dash optimization

Xpress-MP Essentials

Second Edition, February 2002.

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

How to Contact Dash

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

USA, Canada and The Americas
Dash Optimization Inc.
560 Sylvan Avenue
Englewood Cliffs
NJ 07632
USA
Telephone: (201) 567 9445
Fax: (201) 567 9443
email: support-usa@dashoptimization.com

Elsewhere
Dash Optimization Ltd.
Quinton Lodge, Binswood Avenue
Leamington Spa
Warwickshire CV32 5RX
UK
Telephone: +44 1926 315862
Fax: +44 1926 315854
email: support@dashoptimization.com

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

USA, Canada and The Americas
Dash Optimization Inc.
560 Sylvan Avenue
Englewood Cliffs
NJ 07632
USA
Telephone: (201) 567 9445
Fax: (201) 567 9443
email: sales@dashoptimization.com

Elsewhere
Dash Optimization Ltd.
Blisworth House, Church Lane
Blisworth
Northants NN7 3BX
UK
Telephone: +44 1604 858993
Fax: +44 1604 858147
email: sales@dashoptimization.com

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

Contents

Chapter 1
Getting Started

Overview

In this chapter you will:

- meet Xpress-Mosel and the Xpress-Optimizer;
- be introduced to the interfaces through which to use them;
- discover how this book is organized and what you should read.

Introduction

Xpress-MP is a mathematical modeling and optimization software suite, providing tools for the formulation, solution and analysis of linear, quadratic and integer programming problems. Based on the powerful components, Xpress-Mosel and the Xpress-Optimizer, the suite comprises a collection of interfaces, addressing the diversity of users' needs, both in problem solving and embedding Xpress-MP technology within their own custom products.

This book provides an introduction to using the major components of Xpress-MP and, as such, is ideal for getting new users up and running immediately. The pace is relatively gentle and very little knowledge of Mathematical Programming is assumed. For more experienced users, the pages that follow may also provide new or alternative ideas

for using the product suite effectively. Many of the reference manuals that accompany the Xpress-MP software suite assume some knowledge of the material contained in Xpress-MP Essentials and it is recommended that all users familiarize themselves with this guide before continuing.

We begin by briefly describing the components and interfaces of Xpress-MP and explain how they inter-relate. You may have already decided which of the components you will need to use and can go immediately to the relevant chapters of this book. For those who are still trying to determine which are right for them, we provide an overview of the different components to help you decide.

It should be noted that this book does *not* contain any details of the installation and setting up of Xpress-MP products. Such information can be found in the accompanying 'readme' documents (`readme.win`, `readme.unx`) which can be found on the CD-ROM. We strongly recommend that all users consult the document relevant to their operating system before attempting to install and use any of the software contained on the CD-ROM.

Xpress-MP Components and Interfaces

Xpress-Mosel

There is also a third, Xpress-BCL, which is available only to library users. See Chapter 4 for details.

The two basic components of Xpress-MP are Xpress-Mosel and the Xpress-Optimizer. Mosel is an environment for modeling and solving problems, taking as input a model program describing a linear programming (LP) or mixed integer programming (MIP) problem, written in the Mosel model programming language. While small, this language is extensible through the inclusion of library modules which augment its functionality in different ways. The basic premise of Mosel is that there is no separation between a modeling statement and a procedure that actually solves the problem. To this end, the Optimizer is included as a library module, allowing complicated

models to be constructed and solved from within Mosel itself, returning the solution in a user-defined manner.

Using the ODBC library module, Mosel may be used to retrieve data from, and export solution information back to, a wide range of spreadsheets and databases. ODBC is the industry standard for communication with most major databases, and the Mosel `mmodbc` library module supports ODBC connections to any ODBC-enabled product including Excel, Oracle and Access.

Typically a user will use Mosel to extract data from a text file or database to generate a problem instance, call the Optimizer library module to find the optimal solution to the problem and finally send the results back to another ODBC-enabled product for reporting and presentation. Input data can, in fact, be gathered from a number of different data sources and different aspects of the solution can be sent to different files or applications as best suits the requirements.

The Xpress-Optimizer

At the core of the Xpress-MP suite, the Xpress-Optimizer represents decades of research and development in solution methods for linear programming (LP), quadratic programming (QP) and mixed integer programming (MIP) problems. Under continuous development using in-house expertise and through close relationships with research groups worldwide, the Optimizer has proved itself time and again on a wide spectrum of problems.

Although developed to recognize the best strategies for solving most problems, access is also provided to a comprehensive set of controls, allowing users to tune the Optimizer's performance on their own individual problems. This is particularly useful for large mixed integer programming problems, which may be difficult to solve to even approximate optimality.

For the very hardest MIP problems, Xpress-MP provides the parallel MIP Optimizer, designed to work on both a heterogeneous network of PCs and/or workstations, and on shared memory multiprocessor computers, to exploit all the computing power at your disposal. The parallel Optimizer is at the leading edge of large-scale optimization.

Xpress-IVE

Xpress-IVE, the Xpress Interactive Visual Environment, is a complete modeling and optimization development environment running under Microsoft Windows. Presenting Mosel in an easy-to-use Graphical User Interface (GUI), with built-in text editor, IVE can be used for the development, management and execution of multiple model programs.

Combining ease of use with the powerful modeling and optimization features that Xpress-MP users have come to enjoy, IVE is ideal for developing and debugging prototype models.

Console Xpress

Console Xpress comprises the main stand-alone executables, `mosel` and `optimizer`. These are simple yet powerful text-based 'console' interfaces to the Mosel and Optimizer components, respectively. They offer a wide range of features and algorithmic tuning options and are available in essentially identical form on a wide range of platforms including Windows, Linux and other forms of UNIX.

We have found that Console Xpress products satisfy the needs of many commercial users. Having developed your model with Xpress-IVE, you can rapidly build production systems by wrapping Console Xpress components in batch files or shell scripts, and tune the optimization algorithms to solve your largest data instances.

The Xpress-MP Libraries

For more specialized applications, we offer the Xpress-MP Libraries, providing access to the Mosel and Optimizer components from within your own C/C++, Java and Visual Basic applications. The main advantages of the Xpress-MP Libraries are that your own applications can interact tightly with the functionality provided by Xpress-MP and that greater flexibility exists to develop customized applications to suit your needs. Under Windows, the Xpress-MP Libraries are available as DLLs, providing a standard programming interface for integrating Xpress-MP components within your Windows

applications. For UNIX users, the Xpress-MP Libraries are available as shared object libraries.

On one level the Xpress-MP Libraries may be used simply to build stand-alone applications using the basic model building and optimization functionality available in all Xpress-MP products. An application to import the data, generate a problem instance, solve the problem, and export the solution can be developed swiftly, without needing to wade through a vast manual.

For users wishing to build custom applications and develop specialized heuristics, the Xpress-MP Libraries offer a much higher level of functionality. They enable you to load, manipulate and output matrices, solve problems and retrieve the solution, handle multiple problems, and adjust controls. In addition, a sophisticated collection of callbacks from the optimization algorithms and the proven 'Branch and Cut' management tools offer the user unrivaled possibilities to develop advanced optimization applications.

Which Interface Should You Use?

For first-time users or for developing new models, Xpress-IVE provides the simplest interface to the Xpress-MP software. Its integrated development environment gets you modeling quickly, and provides all the solution handling possibilities of the other interfaces.

To operate a production system using modeling and/or optimization, Console Xpress can be used to build batch processing applications. This is also a flexible product for tuning optimization to suit the largest problem instances.

If you are developing software such as Windows applications for your end-user, the Xpress-MP Libraries offer the best interface. They can also be used to good effect if you are investigating a difficult class of problems and wish to develop specialized heuristics to help obtain good solutions.

How to Read this Book

Structure of the Book

This book assumes as a prerequisite that the Xpress-MP software you wish to use has already been installed and no details of installation are given here. For information on the installation of software and the setting up of library components under a number of the major compiler environments, you should refer to the accompanying 'readme' files (`readme.win`, `readme.unx` and `lib\readme.txt`) on the installation CD-ROM.

The following three chapters (2 – 4) are of particular interest to new users, discussing the different Xpress-MP interfaces: Xpress-IVE, Console Xpress and the Xpress-MP Libraries. For the majority of users only one of these will be relevant, depending on the particular interface that you have chosen to use. Each of the chapters will take you through entering and solving a simple problem and introduce many of the practical issues that will be important when optimizing your own problems.

Following this are two chapters on Mosel and its model programming language. The first of these, Chapter 5, "Modeling with Xpress-MP", introduces many of the major features of the modeling language, incrementally developing a simple model and exploring the benefits of versatile modeling. Chapter 6, "Further Mosel Topics", extends some of these ideas, introducing a collection of related topics. The treatment here is only cursory, however, and further details on these topics may be found in the Xpress-Mosel Reference Manual.

The book ends with a glossary of common terms.

Conventions Used

To help you to get the most out of this book, a number of formatting conventions have been employed. On the whole, standard typographical styles have been used, representing programming constructs or program output with a `fixed width font`, whilst equations and equation variables appear in an *italic type face*. Where

it might get confused with program output, user input will be indicated in **bold type face** and this will often also be used to indicate differences between a number of similar programs. Some additional graphical styles and icons have also been used and are as follows:

Notes: These are general points which we want to make sure that you are aware of. They are things that may not otherwise be noticeable in the surrounding text.

Caution: Occasionally we will come across particular points that may be hazardous to the health of your problem or solution if not considered.

Ideas: Experience is a wonderful thing and we want to share it with you. These ideas are often approaches that we have found to work well in the past, or are particular points that have previously been of value to our users.

Debugging: Particularly relevant to the earlier chapters of this book, occasionally users experience difficulty with setting up the software, and these sections draw attention to common errors, pointing to sources of help.

Exercises: Throughout the text we have provided a number of short exercises. These provide breaks from reading about the products to help you try out the ideas in the surrounding section.

Signposting: At various points in the book, a number of alternative routes through the text are possible and depending on your interest and knowledge you may want to skip certain portions of material. This icon indicates the end of a logical section or material and is accompanied by suggestions for what you may read next.

Points in the margin

Side Notes: Some use will also be made of margin notes to stop small points from getting in the way, and margin quotes to draw attention to important concepts in the text.

Summary

In this chapter we have learnt:

- about the main components of the Xpress-MP software suite, Mosel and the Optimizer;
- about the three interfaces to the main components, Xpress-IVE, Console Xpress and the Xpress-MP Libraries;
- which interface is best suited to your needs.

Chapter 2
Xpress-IVE

Overview

In this chapter you will:

- work with projects and model program files in the Xpress-IVE interface;
- input simple models and solve them;
- export and input problems as matrix files;
- learn how to customize the look of the Xpress-IVE environment.

Getting Started

Xpress-IVE is the graphical user interface to Xpress-Mosel and the Xpress-Optimizer for users of Microsoft Windows, providing a simple development environment in which to learn about and use Xpress-MP optimization tools. If you are reading this chapter, we assume that you have already installed IVE. Over the course of this chapter we will learn how to input and solve a simple model, explaining the main features of the interactive visual environment and discuss how solution information and output may be produced in several formats.

The IVE interface brings together the two main components of the Xpress-MP suite, Mosel and the Optimizer, in a single graphical environment for easier use. Implementing the powerful Mosel model

programming language, complicated models can be developed and solved with the minimum of trouble, making IVE the ideal starting point for new users of Xpress-MP software.

Starting Xpress-IVE

You may wish to add a shortcut to IVE to your desktop for easier access.

During installation, a folder of icons is added to the Start Menu. Using this, IVE can be started by clicking on the *Start* button, choosing *Programs*, *Xpress-MP* and finally *Xpress-IVE*. Otherwise, IVE may be started from Windows Explorer by clicking on the *ive.exe* icon in the `xpressmp\bin` directory.

Exercise *Start up IVE now and make sure that the installation was successful.*

All Xpress-MP products employ a security system as an integral part of the software which must be set up during the installation process for all features of the product to be enabled. If set up incorrectly, or if used without a license, Xpress-MP will be started in 'trial mode' and a warning will be displayed along the top of the window stating 'Trial Version — commercial use prohibited'. The software can still be used in this mode, although some restrictions are placed on the size and complexity of the problems that can be handled by it. If this occurs and you have a current license for Xpress-MP software, refer to the 'readme' files for details of setting up the security system correctly.

Entering a Simple Model

Working With Projects

Using IVE, models are created and developed in the context of a *project*. An IVE project can contain any number of files relating to a single model and provides a convenient object with which to work. The *Project Files* pane in the left-most window of your workspace

displays the project currently under development and any files associated with it. Our first task in entering a new model is to create a project to contain it.

Creating a new project

Choose *Project* and then *New Project* from the IVE menu bar. Using the 'Browse' button, choose a parent directory for the project, and enter a name for the project directory. A new directory will be created to contain all the files for the project unless this option is unchecked. Click 'OK' to proceed.

Exercise *Create a project called 'essentials' to work with in this chapter. What sorts of files might you need to include in a project?*

The *Project Files* pane will be updated to display your new project. A project can contain both input files added by you and output files generated during a model run and its compilation. A model program file and any number of files holding external data relevant to the model may be added to a project, whilst output files may include binary model and matrix files. For now, we will just add a model program file to the project.

If the central editor window is not immediately visible, drag the boundary of the Output/Input pane to reveal it.

Adding a model program file

Choose *File*, *New* and then *Blank File* from the IVE menu bar. A dialog box will appear enabling you to specify the filename, file type and to add the file to the current project.

Exercise *Create a new Mosel file named 'simple.mos' and choose to associate it with the current project, 'essentials'.*

The *Project Files* pane should now indicate that the new file '*simple.mos*' is part of your project and a blank file will be open in a new window ready for the model to be entered.

The Model Editor

IVE comes equipped with its own editor for creating and altering model files. Any ASCII file can be viewed, edited and saved from within this, although its usefulness comes from recognizing a model's structure and distinguishing visually between its various components. This feature alone can save much debugging time, allowing most typographical errors to be identified and corrected immediately.

Exercise *Type the model given in Listing 2.1 line by line into the model editor.*

Listing 2.1 A simple model

```
model simple
  uses "mmxprs"

  declarations
    a: mpvar
    b: mpvar
  end-declarations

  first:=  3*a + 2*b <= 400
  second:=   a + 3*b <= 200
  profit:=   a + 2*b

  maximize(profit)

  writeln("Profit is ", getobjval)
end-model
```

The simple model of Listing 2.1 contains just two *decision variables* (`a` and `b`) and two *constraints* (`first` and `second`). Our problem is to maximize the *objective function*, `profit`.

For the moment, since we will be concentrating only on using IVE, we will not discuss the meaning of the model or any of its constituent parts, although in this case it may already be clear. In Chapter 5 we will focus on the Mosel language.

"Keywords..."

As the model is entered, certain *keywords* such as `model`, `uses` and `declarations` are recognized and highlighted in blue. Keywords act as delimiters for the various components of the model program, so it is important to make sure that they are entered correctly. Keywords may be chosen and conveniently entered directly into the model using a dropdown menu, available in the editor by holding down the Ctrl key and space bar simultaneously.

Exercise *Delete the declarations block in your model and leave the cursor at the point where this used to be. By pressing Ctrl-Space, choose 'declarations@@end-declarations' from the list to re-enter it. Ensuring that the model still looks like Listing 2.1*, save it.

"Text searches..."

The IVE model editor comes equipped with its own text search facility, particularly useful when editing long files. By typing a word or phrase in the text box next to the binoculars icon and hitting the Return key, the first instance of that word or phrase is highlighted. Hitting Return again moves on to the next until no further instances can be found in the file and the search wraps to the first occurrence again.

Exercise *Using the IVE model editor's text search facility, find all instances of the word 'profit' in the file 'simple.mos'.*

Compile...Load...Run!

The file that you have just created is perhaps best described as a *model program*. It contains not only model specification statements, but also imperative statements which must be executed. IVE deals with such files firstly by compiling them, then loading the compiled file and finally running it. Every time a model program file is altered, you will have to go through this process to solve the new model. We consider each of these steps in turn.

Compiling a model program file

Choose *Build* and then *Compile* from the IVE menu bar. The file currently active in the model editor is compiled.

As a model program is compiled, information about the compilation process is displayed in the *Build* pane at the bottom of the workspace. This pane is automatically displayed when compilation starts. If any syntax errors are found in the model, they are displayed here, with details of the line and character position where the error was detected and a description of the problem, if available. Warning messages are also displayed in this manner, prefixed by a `W` rather than an `E`. If no errors are detected, the *Build* pane displays 'Compilation successful' and a compiled binary model or BIM file is produced and added to the project.

Exercise *Compile the model program of Listing 2.1. If any errors are detected, check the listing carefully and correct the problem. Check that the binary model file is added to your project.*

The compiled BIM file can now be loaded and run to find a solution to the problem.

Loading and Running a BIM file

Select *Build* and then *Run* from the menu bar. The BIM file for the current model is automatically loaded and then executed.

Exercise *Load and run the file 'simple.bim' to solve the problem. What is the maximum amount of profit that can be obtained in our model?*

When a model program is run, the *Output/Input* pane at the right hand side of the workspace window is selected to display program output. Any output generated by the model is sent to this window and in our case it is here that the objective function value is displayed.

Obtaining Further Information from Xpress-IVE

Knowing the maximum amount of profit that we can make in our model is certainly useful, but what values should the decision variables take in order to produce this? IVE makes all information about the solution available through the *Entities* pane in the left hand window. By expanding the list of decision variables in this pane and hovering over one with the mouse pointer, its solution and reduced cost are displayed. Dual and slack values for constraints may also be obtained.

Solution details can also be obtained by simply hovering over an entity in the model file.

Additionally, by clicking on a variable or constraint in the *Entities* pane, all lines containing occurrences of it in the model are highlighted with a marker and are described in a *Locations* pane at the bottom of the workspace. The markers can subsequently be removed using the 'Clear bookmarks' icon on the IVE toolbar.

