an integer value between a given limit and their upper bound (so-called *semi-continuous integers*): `XPRB_SI`. A third composite type is a *partial integer* which takes integer values from its lower bound to a given limit value and is continuous beyond this value (marked by `XPRB_PI`).

```
#include <iostream>
#include "xprb_cpp.h"

using namespace std;
using namespace ::dashoptimization;

#define NSHARES 10           // Number of shares
#define NRISK 5             // Number of high-risk shares
#define NNA 4               // Number of North-American shares

double RET[] = {5,17,26,12,8,9,7,6,31,21}; // Estimated return in investment
int RISK[] = {1,2,3,8,9}; // High-risk values among shares
int NA[] = {0,1,2,3}; // Shares issued in N.-America

int main(int argc, char **argv)
{
    int s;
    XPRBprob p("FolioSC"); // Initialize a new problem in BCL
    XPRBexpr Risk, Na, Return, Cap;
    XPRBvar frac[NSHARES]; // Fraction of capital used per share

    // Create the decision variables
    for(s=0; s<NSHARES; s++)
    {
        frac[s] = p.newVar("frac", XPRB_SC, 0, 0.3);
        frac[s].setLim(0.1);
    }

    // Objective: total return
    for(s=0; s<NSHARES; s++) Return += RET[s]*frac[s];
    p.setObj(Return); // Set the objective function

    // Limit the percentage of high-risk values
    for(s=0; s<NRISK; s++) Risk += frac[RISK[s]];
    p.newCtr(Risk <= 1.0/3);

    // Minimum amount of North-American values
    for(s=0; s<NNA; s++) Na += frac[NA[s]];
    p.newCtr(Na >= 0.5);

    // Spend all the capital
    for(s=0; s<NSHARES; s++) Cap += frac[s];
    p.newCtr(Cap == 1);

    // Solve the problem
    p.setSense(XPRB_MAXIM);
    p.mipOptimize("");

    // Solution printing
    cout << "Total return: " << p.getObjVal() << endl;
    for(s=0; s<NSHARES; s++)
        cout << s << ": " << frac[s].getSol()*100 << "%" << endl;

    return 0;
}
```

When executing this program we obtain the following output (leaving out the part printed by the Optimizer):

```
Total return: 14.0333
0: 30%
1: 0%
2: 20%
3: 0%
4: 10%
5: 26.6667%
6: 0%
7: 0%
8: 13.3333%
9: 0%
```

Now five securities are chosen for the portfolio, each forming at least 10% and at most 30% of the total investment. Due to the additional constraint, the optimal MIP solution value is again lower than the initial LP solution value.

CHAPTER 12

Quadratic Programming

In this chapter we turn the LP problem from Chapter 10 into a Quadratic Programming (QP) problem, and the first MIP model from Chapter 11 into a Mixed Integer Quadratic Programming (MIQP) problem. The chapter shows how to

- define quadratic objective functions,
- incrementally define and solve problems.

Chapter 7 shows how to formulate and solve the same examples with Mosel and in Chapter 17 the QP problem is input and solved directly with Xpress Optimizer.

12.1 Problem description

The investor may also look at his portfolio selection problem from a different angle: instead of maximizing the estimated return and limiting the portion of high-risk investments he now wishes to minimize the risk whilst obtaining a certain target yield. He adopts the Markowitz idea of getting estimates of the variance/covariance matrix of estimated returns on the securities. (For example, hardware and software company worths tend to move together, but are oppositely correlated with the success of theatrical production, as people go to the theater more when they have become bored with playing with their new computers and computer games.) The return on theatrical productions are highly variable, whereas the treasury bill yield is certain.

Question 1: Which investment strategy should the investor adopt to minimize the variance subject to getting some specified minimum target yield?

Question 2: Which is the least variance investment strategy if the investor wants to choose at most four different securities (again subject to getting some specified minimum target yield)?

The first question leads us to a *Quadratic Programming* problem, that is, a Mathematical Programming problem with a quadratic objective function and linear constraints. The second question necessitates the introduction of discrete variables to count the number of securities, and so we obtain a *Mixed Integer Quadratic Programming* problem. The two cases will be discussed separately in the following two sections.

12.2 QP

To adapt the model developed in Chapter 2 to the new way of looking at the problem, we need to make the following changes:

- New objective function: mean variance instead of total return.

- The risk-related constraint disappears.
- Addition of a new constraint: target yield.

The new objective function is the mean variance of the portfolio, namely:

$$\sum_{s,t \in \text{SHARES}} \text{VAR}_{st} \cdot \text{frac}_s \cdot \text{frac}_t$$

where VAR_{st} is the variance/covariance matrix of all shares. This is a *quadratic objective function* (an objective function becomes quadratic either when a variable is squared, e.g., frac_1^2 , or when two variables are multiplied together, e.g., $\text{frac}_1 \cdot \text{frac}_2$).

The target yield constraint can be written as follows:

$$\sum_{s \in \text{SHARES}} \text{RET}_s \cdot \text{frac}_s \geq \text{TARGET}$$

The limit on the North-American shares as well as the requirement to spend all the money, and the upper bounds on the fraction invested into every share are retained. We therefore obtain the following complete mathematical model formulation:

$$\begin{aligned} &\text{minimize} \quad \sum_{s,t \in \text{SHARES}} \text{VAR}_{st} \cdot \text{frac}_s \cdot \text{frac}_t \\ &\quad \sum_{s \in \text{NA}} \text{frac}_s \geq 0.5 \\ &\quad \sum_{s \in \text{SHARES}} \text{frac}_s = 1 \\ &\quad \sum_{s \in \text{SHARES}} \text{RET}_s \cdot \text{frac}_s \geq \text{TARGET} \\ &\quad \forall s \in \text{SHARES} : 0 \leq \text{frac}_s \leq 0.3 \end{aligned}$$

12.2.1 Implementation with BCL

The estimated returns and the variance/covariance matrix are given in the data file `foliocppqp.dat`:

```
! trs  haw  thr  tel  brw  hgw  car  bnk  sof  elc
0.1   0   0   0   0   0   0   0   0   0 ! treasury
0   19  -2   4   1   1   1  0.5  10   5 ! hardware
0   -2  28   1   2   1   1   0  -2  -1 ! theater
0   4   1  22   0   1   2   0   3   4 ! telecom
0   1   2   0   4 -1.5  -2  -1   1   1 ! brewery
0   1   1   1 -1.5  3.5   2  0.5   1  1.5 ! highways
0   1   1   2   -2   2   5  0.5   1  2.5 ! cars
0  0.5   0   0  -1  0.5  0.5   1  0.5  0.5 ! bank
0  10  -2   3   1   1   1  0.5  25   8 ! software
0   5  -1   4   1  1.5  2.5  0.5   8  16 ! electronics
```

We may read this datafile with the function `XPRBreadarrlinecb`: all comments preceded by `!` and also empty lines are skipped. We read an entire line at once indicating the format of an entry ('g') and the separator (any number of spaces or tabulations).

For the definition of the objective function we now use a *quadratic expression* (equally represented by the class `XPRBexpr`). Since we now wish to minimize the problem, we use the default optimization sense setting and optimization as a continuous problem is again started with the method `lpOptimize` (with empty string argument indicating the default algorithm).

```

#include <iostream>
#include "xprb_cpp.h"

using namespace std;
using namespace ::dashoptimization;

#define DATAFILE "foliocppqp.dat"

#define TARGET 9 // Target yield
#define MAXNUM 4 // Max. number of different assets

#define NSHARES 10 // Number of shares
#define NNA 4 // Number of North-American shares

double RET[] = {5,17,26,12,8,9,7,6,31,21}; // Estimated return in investment
int NA[] = {0,1,2,3}; // Shares issued in N.-America
double VAR[NSHARES][NSHARES]; // Variance/covariance matrix of
// estimated returns

int main(int argc, char **argv)
{
    int s,t;
    XPRBprob p("FolioQP"); // Initialize a new problem in BCL
    XPRBexpr Na,Return,Cap,Num,Variance;
    XPRBvar frac[NSHARES]; // Fraction of capital used per share
    FILE *datafile;

    // Read `VAR' data from file
    datafile=fopen(DATAFILE,"r");
    for(s=0;s<NSHARES;s++)
        XPRBreadarrlinecb(XPRB_FGETS, datafile, 200, "g ", VAR[s], NSHARES);
    fclose(datafile);

    // Create the decision variables
    for(s=0;s<NSHARES;s++)
        frac[s] = p.newVar(XPRBnewname("frac(%d)",s+1), XPRB_PL, 0, 0.3);

    // Objective: mean variance
    for(s=0;s<NSHARES;s++)
        for(t=0;t<NSHARES;t++) Variance += VAR[s][t]*frac[s]*frac[t];
    p.setObj(Variance); // Set the objective function

    // Minimum amount of North-American values
    for(s=0;s<NNA;s++) Na += frac[NA[s]];
    p.newCtr(Na >= 0.5);

    // Spend all the capital
    for(s=0;s<NSHARES;s++) Cap += frac[s];
    p.newCtr(Cap == 1);

    // Target yield
    for(s=0;s<NSHARES;s++) Return += RET[s]*frac[s];
    p.newCtr(Return >= TARGET);

    // Solve the problem
    p.lpOptimize("");

    // Solution printing
    cout << "With a target of " << TARGET << " minimum variance is " <<
        p.getObjVal() << endl;
    for(s=0;s<NSHARES;s++)
        cout << s << ": " << frac[s].getSol()*100 << "%" << endl;

    return 0;
}

```

This program produces the following solution output with a dual-core processor (notice that the default algorithm for solving QP problems is Newton-Barrier, not the Simplex as in all previous examples):

```

Reading Problem FolioQP
Problem Statistics
    3 (      0 spare) rows
   10 (      0 spare) structural columns
   24 (      0 spare) non-zero elements
   76 quadratic elements
Global Statistics
    0 entities          0 sets          0 set members
Minimizing QP FolioQP
Original problem has:
    3 rows              10 cols          24 elements
    76 qobjelem
Presolved problem has:
    3 rows              10 cols          24 elements
    76 qobjelem
Barrier cache sizes : L1=32K L2=8192K
Using AVX support
Cores per CPU (CORESPERCPU): 8
Barrier starts, using up to 8 threads, 4 cores
Matrix ordering - Dense cols.:      9    NZ(L):      92    Flops:      584

Its   P.inf      D.inf      U.inf      Primal obj.      Dual obj.      Compl.
0  1.90e+001  1.85e+002  3.70e+000  8.7840000e+002 -1.1784000e+003  4.5e+003
1  1.69e-001  1.58e+000  3.29e-002  7.1810240e+000 -2.7042733e+002  3.1e+002
2  3.31e-003  1.48e-002  6.45e-004  5.1672666e+000 -1.2127681e+001  1.7e+001
3  6.12e-007  2.66e-015  2.78e-017  1.5558934e+000 -4.8803143e+000  6.4e+000
4  9.71e-017  1.39e-015  2.78e-017  7.2498306e-001  1.4062618e-001  5.8e-001
5  3.64e-017  1.01e-015  5.55e-017  5.6634270e-001  5.2690100e-001  3.9e-002
6  3.97e-017  7.15e-016  5.55e-017  5.5894833e-001  5.5591229e-001  3.0e-003
7  1.22e-016  1.33e-015  5.55e-017  5.5760205e-001  5.5726378e-001  3.4e-004
8  9.18e-017  1.68e-015  5.55e-017  5.5741308e-001  5.5738503e-001  2.8e-005
9  2.36e-016  3.29e-016  5.55e-017  5.5739403e-001  5.5739328e-001  7.5e-007
Barrier method finished in 0 seconds
Uncrunching matrix
Optimal solution found
Barrier solved problem
    9 barrier iterations in 0s

Final objective          : 5.573940299651456e-01
Max primal violation      (abs / rel) : 7.347e-17 / 7.347e-17
Max dual violation        (abs / rel) :      0.0 /      0.0
Max complementarity viol. (abs / rel) : 6.075e-07 / 1.012e-07
All values within tolerances
With a target of 9 minimum variance is 0.557394
0: 30%
1: 7.15401%
2: 7.38237%
3: 5.46362%
4: 12.6561%
5: 5.91283%
6: 0.333491%
7: 29.9979%
8: 1.0997%
9: 6.97039e-06%

```

12.3 MIQP

We now wish to express the fact that at most a given number *MAXNUM* of different assets may be selected into the portfolio, subject to all other constraints of the previous QP model. In Chapter 11 we have already seen how this can be done, namely by introducing an additional set of binary decision variables *buy_s* that are linked logically to the continuous variables:

$$\forall s \in \text{SHARES} : \text{frac}_s \leq \text{buy}_s$$

Through this relation, a variable *buy_s* will be at 1 if a fraction *frac_s* greater than 0 is selected into the

portfolio. If, however, buy_s equals 0, then $frac_s$ must also be 0.