Exercise *Click on the tab to display the Entities pane. What values must the decision variables (*`a` *and* `b`*) take to achieve the stated profit? Clicking on the variable* `a`*, find all lines in the model containing this variable before finally removing the markers.*

"Solution graphs..."

IVE will also provide graphical representations of how the solution is obtained, which are generated by default whenever a problem is optimized. The right hand window contains a number of panes for this purpose, dependent on the type of problem solved and the particular algorithm used. For the problem that we have been considering, the pane *Sim:Obj(iter)* represents the values of the objective function at various iterations when the simplex algorithm is employed.

Exercise *Display the Sim:Obj(iter) pane and resize the window if necessary to view the entire graph and information above it. Moving the mouse pointer over the different areas of the graph, notice that an iteration number and the value of the objective function at that iteration are displayed above the graph.*

In this case, only the initial point and final iteration are displayed on the graph, providing no extra information. Presently, we will see how intermediate iterations can be viewed when we learn how to change the parameters which control this.

Going Further with Xpress-IVE

Setting Control Options

The Mosel model programming language provides a number of possibilities for influencing the solution process by setting control parameters relied on by the Xpress-Optimizer. With such possibilities catered for directly in the language, IVE does not provide duplicate functionality except where the options directly affect the output which IVE produces. Choosing *Build*, followed by *Options*, a full list of the possibilities may be viewed.

Perhaps of most immediate interest is the `LPLOG` control which influences how much detail about solution iterations is produced during optimization. Specifically, the Optimizer outputs information about the start point, the end point and every `LPLOG` iterations between these. Typically set to `100`, the default and Mosel settings for this can be overridden for the simplex algorithm using the first option. Notably, it is this option which affects how many iterations are plotted on the graph produced in the *Sim:Obj(iter)* pane.

Exercise *Checking the 'Ignore Mosel settings' box and setting* `LPLOG` *to* `1`*, click to 'Apply' the settings and re-optimize the problem 'simple'. From the graph, what is the value of the objective function at the first iteration of the algorithm?*

Whilst interesting to consider on small problems such as ours, on larger problems, Optimizer performance can be significantly worse if output must be produced at each iteration. In such cases, setting `LPLOG` to a higher value is preferable. If you are not interested in the solution graphs, it might also be preferable to disable the graphical options altogether, so no resources are wasted during the optimization. The options for achieving this are also available from this same control options menu.

Running Multiple Models

Whilst only one project may be loaded in IVE at a time, a project can contain a number of model files, which can be held in memory and worked on simultaneously. The model currently displayed in the editor window is designated the *active problem*, but by selecting from the open models, any of them can be compiled and run as necessary. To demonstrate this, we will need another model, such as that given in Listing 2.2.

Listing 2.2 The altered model

The only changes in the model between this and Listing 2.1 is the imposition of a new constraint. You can expect the objective function value to decrease if this constraint is binding.

```
model altered
  uses "mmxprs"

  declarations
    a: mpvar
    b: mpvar
  end-declarations

  first:=  3*a + 2*b <= 400
  second:=   a + 3*b <= 200
  third:=  6*a + 5*b <= 800
  profit:=   a + 2*b

  maximize(profit)

  writeln("Profit is ", getobjval)
  writeln(" a = ", getsol(a), "; b = ", getsol(b))
end-model
```

Exercise *Add a second model file 'altered.mos' to the project 'essentials' and type in the model of Listing 2.2.*

As a model is edited in IVE it becomes the active problem. Whenever the *Compile* or *Run* options are called from the *Build* menu, it is the active problem which is affected.

Exercise *Compile and run the new model, 'altered'. What is the maximum profit that may be obtained now? How does this compare to the profit for project 'simple'?*

We have already seen that the *Project Files* pane in the left hand window displays a list of all the files in the current project. A list of all *open* files is also provided in the form of a collection of buttons just above this window, labelled with the file's name. By clicking on these buttons, the associated file can be displayed in the editor window, making it the active problem.

It should be noted, however, that while the active problem may be changed in this way, information in the *Entities* pane is updated only when a compiled model is run. For this reason, solution information in the *Entities* pane need not necessarily refer to the active problem and you should take care when working with several models.

Exercise *Click on the button to redisplay the model 'simple' in the editor window. Notice that all information provided in the Entities pane still relates to the problem 'altered', however.*

Other files in a project may also be opened by clicking on them in the Project Files pane using the right mouse button and choosing *Open/ Show file* from the drop-down menu displayed. Open files may be closed by choosing *Close file* from this same menu and files may be deleted from a project using *Remove from project*.

Exercise *Close the file 'altered.mos' in the editor window and remove this and 'altered.bim' from the current project. Re-run 'simple' to update the information in the Entities pane.*

Writing Matrix Files

Sometimes you may create a model for which you want more control over the optimization process than is afforded by IVE, or you may want to give the model to someone else to use. In both cases output is required in a common format which is understood by the solver program that will eventually be used. IVE allows you to create such *matrix files* in the two main industry standard formats: LP files and MPS files.

You must ensure that the model has been compiled and run before this will work.

Exporting problem matrices

Choose *Build* followed by *Export matrix* from the menu bar and select between either an *MPS matrix file or* a maximization or minimization *LP* as required. Set which constraint represents the objective function before writing the matrix file.

Exercise *Create an MPS matrix file 'simple.mat' from the model program 'simple.mos' and close the model file 'simple.mos'.*

Unlike the MPS format, LP files contain additional information about whether the model describes a maximization or minimization problem. When exporting files in this format, it is important to specify the optimization sense correctly.

Exercise *Using File, Open, view your new matrix file using the IVE model editor. When prompted, choose to add this to your current project.*

Loading Problems from Matrix Files

Xpress-IVE can also be used to load and solve problems available as matrix files, providing a direct interface to the Xpress-Optimizer from within IVE. Although IVE is predominantly a graphical interface for developing models using Mosel, it may sometimes be desirable to use it in this way if, for instance, a user is interested in employing its solution graphing capabilities.

Loading and Solving Matrix Problems

Using *Build*, followed by *Optimize matrix file* from the IVE menu bar, browse for the matrix file to be used. If it is already open in the editor window, the file will be automatically selected. Setting the algorithm, sense and problem type, click on 'Start' to begin solving. Optimizer controls such as `LPLOG` can also be set from this dialog.

Exercise *Load the matrix file 'simple.mat' that you have just created and, setting the integer control parameter* `LPLOG` *to* `1`*, maximize the LP problem.*

"Problem statistics..."

Once a solution to the problem has been found, the *Output/Input* pane will display problem statistics. Often overlooked, these are useful in checking that the matrix loaded actually corresponds to the problem which we wanted to solve. In this case we see that the problem matrix has three rows (corresponding to two constraints and the objective function) and two (structural) columns, another word for the decision variables. The matrix has six nonzero elements because both `a` and `b` appear with nonzero coefficients in all rows. Since this is *not* a (mixed) integer problem, there are no global entities, no sets and hence no set members in the problem.

Following this the value of the objective function at its initial, last and every `LPLOG` iterations is displayed, along with the solution status.

Exercise *Check that the problem statistics correspond to the problem 'simple'. From this or from the Sim:Obj(iter) pane, note how the optimal objective value was obtained.*

Graphing Functions

We have already seen above that IVE can graphically display the objective function value during the optimization, at any iterations as determined by the `LPLOG` control. However, IVE also allows for the graphing of your own functions by embedding the commands `IVEinitgraph` and `IVEaddtograph` in your Mosel program, with output viewed from a *User Graph* pane in the right hand side window. An example is provided in Listing 2.3.

IVEinitgraph

Initializes IVE's user graphing facility. Its two arguments are the names for the x and y-axes respectively.

IVEaddtograph

Plots a point on the user graph. Its two arguments are the values for the point in Cartesian coordinates.

Listing 2.3 Graphing exponential decay

```
model decay
  uses "mmive";

  IVEinitgraph("Decaying oscillator","x","y")

  forall(i in 1..400) do
    IVEaddtograph(i,exp(-0.01*i)*cos(i*0.1))
  end-do
end-model
```

Exercise *Create a new file called 'decay.mos', entering the model of Listing 2.3. Selecting the User Graph pane, run the model and view the function.*

Changing Your Environment

Many aspects of the IVE environment are customizable and may easily be altered if they do not suit the way in which you like to work. Properties such as font faces, color and sizes as well as the placement and visibility of windows within the workspace are all determined by settings that can be changed for a particular project. Default settings are applied whenever a new project is created and are read whenever that project is opened.

Window placement within the IVE workspace is as simple as dragging a window boundary to a new position. Visibility can be changed by selecting/deselecting either of the *Project Bar*, *Info Bar* or *Run Bar* from the *View* menu.

Exercise *Experiment with each of the various options on the View menu, turning each one off and then on to see the effect. Resize any windows to make your environment easier to work with.*

Other settings for a project may be accessed by clicking the right mouse button in the Model Editor window and choosing *Properties*. The dialog box displayed offers tabs to alter Color/Font, Language/ Tabs, Keyboard and other miscellaneous settings. Having made changes, clicking on the 'Apply' or 'OK' buttons updates your window and font information with any changes. This is then saved when the project or application closes.

Exercise *View the settings as they currently stand and change the color used by the editor to display keywords. Click the 'Apply' button for the changes to take effect.*

The IVE model editor can also be used to edit other types of files by setting the language appropriately on the *Language*/*Tabs* pane. Options exist for smart-editing of C/C++, Basic, Java and many other file types. For easier editing of long models, line numbering can also be applied to files from the *Misc* pane, setting the style to decimal.

Getting Help

Having reached this point, you have learnt how to create and manage projects in Xpress-IVE, to enter, alter and save simple models using the Model Editor, and to compile, load and run them to obtain a solution. You have also learnt how to work with several models simultaneously, to export and optimize problems as matrix files and to customize the IVE workspace. Up to this point we have been concentrating solely on how to use the environment provided by IVE. In Chapter 5, however, the basics of the Mosel programming language are explained, allowing you to model and solve your own problems. You may wish to turn there next. The following two chapters contain information on using Console Xpress and the Xpress-MP Libraries and may be safely skipped if you intend to use only IVE. If you require additional help on using either Mosel or the Optimizer, this may be obtained from the Mosel and Optimizer Reference Manuals respectively.

Summary

In this chapter we have learnt how to:

- create and manage projects in IVE;
- create models using the built-in model editor;
- compile, load and run model programs;
- export problems as matrix files;
- import and solve problems from matrix files;
- customize the IVE environment for a project.

Chapter 3
Console Xpress

Overview

In this chapter you will:

- use Xpress-Mosel to compile, load and run a model program file;
- solve linear and integer programing problems and access the solution;
- use the Xpress-Optimizer on its own to solve problems;
- affect the optimization process, setting controls and choosing the algorithm used.

Getting Started

Console Xpress provides a powerful text-based interface to the Xpress-MP software suite, allowing interactive development and batch-mode processing of models by users of all platforms. If you are reading this chapter, we shall assume that you have already installed Console Xpress. Over the course of this chapter we will learn how to input and solve simple models, explaining the major features of the Console interface. Users of the Xpress-MP Libraries may also find this information useful.

The Components of Console Xpress

The Console Xpress interface essentially comprises two major components: Xpress-Mosel and the Xpress-Optimizer. The first of these, Mosel, provides a highly flexible environment in which model specification programs written in the Mosel model programming language may be compiled and run. Matrix output files can be generated in a number of industry standard formats and subsequently loaded into the Optimizer to solve and output a solution. Alternatively, and more usually, the functionality of the Optimizer may be plugged directly into Mosel as a library module, enabling users to solve problems from within Mosel itself. Both of these components can also be usefully employed in batch mode for automated problem solving.

Running one of these components is as simple as typing either `mosel` or `optimizer` at the command prompt. Our first task before going any further will be to check successful installation of the components by testing them.

Testing Xpress-Mosel

Successful installation of Mosel may be checked by issuing the command `mosel` at the shell command prompt and pressing the Return key. The program responds with a short banner, following which the Mosel prompt, `>`, is displayed and Mosel waits for user input.

Exercise *Start Mosel as described above. At the Mosel prompt, type* `quit` *and you should be returned to the shell command prompt.*

Listing 3.1 demonstrates the procedure which you have just gone through, with input from the user highlighted in bold face.

If the header information described does not appear when you follow through the steps outlined here, it may be that the `mosel` and `optimizer` binaries are not in your current path. Check this by changing to the directory containing the binaries and attempt to run them from there. If this is successful, update your path accordingly.

Listing 3.1 Testing Console Xpress-Mosel

```
C:\> mosel
** Xpress-Mosel **
(c) Copyright Dash Associates 1998-zzzz
> quit
Exiting.

C:\>
```

All Xpress-MP products employ a security system as an integral part of the software. This must be set up during the installation process for all features of the product to be enabled. If the security system is not set up correctly, Xpress-MP components will be initialized in 'trial mode', allowing use but placing restrictions on the size and complexity of problems that can be handled. If this occurs and you have been licensed to use Console Xpress, refer to the 'readme' files for details of setting up the security system correctly.

Testing the Xpress-Optimizer

Successful operation of the Optimizer may be checked in a similar manner to Mosel by running `optimizer` from the command prompt. The program again responds with a header supplying the version of the Optimizer installed, followed by a prompt for a *problem name*. If the problem name is left blank and the Return key is pressed, the Optimizer will assume a default problem name of '`$$$$$$$$`' for the purposes of any files that it may need to create. Finally the Optimizer's command prompt, `>`, is displayed and the program waits for user input.

The *problem name* forms the basis for any files that are generated by the Optimizer or Mosel.

Exercise *Run the Optimizer to check that installation was successful and when prompted for a problem name, just hit Return. At the Optimizer prompt, type* `QUIT` *to exit the program.*

Listing 3.2 displays an equivalent session to Listing 3.1 for the Optimizer, again with user input highlighted in bold type.

Listing 3.2 Testing the Console Xpress-Optimizer

```
C:\> optimizer
XPRESS-MP Integer Barrier Optimizer Release xx.yy
(c) Copyright Dash Associates 1984-zzzz
Enter problem name >
Default problem name of $$$$$$$$ assumed
> QUIT

C:\>
```


Solving a Simple Problem with Xpress-Mosel

The Mosel File

Assuming that both Mosel and the Optimizer are running correctly, we can now use Console Xpress to input a model and to solve it. Throughout this chapter we shall largely be working with the simple model given in Listing 3.3, which has just two *decision variables* (`a` and `b`) and two *constraints* (`first` and `second`). The aim is to maximize the *objective function*, `profit`.

For the moment, since we will be concentrating only on *using* Mosel and the Optimizer, we will not discuss the *meaning* of the model or any of its constituent parts, although in this case it may already be clear. In Chapter 5 we will focus on the Mosel model programming language.

Problems such as this are presented to Mosel by way of model input files of the form described in Listing 3.3. These are ASCII text files describing the model syntax as well as any processing information which Mosel is expected to carry out. By default, Mosel files are assumed to have a `.mos` extension and this is the convention that we will adopt in this chapter.

Listing 3.3 A simple model

If you use a word processor to enter the model, make sure you save the file in "text only" format.

```
model simple
  uses "mmxprs"

  declarations
    a: mpvar
    b: mpvar
  end-declarations

  first:=  3*a + 2*b <= 400
  second:=   a + 3*b <= 200
  profit:=   a + 2*b

  maximize(profit)

  writeln("Profit is ", getobjval)
end-model
```


Exercise *Using a text editor, save the model program of Listing 3.3 into a file,* `simple.mos`.

The Three Pillars of Mosel

Solving our problem in Mosel is a three stage process which you will become very familiar with over the following pages. Firstly, the model program must be compiled, following which the compiled program is loaded into Mosel and finally it is run. The commands that Mosel provides to carry out this procedure are as follows:

Using the `-g` flag with `compile` provides additional information for debugging if errors are encountered.

compile

This instructs Mosel to compile a model program file prior to its use. Its single argument is the model's filename. If no extension is given, one of `.mos` will be assumed.

load

A compiled model program can be loaded into Mosel using the `load` command.

run

This instructs Mosel to run the current program. Any arguments following it are assumed to be initializing parameters.

An example of setting parameters in this way may be found in Chapter 5.

Exercise *Starting up Mosel, compile, load and finally run the model program* `simple.mos`*, but do not quit Mosel. What happens?*

A full session describing the process you should have just gone through is provided in Listing 3.4, with user input highlighted in bold.

Listing 3.4 Compiling, loading and running model programs

```
C:\Mosel Files>mosel
** Xpress-Mosel **
(c) Copyright Dash Associates 1998-zzzz
>compile simple
Compiling `simple'...
>load simple
>run
Profit is 171.429
Returned value: 0
>
```

"Compilation errors..."

During the first stage of this process, the model program stored in the file `simple.mos` is compiled into a binary model, or BIM file, `simple.bim`. If any syntax errors are detected in the model, error information is displayed to the console, with details of the line and character position where the error was detected and a description of the error, if available. Warning messages are also displayed in this manner, prefixed by a `W` rather than an `E`. By using the `-g` flag when compiling, extra information is provided which can help identify problems that are harder to spot, such as range errors. If no errors are detected, the compiled model file is subsequently loaded into Mosel.

Since the process of compilation and loading is one that is undergone so frequently, Mosel provides a shortened command for accomplishing both in one step.

Mosel commands can generally be shortened to two letters if there is no conflict with other commands. See the Mosel Reference Manual for details.

cload (cl)

This instructs Mosel to compile a model program and load the compiled binary model file. It may be shortened further to `cl` if required. Its single argument is the model program filename.

In the final stage the compiled program is run and the objective function maximized, as we asked. The optimum value of this is output to the console.

Obtaining Further Information From Mosel

Knowing the maximum amount of profit that we can make in our model is certainly useful, but how can it be achieved? Or rather, what values of the decision variables result in the optimum profit value? Such information could be obtained by altering our model to have the values of `a` and `b` printed to the screen, but if the problem was large and the solution process took a long time, recompiling and solving the problem again would not be an attractive prospect. Fortunately Mosel allows us to query the values of any of the variables or constraints through use of the `display` command.

display

Following this, Mosel will display the value of the object provided as its argument.

Exercise *Using* `display`*, obtain the optimal values of the decision variables. Check that these correspond to* `a = 114.286;b = 28.571`.

Going Further With Mosel

Running in Batch Mode

The full list of Mosel flags may be found by typing `mosel -h` at the command prompt.

One of the main uses of Console Xpress is in batch production systems for solving a large number of problems on a regular basis. While we have so far only detailed the use of Mosel in interactive mode, its use is not restricted to this: using the `-c` flag, Mosel may be run non-interactively from the command line. An example of this is provided in Listing 3.5, where our previous model file is compiled, loaded and run in this manner.

Listing 3.5 Running Mosel in batch mode

```
C:\Mosel Files>mosel -c "cload simple; run"
** Xpress-Mosel **
(c) Copyright Dash Associates 1998-zzzz
>cload simple
Compiling `simple'...
> run
Profit is 171.429
Returned value: 0
>
Exiting.

C:\Mosel Files>
```

Mosel expects a list of commands to follow the `-c` flag, contained in quotes and with commands separated by semi-colons (`;`). By adding the `-s` flag to the expression, Mosel runs silently, outputting only information requested in the model file.

Exercise *Run Mosel in batch mode both without and then with the* `-s` *flag. Note the difference in output between the two.*

Running Multiple Models

Another feature of Mosel enables several different models to be held in memory and worked on simultaneously. One of these is designated the *active problem* and by changing which problem has this status, any of them can be run as necessary. To demonstrate this, we will need another model, such as that given in Listing 3.6.

Listing 3.6 The altered model

The only changes in the model between this and Listing 3.3 is the imposition of a new constraint. You can expect the objective function value to decrease if this constraint is binding.

```
model altered
  uses "mmxprs"

  declarations
    a: mpvar
    b: mpvar
  end-declarations

  first:=  3*a + 2*b <= 400
  second:=   a + 3*b <= 200
  third:=  6*a + 5*b <= 800
  profit:=   a + 2*b

  maximize(profit)

  writeln("Profit is ", getobjval)
  writeln(" a = ", getsol(a), "; b = ", getsol(b))
end-model
```

Exercise *Using* `simple.mos` *as a template, create a new file,* `altered.mos`*, containing the text of Listing 3.6.*

As a model is loaded into Mosel, it becomes the active problem. Whenever the `run` command is called, it is the active problem that will be run, and when the `display` command is called, information about the active problem is returned.

Exercise *Starting up Mosel, load firstly* `simple.bim` *(it should already be compiled) and then compile and load the new problem, 'altered'. Using* `run`*, obtain a solution to the altered problem.*

A full list of all the loaded problems can be obtained by issuing the command `list` at the Mosel prompt. In our case, this will return details about the two problems, 'simple' and 'altered'. The active problem, in this case 'altered', has a star on the left against its name to denote this fact. The active problem can be changed using the `select` command, as shown in Listing 3.7.

Listing 3.7 Viewing and setting the active problem in Mosel

```
>list
* name: altered          number:   2 size: 3832
  Sys. com.: `altered.mos'
  User com.:
- name: simple           number:   1 size: 3176
  Sys. com.: `simple.mos'
  User com.:
>select simple
Model 1 selected.
>list
- name: altered          number:   2 size: 3832
  Sys. com.: `altered.mos'
  User com.:
* name: simple           number:   1 size: 3176
  Sys. com.: `simple.mos'
  User com.:
>
```

Exercise *List the problems currently loaded in Mosel and set 'simple' as the active problem. When you type* `run` *now, it should be this that is solved.*

Further models can be loaded into Mosel using `load` or `cload`. Models can also be removed from Mosel using `delete`. Note that use

of the `delete` command only unloads a model from Mosel and does *not* delete the model file.

list

This lists the problems currently loaded in Mosel. The active problem is distinguished with a star.

select

This allows a new problem to be selected as active. Its single argument is either the name or number of that to be set as the active problem.

delete

Deletes a problem from Mosel.

Exercise *Using* `delete altered`*, unload the new problem from Mosel. Using* `list`*, check that only 'simple' is left loaded.*

Writing Matrix Files

Occasionally a model must be solved and its matrix subsequently altered, requiring more interaction with the user than that provided by Mosel. On the other hand, perhaps the same problem may need to be given to someone else to use. In such cases, a model matrix file is required from Mosel so that the problem can be input into another program. Mosel supports writing matrix files in both MPS and LP format using the `exportprob` command from the Mosel prompt.

`exportprob` can also be used from within the model. For an examples, see Listing 3.11.

exportprob

Used on its own this prints the matrix to the console in LP format. With the `-m` flag, the MPS format is output. By adding a filename as an argument, the matrix is exported to the named file.

Exercise *Using*

```
exportprob -m simple.mat
```

generate the matrix file for the problem 'simple' in MPS format. View the file using a text editor.

Unlike the MPS format, LP files also contain information about whether the model describes a maximization or minimization problem. By default it is assumed that the objective function is to be minimized unless stated otherwise. To export this problem in LP format, the `-p` flag must be used to indicate that the objective function is to be maximized.

Exercise *Output the matrix file for the problem 'simple' to the console in LP format. Check that the matrix has been specified correctly as referring to a maximization problem.*

Working with the Xpress-Optimizer

Solving a Problem

In the exercises above you will have entered model programs into Mosel and called the Optimizer as a library module to solve the problem. The Optimizer can also be used as a stand-alone application, however, and use of it in this way forms the basis for the remainder of the chapter.

The Optimizer accepts problem matrix files as input in either of the main industry standard formats: MPS or LP files. To attempt the exercises in this chapter you will need a valid matrix file, such as the MPS file produced above, `simple.mat`. With this file, all that need be done to solve the problem is to load the completed matrix into the Optimizer and maximize the objective function.

Loading an MPS File into the Optimizer

Start `optimizer` as before and at the prompt enter the problem name, *problem_name*. At the following prompt issue the Optimizer command `READPROB`. Hit return and the Optimizer reads in the file *problem_name*`.mat`.

Maximizing the Objective Function

Once an MPS file has been loaded, issue the Optimizer command `MAXIM` at the following prompt. The objective function is maximized.

You would type `MINIM` instead if this were a minimization problem.

A full session with the Optimizer is given in Listing 3.8, again with user input highlighted in bold face. Having entered the problem name, the command `READPROB` is used to read in the MPS file created by Mosel and then `MAXIM` is called to maximize the objective function: the expression labelled `profit` in Listing 3.3.

Exercise *Following Listing 3.8, load the file* `simple.mat` *into the Optimizer and maximize the objective function. Issue the command* `WRITEPRTSOL` *before quitting.*

Viewing the Solution

Solution information from the Optimizer is output in a number of different ways which we briefly discuss. The output sent to the screen by the `MAXIM` command in Listing 3.8 shows on the lines above the `WRITEPRTSOL` command that an optimal solution to the problem has been found, giving a value of `171.428571` to the objective function. This is our first indication of the solution, along with a set of problem statistics, which are output as the matrix is loaded.

The problem statistics provide a useful sanity check for interpreting the solution. It is from these that we may check that the matrix loaded actually corresponds to the problem that we want to solve. In the current example, Listing 3.8 tells us that the matrix has three rows, corresponding to the two constraints and one objective function. Furthermore, it has two columns, corresponding to the two decision variables. Since both decision variables feature in all three of the rows, we might expect the number of nonzero elements in the matrix

Listing 3.8 A first full session with the Optimizer

```
C:\Optimizer Files> optimizer
XPRESS-MP Integer Barrier Optimizer Release xx.yy
(c) Copyright Dash Associates 1984-zzzz
Enter problem name >simple
>READPROB

Reading Problem simple
Problem Statistics
           3 (      0 spare) rows
           2 (      0 spare) structural columns
           6 (      0 spare) non-zero elements
Global Statistics
          0 entities       0 sets       0 set members
>MAXIM
Presolved problem has: 2 rows 2 cols 4 non-zeros
Crash basis containing 0 structural columns created

   Its        Obj Value  S   Ninf  Nneg    Sum Inf  Time
     0          .000000  p      0     0    .000000     1
     2       171.428571  P      0     0    .000000     1
Uncrunching
     2       171.428571  P      0     0    .000000     1
Optimal solution found
>WRITEPRTSOL
>QUIT

C:\Optimizer Files>
```

The Optimizer is not case-sensitive, so commands may be issued as lower case if desired. We will use upper case here to reflect the descriptions in the reference manual.

to be $3 \times 2 = 6$. Finally, from the global statistics we learn that ours is not a MIP problem: it contains no entities, no (special ordered) sets and consequently no set members.

'Global entities' is used as an umbrella term in Xpress-MP to describe non-continuous variables and special ordered sets.

The Optimizer also produces more detailed output which may be accessed by the user. In Listing 3.8, we have written this to the file `simple.prt` using the `WRITEPRTSOL` command. The same output may be sent to the screen using the `PRINTSOL` command.

Outputting a Full Solution

After optimizing the objective function, use either the PRINTSOL command to send output to the screen, or WRITEPRTSOL to send it to the ASCII solution print file *problem_name*.prt for later viewing

Output from WRITEPRTSOL

The output from WRITEPRTSOL is an ASCII text file suitable for sending to a printer. This file may be naturally split up into three sections to understand its structure, as demonstrated in Listing 3.9.

Listing 3.9 Output from the WRITEPRTSOL command

```
Problem Statistics
Matrix simple                                                    (Section 1)
Objective *OBJ*

RHS *RHS*
Problem has      3 rows and      2 structural columns

Solution Statistics
Maximization performed
Optimal solution found after      2 iterations
Objective function value is   171.428571


Rows Section                                                     (Section 2)
   Number Row    At    Value      Slack Value Dual Value  RHS
 N   1    *OBJ*  BS  171.428571  -171.428571  .000000      .000000
 L   2    second UL  200.000000      .000000  .571429   200.000000
 L   3    first  UL  400.000000      .000000  .142857   400.000000

Columns Section                                                  (Section 3)
   Number Column At    Value      Input Cost   Reduced Cost
 C   4     a     BS  114.285714     1.000000  .000000
 C   5     b     BS   28.571429     2.000000  .000000
```

"Section 1..." The sections are perhaps most usefully considered in the order Section 1, Section 3, Section 2. Section 1 contains summary statistics about the solution process. It gives the matrix (problem) name (`simple`) and the objective function and right hand side names that have been used. Following this are the number of rows (constraints including the objective function) and columns (variables), the fact that it was a maximization problem, that it took two iterations (steps) to solve, and that the best solution has a value of `171.428571`.

"Section 3..." The optimal values of the decision variables are given in Section 3, the *Columns Section*. The *Value* column gives the optimal value of each variable. Additional solution information such as the Input Cost and Reduced Cost for each of the variables is also provided here. See Chapter 7, "Glossary of Terms" for details of these.

"Section 2..." The final section to consider is Section 2, the *Rows Section*. This gives a list of the various constraints and objective function of the problem, along with the value of the "left hand side" of the expression. In particular, the value of the objective function turns up here again. Slack values and Dual values for the constraints can also be obtained from here. See Chapter 7, "Glossary of Terms" for details of these.

Exercise *View the output file* `simple.prt` *which you have just created and identify the various sections. Check your results for the decision variables and objective function with those in the listing.*

Going Further with the Optimizer

Optimization Algorithms

The Optimizer supports a number of different algorithms for solving LP problems, although by default the *dual simplex* algorithm is used. For some problems, however, changing the algorithm may result in a considerable change in the time taken to solve a problem, and either of the *primal simplex* or *Newton barrier* algorithms may produce a

solution in a shorter time. Choosing the 'best' algorithm to use in a given situation is something of an art in itself, but often the best choice is evident from experimenting with the different algorithms on your own problems.

"Choosing an algorithm..."

The simplest way to change the algorithm which will be used on a problem is by setting flags which can be passed to `MAXIM`. These correspond to each of the three algorithms discussed above:

- `-b` use the Newton barrier algorithm;
- `-d` use the dual simplex algorithm;
- `-p` use the primal simplex algorithm.

Listing 3.10 demonstrates the changes from Listing 3.8 to use the Newton barrier method for solving the problem. Only those lines which involve user input are shown here.

Listing 3.10 Using the Newton barrier algorithm

```
C:\Optimizer Files> optimizer
Enter problem name >simple
>READPROB
>MAXIM -b
>WRITEPRTSOL
>QUIT

C:\Optimizer Files>
```

Exercise *Try solving the problem using each of the three algorithms and note the difference in the output produced by each.*

Integer Programming

For the problem considered above, the maximal value of the objective function occurred when the decision variables took values `a = 114.3` and `b = 28.6`, to one decimal place. For some problems, however, fractional solutions such as this may be unacceptable and one or more of the decision variables may be constrained to take only integer values. Integer and mixed integer problems of this kind may also be

solved using Console Xpress and before we end this brief introduction we will consider such a problem

In Listing 3.11, two extra lines have been added to the previous model to specify that `a` and `b` must take integer values. The other changes allow Mosel to export an MPS matrix file 'automatically' when the program is run.

Listing 3.11 Adding integer constraints to our problem

```
model integral
  declarations
    a: mpvar
    b: mpvar
  end-declarations

  first:=  3*a + 2*b <= 400
  second:=   a + 3*b <= 200
  profit:=   a + 2*b

  a is_integer
  b is_integer

  exportprob(EP_MPS,"integral",profit)
end-model
```

Exercise *Create a model file integral.mos containing the statements of Listing 3.11 and run it in Mosel to write an MPS matrix file. Using the Optimizer, solve the LP problem as before. What is the optimal value of the objective function now?*

The problem just solved specified that both `a` and `b` must be integers. Furthermore, the objective function is a linear expression containing only integer multiples of these variables. However, you probably noticed that the optimum value of the objective function just obtained is not integral!

The LP relaxation is the previous problem, ignoring the integrality of any variables.

Finding integer solutions using the Optimizer is a two-stage process. For the first of these, the LP *relaxation* must be solved to find an

optimal solution. For the second, a *global search* must be carried out to find the best integer solution, if one exists. The global search can be called automatically following solution of the LP relaxation by including the `g` flag to `MAXIM` and this is demonstrated in Listing 3.12.

Listing 3.12 Commands to solve the integer problem

```
READPROB
MAXIM -g
WRITEPRTSOL
QUIT
```

`MAXIM -g` is equivalent to calling `MAXIM`, immediately followed by the command `GLOBAL`.

Starting the Global Search for Integer Solutions

Issue the command `MAXIM -g` at the `optimizer` prompt. The *Branch and Bound* method is then employed to find an integer solution to the problem.

When the global search completes, any integer solution found may be viewed using `PRINTSOL` or `WRITEPRTSOL` as previously.

Exercise *Solve the integer problem of Listing 3.11. What values of* `a` *and* `b` *are required for the optimal solution? (They should now be integral.)*

Optimizer Controls

The Optimizer has a number of parameters, whose values may be altered by the user through the setting of various *controls*. With some thought, these can be used to fine tune the Optimizer's performance on particularly difficult problems and can result in significant savings in the solution time. Whilst we do not encourage new users to change the majority of these from their default values, there are a small number with which experimentation may be useful. For the sake of these, we briefly mention how this may be achieved. A full list of all controls which can be set may be found in the Optimizer Reference Manual.

Obtaining the Value of Controls

The value of a control for the Optimizer may be obtained by typing its name at the `optimizer` prompt:

```
> CONTROL
```

Changing the Value of Controls

Controls for the Optimizer may be treated much like standard programming variables and changed through an assignment at the prompt:

```
> CONTROL = value
```

"Changing the algorithm used by default..."

We saw earlier how the algorithm used for optimization may be changed from its default using flags to `MAXIM`. The default algorithm itself is set by the control `DEFAULTALG` which is an integer describing which algorithm should be applied in the following way:

Changing the Default Optimization Algorithm

Set the control variable `DEFAULTALG` to:

`1`	for the algorithm to be determined automatically;
`2`	to use the dual simplex algorithm;
`3`	to use the primal simplex algorithm;
`4`	to use the Newton Barrier algorithm.

The default algorithm may be changed for a complete session using `DEFAULTALG`, removing the need for `p`, `d` and `b` flags to `MAXIM` (or `MINIM`).

Exercise *Issue the control* `DEFAULTALG` *at the* `optimizer` *command prompt to find out its current value as in Listing 3.13.*

Exercise *Set the value of the control* `DEFAULTALG` *to* `4` *so as to use the Newton barrier algorithm. This must be done before the call to* `MAXIM` *which is affected by the value of this variable. Run the program again and view the output.*

Listing 3.13 Obtaining the value of controls

```
C:\Optimizer Files> optimizer
XPRESS-MP Integer Barrier Optimizer Release xx.yy
(c) Copyright Dash Associates 1984-zzzz
Enter problem name >simple
>READPROB
>DEFAULTALG
1
>
```


Getting Help

If you have worked through the chapter this far, you have learnt how to use Mosel to compile, load and run model program files, to access the solution to a problem using the Optimizer as a library module within Mosel and to export problems as matrix files. You have also learnt how to load these files into the Optimizer, to optimize them, view the solution, and change controls affecting how the optimization works. Up to this point we have been concentrating solely on how to enter and optimize a particular existing model. In Chapter 5, however, the basics of the Mosel model programming language are explained, allowing you to model and solve your own problems. You may wish to turn there next. The following chapter contains information about using the Xpress-MP Libraries and may be safely skipped if you intend only to use Xpress-IVE or Console Xpress. If you require additional help on using either Mosel or the Optimizer, this may be obtained from the Mosel and Optimizer Reference Manuals respectively.

Summary

In this chapter we have learnt how to:

- compile, load and run models using Mosel;
- export problem matrix files;
- solve linear problems and access the solutions;
- solve integer problems;
- change the solution algorithm;
- access and alter the values of Optimizer controls.