To limit the number of different shares in the portfolio, we then define the following constraint:

$$\sum_{s \in SHARES} buy_s \leq MAXNUM$$

12.3.1 Implementation with BCL

We may modify the previous QP model or simply append the following lines to the program of the previous section, just after the solution printing: the problem is then solved once as a QP and once as a MIQP in a single program run.

```
XPRBvar buy[NSHARES];           // 1 if asset is in portfolio, 0 otherwise

// Create the decision variables
for(s=0;s<NSHARES;s++)
    buy[s] = p.newVar(XPRBnewname("buy(%d)",s+1), XPRB_BV);

// Limit the total number of assets
for(s=0;s<NSHARES;s++) Num += buy[s];
p.newCtr(Num <= MAXNUM);

// Linking the variables
for(s=0;s<NSHARES;s++) p.newCtr(frac[s] <= buy[s]);

// Solve the problem
p.mipOptimize("");

// Solution printing
cout << "With a target of " << TARGET << " and at most " << MAXNUM <<
      " assets, minimum variance is " << p.getObjVal() << endl;
for(s=0;s<NSHARES;s++)
    cout << s << ": " << frac[s].getSol()*100 << "% (" << buy[s].getSol()
        << ")" << endl;
```

When executing the MIQP model, we obtain the following solution output:

```
Reading Problem FolioQP
Problem Statistics
    14 (    514 spare) rows
    20 (     0 spare) structural columns
    54 (   5056 spare) non-zero elements
    76 quadratic elements
Global Statistics
    10 entities          0 sets          0 set members
Minimizing MIQP FolioQP
Original problem has:
    14 rows              20 cols          54 elements          10 globals
    76 qobjelem
Presolved problem has:
    14 rows              20 cols          54 elements          10 globals
    76 qobjelem
LP relaxation tightened
Will try to keep branch and bound tree memory usage below 14.8Gb
Crash basis containing 0 structural columns created
```

Its	Obj Value	S	Ninf	Nneg	Sum Inf	Time
0	.000000	p	1	4	.100000	0
8	.000000	p	0	0	.000000	0
8	4.609000	p	0	0	.000000	0

Its	Obj Value	S	Nsft	Nneg	Dual Inf	Time
27	.557393	QP	0	0	.000000	0

```
QP solution found
Optimal solution found
```

```

Primal solved problem
  27 simplex iterations in 0s

Final objective           : 5.573934108103899e-01
  Max primal violation    (abs / rel) : 1.804e-16 / 1.804e-16
  Max dual violation      (abs / rel) : 1.776e-15 / 1.776e-15
  Max complementarity viol. (abs / rel) : 2.670e-16 / 1.007e-16
All values within tolerances

```

Starting root cutting & heuristics

Its	Type	BestSoln	BestBound	Sols	Add	Del	Gap	GInf	Time
a		4.094715	.557393	1			86.39		
b		1.839000	.557393	2			69.69		
q		1.825619	.557393	3			69.47		
k		1.419003	.557393	4			60.72		
1	K	1.419003	.557393	4	3	0	60.72	0	
2	K	1.419003	.557393	4	9	2	60.72	0	
3	K	1.419003	.557393	4	7	6	60.72	0	
4	K	1.419003	.557393	4	5	6	60.72	0	
5	K	1.419003	.557393	4	11	5	60.72	0	
6	K	1.419003	.558122	4	8	10	60.67	0	
7	K	1.419003	.570670	4	11	9	59.78	0	
8	K	1.419003	.570670	4	5	12	59.78	0	
9	K	1.419003	.583638	4	5	3	58.87	0	
10	K	1.419003	.612496	4	4	0	56.84	0	
11	K	1.419003	.618043	4	6	5	56.45	0	
12	K	1.419003	.620360	4	8	4	56.28	0	
13	K	1.419003	.620360	4	0	6	56.28	0	

Heuristic search started

Heuristic search stopped

Cuts in the matrix : 14

Cut elements in the matrix : 138

Starting tree search.

Deterministic mode with up to 8 running threads and up to 16 tasks.

Node	BestSoln	BestBound	Sols Active	Depth	Gap	GInf	Time
a 9	1.248762	.919281	5 2	5	26.38		
*** Search completed ***					Time:	0 Nodes:	15
Number of integer feasible solutions found is 5							
Best integer solution found is 1.248762							
Best bound is 1.248752							
Uncrunching matrix							
With a target of 9 and at most 4 assets, minimum variance is 1.24876							
0: 30% (1)							
1: 20% (1)							
2: 0% (0)							
3: 0% (0)							
4: 23.8095% (1)							
5: 26.1905% (1)							
6: 0% (0)							
7: 0% (0)							
8: 0% (0)							
9: 0% (0)							

The log of the Branch-and-Bound search tells us this time that 5 integer feasible solutions have been found (all by the MIP heuristics) and a total of 15 nodes have been enumerated to complete the search. With the additional constraint on the number of different assets the minimum variance is more than twice as large as in the QP problem.

CHAPTER 13

Heuristics

In this chapter we show a simple binary variable fixing solution heuristic that involves a heuristic solution procedure interacting with Xpress Optimizer through

- parameter settings,
- saving and recovering bases, and
- modifications of variable bounds.

Chapter 8 shows how to implement the same heuristic with Mosel.

13.1 Binary variable fixing heuristic

The heuristic we wish to implement should perform the following steps:

1. Solve the LP relaxation and save the basis of the optimal solution
2. *Rounding heuristic*: Fix all variables ‘buy’ to 0 if they are close to 0, and to 1 if they have a relatively large value.
3. Solve the resulting MIP problem.
4. If an integer feasible solution was found, save the value of the best solution.
5. Restore the original problem by resetting all variables to their original bounds, and load the saved basis.
6. Solve the original MIP problem, using the heuristic solution as cutoff value.

Step 2: Since the fraction variables *frac* have an upper bound of 0.3, as a ‘relatively large value’ in this case we may choose 0.2. In other applications, for binary variables a more suitable choice may be $1 - \varepsilon$, where ε is a very small value such as 10^{-5} .

Step 6: Setting a *cutoff value* means that we only search for solutions that are better than this value. If the LP relaxation of a node is worse than this value it gets cut off, because this node and its descendants can only lead to integer feasible solutions that are even worse than the LP relaxation.

13.2 Implementation with BCL

For the implementation of the variable fixing solution heuristic we work with the MIP 1 model from Chapter 11. Through the definition of the heuristic in a separate function we only make minimal changes to the model itself: before solving our problem with the standard call to the method `mipOptimize` we execute our own solution heuristic and the solution printing also has been adapted.

```
#include <iostream>
#include "xprb_cpp.h"
#include "xprs.h"

using namespace std;
using namespace ::dashoptimization;

#define MAXNUM 4                // Max. number of shares to be selected

#define NSHARES 10             // Number of shares
#define NRISK 5                // Number of high-risk shares
#define NNA 4                  // Number of North-American shares

void solveHeur();

double RET[] = {5,17,26,12,8,9,7,6,31,21}; // Estimated return in investment
int RISK[] = {1,2,3,8,9};                // High-risk values among shares
int NA[] = {0,1,2,3};                    // Shares issued in N.-America

XPRBprob p("FolioMIPHeur");             // Initialize a new problem in BCL
XPRBvar frac[NSHARES];                   // Fraction of capital used per share
XPRBvar buy[NSHARES];                    // 1 if asset is in portfolio, 0 otherwise

int main(int argc, char **argv)
{
    int s;
    XPRBexpr Risk, Na, Return, Cap, Num;

    // Create the decision variables (including upper bounds for `frac')
    for(s=0;s<NSHARES;s++)
    {
        frac[s] = p.newVar("frac", XPRB_PL, 0, 0.3);
        buy[s] = p.newVar("buy", XPRB_BV);
    }

    // Objective: total return
    for(s=0;s<NSHARES;s++) Return += RET[s]*frac[s];
    p.setObj(Return);                    // Set the objective function

    // Limit the percentage of high-risk values
    for(s=0;s<NRISK;s++) Risk += frac[RISK[s]];
    p.newCtr(Risk <= 1.0/3);

    // Minimum amount of North-American values
    for(s=0;s<NNA;s++) Na += frac[NA[s]];
    p.newCtr(Na >= 0.5);

    // Spend all the capital
    for(s=0;s<NSHARES;s++) Cap += frac[s];
    p.newCtr(Cap == 1);

    // Limit the total number of assets
    for(s=0;s<NSHARES;s++) Num += buy[s];
    p.newCtr(Num <= MAXNUM);

    // Linking the variables
    for(s=0;s<NSHARES;s++) p.newCtr(frac[s] <= buy[s]);

    // Solve problem heuristically
    p.setSense(XPRB_MAXIM);
    solveHeur();
}
```

```

// Solve the problem
p.mipOptimize("");

// Solution printing
if(p.getMIPStat()==4 || p.getMIPStat()==6)
{
    cout << "Exact solution: Total return: " << p.getObjVal() << endl;
    for(s=0;s<NSHARES;s++)
        cout << s << ": " << frac[s].getSol()*100 << "%" << endl;
}
else
    cout << "Heuristic solution is optimal." << endl;

return 0;
}

void solveHeur()
{
    XPRBbasis basis;
    int s, ifgsol;
    double solval, bsol[NSHARES],TOL;

    XPRSsetintcontrol(p.getXPRSprob(), XPRS_CUTSTRATEGY, 0);
    // Disable automatic cuts
    XPRSsetintcontrol(p.getXPRSprob(), XPRS_HEURSTRATEGY, 0);
    // Disable MIP heuristics
    XPRSsetintcontrol(p.getXPRSprob(), XPRS_PRESOLVE, 0);
    // Switch presolve off
    XPRSgetdblcontrol(p.getXPRSprob(), XPRS_FEASTOL, &TOL);
    // Get feasibility tolerance

    p.mipOptimize("l"); // Solve the LP-relaxation
    basis=p.saveBasis(); // Save the current basis

    // Fix all variables 'buy' for which 'frac' is at 0 or at a relatively
    // large value
    for(s=0;s<NSHARES;s++)
    {
        bsol[s]=buy[s].getSol(); // Get the solution values of 'frac'
        if(bsol[s] < TOL) buy[s].setUB(0);
        else if(bsol[s] > 0.2-TOL) buy[s].setLB(1);
    }

    p.mipOptimize("c"); // Solve the MIP-problem
    ifgsol=0;
    if(p.getMIPStat()==4 || p.getMIPStat()==6)
    {
        // If an integer feas. solution was found
        ifgsol=1;
        solval=p.getObjVal(); // Get the value of the best solution
        cout << "Heuristic solution: Total return: " << p.getObjVal() << endl;
        for(s=0;s<NSHARES;s++)
            cout << s << ": " << frac[s].getSol()*100 << "%" << endl;
    }

    // Reset variables to their original bounds
    for(s=0;s<NSHARES;s++)
        if((bsol[s] < TOL) || (bsol[s] > 0.2-TOL))
        {
            buy[s].setLB(0);
            buy[s].setUB(1);
        }

    p.loadBasis(basis); /* Load the saved basis: bound changes are
                        immediately passed on from BCL to the
                        Optimizer if the problem has not been modified
                        in any other way, so that there is no need to
                        reload the matrix */

    basis.reset(); // No need to store the saved basis any longer
    if(ifgsol==1)

```

```

XPRSsetdblcontrol(p.getXPRSProb(), XPRS_MIPABSCUTOFF, solval+TOL);
// Set the cutoff to the best known solution
}

```

The implementation of the heuristic certainly requires some explanations.