Chapter 4
The Xpress-MP Libraries

Overview

In this chapter you will:

- meet the different Xpress-MP libraries;
- learn how to compile, load and solve problems written in the Mosel language using the Mosel libraries;
- learn how to build and solve models using Xpress-BCL;
- learn how to load and solve problems using the Xpress-Optimizer library;
- learn about using the libraries with different programming languages.

Getting started

The Xpress-MP Libraries provide an interface to the Xpress-Mosel and Xpress-Optimizer engines, allowing users to call Xpress-MP routines from within their own programs. If you are reading this chapter, we shall assume that you have already installed the Xpress-MP Libraries on your system. Over the course of this chapter we will input and solve a simple problem using the subroutine libraries, much as we did for the Xpress-IVE and Console Xpress chapters, as well as exploring other possibilities. You may find it helpful to work through the Console Xpress chapter before reading this, although it will not be assumed that you have done so.

The Components of the Xpress-MP Libraries

Xpress-MP essentially comprises three main components — Mosel, BCL and the Optimizer. The Xpress-MP Libraries constitute an interface to these, providing access to them in a number of different ways. Whilst there are several such libraries available to users, only the three of direct relevance to these will be discussed in any detail in this chapter:

- The Mosel Libraries allow you to compile, load and run model files, directly accessing the Optimizer as a module to solve multiple problems. Moreover, they enable you to take full advantage of the flexible Mosel model programming language within your application code;
- the Xpress-MP Builder Component Library (Xpress-BCL) works slightly differently, providing a separate modeling environment to Mosel, allowing models to be constructed within the user's program, either generating an output file, or passing the matrix directly to the Optimizer;
- the Optimizer Library provides for the input, manipulation and optimization of matrix files, giving access to the solution in a number of ways.

Over the following sections each of these libraries will be introduced, before information of general interest to library users is given. For the most part, examples in this chapter will be given in the C programming language, which is widely used in calling the Xpress-MP libraries and is consequently the language on which the Reference Manuals are based. However, other languages supported include C++, Visual Basic and Java, and the chapter ends with an overview of using the libraries with the last two of these.

Working with the Mosel Libraries

Entering a Simple Model

To begin the chapter, we will discover how the Mosel libraries can be used to input and solve the same simple model that has been used previously. This model, given in Listing 4.1, is written in the Mosel model programming language and has just two decision variables (`a` and `b`) and two constraints (`first` and `second`). The aim is to maximize the objective function, `profit`.

Listing 4.1 A simple model

```
model simple
  uses "mmxprs"

  declarations
    a: mpvar
    b: mpvar
  end-declarations

  first:=  3*a + 2*b <= 400
  second:=   a + 3*b <= 200
  profit:=   a + 2*b

  maximize(profit)

  writeln("Profit is ", getobjval)
end-model
```

Problems such as this are presented to the Mosel libraries by way of model input files. These are ASCII text files describing the model as well as any processing information which Mosel is expected to carry out. By default, Mosel files are assumed to have a `.mos` extension and this is the convention that we will adopt in this chapter.

Exercise *Enter the model of Listing 4.1 into a file* `simple.mos` *using your favorite text editor. You may have already done this if you worked through the exercises in the Console Xpress chapter previously.*

Components of the Mosel Libraries

The Mosel libraries consist of a run time library, `xprm_rt`, and a compiler library, `xprm_mc`. Of the two, the run time library is the main one, containing routines for initialization and termination of Mosel, prior to and following use. The compiler library, by contrast, contains only a single routine, used for the compilation of Mosel model program files. In general, the run time library will often be used on its own. On the occasions when the compiler library is also used, both header files must be included in your code and the initialization routine from the run time library must be called before the compiler routine is used. In this section we will demonstrate use of both of these libraries, resulting in the following general structure for a Mosel library program:

Listing 4.2 Structure of a Mosel library program

```
#include "xprm_rt.h"
#include "xprm_mc.h"

int main(void)
{
  XPRMinit();

  program statements

  XPRMfree();
  return 0;
}
```

The Three Pillars of Mosel

Solving a problem using the Mosel library is a three stage process which you will become very familiar with over the following pages.

The model program, once written, must first be compiled, following which the compiled program is loaded into Mosel and finally it is run. There are three routines provided by the Mosel libraries for these purposes:

XPRMcompmod

This calls Mosel to compile a model file into a *BI*nary *M*odel (or BIM) file. It takes four arguments:

- the first includes options for the compilation process;
- the second is the name of the model source file;
- the third is the name of the destination file and may be `NULL`;
- the fourth contains any commentary text to be saved at the beginning of the output file.

This is the only command in the model compiler library.

Using a '`g`' flag with `XPRMcompmod` provides additional information for debugging if errors are encountered. This is particularly useful with range errors, which are difficult to spot.

XPRMloadmod

This instructs Mosel to load a binary model file and takes two arguments:

- the first is the name of the file;
- the second is the model's internal name and may be `NULL`.

`XPRMloadmod` returns a problem pointer, a variable of type `XPRMmodel`.

XPRMrunmod

Runs the model within Mosel. Its three arguments are:

- the problem pointer;
- a pointer to an area where the result value is returned;
- a string of parameter initializations, which may be `NULL`.

An example of setting parameters in this way may be found in Chapter 5.

Listing 4.3 shows how these may be used to compile, load and run the model saved earlier.

Exercise *Write and run a program to compile, load and run the model* `simple.mos`*. What is the maximum profit obtainable in our model?*

Listing 4.3 Compiling, loading and running model 'simple'

```
#include <stdio.h>
#include "xprm_rt.h"
#include "xprm_mc.h"

int main(void)
{
  XPRMmodel simple;
  int nReturn;

  XPRMinit();
  XPRMcompmod("","simple.mos",NULL,"Simple Example");
  simple = XPRMloadmod("simple.bim",NULL);
  XPRMrunmod(simple,&nReturn,NULL);

  XPRMfree();
  return 0;
}
```

If you are experiencing difficulties with this exercise, it may be that the Mosel libraries cannot find important security system files that it needs, or that it cannot locate other libraries / DLLs. For details of how to set up the Xpress-MP security system, see the 'readme' files on the CD-ROM, which also contain details of setting up the libraries with the most commonly available compilers.

Obtaining Solution Information

Part of the model in Listing 4.1 called upon Mosel to output the maximal value of the objective function once it had been found. In the same way, the model could have been written to output the optimal values of the decision variables, or other such information and we will see examples of this later. However, the Mosel run time library provides a number of functions for obtaining details of the solution such as this, allowing solution information to be used directly in your application. Two such functions are the following:

XPRMgetobjval

Returns the objective function value to double precision. Its single argument is the problem pointer.

XPRMgetvsol

Returns the value of a decision variable to double precision. It takes two arguments:

- the problem pointer;
- a reference to a decision variable.

Returning the objective function value is the simpler of these two. Obtaining values of decision variables is a somewhat more involved process, in which the dictionary entry corresponding to the particular variable name must first be returned by Mosel, using the library function `XPRMfindident`. This returns a generic type, which must be cast as an `XPRMmpvar` variable before finally being employed by `XPRMgetvsol` to return the value. In Listing 4.4 we demonstrate both these functions and their use, with changes from Listing 4.3 highlighted in a bold type face.

Listing 4.4 Obtaining solution information

```
#include <stdio.h>
#include "xprm_rt.h"
#include "xprm_mc.h"

int main(void)
{
  XPRMmodel simple;
  XPRMalltypes atvar;
  XPRMmpvar a, b;
  int nReturn;

  XPRMinit();
  XPRMcompmod("","simple.mos",NULL,"Simple Example");
  simple = XPRMloadmod("simple.bim",NULL);
  XPRMrunmod(simple,&nReturn,NULL);
```

Listing 4.4 Obtaining solution information

```
   XPRMfindident(simple,"a",&atvar);
   a = atvar.mpvar;
   XPRMfindident(simple,"b",&atvar);
   b = atvar.mpvar;

   printf("The maximal profit is %g\n",
      XPRMgetobjval(simple));
   printf("a = %g\nb = %g\n",
      XPRMgetvsol(simple,a), XPRMgetvsol(simple,b));

   XPRMfree();
   return 0;
}
```

Exercise *Alter your previous program to obtain the optimal values of the decision variables,* `a` *and* `b`*.*

The Mosel run time library contains a large number of other functions, similar in nature to `XPRMgetvsol`, for returning the reduced costs (`XPRMgetrcost`), slack values (`XPRMgetslack`) and dual values (`XPRMgetdual`). Details of using these may be found in the Mosel Reference Manual.

Going Further with the Mosel Libraries

Working with Several Models

Users of Xpress-MP often find that they need to develop a number of different models in parallel, regularly alternating between them in their applications. The Mosel libraries allow any number of models to be held in memory and worked on simultaneously, limited only by the constraints of system resources and your license details. As with all Xpress-MP products, model management in the Mosel libraries is

based on the concept of a problem pointer which distinguishes between the models currently loaded and allows you to keep track of which is being worked on at any given instance.

A problem pointer, or reference, is returned by the `XPRMloadmod` routine as we have already seen in the previous exercises and listings. It is this which is passed as the first argument to most of the routines which act on a model, so unlike Console Mosel, there is no concept of an *active model* in this case.

Exercise *Enter the model of Listing 4.5 into a file* `altered.mos` *and adapt your previous program to compile, load and run both 'simple' and 'altered' consecutively.*

Listing 4.5 An altered model

The only changes in the model between this and Listing 4.1 is the imposition of a new constraint. You can expect the objective function value to decrease if this constraint is binding.

```
model altered
  uses "mmxprs"
  declarations
    a, b: mpvar
  end-declarations

  first:=  3*a + 2*b <= 400
  second:=   a + 3*b <= 200
  third:=  6*a + 5*b <= 800
  profit:=   a + 2*b

  maximize(profit)

  writeln("Profit is ", getobjval)
end-model
```

During use, Mosel maintains a list of the models currently in memory, placing new models at the beginning of the list as they are loaded. Models can be selected sequentially from this list using the function `XPRMgetnextmod`, providing a convenient method for cycling through a large number of models. Information about a given model

can be retrieved using `XPRMgetmodinfo`. Once a model is no longer needed, it can be unloaded to free resources using `XPRMunloadmod`.

XPRMgetmodinfo

Returns information about the model specified by a given problem pointer. It takes five arguments following the problem pointer, which are also pointers:

- the first to where the model name may be returned;
- the second to where the model number is returned;
- the third to where system comment is returned;
- the fourth to where user comment is returned;
- the fifth to where the amount of memory used is returned.

Any of these five may be `NULL` if not required.

XPRMgetnextmod

Returns the pointer to the next model in the list after the one given as an argument. If the argument is `NULL`, the first model in the list is returned. If the argument is the last model in the list, `XPRMgetnextmod` returns `NULL`.

XPRMunloadmod

Instructs Mosel to unload a model and free any memory or resources associated to that model. Its single argument is the problem pointer.

Whilst these functions would not normally be used with just two models, Listing 4.6 provides an example of their use which might easily be extended to cover a larger number of models.

Listing 4.6 Working with multiple models

```
#include <stdio.h>
#include "xprm_rt.h"
#include "xprm_mc.h"             /* if needed */

int main(void)
{
  XPRMmodel mod;
  int nReturn;
  const char *name;

  XPRMinit();

  /* uncomment the following two lines if needed */
/*  XPRMcompmod("","simple.mos",NULL,"");
  XPRMcompmod("","altered.mos",NULL,""); */

  XPRMloadmod("simple.bim",NULL);
  XPRMloadmod("altered.bim",NULL); /*first in list*/

  /* return a pointer to the first item in list */
  mod = XPRMgetnextmod(NULL);

  while(mod != NULL)
  {
    XPRMgetmodinfo(mod,&name,NULL,NULL,NULL,NULL);
    printf("\nCurrent problem is '%s'\n", name);
    XPRMrunmod(mod,&nReturn,NULL);
    mod = XPRMgetnextmod(mod);
  }

  /* unload first item in list, i.e. 'altered' */
  XPRMunloadmod(XPRMgetnextmod(NULL));

  XPRMfree();
  return 0;
}
```

Exercise *Enter the and run the program of Listing 4.6. Since you will have compiled both 'simple' and 'altered' in the last exercise, you will not need to include the compiler library* `xprm_mc` *here, and can omit the two lines using* `XPRMcompmod`*.*

Run Time Management

Once a model is loaded, it is run using the `XPRMrunmod` routine. A user can check to see if a given model is still running at any time using `XPRMisrunmod` and a run can be terminated using `XPRMstoprunmod`.

XPRMisrunmod

This checks if a model is still running. Its single argument is the problem pointer and it returns `1` if the model is running and `0` otherwise.

XPRMstoprunmod

This interrupts the run of a model. Its single argument is the problem pointer.

These two routines are frequently used together in applications with large modeling problems which are likely to take a long time to solve. Listing 4.7 provides an example of this, where the 'Ctrl-C' key combination can be used to interrupt a model in the middle of a run.

Listing 4.7 Ending a model run cleanly

```
#include <stdio.h>
#include <signal.h>
#include "xprm_rt.h"

XPRMmodel mod;

void end_run(int sig)
{
  if(XPRMisrunmod(mod)) XPRMstoprunmod(mod);
}
```

Listing 4.7 Ending a model run cleanly

```
int main(void)
{
  int nReturn;

  XPRMinit();
  mod = XPRMloadmod("bigmodel.bim",NULL);
  signal(SIGINT,end_run); /* Redirect Ctrl-C */

  XPRMrunmod(mod,&nReturn,NULL);
  XPRMfree();
  return 0;
}
```

Writing Matrix Files

Occasionally a model must be solved and its matrix subsequently altered, requiring more interaction with the user than that provided by Mosel. On the other hand, perhaps the same problem may need to be passed to someone else for use. In such cases, a matrix file is required from Mosel so that the problem can be input into another program. The Mosel libraries support writing of matrix files in both MPS and LP format using the `XPRMexportprob` routine.

XPRMexportprob

This instructs Mosel to write a matrix file in MPS or LP format. It takes four arguments:

- the problem pointer;
- format of the output, which can be one of `"m"` for MPS format, `""` for LP format (minimization), `"p"` for LP format (maximization) or `"s"` for scrambled names;
- a filename for the output — if `NULL`, the output is printed to the console;
- the objective to use, which may be `NULL`.

Exercise *Alter your model to write an MPS matrix file,* `simple.mat`*, for the problem. Now generate an appropriate LP format matrix file,* `simple.lp`*.*

Use of `XPRMexportprob` to create an MPS file is demonstrated in Listing 4.8. Compare this with what you have just done.

Listing 4.8 Generating an MPS matrix file

```
#include <stdio.h>
#include "xprm_rt.h"

int main(void)
{
  XPRMmodel simple;
  int nReturn;

  XPRMinit();
  simple = XPRMloadmod("simple.bim",NULL);
  XPRMrunmod(simple,&nReturn,NULL);
  XPRMexportprob(simple,"m","simple",NULL);
  XPRMfree();
  return 0;
}
```

If no filename extension is given, MPS files will have `.mat` appended and LP files will have `.lp` appended.

Having got this far, you should now be able to use the Mosel libraries to compile, load and solve model programs, access their solution values from your application, handle multiple problems and export matrix files in standard formats. In the next chapter we will learn more about the Mosel model programming language itself, enabling you to create your own models to solve in this way. In the following section, however, we will see how the same tasks as those above may be achieved using BCL. If you will not be using BCL, you may want to skip to page 74 where the Optimizer library will be introduced, view the material contained in the section "Getting the Best out of the Xpress-MP Libraries" on page 95, or find out more about the Mosel language in Chapter 5.

Working with Xpress-BCL

We have already seen how the Mosel libraries can be used to compile, load and run model program files, invoking the Optimizer to find a solution. Whilst this will often suffice for most needs, sometimes you might want more flexibility, creating models directly from within your own programs. There are essentially three ways in which this might be done:

- use the program to create a model file and compile, load and run it using the Mosel libraries;
- use BCL to build the model dynamically;
- use the program to construct the model locally and load it directly into the Optimizer with the Optimizer library.

The first of these is related to the material that has gone before and will not be considered further. The second is the subject of this section and the third will be discussed in the material on the Optimizer library following.

The Xpress-MP Builder Component Library (Xpress-BCL) provides a different library environment in which models may be created. BCL programs may be relatively simple, geared toward building a specific type of model, as we will do in the next few pages. More usually, however, BCL is used for the development of complete software systems containing mathematical programming models and optimization.

Completed models may be output by BCL in a number of file formats. Alternatively, they may be loaded directly into the Optimizer and either solved directly using BCL, or manipulated further using Optimizer library commands. Each of these possibilities will be discussed.

As in previous sections and chapters, the approach here will again be to use BCL to build the simple model described in Listing 4.1 on page 49. You may therefore find it helpful to keep a finger in that page so you can refer back to the model as necessary.

A First Formulation for the Model

To build our simple model using BCL requires a translation of the model statements into the Builder language. We begin this in Listing 4.9.

Listing 4.9 Declaration of structures and initialization

```
XPRBvar a, b;
XPRBctr first, second, profit;
XPRBprob prob;

XPRBinit("");
prob = XPRBnewprob("simple");
```

There are just five simple statements here, declaring the decision variables, constraints and problem pointer, initializing the library and creating the problem.

Although none of the examples so far have shown checking of return codes, we recommend that such checks always be carried out. See "Error Checking" on page 95 for details.

XPRBinit

Initializes the BCL environment. Its single argument may be used to specify the path used in searching for the Xpress-MP security files. If it is a null string, the default search paths are used. The return code from this should always be checked: a nonzero value indicates failure and the program should terminate.

XPRBnewprob

Creates a new problem and sets its name. It takes a single argument which is the problem name and returns a pointer to the problem created, a variable of type `XPRBprob`.