In this example for the first time we use the *direct access to Xpress Optimizer*. To do so, we need to include the Optimizer header file `xprs.h`. The Optimizer functions are applied to the problem representation (of type `XPRSProb`) held by the Optimizer which can be retrieved with the method `getXPRSProb` of the BCL problem. For more detail on how to use the BCL and Optimizer libraries in combination the reader is referred to the 'BCL Reference Manual'. The complete documentation of all Optimizer functions and parameters is provided in the 'Optimizer Reference Manual'.

Parameters: The solution heuristic starts with parameter settings for the Xpress Optimizer. Switching off the automated cut generation (parameter `XPRS_CUTSTRATEGY`) and the MIP heuristics (parameter `XPRS_HEURSTRATEGY`) is optional, whereas it is required in our case to disable the presolve mechanism (a treatment of the matrix that tries to reduce its size and improve its numerical properties, set with parameter `XPRS_PRESOLVE`), because we interact with the problem in the Optimizer in the course of its solution and this is only possible correctly if the matrix has not been modified by the Optimizer.

In addition to the parameter settings we also retrieve the feasibility tolerance used by Xpress Optimizer: the Optimizer works with tolerance values for integer feasibility and solution feasibility that are typically of the order of 10^{-6} by default. When evaluating a solution, for instance by performing comparisons, it is important to take into account these tolerances.

The fine tuning of output printing mentioned in Chapter 10 can be obtained by setting the parameters `XPRS_LPLOG` and `XPRS_MIPLOG` (both to be set with function `XPRSsetintcontrol`).

Optimization calls: We use the optimization method `mipOptimize` with the argument "1", indicating that we only want to solve the top node LP relaxation (and not yet the entire MIP problem). To continue with MIP solving from the point where we have stopped the algorithm we use the argument "c".

Saving and loading bases: To speed up the solution process, we save (in memory) the current basis of the Simplex algorithm after solving the initial LP relaxation, before making any changes to the problem. This basis is loaded again at the end, once we have restored the original problem. The MIP solution algorithm then does not have to re-solve the LP problem from scratch, it resumes the state where it was 'interrupted' by our heuristic.

Bound changes: When a problem has already been loaded into the Optimizer (e.g. after executing an optimization statement or following an explicit call to method `loadMat`) bound changes via `setLB` and `setUB` are passed on directly to the Optimizer. Any other changes (addition or deletion of constraints or variables) always lead to a complete reloading of the problem.

The program produces the following output. As can be seen, when solving the original problem for the second time the Simplex algorithm performs 0 iterations because it has been started with the basis of the optimal solution saved previously.

```

Reading Problem FolioMIPHeur
Problem Statistics
      14 (      0 spare) rows
      20 (      0 spare) structural columns
      49 (      0 spare) non-zero elements
Global Statistics
      10 entities          0 sets          0 set members
Maximizing MILP FolioMIPHeur
Original problem has:
      14 rows          20 cols          49 elements          10 globals
Will try to keep branch and bound tree memory usage below 14.8Gb
Starting concurrent solve with dual

Concurrent-Solve,    0s
      Dual
      objective    dual inf

```

```

D 14.066667 .0000000
----- optimal -----
Concurrent statistics:
    Dual: 5 simplex iterations, 0.00s
Optimal solution found

    Its      Obj Value      S   Ninf  Nneg   Sum Dual Inf   Time
      5      14.066667      D     0     0       .000000      0
Dual solved problem
    5 simplex iterations in 0s

Final objective           : 1.406666666666667e+01
Max primal violation      (abs / rel) :      0.0 /      0.0
Max dual violation        (abs / rel) :      0.0 /      0.0
Max complementarity viol. (abs / rel) :      0.0 /      0.0
All values within tolerances
*** Search unfinished ***   Time:      0 Nodes:      0
Number of integer feasible solutions found is 0
Best bound is 14.066667

Starting root cutting & heuristics

    Its Type   BestSoln   BestBound   Sols   Add   Del   Gap   GInf   Time
a      13.100000   14.066667     1
Heuristic search started
Heuristic search stopped
*** Search completed ***   Time:      0 Nodes:      1
Number of integer feasible solutions found is 1
Best integer solution found is 13.100000
Best bound is 13.100014
Heuristic solution: Total return: 13.1
0: 20%
1: 0%
2: 30%
3: 0%
4: 20%
5: 30%
6: 0%
7: 0%
8: 0%
9: 0%
Maximizing MILP FolioMIPHeur
Original problem has:
    14 rows      20 cols      49 elements      10 globals
Will try to keep branch and bound tree memory usage below 14.8Gb
Starting concurrent solve with dual

Concurrent-Solve, 0s
    Dual
    objective dual inf
D 14.066667 .0000000
----- optimal -----
Concurrent statistics:
    Dual: 0 simplex iterations, 0.00s
Optimal solution found

    Its      Obj Value      S   Ninf  Nneg   Sum Dual Inf   Time
      0      14.066667      D     0     0       .000000      0
Dual solved problem
    0 simplex iterations in 0s

Final objective           : 1.406666666666667e+01
Max primal violation      (abs / rel) :      0.0 /      0.0
Max dual violation        (abs / rel) :      0.0 /      0.0
Max complementarity viol. (abs / rel) :      0.0 /      0.0
All values within tolerances

Starting root cutting & heuristics

```

```

    Its Type      BestSoln    BestBound    Sols    Add    Del    Gap    GInf    Time
Heuristic search started
Heuristic search stopped

Starting tree search.
Deterministic mode with up to 8 running threads and up to 16 tasks.

    Node      BestSoln    BestBound    Sols Active    Depth    Gap    GInf    Time
*** Search completed ***      Time:      0 Nodes:      5
Problem is integer infeasible
Number of integer feasible solutions found is 0
Best bound is      13.100001
Heuristic solution is optimal.
```

Observe that the heuristic found a solution of 13.1, and that the MIP optimizer without the heuristic could not find a better solution (hence the infeasible message). The heuristic solution is therefore optimal.

III. Getting started with the Optimizer

CHAPTER 14

Matrix input

In this chapter we show how to

- initialize Xpress Optimizer,
- load matrices in MPS or LP format into the Optimizer,
- solve a problem, and
- write out the solution to a file.

14.1 Matrix files

With Xpress, the user has the choice between two matrix formats: extended MPS and extended LP format, the latter being in general more easily human-readable since constraints are printed in algebraic form. Such matrices may be written out by Xpress Optimizer, but more likely they will have been generated by some other tool.

If the optimization process with Xpress Optimizer is started from within a Mosel or BCL program, then the problem matrix is loaded in memory into the solver without writing it out to a file (which would be expensive in terms of running time). However, both tools may also be used to produce matrix files (see Chapter 9 for matrix generation with Mosel and Chapter 10 for BCL).

14.2 Implementation

To load a matrix into Xpress Optimizer we need to perform the following steps:

1. Initialize Xpress Optimizer.
2. Create a new problem.
3. Read the matrix file.

The following C program `folioinput.c` (similar interfaces exist for Java and C#) shows how to load a matrix file, solve it, and write out the results. For clarity's sake we have omitted all error checking in this program. In general it is recommended to test the return value of the initialization function and also whether the problem has been created and read correctly.

To use Xpress Optimizer, we need to include the header file `xprs.h`.

```

#include <stdio.h>
#include "xprs.h"

int main(int argc, char **argv)
{
    XPRSPprob prob;

    XPRSinit(NULL);                /* Initialize Xpress Optimizer */
    XPRScreateprob(&prob);         /* Create a new problem */

    XPRSreadprob(prob, "Folio", ""); /* Read the problem matrix */

    XPRSchgobjsense(prob, XPRS_OBJ_MAXIMIZE); /* Set sense to maximization */
    XPRSloptimize(prob, "");        /* Solve the problem */

    XPRSwriteprtsol(prob, "Folio.prt", ""); /* Write results to `Folio.prt' */

    XPRSdestroyprob(prob);         /* Delete the problem */
    XPRSfree();                   /* Terminate Xpress */

    return 0;
}

```

14.3 Compilation and program execution

If you have followed the standard installation procedure of Xpress Optimizer, you may compile this file with the following command under Windows:

```
cl /MD /I%XPRESSDIR%\include %XPRESSDIR%\lib\xprs.lib folioinput.c
```

For Linux or Solaris use

```
cc -D_REENTRANT -I${XPRESSDIR}/include -L${XPRESSDIR}/lib folioinput.c -o folioinput -lxprs
```

For other systems please refer to the example makefile provided with the corresponding distribution.

If we run this program with the matrix `Folio.mat` produced by BCL for the LP example problem of Chapter 2, then we obtain an output file `Folio.prt` with the following contents:

```

Problem Statistics
Matrix FolioLP
Objective *OBJ*

RHS *RHS*
Problem has      4 rows and      10 structural columns

Solution Statistics
Maximization performed
Optimal solution found after      5 iterations
Objective function value is      14.066659

Rows Section
  Number  Row  At  Value  Slack Value  Dual Value  RHS
N      1  *OBJ*  BS   14.066659  -14.066659   .000000   .000000
E      2   Cap  EQ    1.000000   .000000    8.000000  1.000000
G      3   NA   LL    .500000   .000000   -5.000000  .500000
L      4  Risk  UL    .333333   .000000   23.000000  .333333

Columns Section
  Number  Column  At  Value  Input Cost  Reduced Cost
C      5  frac   UL    .300000    5.000000    2.000000
C      6  frac_1  LL    .000000   17.000000   -9.000000
C      7  frac_2  BS    .200000   26.000000    .000000
C      8  frac_3  LL    .000000   12.000000  -14.000000

```

C	9	frac_4	BS	.066667	8.000000	.000000
C	10	frac_5	UL	.300000	9.000000	1.000000
C	11	frac_6	LL	.000000	7.000000	-1.000000
C	12	frac_7	LL	.000000	6.000000	-2.000000
C	13	frac_8	BS	.133333	31.000000	.000000
C	14	frac_9	LL	.000000	21.000000	-10.000000

The upper half contains some statistics concerning the problem size and the solution algorithm: the optimal LP solution found has a value of 14.066659. The `Rows` Section gives detailed solution information for the constraints in the problem. The solution values for the decision variables are located in the column labeled `Value` of the `Columns` Section.