Following declaration, adding the variables is a relatively simple task, although more information is required than is evident from Listing 4.1. We must additionally set the type and bounds, as in Listing 4.10.

Listing 4.10 Adding the decision variables

```
a = XPRBnewvar(prob,XPRB_PL,"a",0,200);
b = XPRBnewvar(prob,XPRB_PL,"b",0,200);
```

The bounds here are derived from the constraint `Second`, although lower values may easily be deduced.

XPRBnewvar

Adds a single variable. This takes five arguments:

- the first argument is the problem pointer;
- the second argument defines the type — `XPRB_PL` means that it is a continuous, non-negative variable;
- the third argument is the name as it appears in the output file and this is also the name which will later be passed to the Optimizer;
- the last two specify the lower and upper bounds for the variable. The latter may be set to infinity if there is no bound.

It returns a variable of type `XPRBvar`.

With the decision variables defined, the constraints may be added, in this case incrementally, as in Listing 4.11.

Listing 4.11 Incrementally adding constraints

```
first = XPRBnewctr(prob,"First",XPRB_L);
XPRBaddterm(prob,first,a,3);
XPRBaddterm(prob,first,b,2);
XPRBaddterm(prob,first,NULL,400);
```

XPRBnewctr

Creates a new constraint. This takes two arguments following the problem pointer, the first of which is the constraint name, and the second of which is the constraint type. `XPRB_L` means it is a $\leq$ constraint. It returns a variable of type `XPRBctr`.

XPRBaddterm

Adds a term to the constraint. This takes three arguments following the problem pointer:

- the first is a reference to a constraint, as resulting from `XPRBnewctr`;
- the second is a variable, or `NULL` to add a constant to the right hand side;
- the third is the value of the coefficient, or the right hand side if the second argument is `NULL`.

Thus the second line in Listing 4.11 adds `3*a` to the constraint named `First`.

The objective function is added in much the same way as the constraints without setting the right hand side term, however. It must then be specified as the objective function using the `XPRBsetobj` command. We demonstrate this in Listing 4.12.

Listing 4.12 Adding the objective function

The row-type for objective functions is typically 'unconstrained', `XPRB_N`.

```
profit = XPRBnewctr(prob,"Profit",XPRB_N);
XPRBaddterm(prob,profit,a,1);
XPRBaddterm(prob,profit,b,2);
XPRBsetobj(prob,profit);
```

Solving the Problem with BCL

With the model constructed, the next task is to find a solution. In common with Mosel, BCL also contains functionality for loading problems into the Optimizer, solving them and accessing the result. The function `XPRBmaxim` may be used to call the Optimizer to find a solution. Following this, the two functions `XPRBgetobjval` and `XPRBgetsol` can be used to return the optimal value of the objective function and values of the decision variables respectively. Finally, standard print functions may be used to display these values to the screen.

XPRBmaxim

Calls the Optimizer to solve the problem. It takes two arguments, the first of which is the problem pointer and the second is a string of option flags.

XPRBgetobjval

Returns the optimal objective function value.

XPRBgetsol

Returns the optimal values for the decision variables. It takes two arguments, the first of which is the problem pointer and the second is the variable whose value is required.

Putting all this together, a complete description of the model can be found in Listing 4.13.

Listing 4.13 A first formulation for the model

The `xprb.h` header file contains all the definitions for the BCL functions.

```
#include <stdio.h>
#include "xprb.h"

int main(void)
{
  double objval;
  XPRBvar a, b;
  XPRBctr first, second, profit;
  XPRBprob prob;

  if(XPRBinit("")) return 1;
  prob = XPRBnewprob("simple");

  /* adding the variables */
  a = XPRBnewvar(prob,XPRB_PL,"a",0,200);
  b = XPRBnewvar(prob,XPRB_PL,"b",0,200);
```

Listing 4.13 A first formulation for the model

```
   /* adding the constraints */
   first = XPRBnewctr(prob,"First",XPRB_L);
   XPRBaddterm(prob,first,a,3);
   XPRBaddterm(prob,first,b,2);
   XPRBaddterm(prob,first,NULL,400);

   second = XPRBnewctr(prob,"Second",XPRB_L);
   XPRBaddterm(prob,second,a,1);
   XPRBaddterm(prob,second,b,3);
   XPRBaddterm(prob,second,NULL,200);

   profit = XPRBnewctr(prob,"Profit",XPRB_N);
   XPRBaddterm(prob,profit,a,1);
   XPRBaddterm(prob,profit,b,2);
   XPRBsetobj(prob,profit);

   XPRBmaxim(prob,"");

   /* retrieving the solution */
   objval = XPRBgetobjval(prob);
   printf("The maximum profit is %f\n", objval);
   printf(" a = %f, b = %f\n",XPRBgetsol(prob,a),
      XPRBgetsol(prob,b));

   XPRBdelprob(prob);
   XPRBfree();
   return 0;
}
```

The final two commands remove the problem from memory and free other system resources. It is important to make sure that these are called at the end of every program.

XPRBdelprob

Removes a problem from memory. Its single argument is the problem pointer.

XPRBfree

Frees other system resources currently being used.

Exercise *Type in the BCL program of Listing 4.13 to enter and solve the simple problem of Listing 4.1, returning the solution values to screen.*

By default, no program output is provided by BCL unless it is written into the application explicitly. This can be advantageous if we only want to return the values of the decision variables and the objective function, as above, but there are times when you might want to know more about how the optimization is progressing before a solution is returned. In such circumstances, console output in the form of a log file may be provided by increasing the *message level* from `0`. When set to `3`, BCL outputs all messages, including any passed to it by the Optimizer, giving access to the problem statistics and the objective function value at various points during the solution process. If this is required, the message level must be set before the problem is input.

Exercise *Add the line*

```
XPRBsetmsglevel(prob,3);
```

to the above program directly following the `XPRBnewprob` *command and rebuild the program. Run it and view the output.*

Formulating the Problem Using Array Variables

The incremental nature of building up the program, employed above, can result in programs which are unnecessarily lengthy. However, an alternative approach can be employed, making the program more concise, by framing the model in terms of variable arrays. This we now demonstrate.

We begin the description by declaring a new array variable, `x`, shown in Listing 4.14.

Listing 4.14 Array variables

```
XPRBarrvar x;
    ...
x = XPRBnewarrvar(prob,2,XPRB_PL,"x",0,200);
```

The two new statements here declare the variable type and then provide information about it, much as with the XPRBnewvar function previously. The additional argument gives the size of the array.

XPRBnewarrvar

Defines a new array of variables. The first argument is the problem pointer, the second states the size of the array. The next four give the same information as XPRBnewvar. It returns a variable of type XPRBarrvar.

The only constraint that need be declared this time is the objective function, since this will eventually be passed to XPRBsetobj. The other constraints will be defined using XPRBnewarrsum functions. Listing 4.15 shows the relevant sections.

Listing 4.15 Array constraints

```
double FIRST[]  = {3,2};
double SECOND[] = {1,3};
double PROFIT[] = {1,2};
    ...
XPRBctr profit;
    ...
XPRBnewarrsum(prob,"First",x,FIRST,XPRB_L,400);
XPRBnewarrsum(prob,"Second",x,SECOND,XPRB_L,200);
profit=XPRBnewarrsum(prob,"Profit",x,PROFIT,XPRB_N,0);
XPRBsetobj(prob,profit);
```

In particular, we define here three arrays holding the coefficients for each of the left hand sides of the constraints, calling them FIRST, SECOND and PROFIT. These are then multiplied (as a scalar product) by the array variable, x, to form the left hand sides of the constraints.

XPRBnewarrsum

Defines a new constraint left hand side by taking two one-dimensional arrays, one of variables and one containing coefficients and multiplying them component-wise. This may be represented mathematically as:

$$\underline{a} \cdot \underline{x} = \sum_i a_i x_i$$

It takes five arguments after the problem pointer:

- the first is the constraint name;
- the second is the variable array;
- the third is the array of coefficients;
- the fourth is the constraint type;
- the final argument is the constraint bound.

The final argument is ignored if the constraint type is 'non-binding', `XPRB_N`.

The model ends in the usual way by solving the problem and outputting a solution. A full listing is given in Listing 4.16.

Listing 4.16 The full model, 'arraymod'

```
#include <stdio.h>
#include "xprb.h"

double FIRST[]  = {3,2};
double SECOND[] = {1,3};
double PROFIT[] = {1,2};

int main(void)
{
  double objval;
  XPRBarrvar x;
  XPRBctr profit;
  XPRBprob prob;

  if(XPRBinit("")) return 1;
  prob = XPRBnewprob("arraymod");

  /* adding the variables */
  x = XPRBnewarrvar(prob,2,XPRB_PL,"x",0,200);
```

Listing 4.16 The full model, 'arraymod'

```
   /* adding the constraints */
   XPRBnewarrsum(prob,"First",x,FIRST,XPRB_L,400);
   XPRBnewarrsum(prob,"Second",x,SECOND,XPRB_L,200);
   profit=XPRBnewarrsum(prob,"Profit",x,PROFIT,
                        XPRB_N,0);
   XPRBsetobj(prob,profit);

   XPRBmaxim(prob,"");

   /* retrieving the solution */
   objval = XPRBgetobjval(prob);
   printf("The maximum profit is %f\n", objval);
   printf(" a = %f, b = %f\n",XPRBgetsol(prob,x[0]),
      XPRBgetsol(prob,x[1]));
   XPRBdelprob(prob);
   XPRBfree();
   return 0;
}
```

Exercise *Type in the new model, build it and run it to view the solution.*

Going Further with BCL

Integer Programming

For the simple problem of Listing 4.1, we have seen that the maximum value of the objective function occurs when the decision variables take the values `a = 114.3` and `b = 28.6`, to one decimal place. For some problems, however, fractional solution values such as these may not be appropriate, perhaps because the solution is only meaningful if the variables are integral.

Working with integer variables in BCL is simply a case of specifying the variable type appropriately when it is created with `XPRBnewvar` or `XPRBnewarrvar`. Where we previously worked with continuous variables, of type `XPRB_PL`, integer variables have type `XPRB_UI`. The relevant change to implement this in your last example would be:

A full list of all the variable types which are available can be found in the BCL Reference Manual.

```
x = XPRBnewarrvar(prob,2,XPRB_UI,"x",0,200);
```

The new problem is not as simple as the original, however, requiring a more complicated method to find a solution — a *global search* must now be employed to find an integer solution. This is employed using the optional `g` flag with the solution routine:

```
XPRBmaxim(prob,"g");
```

With these two changes made, your program is ready to be run.

Exercise *Alter the model of Listing 4.16 to make both* `a` *and* `b` *integer variables and solve the global problem. What values of the decision variables are necessary to achieve the maximum profit?*

In the situation where only a few of the variables in your array need be integral, a little more work is involved. Since all variables will be declared to be of the same type when the array is created, those of different types should subsequently be changed using one or more `XPRBsetvartype` commands. In such circumstances it may be simpler to construct the model using a series of `XPRBnewvar` statements and placing these variables in a C array.

Writing Matrix Files

Occasionally you will want to output your model as a matrix file for independent input into a solver, or to pass to another person. BCL provides the command `XPRBexportprob` for just such a purpose.

XPRBexportprob

This exports the problem as a matrix file in the two industry standard formats. It takes two arguments after the problem pointer:

- the first is the matrix output file format. This is one of `XPRB_MPS` or `XPRB_LP` depending on whether an MPS or LP format file is required;
- the second is the name of the output file *without an extension*.

Exercise *Alter the previous program to write an MPS file rather than solving the problem.*

A full program which achieves this is provided in Listing 4.17. A matrix file `arraymod.mat` is produced, but otherwise the program runs silently.

Listing 4.17 Writing an MPS matrix file

```
#include <stdio.h>
#include "xprb.h"

double FIRST[]  = {3,2};
double SECOND[] = {1,3};
double PROFIT[] = {1,2};

int main(void)
{
  XPRBarrvar x;
  XPRBctr profit;
  XPRBprob prob;

  if(XPRBinit("")) return 1;
  prob = XPRBnewprob("arraymod");

  x = XPRBnewarrvar(prob,2,XPRB_PL,"x",0,200);
```

Listing 4.17 Writing an MPS matrix file

```
   XPRBnewarrsum(prob,"First",x,FIRST,XPRB_L,400);
   XPRBnewarrsum(prob,"Second",x,SECOND,XPRB_L,200);
   profit=XPRBnewarrsum(prob,"Profit",x,PROFIT,
                        XPRB_N,0);
   XPRBsetobj(prob,profit);

   XPRBexportprob(prob,XPRB_MPS,"arraymod");
   XPRBdelprob(prob);
   XPRBfree();
   return 0;
}
```

In contrast to MPS format, LP matrix files also contain information about whether the objective function is to be maximized or minimized. Unless this is specified, the problem is assumed to be a minimization problem by default. To write an LP file for this problem, an extra line should be added before the matrix file is exported:

```
XPRBsetsense(prob,XPRB_MAXIM);
```

This changes the sense of the problem to maximization.

Exercise *Alter the model above to write an LP matrix file for the problem. Using a text editor or similar, check that the sense has been set correctly.*

In this brief introduction to BCL, it is not possible to cover the full range of possibilities offered and the above is intended just to provide a flavor of the library. For further details you should consult the BCL Reference Manual or the many examples contained on the Xpress-MP CD-ROM. In the following sections we turn to describing the Xpress-Optimizer Library, allowing direct access to the Optimizer from applications. This may usefully be combined with BCL to provide additional functionality for those needing to progressively develop and alter models at the matrix level. Details of how the two can be used together may be found in the section "Combining BCL with the Optimizer Library" on page 95. You may also find it useful to consult the rest of the section "Getting the Best out of the Xpress-MP Libraries" on page 95.

Working with the Xpress-Optimizer Library

The Xpress-Optimizer Library allows library users access to the full power of the Optimizer from within their own applications. Providing all the functionality enjoyed by Console users, the Optimizer library augments this with a number of 'advanced' functions for matrix manipulation and enhanced interaction with the solution process. Additionally, the library provides users with a thread-safe Optimizer, allowing simultaneous handling of a number of different problems, limited only by your system resources and license details. In the same manner as for the Mosel libraries and BCL, the Optimizer library distinguishes between problems using a problem pointer which is passed to each function that will operate on it. As with the other libraries discussed above, there is no concept of an *active problem* here.

Solving an LP Problem with the Optimizer Library

The Optimizer accepts matrix files in either MPS or LP format as valid input. To attempt the exercises in this section you will need access to a valid matrix file, such as you may have generated in the exercises previously in this chapter, or obtained from Chapter 3, "Console Xpress". Throughout this section we will refer to the matrix file `simple.mat` generated from the model in Listing 4.1 above.

The Optimizer library provides functions for initializing the library prior to use and for freeing system resources at the end. Generally these two functions, `XPRSinit` and `XPRSfree`, enclose all other library commands and both must appear in any Optimizer program. To load a matrix into the Optimizer, a problem pointer must first be created for that problem which must then be passed to every function concerned with manipulating it. It does this using `XPRScreateprob`.

XPRSinit

This initializes the Optimizer library prior to use. It takes a single argument which is a pointer to where the password file is located. If this is `NULL`, the standard installation directories are searched.

XPRScreateprob

This sets up a problem pointer for a particular problem. It takes a single argument into which a problem pointer is to be returned. This is then passed as the first argument to all functions which operate on the problem.

XPRSdestroyprob

Removes the problem specified by a problem pointer.

XPRSfree

This releases any memory currently used by the Optimizer and closes any open files.

To find a solution to the problem of Listing 4.1, the matrix file must be input before the objective function can subsequently be maximized. A full program solving this problem is given in Listing 4.18.