CHAPTER 15

Inputting and solving a Linear Programming problem

In this chapter we take the example formulated in Chapter 2 and show how to input and solve this problem with Xpress Optimizer. In detail, we shall discuss the following topics:

- transformation of an LP model into matrix format,
- LP problem input with Xpress Optimizer,
- solving and solution output.

Chapter 3 shows how to formulate and solve this example with Mosel and Chapter 10 does the same for BCL.

15.1 Matrix representation

As a first step in the transformation of the mathematical problem into the form required by the LP problem input function of Xpress Optimizer we write the problem in the form of a table where the columns represent the decision variables and the rows are the constraints. All non-zero coefficients are then entered into this table, resulting in the problem *matrix*, completed by the operators and the constant terms (the latter are usually referred to as the *right hand side*, RHS, values).

Table 15.1: LP matrix

		<i>frac</i> ₁	<i>frac</i> ₂	<i>frac</i> ₃	<i>frac</i> ₄	<i>frac</i> ₅	<i>frac</i> ₆	<i>frac</i> ₇	<i>frac</i> ₈	<i>frac</i> ₉	<i>frac</i> ₁₀		
		0	1	2	3	4	5	6	7	8	9	Oper.	RHS
Risk	0		²	⁵	⁸					¹⁵	¹⁷	≤	1/3
MinNA	1	⁰	³	⁶	⁹							≥	0.5
Allfrac	2	¹	⁴	⁷	¹⁰	¹¹	¹²	¹³	¹⁴	¹⁶	¹⁸	=	1
		↑	↑										
		<i>rowidx</i>	<i>matval</i>										
<i>colbeg</i>		0	2	5	8	11	12	13	14	15	17	19	
<i>nelem</i>		2	3	3	3	1	1	1	1	2	2		

The matrix specified to Xpress Optimizer does not consist of the full *number_of_rows* x *number_of_columns* table; instead, only the list of non-zero coefficients is given and an indication where they are located. The superscripts in the table above indicate the order of the matrix entries in this list. The coefficient values will be stored in the array *matval*, the corresponding row numbers in the array *rowidx*, the values of the first few entries of these arrays are printed in italics to highlight them (see the code example in the following section for the full definition of these arrays). To complete

this information, the array `colbeg` contains the index of the first entry per column and the array `nelem` the number of entries per column.

15.2 Implementation with Xpress Optimizer

The following C program `foliolp.c` shows how to input and solve this LP problem with Xpress Optimizer. We have also added printing of the solution. Before trying to access the solution, the LP problem status is checked (see the 'Optimizer Reference Manual' for further explanation). To use Xpress Optimizer, we need to include the header file `xprs.h`.

To load a problem into Xpress Optimizer we need to perform the following steps:

1. Initialize Xpress Optimizer.
2. Create a new problem.
3. Load the matrix data.

```
#include <stdio.h>
#include <stdlib.h>
#include "xprs.h"

int main(int argc, char **argv)
{
    XPRSProb prob;
    int s, status;
    double objval, *sol;

    /* Problem parameters */
    int ncol = 10;
    int nrow = 3;

    /* Row data */
    char rowtype[] = { 'L','G','E' };
    double rhs[] = {1.0/3, 0.5, 1};

    /* Column data */
    double obj[] = { 5, 17, 26, 12, 8, 9, 7, 6, 31, 21 };
    double lb[] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
    double ub[] = { 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3 };

    /* Matrix coefficient data */
    int colbeg[] = { 0, 2, 5, 8, 11, 12, 13, 14, 15, 17, 19 };
    int rowidx[] = { 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 2, 2, 2, 0, 2, 0, 2 };
    double matval[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 };

    XPRSinit(NULL); /* Initialize Xpress Optimizer */
    XPRScreateprob(&prob); /* Create a new problem */

    /* Load the problem matrix */
    XPRSloadlp(prob, "FolioLP", ncol, nrow, rowtype, rhs, NULL,
               obj, colbeg, NULL, rowidx, matval, lb, ub);

    XPRSchgobjsense(prob, XPRS_OBJ_MAXIMIZE); /* Set sense to maximization */
    XPRSloptimize(prob, ""); /* Solve the problem */

    XPRSgetintattrib(prob, XPRS_LPSTATUS, &status); /* Get LP sol. status */

    if(status == XPRS_LP_OPTIMAL)
    {
        XPRSgetdblattrtrib(prob, XPRS_LPOBJVAL, &objval); /* Get objective value */
        printf("Total return: %g\n", objval);

        sol = (double *)malloc(ncol*sizeof(double));
    }
}
```

```

    XPRSgetlpcol(sol, NULL, NULL, NULL); /* Get primal solution */
    for(s=0;s<ncol;s++) printf("%d: %g%%\n", s+1, sol[s]*100);
}

XPRSdestroyprob(prob); /* Delete the problem */
XPRSfree(); /* Terminate Xpress */

return 0;
}

```

Instead of defining `colbeg` with one extra entry for the last+1 column we may give the numbers of coefficients per column in the array `nelem`:

```

/* Matrix coefficient data */
int colbeg[] = {0, 2, 5, 8, 11,12,13,14,15, 17};
int nelem[] = {2, 3, 3, 3, 1, 1, 1, 1, 2, 2};
int rowidx[] = {1,2,0,1,2,0,1,2,0,1,2, 2, 2, 2, 2, 0,2, 0,2};
double matval[] = {1,1,1,1,1,1,1,1,1,1,1, 1, 1, 1, 1, 1,1, 1,1};

...
/* Load the problem matrix */
XPRSloadlp(prob, "FolioLP", ncol, nrow, rowtype, rhs, NULL,
           obj, colbeg, nelem, rowidx, matval, lb, ub);

```

The seventh argument of the function `XPRSloadlp` remains empty for our problem since it is reserved for range information on constraints.

The second argument of the optimization function `XPRSloptimize` indicates the algorithm to be used: an empty string stands for the default LP algorithm. After solving the problem we check whether the LP has been solved and if so, we retrieve the objective function value and the primal solution for the decision variables.

15.3 Compilation and program execution

If you have followed the standard installation procedure of Xpress Optimizer, you may compile this file with the following command under Windows:

```
cl /MD /I%XPRESSDIR%\include %XPRESSDIR%\lib\xprs.lib foliolp.c
```

For Linux or Solaris use

```
cc -D_REENTRANT -I${XPRESSDIR}/include -L${XPRESSDIR}/lib foliolp.c -o foliolp -lxprs
```

For other systems please refer to the example makefile provided with the corresponding distribution.

Running the resulting program will generate the following output:

```

Total return: 14.0667
1: 30%
2: 0%
3: 20%
4: 0%
5: 6.66667%
6: 30%
7: 0%
8: 0%
9: 13.3333%
10: 0%

```

Under Unix this is preceded by the log of Xpress Optimizer:

```

Reading Problem FolioLP
Problem Statistics
      3 (      0 spare) rows
     10 (      0 spare) structural columns
     19 (      0 spare) non-zero elements
Global Statistics
      0 entities          0 sets          0 set members
Maximizing LP FolioLP
Original problem has:
      3 rows          10 cols          19 elements
Presolved problem has:
      3 rows          10 cols          19 elements

      Its      Obj Value      S      Ninf      Nneg      Sum Inf      Time
      0         42.600000      D        2        0         .000000         0
      5         14.066667      D        0        0         .000000         0
Uncrunching matrix
      5         14.066667      D        0        0         .000000         0
Optimal solution found

```

Windows users can retrieve the Optimizer log by redirecting it to a file. Add the following line to your program immediately after the problem creation:

```
XPRSsetlogfile(prob, "logfile.txt");
```

The Optimizer log displays the size of the matrix, 3 rows (i.e. constraints) and 10 columns (i.e. decision variables), and the log of the LP solution algorithm (here: 'D' for dual Simplex). The output produced by our program tells us that the maximum return of 14.0667 is obtained with a portfolio consisting of shares 1, 3, 5, 6, and 9. 30% of the total amount are spent in shares 1 and 6 each, 20% in 3, 13.3333% in 9 and 6.6667% in 5. It is easily verified that all constraints are indeed satisfied: we have 50% of North-American shares (1 and 3) and 33.33% of high-risk shares (3 and 9).

CHAPTER 16

Mixed Integer Programming

This chapter extends the LP problem from Chapter 2 to a Mixed Integer Programming (MIP) problem. It describes how to

- transform a MIP model into matrix format,
- input MIP problems with different types of discrete variables into Xpress Optimizer,
- solve MIP problems and output the solution.

Chapter 6 shows how to formulate and solve this example with Mosel and in Chapter 11 the same is done with BCL.

16.1 Extended problem description

The investor is unwilling to have small share holdings. He looks at the following two possibilities to formulate this constraint:

1. Limiting the number of different shares taken into the portfolio to 4.
2. If a share is bought, at least a minimum amount 10% of the budget is spent on the share.

We are going to deal with these two constraints in two separate models.

16.2 MIP model 1: limiting the number of different shares

To be able to count the number of different values we are investing in, we introduce a second set of variables buy_s in the LP model developed in Chapter 2. These variables are *indicator variables* or *binary variables*. A variable buy_s takes the value 1 if the share s is taken into the portfolio and 0 otherwise.

We introduce the following constraint to limit the total number of assets to a maximum of 4 different ones. It expresses the constraint that at most 4 of the variables buy_s may take the value 1 at the same time.

$$\sum_{s \in SHARES} buy_s \leq 4$$

We now still need to link the new binary variables buy_s with the variables $frac_s$, the quantity of every share selected into the portfolio. The relation that we wish to express is 'if a share is selected into the portfolio, then it is counted in the total number of values' or 'if $frac_s > 0$ then $buy_s = 1$ '. The following inequality formulates this implication:

$$\forall s \in SHARES : frac_s \leq buy_s$$

If, for some s , $frac_s$ is non-zero, then buy_s must be greater than 0 and hence 1. Conversely, if buy_s is at 0, then $frac_s$ is also 0, meaning that no fraction of share s is taken into the portfolio. Notice that these constraints do not prevent the possibility that buy_s is at 1 and $frac_s$ at 0. However, this does not matter in our case, since any solution in which this is the case is also valid with both variables, buy_s and $frac_s$, at 0.

16.2.1 Matrix representation

The mathematical model can be transformed into the following table. Compared to the LP matrix of the previous chapter, we now have ten additional columns for the variables buy_s and ten additional rows for the constraints linking the two types of variables. Notice that we have to transform the linking constraints so that all terms involving decision variables are on the left hand side of the operator sign.

Table 16.1: MIP matrix

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	Op.	RHS
Risk	0	<i>1</i> ³	<i>1</i> ⁷	<i>1</i> ¹¹					<i>1</i> ²³	<i>1</i> ²⁶											≤	1/3
MinNA	1	<i>1</i> ⁰	<i>1</i> ⁴	<i>1</i> ⁸	<i>1</i> ¹²																≥	0.5
Allfrac	2	<i>1</i> ¹	<i>1</i> ⁵	<i>1</i> ⁹	<i>1</i> ¹³	<i>1</i> ¹⁵	<i>1</i> ¹⁷	<i>1</i> ¹⁹	<i>1</i> ²¹	<i>1</i> ²⁴	<i>1</i> ²⁷										=	1
Maxnum	3										<i>1</i> ²⁹	<i>1</i> ³¹	<i>1</i> ³³	<i>1</i> ³⁵	<i>1</i> ³⁷	<i>1</i> ³⁹	<i>1</i> ⁴¹	<i>1</i> ⁴³	<i>1</i> ⁴⁵	<i>1</i> ⁴⁷	≤	4
Linking	4	<i>1</i> ²									<i>-1</i> ³⁰										≤	0
	5	<i>1</i> ⁶										<i>-1</i> ³²									≤	0
	6		<i>1</i> ¹⁰										<i>-1</i> ³⁴								≤	0
	7			<i>1</i> ¹⁴										<i>-1</i> ³⁶							≤	0
	8				<i>1</i> ¹⁶										<i>-1</i> ³⁸						≤	0
	9					<i>1</i> ¹⁸										<i>-1</i> ⁴⁰					≤	0
	10						<i>1</i> ²⁰										<i>-1</i> ⁴²				≤	0
	11							<i>1</i> ²²										<i>-1</i> ⁴⁴			≤	0
	12								<i>1</i> ²⁵										<i>-1</i> ⁴⁶		≤	0
	13									<i>1</i> ²⁸										<i>-1</i> ⁴⁸	≤	0
		↑	↑																			
		rowidx matval																				
colbeg	0	3	7	11	15	17	19	21	23	26	29	31	33	35	37	39	41	43	45	47	49	

The superscripts for the matrix coefficients indicate again the order of the entries in the arrays `rowidx` and `matval`, the first three entries of which are highlighted (printed in italics).

16.2.2 Implementation with Xpress Optimizer

In addition to the structures related to the matrix coefficients that are in common with LP problems, we now also need to specify the MIP-specific information, namely the types of the MIP variables (here all marked 'B' for *binary variable*) in the array `miptype` and the corresponding column indices in the array `mipcol`.

Another common type of discrete variable is an *integer variable*, that is, a variable that can only take on integer values between given lower and upper bounds. These variables are defined with the type 'I'. In the following section (MIP model 2) we shall see yet another example of discrete variables, namely semi-continuous variables.

```
#include <stdio.h>
#include <stdlib.h>
#include "xprs.h"

int main(int argc, char **argv)
{
```

```

XPRSprob prob;
int s, status;
double objval, *sol;

/* Problem parameters */
int ncol = 20;
int nrow = 14;
int nmip = 10;

/* Row data */
char rowtype[] = { 'L', 'G', 'E', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L' };
double rhs[] = { 1.0/3, 0.5, 1, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

/* Column data */
double obj[] = { 5, 17, 26, 12, 8, 9, 7, 6, 31, 21, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
double lb[] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
double ub[] = { 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 };

/* Matrix coefficient data */
int colbeg[] = { 0, 3, 7, 11, 15, 17, 19, 21, 23, 26, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49 };
int rowidx[] = { 1, 2, 4, 0, 1, 2, 5, 0, 1, 2, 6, 0, 1, 2, 7, 2, 8, 2, 9, 2, 10, 2, 11, 0, 2, 12, 0, 2,
13, 3, 4, 3, 5, 3, 6, 3, 7, 3, 8, 3, 9, 3, 10, 3, 11, 3, 12, 3, 13 };
double matval[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1 };

/* MIP problem data */
char miptype[] = { 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B' };
int mipcol[] = { 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 };

XPRSinit(NULL); /* Initialize Xpress Optimizer */
XPRScreateprob(&prob); /* Create a new problem */

/* Load the problem matrix */
XPRSloadglobal(prob, "FolioMIP1", ncol, nrow, rowtype, rhs, NULL,
obj, colbeg, NULL, rowidx, matval, lb, ub,
nmip, 0, miptype, mipcol, NULL, NULL, NULL, NULL, NULL);

XPRSschgobjsense(prob, XPRS_OBJ_MAXIMIZE); /* Set sense to maximization */
XPRSmipoptimize(prob, ""); /* Solve the problem */

XPRSgetintattrib(prob, XPRS_MIPSTATUS, &status); /* Get MIP sol. status */

if((status == XPRS_MIP_OPTIMAL) || (status == XPRS_MIP_SOLUTION))
{
XPRSgetdblattrib(prob, XPRS_MIPOBJVAL, &objval); /* Get objective value */
printf("Total return: %g\n", objval);

sol = (double *)malloc(ncol*sizeof(double));
XPRSgetmipsol(prob, sol, NULL); /* Get primal solution */
for(s=0; s<ncol/2; s++)
printf("%d: %g%g (%g)\n", s, sol[s]*100, sol[ncol/2+s]);
}

XPRSdestroyprob(prob); /* Delete the problem */
XPRSfree(); /* Terminate Xpress */

return 0;
}

```

To load the problem into Xpress Optimizer we now use the function `XPRSloadglobal`. The first 14 arguments of this function are the same as for `XPRSloadlp`. The use of the 19th argument will be discussed in the next section; the remaining four arguments are related to the definition of SOS (Special Ordered Sets)—the value 0 for the 16th argument indicates that there are none in our problem.

In this program, not only the function for loading the problem but also those for solving and solution access have been adapted to the problem type: we now solve a MIP problem via a Branch-and-Bound search (the second argument " " of the optimization function `XPRSmipoptimize` stands for 'default

MIP algorithm'). We then retrieve the MIP solution status and if an integer feasible solution has been found we print out the objective value of the best integer solution found and the corresponding solution values of the decision variables.

Running this program produces the following solution output. The maximum return is now lower than in the original LP problem due to the additional constraint. As required, only four different shares are selected to form the portfolio:

```
Total return: 13.1
0: 20% (1)
1: 0% (0)
2: 30% (1)
3: 0% (0)
4: 20% (1)
5: 30% (1)
6: 0% (0)
7: 0% (0)
8: 0% (0)
9: 0% (0)
```

16.3 MIP model 2: imposing a minimum investment in each share

To formulate the second MIP model, we start again with the LP model from Chapters 2 and 15. The new constraint we wish to formulate is 'if a share is bought, at least a minimum amount 10% of the budget is spent on the share.' Instead of simply constraining every variable $frac_s$ to take a value between 0 and 0.3, it now must either lie in the interval between 0.1 and 0.3 or take the value 0. This type of variable is known as *semi-continuous variable*. In the new model, we replace the bounds on the variables $frac_s$ by the following constraint:

$$\forall s \in SHARES : frac_s = 0 \text{ or } 0.1 \leq frac_s \leq 0.3$$

16.3.1 Matrix representation

This problem has the same matrix as the LP problem in the previous chapter, and so we do not repeat it here. The only changes are in the specification of the MIP-related column data.

16.3.2 Implementation with Xpress Optimizer

The following program `foliomip2.c` loads the MIP model 2 into the Optimizer. We have the same matrix data as for the LP problem in the previous chapter, but the variables are now semi-continuous, defined by the type marker 'S'. By default, Xpress Optimizer assumes a continuous limit of 1, we therefore specify the value 0.1 in the array `sclim`. Please note in this context that limits for semi-continuous and semi-continuous integer variables given in the array `sclim` are overwritten by the value in the array `lb` if the latter is different from 0.

Other available composite variable types are *semi-continuous integer variables* that take either the value 0 or an integer value between a given limit and their upper bound (marked by 'R') and *partial integers* that take integer values from their lower bound to a given limit value and are continuous beyond this value (marked by 'P').

```
#include <stdio.h>
#include <stdlib.h>
#include "xprs.h"

int main(int argc, char **argv)
{
    XPRSProb prob;
    int s, status;
    double objval, *sol;
```

```

/* Problem parameters */
int ncol = 10;
int nrow = 3;
int nmip = 10;

/* Row data */
char rowtype[] = { 'L','G','E'};
double rhs[] = {1.0/3,0.5, 1};

/* Column data */
double obj[] = { 5, 17, 26, 12, 8, 9, 7, 6, 31, 21};
double lb[] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
double ub[] = {0.3,0.3,0.3,0.3,0.3,0.3,0.3,0.3,0.3,0.3};

/* Matrix coefficient data */
int colbeg[] = {0, 2, 5, 8, 11,12,13,14,15, 17, 19};
int rowidx[] = {1,2,0,1,2,0,1,2,0,1,2, 2, 2, 2, 2, 0,2, 0,2};
double matval[] = {1,1,1,1,1,1,1,1,1,1, 1, 1, 1, 1, 1,1, 1,1};

/* MIP problem data */
char miptype[] = {'S','S','S','S','S','S','S','S','S','S'};
int mipcol[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
double sclim[] = {0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1};

XPRSinit(NULL); /* Initialize Xpress Optimizer */
XPRScreateprob(&prob); /* Create a new problem */

/* Load the problem matrix */
XPRSloadglobal(prob, "FolioSC", ncol, nrow, rowtype, rhs, NULL,
obj, colbeg, NULL, rowidx, matval, lb, ub,
nmip, 0, miptype, mipcol, sclim, NULL, NULL, NULL, NULL);

XPRSchgobjsense(prob, XPRS_OBJ_MAXIMIZE); /* Set sense to maximization */
XPRSmipoptimize(prob, ""); /* Solve the problem */

XPRSgetintattrib(prob, XPRS_MIPSTATUS, &status); /* Get MIP sol. status */

if((status == XPRS_MIP_OPTIMAL) || (status == XPRS_MIP_SOLUTION))
{
XPRSgetdblattrib(prob, XPRS_MIPOBJVAL, &objval); /* Get objective value */
printf("Total return: %g\n", objval);

sol = (double *)malloc(ncol*sizeof(double));
XPRSgetmipsol(prob, sol, NULL); /* Get primal solution */
for(s=0;s<ncol;s++) printf("%d: %g%%\n", s, sol[s]*100);
}

XPRSdestroyprob(prob); /* Delete the problem */
XPRSfree(); /* Terminate Xpress */

return 0;
}

```

When executing this program we obtain the following output:

```

Total return: 14.0333
0: 30%
1: 0%
2: 20%
3: 0%
4: 10%
5: 26.6667%
6: 0%
7: 0%
8: 13.3333%
9: 0%

```

Now five securities are chosen for the portfolio, each forming at least 10% and at most 30% of the total investment. Due to the additional constraint, the optimal MIP solution value is again lower than the initial LP solution value.

CHAPTER 17

Quadratic Programming

In this chapter we turn the LP problem from Chapter 15 into a Quadratic Programming (QP) problem, showing how to

- transform a QP model into matrix format,
- input and solve QP problems with Xpress Optimizer.

Chapter 7 shows how to formulate and solve this example with Mosel and in Chapter 12 the same is done with BCL.

17.1 Problem description

The investor may also look at his portfolio selection problem from a different angle: instead of maximizing the estimated return and limiting the portion of high-risk investments he now wishes to minimize the risk whilst obtaining a certain target yield. He adopts the Markowitz idea of getting estimates of the variance/covariance matrix of estimated returns on the securities. Which investment strategy should the investor adopt to minimize the variance subject to getting a minimum target yield of 9?

17.2 QP model

To adapt the model developed in Chapter 2 to the new way of looking at the problem, we need to make the following changes:

- New objective function: mean variance instead of total return.
- The risk-related constraint disappears.
- Addition of a new constraint: target yield.

The new objective function is the mean variance of the portfolio, namely:

$$\sum_{s,t \in \text{SHARES}} \text{VAR}_{st} \cdot \text{frac}_s \cdot \text{frac}_t$$

where VAR_{st} is the variance/covariance matrix of all shares. This is a *quadratic objective function* (an objective function becomes quadratic either when a variable is squared, e.g., frac_1^2 , or when two variables are multiplied together, e.g., $\text{frac}_1 \cdot \text{frac}_2$).