Listing 4.18 Solving an LP problem with the Optimizer library

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

int main(void)
{
  XPRSprob prob;

  XPRSinit(NULL);
  XPRScreateprob(&prob);

  XPRSreadprob(prob,"simple","");
  XPRSmaxim(prob,"");
  XPRSwriteprtsol(prob);

  XPRSdestroyprob(prob);
  XPRSfree();
  return 0;
}
```

XPRSreadprob

Reads a problem file in MPS or LP format. It takes three arguments:

- the first is the problem pointer;
- the second is the filename of the problem, *without its extension*;
- the third is a list of possible flags.

XPRSmaxim

Begins the search for a maximal solution to the problem. It takes two arguments, the first of which is the problem pointer and the second is a list of possible flags.

XPRSwriteprtsol

Outputs the solution to a file, *problem_name*`.prt`.

Exercise *Type in the Optimizer program from Listing 4.18 and run it to input the matrix file* `simple.mat`*, maximize its objective function and output the solution using* `XPRSwriteprtsol`*.*

Obtaining the Solution Using `XPRSwriteprtsol`

There are a number of methods for obtaining solution information using the Optimizer library. We will meet others later, but for now the simplest is to use the command `XPRSwriteprtsol`, which writes a solution file suitable for sending to a line printer. This file can be naturally split up into three sections as demonstrated in Listing 4.19.

"Section 1..." The sections are perhaps most usefully considered in the order Section 1, Section 3, Section 2. Section 1 contains summary statistics about the solution process. It gives the matrix (problem) name (`simple`) and the objective function and right hand side names that have been used. Following this are the number of rows and columns, the fact that it was a maximization problem, that it took two iterations (steps) to solve and that the best solution has a value of `171.428571`.

"Section 3..." The optimal values of the decision variables are given in Section 3, the *Columns Section*. The *Value* column gives the optimal value of the variable. Additional solution information such as the Input Cost and

Listing 4.19 Output from the `XPRSwriteprtsol` command

```
Problem Statistics
Matrix simple                                                    (Section 1)
Objective *OBJ*

RHS *RHS*
Problem has      3 rows and      2 structural columns

Solution Statistics
Maximization performed
Optimal solution found after      2 iterations
Objective function value is   171.428571


Rows Section                                                     (Section 2)
   Number Row    At    Value      Slack Value Dual Value  RHS
 N   1    *OBJ*  BS  171.428571  -171.428571  .000000       .000000
 L   2    second UL  200.000000       .000000  .571429   200.000000
 L   3    first  UL  400.000000       .000000  .142857   400.000000

Columns Section                                                  (Section 3)
   Number Column At    Value      Input Cost   Reduced Cost
 C   4     a     BS  114.285714     1.000000  .000000
 C   5     b     BS   28.571429     2.000000  .000000
```

Reduced Cost for each of the variables is also provided here. See Chapter 7, "Glossary of Terms" for details of these.

"Section 2..." The final section to consider is Section 2, the *Rows Section*. This gives a list of the various constraints and objective function of the problem, along with the value of the "left hand side" of the expression. In particular, the value of the objective function turns up here again. Slack values and Dual values for the constraints can also be obtained from here. See Chapter 7, "Glossary of Terms" for details of these.

Exercise *View the output file* `simple.prt` *which you have just created and identify the various sections. Check your results for the decision variables and objective function with those in the listing.*

Going Further with the Optimizer Library

Changing the Optimization Algorithm

The Optimizer solves LP problems by default using the *dual simplex* algorithm. However, *primal simplex* and *Newton barrier* algorithms are also supported and on certain problems changing the algorithm may result in a considerable change in the time taken to solve a problem. Choosing the 'best' algorithm to use in a given situation is something of an art in itself, but often the best choice is evident by experimenting with the different algorithms on problems which are similar in nature to your own.

The simplest way to change the algorithm is by setting a flag which can be passed to the optimization routines. These correspond to each of the three algorithms discussed above:

- `b` use the Newton barrier algorithm;
- `d` use the dual simplex algorithm;
- `p` use the primal simplex algorithm.

Listing 4.20 demonstrates using the Newton barrier method for the optimization.

Listing 4.20 Changing the optimization algorithm

```
XPRScreateprob(&prob);
XPRSreadprob(prob,"simple","");
XPRSmaxim(prob,"b");
XPRSwriteprtsol(prob);
XPRSdestroyprob(prob);
```

Exercise *Try solving the problem using each of the three algorithms. For a problem as simple as this, there will be no noticeable difference in solution times.*

Integer Programming

For the simple problem considered above, the maximal value of the objective function is obtained when the decision variables take values of `a = 114.3` and `b = 28.6`, to one decimal place. For some problems, however, fractional solutions such as this may be unacceptable and one or more of the decision variables may be constrained to take only integer values. Listing 4.21 provides the necessary changes to our model if we wish to seek only integer solutions and export the problem as an MPS matrix file.

Listing 4.21 Introducing integer variables in our problem

```
model integral
  declarations
    a: mpvar
    b: mpvar
  end-declarations

  first:=  3*a + 2*b <= 400
  second:=   a + 3*b <= 200
  profit:=   a + 2*b

  a is_integer
  b is_integer

  exportprob(EP_MPS,"integral",profit)
end-model
```

Exercise *Use the Mosel libraries to compile, load and run the integer problem of Listing 4.21, exporting an MPS matrix file. Load this into the Optimizer using the library and solve the problem as before. What is the value of the objective function now?*

The *LP relaxation* is the previous problem, ignoring the integrality of any variables.

If you have run your program on this new model file, you will probably have noticed that the solution returned remains unchanged, despite our imposition of integer-valued variables. The reason for this is that the search for optimal integer solutions is a two-stage process. For the first of these, the LP *relaxation* must be solved to find an optimal

solution. For the second, a *global search* is carried out to find the best integer solution, assuming one exists. The global search can be called automatically following solution of the LP relaxation by passing the `g` flag to `XPRSmaxim` and this is demonstrated in Listing 4.22 where we again show only a small portion of the program, demonstrating the additional command in bold face. If we run this and view the solution produced, the decision variables should, finally, both take integer values.

Listing 4.22 Solving a MIP Problem with the Optimizer Library

```
XPRScreateprob(&prob);
XPRSreadprob(prob,"integral","");
XPRSmaxim(prob,"g");
XPRSwriteprtsol(prob);
XPRSdestroyprob(prob);
```

Calling `XPRSmaxim` with the `g` flag is equivalent to calling `XPRSmaxim` without any flag and then calling the `XPRSglobal` command directly afterward.

Exercise *Alter your program to find integer solutions to the problem. What is the largest value of the objective function in this case and what values should the decision variables take to achieve it?*

Exercise *Alter the model again to keep one of the decision variables integral but to remove this condition on the other. What is the optimal value of the objective function now?*

Presolve and Everything After

The Optimizer makes use of a number of procedures for simplifying problems prior to optimization to make them easier to solve. This collection of algorithms is known as *presolve*. During presolve, redundant rows and columns in the matrix may be removed with the result that the presolved matrix will often look very different from the original problem. Whilst the `XPRSwriteprtsol` routine only outputs information related to the original problem, other routines exist which are affected by the matrix state and can return the presolved solution if the matrix is still in its presolved form. Obtaining presolved solutions rather than a solution to the original problem is a principle source of confusion for many users. It is therefore important, when accessing the matrix or a solution, to know what to expect.

"LP problems..." Presolve is called automatically by the solution routines, `XPRSmaxim` and `XPRSminim`. Following the optimization of *linear* programming problems, the matrix is also automatically *postsolved*, reinstating the original matrix and returning a solution to the original problem.

"IP problems..." If the matrix contains non-continuous variables (e.g. integer variables) or special ordered sets, it is left in its presolved state following `XPRSmaxim` (and `XPRSminim`), since a call to the global search is expected to follow. In particular, if one of the solution routines is called without the `g` flag, as we did above, then the matrix will remain in its presolved form.

During the global search, integer solutions are postsolved and written to the solution file as they are identified, making a solution to the original problem available. However, postsolving of the entire matrix is *never* carried out after a global search and the original matrix is not restored. If further access to the matrix is subsequently required, it must be reloaded, or copied *before* calling any of the solution routines.

Note also that many of the advanced library functions will not work with a presolved matrix and care should be taken to ensure that the matrix will be in the correct form before calling them. An example of this is the `XPRSaddrows` command which we will meet later.

Whilst we can only scratch the surface here, further details about presolve may be found in the Optimizer Reference Manual. For additional information, we recommend that you consult that manual.

Controls and Problem Attributes

There are a number of parameters to the Optimizer, whose values may be altered by the user for a particular problem through the setting of various *controls*. With some thought, these can be used to fine tune the Optimizer's performance on particularly difficult problems and can result in significant savings in the solution time. Whilst we do not encourage new users to change the majority of these from their default values, there are a small number with which experimentation may be useful. For the sake of these, we briefly mention how this may be achieved.

Additionally, the optimization process makes available a number of *attributes* of the particular problem being solved which may be retrieved in a similar way to the control values. A full list of all controls which can be set and problem attributes which can be retrieved may be found in the Optimizer Reference Manual

Both the controls and the problem attributes have a type which is one of *integer*, *double* or *character string* and the way they are handled and accessed depends on the type. For the moment we concentrate solely on the controls. There are three library functions allowing the values of controls to be obtained, namely `XPRSgetintcontrol`, `XPRSgetdblcontrol` and `XPRSgetstrcontrol`. To access the value of a given control for a particular problem, the correct function for its type must be employed.

We have just seen that when `XPRSmaxim` is invoked the presolve algorithms are automatically called to simplify the problem before the solution algorithm begins. This default behavior is governed by the control `PRESOLVE`, which is an integer describing whether or not presolve should be used during optimization.

Turning Presolve On or Off

Set the control variable `PRESOLVE` to:

`0` to call the solution algorithm *without* presolving;

`1` to call presolve before the solution algorithm;

Exercise *Retrieve the value of the control* `PRESOLVE` *to the integer variable* `presolve` *using*

```
XPRSgetintcontrol(prob,XPRS_PRESOLVE,&presolve);
```

and confirm that this corresponds to presolve being set on by default.

It should be noted that when controls are accessed from within the Xpress-MP libraries, their names differ from those in Console Xpress in that they are prefixed by `XPRS_`. This is an important point which is easily forgotten since controls are usually described without this prefix.

To set new values to the controls, the functions `XPRSsetintcontrol`, `XPRSsetdblcontrol` and `XPRSsetstrcontrol` should be used, setting new values to integer, double and string controls respectively.

Exercise *Use* `XPRSsetintcontrol(prob,XPRS_PRESOLVE,0);` *to set the value of the control* `PRESOLVE` *to* `0` *in the previous program, turning off presolve. This must be done before the call to* `XPRSmaxim` *which is affected by the value of this variable. Run the program again and view the output.*

"Problem attributes..."

The final set of functions of this type allow access to various attributes of the problem, set by the Optimizer during the solution process. The `XPRSgetintattrib`, `XPRSgetdblattrib` and `XPRSgetstrattrib` functions are equivalent to those for getting control values and are used in the same way. An example of this may be given in relation to the integer problem attribute `LPSTATUS`, whose value provides details about whether the solution is optimal, infeasible, unbounded or unfinished. Listing 4.23 shows how this may be used to good effect.

Listing 4.23 Checking the solution status

```
int lpstatus;
   ...
XPRSmaxim(prob,"");

XPRSgetintattrib(prob,XPRS_LPSTATUS,&lpstatus);
if(lpstatus == XPRS_LP_OPTIMAL)
  XPRSwriteprtsol(prob);
else
  printf("Solution is not optimal\n");
```

A full list of all the possible values of `LPSTATUS` may be found with the Problem Attributes in the Optimizer Reference Manual.

Although we do not discuss this further here, another example can be found later when use of the Optimizer library in modeling is explored.

Interacting with the Optimization Process

For large problems taking a long time to solve, it is often helpful to be able to obtain more information from the Optimizer during the optimization process. If a log file has been set up, it is possible to consult this to determine how the solution is progressing, but this is inconvenient and inflexible. For such situations, the Optimizer library provides a set of 'advanced' functions, known as *callbacks*, allowing a return to your program at various points during the optimization process to run your own functions.

Log files may be set in place using the `XPRSsetlogfile` command. See the Optimizer Reference Manual for details of this.

The majority of the callbacks are associated with various aspects of the global search and are described in the Optimizer Reference Manual. There are a few, however, which are called in more general situations, perhaps the simplest of which is `XPRSsetcbmessage`. This sets a function which is called every time a text line would be output by the Optimizer and a (partial) example of usage may be found in Listing 4.24. This is a particularly simple example, merely writing each line of standard output generated by the Optimizer to the screen, with a suitable prefix.

Listing 4.24 Using callbacks with the Optimizer library

```
void XPRS_CC callback(XPRSprob prob, void *user,
     const char *msg, int len, int msgtype)
{
  char *usermsg = (char *)user;
  if(msgtype>=0) printf("%s: %s\n",usermsg,msg);
}

int main(void)
{
  XPRSprob prob;
  char *usermsg = "simple.callback";

  XPRSinit(NULL);
  XPRScreateprob(&prob);
  XPRSsetcbmessage(prob,callback,usermsg);
     ...
}
```

`usermsg` can be `NULL` here if no extra information is to be passed into the callback .

XPRSsetcbmessage

Sets up a function to be executed each time a line of output is generated by the Optimizer. Its syntax is:

```
int XPRSsetcbmessage(XPRSprob prob,
  void (*uopf)(XPRSprob this_prob,
  void *this_user, const char *this_msg,
  int this_len, int this_msgtype),
  void *user);
```

where `this_user` is a user-definable pointer allowing your own data to be passed into the callback function, `this_msg` is the character string output, `this_len` is its length and `this_msgtype` is an integer defining the message level.

Exercise *Add a callback to your program, which prints Optimizer output to the screen, such as that in Listing 4.24.*

The message type may be used to 'screen' the lines output by the Optimizer, since the message type varies according to the message level of the output in the following way:

Message Type	Message Level
1	normal messages (informational)
2	normal messages (debugging)
3	warning messages
4	error messages

Exercise *Alter the callback function in your program to format messages differently when printed, depending on their level.*

The Optimizer Reference Manual contains a full list of all the callbacks that may be set by library users. You may find it useful to familiarize yourself with the possibilities that these provide and particularly so if it is likely that you will be solving large integer problems on a regular basis.

Loading Models from Memory

The Optimizer library is the largest Xpress-MP library, with functions which fall roughly into two categories: those 'basic' functions which, for the most part, are the counterparts of Console Xpress commands, and more advanced commands which are available only to library users.

The advanced library functions contain all the tools necessary to load problems directly into the Optimizer from memory-resident arrays and to manipulate them once there. In this section we demonstrate how our simple problem (Listing 4.1) may be entered directly using these functions and show how the Optimizer and the library program can interact further.

Loading LP Problems with the Optimizer Library

In modeling parlance we always speak of constraints and variables when describing problems, but the equivalent structures within the Optimizer are the rows and columns of the matrix. You will already have seen this in the file produced by `XPRSwriteprtsol` earlier. When a model is exported by Mosel, the problem description is converted into a matrix format that can be read into the Optimizer for solving. In this section we will explore how you can create this matrix yourself directly with the Optimizer library. We do this using the 'advanced' function `XPRSloadlp`.

XPRSloadlp

Loads an LP problem into the Optimizer data structures. A full list of all its arguments may be found in the Optimizer Reference Manual.

Since `XPRSloadlp` takes so many arguments, it is perhaps more sensible to describe its use by way of an example. You may find it useful to have a finger in page 49 and look at Listing 4.1 alongside this.

In Listing 4.25 the various arguments to `XPRSloadlp` are declared and assigned values for the same simple problem discussed earlier.

Listing 4.25 Declaring arguments for `XPRSloadlp`

```
char probname[]    = "advanced";
int ncol           = 2;
int nrow           = 2;

char qrtype[]      = {'L','L'};
double rhs[]       = {400.0, 200.0};
int mstart[]       = {0, 2, 4};
int mrwind[]       = {0, 1, 0, 1};
double dmatval[]   = {3.0, 1.0, 2.0, 3.0};
double objcoefs[]  = {1.0, 2.0};

double dlb[]       = {0.0, 0.0};
double dub[]       = {200.0, 200.0};
```

- The first of these specifies a name for the problem, which this time we have chosen to be 'advanced'. As previously, the problem name is the basis for any files that are created, and must be specified.
- Following this we define the number of columns (variables) in the matrix to be `2` (`a` and `b`) and the number of rows (constraints, *not* including the objective function) to be `2` also.

The next six arrays specify details of the constraints in the model for constructing the matrix rows.

- To begin with, we ignore the objective function as a row and consider only the 'true' constraints. The first array, `qrtype`, specifies the type of each constraint, which were both $\leq$ in our model. This is passed to `XPRSloadlp` using an '`L`' for each row.
- Following this, the right hand sides of the constraints are defined, one row at a time. The first value comes from the constraint $3a + 2b \leq 400$, while the second comes from $a + 3b \leq 200$.

- The following two arrays describe the offsets and starting points for elements in the matrix that will shortly be defined. These will be described in detail in the paragraphs below.
- The array `dmatval` then contains the actual coefficients from the constraints which will become the matrix elements. This is a one-dimensional array running vertically, taking nonzero coefficients from each constraint, one variable at a time. Effectively we specify the matrix column-wise from the left.
- Finally the coefficients of the objective function are specified.

The last two lines provide upper and lower bounds on the variables in the model.

"Offsets and start points..."

A matrix with a large number of zero entries is said to be *sparse*.

Perhaps the most interesting elements of this listing are the arrays `mstart` and `mrwind`. Defining a matrix in terms of the offsets and starting points for its entries provides an extremely efficient method of describing problems which have a large number of zero entries. Since the simple example that we have been considering up to this point does not contain any zeros, their use is perhaps not as evident as it might be. By way of an aside, therefore, consider briefly the following problem:

Problem 1

```
Maximize:   0.1x1 + 0.2x2 + x3
Subject to:  x1 + 2x2                          = 0
                        x3 + x4 -      x5      = 0
                             x4 - 0.27x5 >= 0
                  3x1          + x4 - 0.4 x5 <= 0
```

The matrix in Problem 1 would have 20 entries, with ten of them zero. In common with this model, large problems are *usually* sparse and `XPRSloadlp` is geared towards this. Requiring problems to be specified in a sparse format is considerably more efficient and an example of how this might be done is provided in Listing 4.26.

Listing 4.26 Entering sparse matrix data

```
int mstart[]     = {0,  2,3,4,     7,           10};
int mrwind[]     = {0,3,0,1,1,2,3, 1,    2,   3};
double dmatval[] = {1,3,2,1,1,1,1,-1,-0.27,-0.4};
```

The entries in the matrix `mstart` tell the Optimizer that the data in `dmatval` may be divided up into columns in the matrix according to:

Don't forget, all row and column indices run from `0` in C!

`dmatval[0]`, `dmatval[1]` in column `0`;
`dmatval[2]` in column `1`;
`dmatval[3]` in column `2`;
`dmatval[4]`, `dmatval[5]`, `dmatval[6]` in column `3`;
`dmatval[7]`, `dmatval[8]`, `dmatval[9]` in column `4`.

Thus, the entries in `mstart` define the starting points (and end points) for the columns within the stream of data provided in `dmatval`. The `mstart` array thus contains one more entry than the number of columns, as data for column `i` begins at `mstart[i]` and finishes at `mstart[i+1]-1`. With the columns defined, information contained in `mrwind` places the entries for each column into their correct rows, with:

`dmatval[0]` in row `0`; `dmatval[1]` in row `3`;
`dmatval[2]` in row `0`;
`dmatval[3]` in row `1`;
`dmatval[4]` in row `1`; `dmatval[5]` in row `2`; `dmatval[6]` in row `3`;
`dmatval[7]` in row `1`; `dmatval[8]` in row `2`; `dmatval[9]` in row `3`.

Once the positions have been specified, the elements themselves are determined from the data in `dmatval`.

"*Anyway...*" Returning to our problem, 'advanced', Listing 4.25 places the first two elements of `dmatval` in rows `0` and `1` of column `0` and the third and fourth elements of `dmatval` in rows `0` and `1` of column `1` in the matrix.

Using `XPRSaddnames`, names can also be added to the variables within the matrix once it has been defined. In Listing 4.27 we show this, along with the definition of the matrix using `XPRSloadlp`. The variable names are stored in the character array `colnames` as a null-terminated list (hence the `\0` character between the two).

Listing 4.27 Loading the problem and adding variable names