The target yield constraint can be written as follows:

$$\sum_{s \in \text{SHARES}} RET_s \cdot frac_s \geq 9$$

The limit on the North-American shares as well as the requirement to spend all the money, and the upper bounds on the fraction invested into every share are retained. We therefore obtain the following complete mathematical model formulation:

$$\begin{aligned} & \text{minimize} \quad \sum_{s,t \in \text{SHARES}} VAR_{st} \cdot frac_s \cdot frac_t \\ & \sum_{s \in \text{NA}} frac_s \geq 0.5 \\ & \sum_{s \in \text{SHARES}} frac_s = 1 \\ & \sum_{s \in \text{SHARES}} RET_s \cdot frac_s \geq 9 \\ & \forall s \in \text{SHARES} : 0 \leq frac_s \leq 0.3 \end{aligned}$$

17.3 Matrix representation

For the problem input into Xpress Optimizer, the mathematical model is transformed into the following constraint matrix (Table 17.1).

Table 17.1: QP matrix

		<i>frac₁</i>	<i>frac₂</i>	<i>frac₃</i>	<i>frac₄</i>	<i>frac₅</i>	<i>frac₆</i>	<i>frac₇</i>	<i>frac₈</i>	<i>frac₉</i>	<i>frac₁₀</i>		
		0	1	2	3	4	5	6	7	8	9	Oper.	RHS
MinNA	0	<i>1⁰</i>	<i>1³</i>	<i>1⁶</i>	<i>1⁹</i>							≥	0.5
Allfrac	1	<i>1¹</i>	<i>1⁴</i>	<i>1⁷</i>	<i>1¹⁰</i>	<i>1¹²</i>	<i>1¹⁴</i>	<i>1¹⁶</i>	<i>1¹⁸</i>	<i>1²⁰</i>	<i>1²²</i>	=	1
Yield	2	<i>5²</i>	<i>17⁵</i>	<i>26⁸</i>	<i>12¹¹</i>	<i>8¹³</i>	<i>9¹⁵</i>	<i>7¹⁷</i>	<i>6¹⁹</i>	<i>31²¹</i>	<i>21²³</i>	≥	9
		↑	↑										
		<i>rowidx</i>	<i>matval</i>										
<i>colbeg</i>		0	3	6	9	12	14	16	18	20	22	24	

As in the previous chapters, the superscripts for the matrix coefficients indicate the order of the entries in the arrays *rowidx* and *matval*, the first three entries of which are highlighted (printed in italics).

The coefficients of the quadratic objective function are given by the following variance/covariance matrix (Table 17.2).

17.4 Implementation with Xpress Optimizer

The following program `folioqp.c` loads the QP problem into Xpress Optimizer and solves it. Notice that the quadratic part of the objective function must be specified in triangular form, that is, either the lower or the upper triangle of the original matrix. Here we have chosen the upper triangle, which means that instead of $4 \cdot frac_2 \cdot frac_4 + 4 \cdot frac_4 \cdot frac_2$ we only specify the sum of these terms, $8 \cdot frac_2 \cdot frac_4$. Due to the input conventions of the Optimizer the values of the main diagonal also need to be multiplied with 2. As with the matrix coefficients, only quadratic terms with non-zero coefficients are specified to the Optimizer (hence the spaces in the array definitions below).

Table 17.2: Variance/covariance matrix

	<i>frac</i> ₁	<i>frac</i> ₂	<i>frac</i> ₃	<i>frac</i> ₄	<i>frac</i> ₅	<i>frac</i> ₆	<i>frac</i> ₇	<i>frac</i> ₈	<i>frac</i> ₉	<i>frac</i> ₁₀
	0	1	2	3	4	5	6	7	8	9
<i>frac</i> ₁	0	0.1								
<i>frac</i> ₂	1	19	-2	4	1	1	1	0.5	10	5
<i>frac</i> ₃	2		-2	28	1	2	1		-2	-1
<i>frac</i> ₄	3		4	1	22		1	2	3	4
<i>frac</i> ₅	4		1	2		4	-1.5	-2	-1	1
<i>frac</i> ₆	5		1	1	1	-1.5	3.5	2	0.5	1
<i>frac</i> ₇	6		1	1	2	-2	2	5	0.5	1
<i>frac</i> ₈	7		0.5			-1	0.5	0.5	1	0.5
<i>frac</i> ₉	8		10	-2	3	1	1	1	0.5	25
<i>frac</i> ₁₀	9		5	-1	4	1	1.5	2.5	0.5	8

```

#include <stdio.h>
#include <stdlib.h>
#include "xprs.h"

int main(int argc, char **argv)
{
    XPRSProb prob;
    int s, status;
    double objval, *sol;

    /* Problem parameters */
    int ncol = 10;
    int nrow = 3;
    int nqt = 43;

    /* Row data */
    char rowtype[] = {'G','E','G'};
    double rhs[] = {0.5, 1, 9};

    /* Column data */
    double obj[] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    double lb[] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    double ub[] = {0.3,0.3,0.3,0.3,0.3,0.3,0.3,0.3,0.3,0.3};

    /* Matrix coefficient data */
    int colbeg[] = {0, 3, 6, 9, 12, 14, 16, 18, 20, 22, 24};
    int rowidx[] = {0,1,2,0,1, 2,0,1, 2,0,1, 2,1,2,1,2,1,2,1, 2,1,2};
    double matval[] = {1,1,5,1,1,17,1,1,26,1,1,12,1,8,1,9,1,7,1,6,1,31,1,21};

    /* QP problem data */
    int qcoll[] = {0,
        1,1,1,1,1,1,1,1,1,
        2,2,2,2,2, 2,2,
        3, 3,3, 3,3,
        4,4,4,4,4,4,4,
        5,5,5,5,5,5,
        6,6,6,6,
        7,7,7,
        8,8,
        9};

    int qcol2[] = {0,
        1,2,3,4,5,6,7,8,9,
        2,3,4,5,6, 8,9,
        3, 5,6, 8,9,
        4,5,6,7,8,9,
        5,6,7,8,9,
        6,7,8,9,
        7,8,9,
        8,9,
        9};

```

```

double qval[] = {0.1,
                 19,-2, 4,1,   1, 1,0.5, 10,  5,
                 28, 1,2,   1, 1,   -2, -1,
                 22,   1, 2,     3,  4,
                 4,-1.5,-2, -1,  1,  1,
                 3.5, 2,0.5,  1,1.5,
                 5,0.5,  1,2.5,
                 1,0.5,0.5,
                 25,  8,
                 16};

for(s=0;s<nqt;s++) qval[s]*=2;

XPRSinit(NULL); /* Initialize Xpress Optimizer */
XPRScreateprob(&prob); /* Create a new problem */
/* Load the problem matrix */
XPRSloadqp(prob, "FolioQP", ncol, nrow, rowtype, rhs, NULL,
            obj, colbeg, NULL, rowidx, matval, lb, ub,
            nqt, qcol1, qcol2, qval);

XPRSchgobjsense(prob, XPRS_OBJ_MINIMIZE); /* Set sense to maximization */
XPRSlpoptimize(prob, ""); /* Solve the problem */

XPRSgetintattrib(prob, XPRS_LPSTATUS, &status); /* Get solution status */

if(status == XPRS_LP_OPTIMAL)
{
    XPRSgetdblattrib(prob, XPRS_LPOBJVAL, &objval); /* Get objective value */
    printf("Minimum variance: %g\n", objval);

    sol = (double *)malloc(ncol*sizeof(double));
    XPRSgetlpval(prob, sol, NULL, NULL, NULL); /* Get primal solution */
    for(s=0;s<ncol;s++) printf("%d: %g%%\n", s, sol[s]*100);
}

XPRSdestroyprob(prob); /* Delete the problem */
XPRSfree(); /* Terminate Xpress */

return 0;
}

```

A QP problem is loaded into the Optimizer with the function `XPRSloadqp`. This function takes the same arguments as function `XPRSloadlp` with four additional arguments at the end for the quadratic part of the objective function: the number of quadratic terms, `nqt`, the column numbers of the variables in every quadratic term (`qcol1` and `qcol2`) and their coefficient, `qval`.

If we wish to load a Mixed Integer Quadratic Programming (MIQP) problem, then we need to use the function `XPRSloadqglobal` that takes the same arguments as `XPRSloadqp` plus the nine arguments for MIP problems introduced with function `XPRSloadglobal` in the previous chapter.

As opposed to the previous examples we now minimize the objective function. Notice that for solving and solution access we use the same functions as for LP problems. When solving MIQP problems, correspondingly we need to use the MIP solving and solution functions presented in Chapter 16.

Executing this program produces the following output:

```

Minimum variance: 0.557394
0: 30%
1: 7.15401%
2: 7.38237%
3: 5.46362%
4: 12.6561%
5: 5.91283%
6: 0.333491%
7: 29.9979%
8: 1.0997%
9: 6.97039e-06%

```

All but the last share are selected into the portfolio (the value printed for 9 is so close to 0 that Xpress Optimizer interprets it as 0 with its default tolerance settings).

Appendix

APPENDIX A

Going further

A.1 Installation, licensing, and trouble shooting

Detailed information on how to install Xpress is provided with every distribution (see subdirectory `docs`). The 'Xpress Installation Guide' is also accessible online from the [Xpress online documentation website](#). To obtain a license key please contact your nearest Xpress sales office.

Should you encounter any problems with installing the software or setting up the license, please contact Xpress Support:

support@fico.com

(Please include 'Xpress', in the subject line.)

You may also consult the Xpress FAQs and discussion groups on the FICO website:

<https://community.fico.com/community/fico-optimization-community/xpressdiscuss>

A.2 User guides, reference manuals, and other publications

Under the following address you may find the complete online documentation for Xpress:

<http://www.fico.com/fico-xpress-optimization/docs/latest>

The whitepapers and all of the documents referred to in the following sections are included in PDF format in the Xpress distribution, for an overview direct your webbrowser to the subdirectory `docs` of your Xpress installation directory.

A useful online resource is the searchable database of Xpress examples that you can reach following this link:

<http://examples.xpress.fico.com/example.pl>

A.2.1 Modeling

The book 'Applications of Optimization with Xpress-MP' (Dash Optimization, 2002) shows how to formulate and solve a large number of application problems with Xpress:

http://examples.xpress.fico.com/example.pl#mosel_app

A.2.2 Mosel

For a more in-depth introduction to working with Mosel, we suggest to read the 'Mosel User Guide'.

The 'Mosel Language Reference Manual' provides a complete documentation of the Mosel language, also including the features defined by the modules of the Mosel distribution (*mmxprs*, *mmodbc*, *mmsvg*, etc.).

The whitepaper 'Using ODBC and other database interfaces with Mosel' discusses examples of data exchange with spreadsheets and databases. The topic of I/O drivers is covered more generally by the whitepaper 'Generalized file handling in Mosel'.

The Mosel Compiler and Mosel Run-time libraries are documented in the 'Mosel Libraries Reference Manual'.

To learn how to implement your own Mosel modules, please refer to the 'Mosel NI User Guide'.

The Mosel Native Interface is documented in the 'Mosel NI Reference Manual'.

A.2.3 BCL

The 'BCL Reference Manual' contains further examples of the use of BCL and a complete documentation of all C library functions and C++ classes.

For Java, a separate '[BCL Java on-line documentation](#)' is available, and similarly the '[BCL .NET on-line documentation](#)' for C#.

A.2.4 Optimizer

All functions of the Optimizer library are documented in the 'Xpress Optimizer Reference Manual'. In this manual you also find exhaustive lists of all problem attributes and control parameters that may be used with Xpress Optimizer.

An introduction to fully automated tuning of the optimization algorithms for your problems is provided in the section 'Using the Tuner' of the 'Xpress Optimizer Reference Manual'.

A.2.5 Other solvers and solution methods

The FICO Xpress Optimization suite comprises some other products that have not been mentioned in this manual since they are typically reserved for more advanced uses. Each of these components comes with its own documentation. However, reading the introduction to Mosel in the first part of this manual is recommended to all first-time users who wish to employ these other products as Mosel modules.

Xpress NonLinear comprises a set of solvers for solving general Non-linear Programming (NLP) problems to (local) optimality, including the *Successive Linear Programming (SLP)* solver *Xpress SLP* and also the NLP solver *Knitro*. *Xpress NonLinear* is provided in the form of a Mosel module, *mmxnlp*, *Xpress SLP* can also be used as a C library or in console mode. For further detail see the 'Xpress NonLinear Reference Manual'.

Constraint Programming (CP) is an approach to problem solving that has been particularly successful for dealing with nonlinear constraint relations over discrete variables, such as frequently occur in scheduling and planning applications. The *Xpress Kalis* software, an interface to the Artelys Kalis constraint solver is provided in the form of a Mosel module, *kalis*, which defines aggregate modeling objects specialized for scheduling and planning problems. For a description of this software see the 'Xpress Kalis Reference Manual' and 'Xpress Kalis User Guide'.

APPENDIX B

Glossary

Basis: when solving an LP problem with the Simplex algorithm, the basis provides the complete information about which variables and constraints are active in a given solution. It can therefore be used to save and quickly restore the status of the solution algorithm at a given point.

Binary model file (BIM file): a compiled version of the `.mos` model file that is portable across all platforms for which Mosel is available. It does *not* include any data read from external files. These must still be provided in separate files, thus making it possible to run the same BIM file with different data sets.

Bound: equality or inequality constraint on a single decision variable. When working with Xpress Optimizer through Mosel or BCL, bounds may be changed without having to reload the problem.

Branch-and-Bound: solution method for MIP problems consisting of an enumeration of the feasible values of the discrete variables (*branching*) coupled with LP techniques (providing *bounding* information). Typically represented in the form of a *Branch-and-Bound tree* where every *node* stands for the solution of an LP problem, and the connections between these nodes are the bound changes or added constraints. Such enumerative methods may lead to a computational explosion, even for relatively small problem instances, so that it is not always realistic to solve MIP problems to optimality.

Branch-and-Cut: solution algorithm for MIP problems similar to Branch-and-Bound. At some or all nodes of the search tree violated *cuts* are added to the problem to tighten the LP relaxation.

Builder Component Library (BCL): model builder library for developing a model directly in a programming language. BCL allows users to formulate their models with objects (decision variables, constraints, index sets) similar to those of a dedicated modeling language.

Constraint: relation between decision variables. Constraint types include *equality constraints* (operator = in Mosel and == in BCL C++), *inequality constraints* (operators >= and <= in Mosel and BCL C++), and *integrality conditions*. *Bounds* are special cases of inequality or equality constraints.

Cut: also called *valid inequality*; additional constraint in MIP problems that is not required to characterize the set of integer solutions, but must be satisfied by all feasible solutions. Cuts *tighten* the LP relaxation by drawing the LP solution space closer to the convex hull of the MIP solution space.

Decision variable (or *variable* for short): unknown that needs to be assigned a value by the solution algorithm. The basic variable type is a *continuous variable* (a variable taking values from a continuous domain between a given lower and upper bound). *Discrete variable* types include *binary variables* (also called *indicator variables*; variables that may only take the values 0 or 1); *integer variables* (taking values in an integer range between given lower and upper bounds); *semi-continuous variables* (either 0 or values from a continuous interval between a given limit and upper bound); *semi-continuous integer variables* (either 0 or integer values between a given limit and upper bound); *partial integer variables* (integer-valued from the lower bound to a given limit and continuous beyond this limit value)

Declaration: the declaration of an object states its form and type and usually precedes the *definition* of its contents. With Mosel, the declaration of basic types and linear constraints is optional, but decision variables must always be declared; subroutines must be declared if they are used in a model prior to

their definition.

Dense array: arrays in Mosel can be either dense or sparse. By default arrays in Mosel are dense, that is, every possible index tuple is associated to a cell in the array. A dense array is *fixed* if its index sets are constant or have been *finalized* to make them *static*; *non-fixed* arrays can increase dynamically with the contents assigned to them, however it is generally more efficient to finalize their index sets as early as possible, this also allows Mosel to check for 'out of range' errors that cannot be detected if the sets are dynamic.

Dynamic set, dynamic array: sets and arrays in Mosel can be marked explicitly as *dynamic*. Dynamic sets cannot be finalized to make them static; dynamic arrays and hashmap arrays are two forms of (sparse) arrays for storing and efficiently enumerating sparse data tables. Non-fixed dense arrays are sometimes referred to as implicitly dynamic arrays, particularly for earlier versions of Mosel.

Heuristic: algorithm for finding feasible solution(s) to a problem. Some heuristics guarantee a bound on the solution quality but usually no proof of optimality is possible.

Index set: set used for indexing an array. Using *string indices* may help to make the output produced by Mosel or BCL more easily understandable.

Interactive Visual Environment (IVE): development environment for Mosel that provides, amongst many other tools, graphical displays of solution information.

Linear Programming (LP) problem: a Mathematical Programming problem where all constraints and the objective function are linear expressions of the decision variables, and the variables have continuous domains—i.e., they can take on any, usually non-negative, real values. A well-understood case for which efficient algorithms (Simplex, interior point) are known.

Loop: Loops group actions that need to be repeated a certain number of times, either for all values of some index or counter (*forall* in Mosel) or depending on whether a condition is fulfilled or not (*while*, *repeat until* in Mosel).

LP relaxation: in a MIP problem, the LP relaxation is obtained by dropping the integrality conditions on the decision variables.

Mathematical Programming problem (or *problem* for short): a set of decision variables, constraints over these variables and an objective function to be maximized or minimized.

Matrix: the matrix representation of Mathematical Programming problems with linear constraints is a table where the *columns* are the variables and the *rows* represent the constraints. The table entries are the coefficients of the variables in the constraints, usually stored in *sparse format*, that is, only the non-zero entries are given.

Mixed Integer Programming (MIP) problem: a Mathematical Programming problem where constraints and objective function are linear just as in LP and variables may have either discrete or continuous domains. To solve this type of problems, LP techniques are coupled with an enumeration (known as *Branch-and-Bound*).

Modeling language: a high-level language (such as the Mosel language) that allows the user to state Mathematical Programming problems in a form close to their algebraic representation. Carries out automatically the transformation to the representation required by the solver(s).

Model: algebraic representation of a problem; also employed to denote the implementation with a modeling tool such as Mosel or BCL.

Module: also called *dynamic shared object (DSO)*; dynamic library written in the C programming language that observes the conventions set out by the Mosel Native Interface. Modules enable users to extend the Mosel language with new features (e.g. to implement problem-specific data handling, or connections to external solvers or solution algorithms). Modules of the Xpress distribution include access to Xpress Solver (Xpress Optimizer for LP, MIP, QP, Xpress NonLinear for NLP, and Xpress Kalis for CP), data handling facilities (e.g. via ODBC) and access to system functions, graphing capabilities, distributed and remote computing functionality via the *Mosel Distributed Framework*, and also

interfaces to statistics packages such as R or Matlab.

Mosel: modeling and solving environment comprising the *Mosel language* (a modeling and programming language), the *Mosel libraries* (for embedding Mosel models into applications), and the *Mosel Native Interface* (opening up the Mosel language to external additions in the form of *modules*).

Mosel Native Interface (NI): a subroutine library giving access to Mosel models during their execution; defines also the conventions to be observed by Mosel modules. The NI enables users to extend the Mosel language with new features.

Newton-Barrier algorithm: also *interior point algorithm*; solution algorithm for LP and QP problems that proceeds from some initial interior point in the set of feasible solutions towards an optimal solution without touching the border of the feasible set.

Non-linear Programming (NLP) problem: a Mathematical Programming problem with non-linear constraints or objective function. Frequently heuristic or approximation methods are employed to find good (locally optimal) solutions. A method provided by Xpress for solving problems of this type is *Successive Linear Programming (SLP)*.

Objective function: an expression of decision variables to be minimized or maximized (in this manual only linear or quadratic expressions are considered).

Optimization: finding a feasible solution to a problem that minimizes or maximizes a given objective function.

Optimizer: the Xpress solver for LP, MIP, and QP. Available in the form of a library or a standalone program.

Overloading: subroutines that are defined in several versions for different types or numbers of arguments; operators that are defined for different operand types or combinations of operand types.

Parameter: depending on the context this term has several slightly different meanings: the settings of *model parameters* (in Mosel) may be changed at run-time, for instance to define different input data sets; *problem parameters* (in the Optimizer usually called *problem attributes*) provide access to information about a problem (e.g. solution status) and are typically read-only; *algorithm control parameters* are used to control algorithmic settings (choice of the solution algorithm, tolerances, etc.).

Problem instance: a Mathematical Programming problem complete with a specific data set.

Quadratic Programming (QP) problem: differs from LP problems in that there are quadratic terms in the objective function (the constraints remain linear). The decision variables may be continuous or discrete, in the latter case we speak of *Mixed Integer Quadratic Programming (MIQP)*.

Range set: (in Mosel) a set of consecutive integers.

Right hand side (RHS): constant term of a (linear) constraint; a standard format (used, for example, in the matrix representation) is to write all terms involving decision variables on the left of the operator sign and the constant term on its right side.

Selection statement: statement to express a selection between different actions to be taken in a program. In Mosel these are *if/then/elif/then/else/end-if* and *case*.

Simplex algorithm: solution algorithm for LP problems. The idea of the Simplex algorithm is moving from vertex to vertex of the polytope ('simplex') that represents the set of feasible solutions for an LP problem, to improve the objective function value.

Solution: this term may be used with two different meanings: it may denote an assignment of values to all decision variables that satisfies all constraints (*feasible solution*). In optimization problems where the best possible solution is sought—i.e., a solution minimizing or maximizing a given objective function, the term solution usually is equivalent to *optimal solution*.

Solver: software used to solve (usually optimize) a problem. With Xpress we use Xpress Solver (comprising Xpress Optimizer, Xpress NonLinear and Xpress Kalis).

Sparse array: arrays in Mosel can be either dense or sparse. Sparse arrays are created empty and may grow on demand as their entries are created or get assigned values. Mosel has two types of sparse arrays: *dynamic* arrays require less memory and are faster for linear enumeration, *hashmap* arrays are faster for random access.

Status information: Mosel, BCL, and the Optimizer define different *parameters* providing status information, such as the LP or MIP status that tell the user among others whether the problem has been solved correctly and a solution is available.

Subroutine: substructures allowing programs to be broken down into smaller subtasks that are easier to understand and to work with. In Mosel, subroutines may be employed in the form of procedures or functions. *Procedures* are called as a program statement, they have no return value, *functions* must be called in an expression that uses their return value.

Successive Linear Programming (SLP): method for solving NLP problems via a sequence of LP problems.

APPENDIX C

Contacting FICO

FICO provides clients with support and services for all our products. Refer to the following sections for more information.

Product support

FICO offers technical support and services ranging from self-help tools to direct assistance with a FICO technical support engineer. Support is available to all clients who have purchased a FICO product and have an active support or maintenance contract. You can find support contact information and a link to the Customer Self Service Portal (online support) on the Product Support home page (www.fico.com/en/product-support).

The FICO Customer Self Service Portal is a secure web portal that is available 24 hours a day, 7 days a week from the Product Support home page. The portal allows you to open, review, update, and close cases, as well as find solutions to common problems in the FICO Knowledge Base.

Please include 'Xpress' in the subject line of your [support queries](#).