```
XPRSprob prob;
char colnames[] = "a\0b";
    ...
XPRSinit(NULL);
XPRScreateprob(&prob);
XPRSloadlp(prob, probname, ncol, nrow, qrtype, rhs,
     NULL, objcoefs, mstart, NULL, mrwind, dmatval,
     dlb, dub);
    ...
XPRSaddnames(prob,2,colnames,0,ncol-1);
```

The two NULLs here relate to row range values and number of nonzeros in each column. Because of the way we have specified the problem, these are not needed. See the Optimizer Reference Manual for details.

XPRSaddnames

Adds row or column names to the matrix. It takes five arguments:

- the first argument is the problem;
- the second can take values of either 1 or 2 depending on whether the names are for rows or columns;
- the third is a null-terminated array of names;
- the final two arguments specify the start and end rows (or columns) in the matrix for assigning the names.

Obtaining a Solution to the Problem

With the problem matrix defined, the objective function can now be maximized as before using XPRSmaxim (see Listing 4.18). However, while previously we have always output the solution to file for viewing or printing, the Optimizer library provides another function, XPRSgetsol, which enables the solution to be accessed directly from your program.

XPRSgetsol

Retrieves a solution to the problem. It takes four arguments after the problem, any of which may be NULL depending on what information about the solution is required.

In Listing 4.28 we show how this can be employed to obtain the optimal values of the various variables and output them to the screen.

Listing 4.28 Obtaining and displaying variable values

```
#include <stdlib.h>

double *vars;
    ...
vars = malloc(ncol*sizeof(double));
XPRSgetsol(prob,vars,NULL,NULL,NULL);
printf("a = %2.1f\nb = %2.1f\n", vars[0], vars[1]);
```

Perhaps the only part of the solution that `XPRSgetsol` cannot supply us with is the optimal value of the objective function. Indeed, there are no advanced library functions that return this value. Rather it is stored in the problem attribute `LPOBJVAL` which must be accessed in the same manner as for the problem status previously. The code for doing this is given in Listing 4.29.

Listing 4.29 Obtaining and displaying the objective value

```
double lpobjval;
    ...
XPRSgetdblattrib(prob, XPRS_LPOBJVAL, &lpobjval);
printf("The objective value is %2.1f\n", lpobjval);
```

Exercise *Putting together the ideas of Listing 4.25 – Listing 4.29, write an Optimizer program which directly loads and solves the problem of Listing 4.1 and outputs the solution to the screen. Check that the values obtained agree with those from previous exercises.*

Altering the Problem Matrix

The Optimizer library also provides the possibility of altering our matrix once it has been loaded. As an example, we will briefly discuss how an extra constraint can be added to the problem using `XPRSaddrows`. Suppose that we wish to add the extra constraint,

XPRSaddrows

Adds extra constraint rows to the matrix in the Optimizer data structures.

Compare this with the '*altered*' model of Listing 4.5 earlier.

$6a + 5b \leq 800$ to our problem. In Listing 4.30 we show the extra lines necessary to achieve this. Much as for `XPRSloadlp`, we specify coefficients of the variables in the linear constraint expression, with `aclind` specifying the column indices for the elements of `amatval` and `astart` specifying the offsets of the start of the elements for the new row. In this case a new row will be added with `amatval[0]` (in column `0`) at element `0` in the matrix and `amatval[1]` (in column `1`) at element `3`, specifying the nonzeros in the matrix columnwise. The second and third arguments of `XPRSaddrows` are the number of new rows and the number of nonzeros in the added rows.

Listing 4.30 Adding an extra row in the matrix

```
char atype[] = {'L'};
double arhs[] = {800.0};
int astart[] = {0,2};
int aclind[] = {0,1};
double amatval[] = {6.0,5.0};
    ...
XPRSaddrows(prob,1,2,atype,arhs,NULL,astart,aclind,
     amatval);
```

Exercise *Alter the previous program to add a new row to the problem before solving it. Run the program and the new optimal value of the objective function should have reduced to* `169.2`*. What values of the decision variables are necessary to achieve this?*

Loading MIP Problems with the Optimizer Library

'Global entities' is used as an umbrella term in Xpress-MP to describe any binary, integer, or other non-continuous variables and special ordered sets.

Using the Mosel model programming language, we saw earlier that changing the problem to insist on integer-valued decision variables was merely a case of adding two extra lines to the model program file. Similarly, we can specify that the decision variables should be integral using the Optimizer library, although the problem can no longer be loaded using `XPRSloadlp` — we must instead use the function, `XPRSloadglobal`.

XPRSloadglobal

Loads a problem with global entities into the Optimizer data structures. The first 14 of its arguments are identical to those of `XPRSloadlp`. For a list of the remaining ones, consult the Optimizer Reference Manual.

The first 14 arguments of `XPRSloadglobal` are exactly the same as those for `XPRSloadlp`, so need no further discussion. Of the remaining ones, only a few warrant a mention here. We declare and set them in Listing 4.31.

Listing 4.31 Loading integer problems using the libraries

```
int ngents = 2;
int nsets = 0;
char qgtype[] = {'I','I'};
int mgcols[] = {0,1};
   ...
XPRSloadglobal(prob, probname, ncol, nrow, qrtype,
     rhs, NULL, objcoefs, mstart, NULL, mrwind,
     dmatval, dlb, dub, ngents, nsets, qgtype, mgcols,
     NULL, NULL, NULL, NULL, NULL);
```

For details of the extra arguments for problems with SOSs, see the Optimizer Reference Manual.

- Of the new arguments, `ngents` states the number of global entities in the problem, which in our case is two — the two variables should be integers.

- The integer `nsets` states the number of *Special Ordered Sets* (SOSs) in the problem. We use none here.
- The array `qgtype` allows us to set both variables to be integral.
- `mgcols` provides the indices of the variables which should be set to these types.

Variable indices are the position of each in the variable array, counting from `0`.

The final five arguments are all concerned with SOSs. Since we do not have any of these, they have all been set to `NULL`.

Once a MIP problem has been loaded with `XPRSloadglobal`, you can maximize it using `XPRSmaxim` (with the `"g"` flag specified) as described earlier (see Listing 4.22).

It should be noted that for mixed integer problems such as this, the problem attribute `LPOBJVAL` should *not* be used to return the optimal value of the objective function. Instead the attribute `MIPOBJVAL` must be consulted. During the global (MIP) search, a number of LP problems will be solved, each one refining the set of possible solutions by placing additional bounds on the various global variables. After the global search has completed, the attribute `LPOBJVAL` will hold the optimal value of the objective function for the *last LP problem that was solved*. However, this need not be the global maximum (over all the LP problems which had to be solved), which is instead stored in `MIPOBJVAL`. For interested users, further details of this may be found in the Optimizer Reference Manual.

Exercise *Alter the original problem (the one with only two constraints) to load the integer problem. Solve this and confirm that the solution agrees with the last integer problem we solved.*

Exercise *Change this to load a new global problem with the additional constraint* $6a + 5b \leq 800$. *This should have an optimal solution when* `a` *is* `107` *and* `b` *is* `31`. *Solve it and check this.*

Much more information on modeling with this library can be found in the Optimizer Reference Manual, where all the functions which are available for changing and solving loaded problems are documented. Hopefully we have been able at least to give the flavor of this method of modeling, although the treatment has obviously not been exhaustive. We end this chapter now with some general comments

which apply to users of more than one of the Xpress-MP libraries before describing the use of the Xpress-MP Libraries with other languages.

Getting the Best out of the Xpress-MP Libraries

Error Checking

Almost all library functions return an integer value on exit which is zero if the function call completes successfully and nonzero otherwise. By catching the return values of functions, your program can respond early to possible errors and by checking these values it becomes easier to diagnose problems. Whilst we have not used error checking in any of the previous listings in this chapter, this has been done purely for clarity and we recommend that users employ error checking within all their programs, regardless of the library used.

An example of basic error checking using the Optimizer library is given in Listing 4.32. This example is particularly simple, merely printing a message and exiting when an error occurs. In practice, of course, more complicated procedures for dealing with errors within your code are possible and we leave these for you to experiment with.

Exercise *Alter one of the programs that you have written in this chapter to include error checking.*

Combining BCL with the Optimizer Library

In certain circumstances, the basic BCL functions will not be sufficient to perform all the tasks within the Optimizer that a user may require. The solution is to build the problem and load it into the Optimizer using BCL and then call on the Optimizer functions afterward. This enables you to use the full capabilities of the Optimizer library on the problem, but with the drawback that the problem will no longer be accessible by BCL after this point. If you require access by BCL

Listing 4.32 Basic error checking

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

void error(XPRSprob my_prob, const char *function)
{
  char errmsg[256];
  XPRSgetlasterror(my_prob,errmsg);
  printf("%s did not execute correctly:\n\t%s\n",
     function, errmsg);
  if(my_prob != NULL) XPRSdestroyprob(my_prob);
  XPRSfree();
  exit(1);
}

int main(void)
{
  XPRSprob prob=NULL;

  if(XPRSinit(NULL))
     error(prob,"XPRSinit");

  if(XPRScreateprob(&prob))
     error(prob,"XPRScreateprob");
  if(XPRSreadprob(prob,"simple",""))
     error(prob,"XPRSreadprob");
  if(XPRSmaxim(prob,""))
     error(prob,"XPRSmaxim");
  if(XPRSwriteprtsol(prob))
     error(prob,"XPRSwriteprtsol");
  if(XPRSdestroyprob(prob))
     error(prob,"XPRSdestroyprob");

  XPRSfree();
  return 0;
}
```

throughout the solution process, you will need to perform the optimization tasks using the BCL solution functions, as above.

Combining BCL with the Optimizer library can be achieved on two levels. In the simplest case, BCL can be used to write an MPS file (as in Listing 4.16) which can then be read into the Optimizer using library functions. However, the generation and reading of ASCII files involves disk access, which can take a significant amount of time. For this reason, BCL can also be used to pass the problem directly to the Optimizer. There are some complications involved with combining the libraries in this manner, due mainly to the different problem pointers used by BCL and the Optimizer library. Consequently, any problem used must initially be created by BCL and then subsequently passed to the Optimizer library. Listing 4.33 demonstrates how this may be achieved.

Listing 4.33 Combining BCL and the Optimizer library

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

double FIRST[]  = {3,2};
double SECOND[] = {1,3};
double PROFIT[] = {1,2};

int main(void)
{
  XPRSprob opt;  /* Optimizer problem pointer */
  XPRBprob bcl;  /* BCL problem pointer */
  XPRBarrvar x;
  XPRBctr profit;

  bcl = XPRBnewprob("combined");

  x = XPRBnewarrvar(bcl,2,XPRB_PL,"x",0,200);

  XPRBnewarrsum(bcl,"First",x,FIRST,XPRB_L,400);
  XPRBnewarrsum(bcl,"Second",x,SECOND,XPRB_L,200);
  profit=XPRBnewarrsum(bcl,"Profit",x,PROFIT,
     XPRB_N,0);
  XPRBsetobj(bcl,profit);
```

Listing 4.33 Combining BCL and the Optimizer library

```
   /* load the matrix into the Optimizer */
   XPRBloadmat(bcl);
   /* obtain the problem pointer for the Optimizer */
   opt = (XPRSprob)XPRBgetXPRSprob(bcl);

   XPRSmaxim(opt,"");
   XPRSwriteprtsol(opt);

   /* tidy up using the BCL function */
   XPRBdelprob(bcl);
   XPRBfree();
   return 0;
}
```

In this example, the BCL `XPRBnewprob` function initializes both BCL and the Optimizer at the same time, removing the need for a separate call to `XPRSinit`. The output from this function is a BCL problem pointer which is used by all BCL functions operating on the problem. The matrix is then loaded into the Optimizer using `XPRBloadmat`, after which an Optimizer problem pointer is needed. This is obtained by BCL using the function `XPRBgetXPRSprob` and thereafter the Optimizer functions can be used in exactly the same manner as previously.

XPRBloadmat

Loads a matrix into the Optimizer for Optimizer functions to operate on.

XPRBgetXPRSprob

Returns an Optimizer problem pointer for the problem just loaded into the Optimizer by BCL.

Exercise *Build the example of Listing 4.33 and try running it. Notice that no MPS file is created and the libraries work together by passing the completed matrix from BCL directly to the Optimizer.*

It is worth remarking once again that after Optimizer functions have been called, it will no longer be possible to access the problem from BCL. If this is required, the BCL solution routines should be used in preference to the Optimizer ones.

Using Visual Basic with the Xpress-MP Libraries

The Xpress-Optimizer library can be used with Visual Basic, either the stand-alone product, or Visual Basic for Applications (VBA) as implemented in Excel, Access etc. We shall simply use 'VB' here to refer to both of these. If you do not use VB you can safely skip this section.

The Xpress-MP library functions can be imported into your VB project using the external function declarations defined in the VB module `xprs.bas`, included as part of the Xpress-MP Libraries distribution. The Xpress-MP library functions themselves take the same names as the C functions, allowing users to easily translate between the prototypes in the Reference Manuals and their VB equivalents. They all produce a `Long` return value. An example of this is given in Listing 4.34 which is a VB version of our previous problem, Listing 4.18.

A copy of this module should be added to your project since you may eventually customize it when defining function prototypes that use `NULL` arguments.

Listing 4.34 Using the Xpress-MP libraries in VB

```
Public Sub main()
  Dim nReturn As Long

  nReturn = XPRSinit("")
  nReturn = XPRScreateprob(prob);
  nReturn = XPRSreadprob(prob,"simple","")
  nReturn = XPRSmaxim(prob,"")
  nReturn = XPRSwriteprtsol(prob)
  nReturn = XPRSdestroyprob(prob)
  nReturn = XPRSfree()
End Sub
```

We recommend that you check the return values of all library functions, although for clarity we have not done so here. See "Error Checking" on page 95 for details.

Over the remainder of this section, we will briefly describe some of the differences between the C and VB interfaces and mention some of the pitfalls for users of the libraries in this environment. The approach taken will be summary in nature, so users may find it useful to consult the Optimizer Reference Manual alongside this.

Included Files

Part of the Xpress-MP distribution, the file `modbyte.bas` contains a number of useful function definitions which we will use in this chapter and we shall assume that it is included in the source of any program described in the listings. In particular, the function `ByteConversion` is defined here and consequently a number of the following listings may not work without this. It may be found in the directory `\examples\optimizer\vb` on the CD-ROM.

Principal Structures

Argument Types: VB parameters passed to the routines are related to the C parameters in the following way:

C argument type	VB argument type
`int x`	`ByVal x As Long`
`int *x`	`x As Long`
`double x`	`ByVal x As Double`
`double *x`	`x As Double`
`char *x`	`ByVal x As String` or `x As Byte`

Constants: The C constants become VB global constants with the same names, for example:

```
Global Const XPRS_PLUSINFINITY = 1E+20
Global Const XPRS_LPLOG = 9
```

Numerical Arrays: A number of routines in the Optimizer library require numerical arrays to be passed as one of their arguments. An example is given by the routine `XPRSgetobj`, whose second argument

is a double precision array. An example passing this to the Optimizer from VB is given in Listing 4.35. The important thing to note is that only the first element of the array is passed and not the array itself. This is the case for all numerical arrays passed between Xpress-MP and VB.

Listing 4.35 Passing numerical arrays

```
Dim gpObjCoef(2) As Double
Dim nReturn As Long
      ...
nReturn = XPRSgetobj(prob,gpObjCoef(0),0,2)
```

Character Arrays: It is often necessary to pass character arrays to various library subroutines. In VB, the character array is passed as a `Byte` variable type, for example in the definition of `XPRSchgbounds`:

```
Declare Function XPRSchgbounds Lib "XPRS.DLL" _
     (..., qbtype As Byte,...) As Long
```

To use this, the byte array must be initialized in your VB code using the `Asc` function as in Listing 4.36. This initializes the `qbtype` array to `"LU"` ready to be passed to the `XPRSchgbounds` routine.

Listing 4.36 Initializing byte arrays in VB

```
Dim qbtype(1) As Byte
      ...
qbtype(0) = Asc("L")
qbtype(1) = Asc("U")
```

"Passing character arrays..."

Some Optimizer library routines require the user to *pass* character arrays containing multiple strings. This should also be done using the `Byte` data type. To separate each of the strings it is necessary to use the C string terminator (a byte value of 0) and this requires some conversion to be done in VB. To make this conversion easy for users of Xpress-MP, the file `modbyte.bas` has been included in the VB examples supplied with the Xpress-MP libraries. This module contains

the `ByteConversion` function which takes a VB array of strings and returns a byte array that can be passed to Xpress-MP routines. Listing 4.37 shows how this may be used.

Listing 4.37 Using the `ByteConversion` function

```
Dim sRowName(3) As String
Dim bRowName() As Byte

sRowName(0) = "c1": sRowName(1) = "c2"
sRowName(2) = "c3": sRowName(3) = "c4"

' Transform the row names in bytes
bRowName = ByteConversion(sRowName,4)
```


"Returning character arrays..."

Some routines in the Optimizer library *return* character arrays to VB in the form of `Byte` arrays. These byte arrays can be translated to strings for easier manipulation in VB. For example, to convert a three element byte array to a three character string, the code in Listing 4.38 can be used.

Listing 4.38 Converting a byte array to a character string