Product education

FICO Product Education is the principal provider of product training for our clients and partners. Product Education offers instructor-led classroom courses, web-based training, seminars, and training tools for both new user enablement and ongoing performance support. For additional information, visit the Product Education homepage at www.fico.com/en/product-training or email producteducation@fico.com.

Product documentation

FICO continually looks for new ways to improve and enhance the value of the products and services we provide. If you have comments or suggestions regarding how we can improve this documentation, let us know by sending your suggestions to techpubs@fico.com.

Please include your contact information (name, company, email address, and optionally, your phone number) so we may reach you if we have questions.

Sales and maintenance

If you need information on other Xpress Optimization products, or you need to discuss maintenance contracts or other sales-related items, contact FICO by:

- Phone: +1 (408) 535-1500 or +44 207 940 8718
- Web: www.fico.com/optimization and use the available contact forms

Related services

Strategy Consulting: Included in your contract with FICO may be a specified amount of consulting time to assist you in using FICO Optimization Modeler to meet your business needs. Additional consulting time can be arranged by contract.

Conferences and Seminars: FICO offers conferences and seminars on our products and services. For announcements concerning these events, go to www.fico.com or contact your FICO account representative.

FICO Community

The FICO Community is a great resource to find the experts and information you need to collaborate, support your business, and solve common business challenges. You can get informal technical support, build relationships with local and remote professionals, and improve your business practices. For additional information, visit the FICO Community (community.fico.com/welcome).

About FICO

FICO (NYSE:FICO) powers decisions that help people and businesses around the world prosper. Founded in 1956 and based in Silicon Valley, the company is a pioneer in the use of predictive analytics and data science to improve operational decisions. FICO holds more than 165 US and foreign patents on technologies that increase profitability, customer satisfaction, and growth for businesses in financial services, telecommunications, health care, retail, and many other industries. Using FICO solutions, businesses in more than 100 countries do everything from protecting 2.6 billion payment cards from fraud, to helping people get credit, to ensuring that millions of airplanes and rental cars are in the right place at the right time. Learn more at www.fico.com.

Index

Symbols

`;`, 16
`==`, 68

A

annotation, 57
argument
 subroutine, 46
array
 definition, 15
 dense, 114
 dynamic, 23, 114
 sparse, 116

B

basis, 113
 loading, 47, 88
 saving, 47, 88
BCL, 6, 113
 initialization, 67
 model, 67
BIM file, 25, 50, 113
binary variable, 32, 73, 99, 113
bound, 9, 68, 113
 modification, 48, 88
Branch-and-Bound, 4, 35, 75, 101, 113
Branch-and-Bound tree, 35, 113
Branch-and-Cut, 113

C

case, 115
code completion, 17
column, 95, 114
comments, 16, 22, 71, 80
companion file, 63
comparison tolerance, 48
compilation, 68, 93, 97
constraint, 113
 definition, 15, 67
 equality, 9, 68, 113
 inequality, 9, 113
 linear, 15, 67
 name, 28
Constraint Programming, 112
continuous variable, 113
control parameter, 115
CP, see Constraint Programming
cut, 35, 76, 113
cut generation, 35, 47, 76, 88
cutoff value, 44, 85

D

data

 input from file, 22, 30, 40, 71, 80
data file, 22, 30, 40, 69, 80
data handling, 5
debugger window, 13
debugging, 17
decision variable, 9, 113
 creation, 67
 declaration, 15, 23
 type, 33, 74, 100
declaration, 113
declarations, 15, 47
dense array, 114
deployment template, 49
discrete variable, 33, 74, 113
distributed computing, 5, 114
DSO, see dynamic shared object
dynamic array, 23, 114, 116
dynamic set, 23, 114
dynamic shared object, 114

E

editor window, 13
elif/then, 28
embedding, 5, 49
empty line, 16
end-do, 28
end-function, 46
end-model, 15
end-procedure, 46, 47
equality constraint, 9, 68, 113
error handling, 16, 72, 92
exportprob, 53
expression
 linear, 15, 67
 quadratic, 80

F

F_APPEND, 23
F_OUTPUT, 23
fclose, 23
feasibility tolerance, 47, 88
feasible solution, 115
finalize, 23, 114
fixed array, 114
fopen, 23
forall, 15, 28, 114
forall/do, 28
forming, 16
 output, 23
forward, 45
function, 46, 116

G

getLPStat, 72
 getMIPStat, 75
 getparam, 47
 getprobat, 28
 getXPRSprob, 88
 graph drawing, 28
 graphical user interface, 5, 114

H

hashmap array, 116
 heuristic, 114
 variable fixing, 44, 85

I

if/then/else/end-if, 28, 115
 if/then/end-if, 28
 indentation, 16
 index
 string, 20, 69, 114
 index set, 71, 114
 indicator variable, 32, 73, 99, 113
 inequality constraint, 9, 113
 initialization
 BCL, 67
 explicit, 72
 Mosel, 51
 Optimizer, 92
 initializations from, 23
 initializations to, 23
 input file, 22, 71
 Insight, see Xpress Insight
 integer variable, 33, 74, 100, 113
 integrality condition, 35, 113
 interior point algorithm, 115
 is_binary, 33
 is_integer, 33
 is_partint, 36
 is_semcont, 36
 is_semint, 36
 IVE, 114

L

language
 modeling, 5
 solving, 5
 library
 embedding, 5
 limit value, 77, 102, 113
 line break, 16
 linear constraint, 15, 67
 linear expression, 15, 67
 Linear Programming, 4, 12, 66, 95, 114
 optimization information, 19, 69, 94, 98
 problem status, 72, 96
 relaxation, 35, 114
 list, 29
 loadMat, 88
 loadprob, 48
 loop, 15, 28, 114

optimization, 27

LP, see Linear Programming

LP format, 53, 72, 92

lpOptimize, 68, 80

M

mathematical model, 10
 Mathematical Programming, 4
 Mathematical Programming problem, 4, 114
 Matlab, 5, 115
 matrix, 114
 export, 53, 72
 format, 53, 72, 92
 import, 92
 matrix representation, 95, 100, 106, 115
 maximize, 68
 maximize, 15, 17, 35, 47
 minimize, 40
 MIP, see Mixed Integer Programming
 MIP heuristics, 35, 47, 76, 88
 mipOptimize, 72, 75, 88
 MIQP, see Mixed Integer Quadratic Programming
 Mixed Integer Programming, 4, 32, 73, 99, 114
 optimization information, 34, 76, 88
 problem status, 75, 102
 search, 35, 76, 101
 Mixed Integer Quadratic Programming, 4, 38, 79, 108, 115
 optimization information, 43, 84
 mminsight, 53
 model, 114
 BCL, 67
 embedding, 5
 incremental definition, 42, 83
 Mosel, 14
 parameter, 24, 115
 model, 15
 model building, 4, 8
 modeling language, 5, 114
 modeling objects, 6, 113
 modeling style, 24
 module, 5, 114
 Mosel, 115
 environment, 5
 language, 5, 115
 libraries, 5, 115
 model, 14
 Native Interface, 5, 115
 standalone, 24
 mosel, 25
 Mosel Compiler Library, 51
 Mosel Run-time Library, 51
 MPS format, 53, 72, 92
 mpvar, 15

N

names
 constraint, 28
 defining, 68
 namespace, 67

Newton-Barrier, 81, 115
 newVar, 67
 NI, see Mosel Native Interface
 NLP, see Non-linear Programming
 node, 35, 113
 noimplicit, 15
 Non-linear Programming, 4, 112, 115

O
 objective function, 10, 115
 definition, 15, 67
 quadratic, 39, 80, 105
 ODBC, 5, 114
 optimal solution, 115
 optimization, 15, 68, 96, 115
 loop, 27, 41
 optimization project, 5
 Optimizer, see Xpress Optimizer, 15, 68, 115
 direct access, 88
 option
 noimplicit, 15
 output, 15, 19, 68, 96
 formatting, 23
 redirection, 98
 output file, 24, 52, 93
 overloading, 47, 115

P
 parameter, 115
 parameter settings, 24, 35, 47, 52, 88, 115
 parameters, 24, 115
 partial integer variable, 36, 77, 102, 113
 presolving, 35, 47, 88
 print, 71
 printing, 15, 68
 problem
 change, 48, 88
 creation, 67
 input, 96
 instance, 52, 115
 matrix, 95, 114
 parameter, 115
 problem status, 28, 115, 116
 LP, 72, 96
 MIP, 75, 102
 QP, 108
 procedure, 46, 116
 programming language, 5
 project navigation, 13

Q
 QP, see Quadratic Programming
 quadratic expression, 80
 quadratic objective function, 39, 80, 105
 Quadratic Programming, 4, 38, 79, 105, 115
 algorithm, 81
 optimization information, 81
 problem status, 108

R
 R, 5, 115

range set, 15, 115
 remote computing, 5, 114
 repeat until, 114
 return value, 47
 right hand side, 24, 95, 115
 row, 95, 114

S
 selection statement, 115
 semi-continuous integer, 36, 77, 102, 113
 semi-continuous variable, 36, 77, 102, 113
 set
 definition, 15
 dynamic, 23, 114
 range, 15, 115
 setLB, 88
 setlb, 48
 setLim, 77
 setMsgLevel, 69
 setObj, 68
 setparam, 35, 47
 setSense, 68, 72
 setUB, 88
 setub, 48
 Simplex, 19, 81, 88, 115
 SLP, see Successive Linear Programming
 solution, 115
 feasible, 115
 optimal, 115
 solution heuristic, 44, 85
 solution information, 19, 116
 retrieving, 52
 solver, 115
 solving, 15, 68, 96, 115
 space, 16
 sparse array, 116
 sparse format, 114
 strftime, 23
 string indices, 20, 69, 114
 subroutine, 46, 116
 subroutine definition
 overloading, 47
 Successive Linear Programming, 4, 112, 116
 sum, 29
 system functions, 5, 114

T
 tolerance value, 47, 88

U
 user graph, 28, 41
 user module, 5, 114
 uses, 15, 30

V
 valid inequality, see cut
 variable, see decision variable
 VDL, 57, 60
 view definition, 57

W

web deployment, 5
while, 114
Workbench, see Xpress Workbench
 starting, 12
write, 15
writeln, 15

X

XPRB_BV, 74
XPRB_PI, 77
XPRB_SC, 77
XPRB_SI, 77
XPRB_UI, 74
XPRBctr, 71
XPRBexpr, 80
XPRBindexSet, 71
XPRBprob, 71
XPRBreadarrlinecb, 80
XPRBreadlinecb, 71
XPRBsos, 71
XPRBvar, 71
Xpress Insight, 5, 53
Xpress Kalis, 112
Xpress Mosel, see Mosel
Xpress NonLinear, 6, 112
Xpress Optimizer, 6
Xpress SLP, 112
Xpress Solver, 6, 115
Xpress-BCL, see BCL
XPRS_CONT, 47
XPRS_CUTSTRATEGY, 35, 47, 88
XPRS_HEURSTRATEGY, 35, 47, 88
XPRS_LPLOG, 88
XPRS_LPSTOP, 47
XPRS_MIPLOG, 88
XPRS_OPT, 28
XPRS_PRESOLVE, 35, 47, 88
XPRSchgobjsense, 108
XPRSloadglobal, 101
XPRSloadlp, 97
XPRSloadqglobal, 108
XPRSloadqp, 108
XPRSloptimize, 97
XPRSmipoptimize, 101
XPRSpob, 88
XPRSsetintcontrol, 88
XPRSsetlogfile, 98

Z

ZEROTOL, 48