```
For i = 0 To 2
  sRowType = sRowType + Chr(qrtype(i))
Next i
```

"Converting arrays from `Byte` to `String`..."

Other routines in the Optimizer library return arrays of strings to VB which must be converted from `Byte` to `String` to be useful. For example, the `XPRSgetnames` routine has a second argument which is a byte array, receiving row or column names from the Optimizer. This can be easily translated to an array of strings using the code in Listing 4.39.

NULL Arguments: Certain routines can be called from C with null arguments, meaning that the argument is to be ignored. To do the equivalent of passing a null argument in VB, a compatible data structure that might be used is the `vbNullString` constant, which is

Listing 4.39 Translating a byte array to a string array in VB

```
Dim nReturn As Long
Dim nmLen As Long
Dim i, j As Long
Dim nRow As Long
Dim bV() As Byte
Dim sV() As String

nReturn = XPRSgetintcontrol(prob, _
     XPRS_MPSNAMELENGTH, nmLen)
nReturn = XPRSgetintattrib(prob, XPRS_ROWS, nRow)

ReDim bV(nmLen * nRow + nRow - 1)
ReDim sV(nRow - 1)

nReturn = XPRSgetnames(prob, 2, bV(0), 0, nRow - 1)
For i = 0 To nRow - 1
  For j = 0 To nmLen - 1
    sV(i) = sV(i) + Chr(bV(i * (nmLen + 1) + j))
  Next j
Next i
```


of type `String`. To use this, the function prototype in the `xprs.bas` must often be changed and alternative prototypes may be declared for each of the required configurations of null arguments in the function argument list. As an example, suppose we wish to do the equivalent of the C code:

```
nReturn = XPRSgetsol(prob,x,NULL,NULL,NULL);
```

where the last three quantities in the argument list are not required. In VB, the `XPRSgetsol` function prototype in the `xprs.bas` file is declared as:

```
Declare Function XPRSgetsol Lib "XPRS.DLL" _
     (ByVal XPRSprob As Long,x As Double, _
     slack As Double,dual As Double,dj As Double) _
     As Long
```

The customized definition of `XPRSgetsol` which now allows for the use of `vbNullString` with the final three arguments is then:

```
Declare Function XPRSgetsolx Lib "XPRS.DLL" _
     Alias "XPRSgetsol" (ByVal XPRSprob As Long, _
     x As Double, ByVal slack As String, _
     ByVal dual As String, ByVal dj As String) _
     As Long
```

The new function with a modified argument specification may subsequently be used as in Listing 4.40.

Listing 4.40 Using null arguments in VB

```
Dim x() As Double
Dim cols As Long

nReturn = XPRSgetintattrib(prob,XPRS_COLS,cols)
ReDim x(cols-1)
Call XPRSgetsol(prob, x(0), vbNullString, _
     vbNullString, vbNullString)
```

Note that the prototype `XPRSgetsolx` and some other useful prototypes for the `XPRSgetsol` routine are declared in `xprs.bas`. You may find it easier to add your customized prototypes into a central `xprs.bas` file as you need them.

Using Callbacks in Visual Basic

Optimizer callbacks can be used in Visual Basic by declaring a function or subroutine with the correct parameters and passing its address to the appropriate `XPRSsetcb` routine. This may be obtained using the `AddressOf` operator. However, since the callbacks also allow user-defined objects to be passed into the callback function as the final argument, often a new prototype for the callback must be defined. An example of this was seen above, where the `XPRSgetsol` function prototype had to be altered to accept `vbNullString` as an argument. An example in this context is given in Listing 4.41.

In this example, two strings and two integers are to be passed into the callback, to which end an object, `MyInfo`, is defined containing such data. Following this, the callback function `XPRSsetcblplog` has a

Listing 4.41 Using callbacks in Visual Basic

```
Public Type MyInfo
  int1 As Long
  int2 As Long
  var1 As Variant
  var2 As Variant
End Type
   ...

Declare Function XPRSsetcblplog Lib "XPRS.DLL" _
     (ByVal XPRSprob As Long, _
     ByVal Address As Long, _
     ByRef p As MyInfo) As Long
   ...

Public Function lplog(ByVal prob As Long, _
     ByRef mi As MyInfo) As Long
   ...
End Function
   ...
mi1.var1 = "The first string"
mi1.var2 = "The second string"
mi1.int1 = 1234
mi1.int2 = 5678
nReturn = XPRSsetcblplog(prob, AddressOf lplog, mi1)
```


new prototype defined to accept data of type `MyInfo` as its second argument. When the callback is set up, a reference to the object `mi1` is passed into the function.

This provides just one example of the use of callbacks under Visual Basic. However, several others may be found on the Xpress-MP CD-ROM.

Using Java with the Xpress-MP Libraries

The BCL and Optimizer libraries may also be used with Java, requiring a recent Java Virtual Machine (JVM) supporting the standard Java Native Interface. Microsoft JVMs are non-standard and are not supported.

The classes of the Java libraries are contained in the package:

```
com.dashoptimization.*;
```

It is recommended that this package be imported in the Java source files which use the Xpress-MP Libraries. The entry point to BCL is the class `XPRBprob` found in this package; for the Optimizer, it is the class `XPRSprob`.

The Java Builder Component Library (BCL)

The Java interface of BCL provides the full functionality of the C version except for the data input, output and error handling, where the standard Java functions should be used. The modeling entities, such as variables, constraints and problems are all converted into classes, and their associated functions into methods of the corresponding class in Java. All such classes (for example `XPRBprob`, `XPRBvar`, `XPRBctr`, `XPRBsos`, `XPRBindexSet`, `XPRBbasis`) take the same name, with the exception of `XPRBindexSet`, where the Java capitalization convention has been employed. Full details of the Java interface can be found in the BCL Reference Manual and the JavaDoc documentation on the CD-ROM.

An example of use is given in Listing 4.42, which is a Java version of the example used throughout the chapter. Several features of this example are worthy of note. The Java interface makes use of an `XPRB` class containing methods relating to the initialization and general status of the software (including `init` and `free`) and the definition of all parameters. In practice, this means that any parameter with the prefix `XPRB_` in standard BCL must be referred to as a constant of the class `XPRB` in Java. For example, `XPRB_BV` in standard BCL becomes `XPRB.BV` here.

Listing 4.42 Using BCL in Java

```
import com.dashoptimization.*;
public class simple
{
  static XPRBprob p;
  public static void main(String[] args)
  {

    XPRBvar a;
    XPRBvar b;

    XPRB.init();
    p = new XPRBprob("simple");

    a = p.newVar("a",XPRB.PL,0,200);
    b = p.newVar("b",XPRB.PL,0,200);

    p.newCtr("First",a.mul(3).add(b.mul(2)).lEql(400));
    p.newCtr("Second",a.add(b.mul(3)).lEql(200));
    p.setObj("Profit",a.add(b.mul(2)));

    p.maxim("");

    System.out.println("Profit: "+p.getObjVal());
    System.out.println(" a = "+a.getSol());
    System.out.println(" b = "+b.getSol());

    XPRB.free();
  }
}
```

In Java it is not possible to overload the algebraic operators for the definition of constraints as it is in C++. Instead, a number of classes have been added to help in this process. For example, linear expressions (class `XPRBlinExp`) are required in the definition of constraints and Special Ordered Sets; quadratic expressions (class `XPRBquadExp`) are used to define quadratic objective functions; linear relations (class `XPRBlinRel`) may be used as an intermediary in the definition of constraints. Further, a set of simple methods, such as `add` or `eql`, are provided which may be overloaded to accept various types and numbers of parameters.

The method `isValid` is a useful addition to BCL which should be used in conjunction with methods `getVarByName`, `getCtrByName` etc. and may require some explanation. These methods always return an object of the desired type, unlike the corresponding functions in standard BCL, which return a `NULL` pointer if the object was not found. Only with the method `isValid` is it possible to test whether the object is a valid object, i.e. whether it is contained in a problem definition. The status of all such objects should be checked in this way before use.

For the remainder of this section differences between the C and Java interfaces will be discussed in the context of the Optimizer library. However, the majority of the issues discussed later in the "Principal Structures" section are relevant to both libraries and BCL users should find little difficulty in applying the points raised there.

The Java Optimizer Library

Most of the routines (methods) described in the Optimizer Reference Manual are now member functions of the `XPRSprob` class. The `XPRSprob` member functions essentially take their names from their C counterparts, albeit dropping their `XPRS` prefix and employing the Java capitalization convention, allowing users to easily translate between the C prototypes and their Java equivalents. There are a few significant variations, however, which we discuss in these pages. For more details of the Java interface, see the JavaDoc documentation on the CD-ROM.

An example of use is given in Listing 4.43 which is again a Java version of the example used throughout the chapter.

Listing 4.43 Using the Optimizer library in Java

```
import com.dashoptimization.*;
import java.io.*;

public class simple
{
  public static void main(String[] args)
  {
```

Listing 4.43 Using the Optimizer library in Java

```
      try
      {
        XPRS.init();
      }
      catch (Exception e)
      {
        System.out.println("Cannot initialize");
      }

      try
      {
        XPRSprob prob = new XPRSprob();
        prob.readProb("simple","");
        prob.maxim("");
        prob.writePrtSol();
       }

      catch(XPRSprobException xpe)
      {
        xpe.printStackTrace();
      }
      XPRS.free();
    }
  }
```

Noteworthy in this example, the Java interface makes use of an `XPRS` class, which contains static members only. These include constant, control and attribute names as well as the two methods `init` and `free`. The function `init` is overloaded, so no `null` or null string argument is required when standard paths are to be checked for license details. `init` throws a standard exception, rather than an `XPRSprobException`, whilst `free` does not throw an exception. See the section "Controls and Problem Attributes" on page 111 for examples of use of the control and problem attribute names.

The Java interface contains no equivalent of the C Optimizer functions `XPRScreateprob` and `XPRSdestroyprob`. The creation and destruction of problem objects is handled in the constructor and `finalize` methods of `XPRSprob`.

Principal Structures

Data Types: Java parameters passed to the routines are related to the C types in the following way:

C argument type	Java argument type
`int*` (array of int)	`int []`
`int*` (reference to an int)	**`IntHolder`**
`double*` (array of double)	`double []`
`double*` (reference to a double)	**`DoubleHolder`**
`char*` (array of char)	`byte []`
`char*` (pointer to a '`\0`' terminated string *passed* to a routine)	`java.lang.String`
`char*` (pointer to a '`\0`' terminated string *received* from a routine)	**`StringHolder`**

Note that the types `IntHolder`, `DoubleHolder` and `StringHolder` are all in the `com.dashoptimization` package.

Numerical Arrays: It is straightforward to use Java arrays in order to pass arrays of integers and doubles. It is worth mentioning that arrays must always be allocated *before* the function call, for example

```
double sol[] = new double[
     prob.getIntAttrib(XPRS.COLS)];
prob.minim("");
prob.getSol(sol, null, null, null);
```

Failure to do so will crash the JVM.

Numerical References: In order to get integers and doubles back from functions, as one would do using pointers in C, one of two wrapper classes can be used: `IntHolder` and `DoubleHolder`. Each of these classes contains a single public member variable of the respective type, named `value`. For example:

```
IntHolder ni=new IntHolder();
prob.getIndex(1,"rowname",ni);
```

```
System.out.println("Index of 'rowname' is:
     "+ni.value);
```

If you need the integer (or double) to be passed both ways, make sure you initialize the value field with the appropriate value before passing it to the function.

Character Arrays and Strings: Some Optimizer routines require strings passed to them as parameters, without modifying them. In this case, `java.lang.String` may be used, for example

```
prob.readProb(args[0],"");
```

where `args[0]` is a `java.lang.String`. If the string passed needs to be modified by the called function, the class `StringHolder` will be used, for example:

```
StringHolder p_name=new StringHolder();
prob.getProbName(p_name);
System.out.println("The problem name is: "
     +p_name.value);
```

Finally, if an array of characters is required by a routine, use a Java array of `byte[]` which has been allocated in advance.

Controls and Problem Attributes: All the control and problem attribute names used by the Optimizer are static final members of the `XPRS` class. Here are two examples of how they can be used:

```
// get a problem attribute
double sol[] = new double[
     prob.getIntAttrib(XPRS.COLS)];
// set a control
prob.setIntControl(XPRS.LPLOG,10);
```

Checking Errors: In certain situations all the `XPRSprob` member functions generate exceptions that must be caught. The exception class `XPRSprobException` is derived from `java.lang.Exception` to which it adds two member functions:

```
public int getCode();
```

which returns the Optimizer error code associated with the exception and

```
public XPRSprob getXPRSprob();
```

which returns a reference to the problem object that caused the exception to be thrown.

In certain cases the `XPRSprob` object will remain valid after the exception is thrown and this may be determined by checking the value of `getCode` (See Chapter 9, "Return Codes & Error Messages" in the Optimizer Reference Manual). Where the `XPRSprob` remains valid, and depending on the context of the exception within the user's application, the exception can be recovered from, allowing the program to continue. An example of this is where a program should start by trying to load a warm-start basis from file. If this fails because the file does not exist (therefore causing an exception) the default initial basis can be used instead. By adding `try-catch` blocks around particular sections of code, such recoverable exceptions may be dealt with. An example is given in Listing 4.44.

Listing 4.44 Handling recoverable exceptions

```
try
{
  XPRSprob prob = new XPRSprob();
     ...
}

catch(XPRSprobException xpe)
{
  xpe.printStackTrace();
  System.out.println("Exit code: "+xpe.getCode());
  System.out.println("Message: "+xpe.getMessage());
  if(xpe.getXPRSprob()!=null)
  {
    ...
  }
}
```

Using Callbacks in Java

In order to take advantage of callbacks from the Optimizer, a flexible event structure is used by the Java Optimizer library API. Each callback is defined as an event in a Listener interface. For an object to be the target of a callback, its class must implement the corresponding listener (e.g. `XPRSmessageListener`, `XPRSlpLogListener`, etc.). Then the object must be registered with the `XPRSprob` object. When an object is no longer interested in receiving events (callbacks), it should be explicitly unregistered. The methods used to register and unregister listeners are:

```
public void add<listener name>Listener(
     Listener listener, Object data)
     throws XPRSprobException;
```

and

```
public void remove<listener name>Listener()
     throws XPRSprobException;
```

Be careful not to destroy the listener object while the Optimizer may still call it.

Note that for coding convenience, any `Object` reference can be passed when registering the listener. The same object reference will be passed back as a parameter to the corresponding *Event* method.

Listing 4.45 provides the simplest example of how the Message callback can be used (the changes from the code in Listing 4.43 are highlighted in bold face).

Listing 4.45 Using callbacks with Java

```
import com.dashoptimization.*;
import java.io.*;

public class CbTest implements XPRSmessageListener
{
  public static void main(String[] args)
  {
```

Listing 4.45 Using callbacks with Java

```
    try
    {
      XPRS.init();
    }
    catch (Exception e)
    {
      System.out.println("Cannot initialize");
    }

    CbTest ML = new CbTest();

    try
    {
      XPRSprob prob = new XPRSprob();

      prob.addMessageListener(ML,null);

      prob.readProb("simple","");
      prob.maxim("");
      prob.writePrtSol();

      prob.removeMessageListener();
    }
    catch(XPRSprobException xpe)
    {
      xpe.printStackTrace();
    }
    XPRS.free();
  }

  public void XPRSmessageEvent(XPRSprob prob,
     Object data, String msg, int len, int type)
  {
    System.out.println(msg);
  }
}
```

Note that for each listener interface the Optimizer will send an event to at most one listener object. A complete list of callbacks and their associated listeners and events may be obtained from the Optimizer Reference Manual following this convention. Refer to the online

documentation for further details of this, or to the Optimizer Reference Manual for usage.

Getting Help

Having worked through the chapter this far, you have learnt how to compile, load and run model programs using the Mosel Libraries, handle multiple models simultaneously and create matrix files as output. You have learnt how to build and solve problems using BCL and write matrix files from this library as well. You have also learnt how to load these files into the Optimizer using the Optimizer library, to optimize them, view the solution and change control variables affecting how the optimization works. Finally you saw how models may be constructed directly within the Optimizer using the 'advanced' functions, and alternative languages were discussed. In Chapter 5 we will focus on the powerful Mosel model programming language to teach you how to construct your own models. Additional help with using the libraries may be found in the form of examples on the Xpress-MP CD-Rom and in the Mosel, BCL and Optimizer Reference Manuals.

Summary

In this chapter we have learnt how to:

- use the Mosel Libraries to compile, load and run models;
- use BCL to build and solve problems;
- solve linear problems and access the solutions using the Optimizer library;
- solve integer problems;
- input problems with the Optimizer library;
- use the Xpress-MP Libraries with other languages.

Chapter 5
Modeling with Xpress-Mosel

Overview

In this chapter, you will:

- construct simple models using the Mosel model programming language;
- develop versatile models through the use of subscripted variables, and constants;
- learn about separation of model data from model structure;
- learn about the use of data files and parameters.

Introduction

Over the course of the last few chapters we have seen how each of the interfaces for Xpress-MP can be used to load and solve a simple model, obtaining the optimal solution to the linear problem. For this chapter, however, we begin to explore the Mosel model programing language in greater detail, learning how to construct new models and to interpret the solution produced. At the same time, the desirability of flexible modeling will also be explored, as well as that of separating

model structure from the data which makes up a particular instance of that model.

The Xpress-Mosel model programing language can often be used in many ways to model the same problem and the examples given here will represent just one approach to this. Through these, we will aim to introduce a large part of the language, although foremost in our minds will always be the development of good modeling practice.

Constructing our First Model

The Burglar Problem

Burglar Bill breaks into a house one night with a sack to carry away items of interest to him. He identifies a number of items which have the following weights and estimated values:

	Weight	Value
Camera	2	15
Necklace	20	100
Vase	20	90
Picture	30	60
TV	40	40
Video	30	15
Chest	60	10
Brick	10	1

Bill can only carry items up to a total weight of 102 pounds. Subject to this, his aim is to maximize the estimated value of the items that he takes.

Problem Specification

Formulating the Burglar Problem mathematically is relatively simple. Suppose that we have a decision variable, *camera*, which has the value 1 if Bill takes the camera and 0 otherwise. Suppose also that we have a similar set of variables for the other items. The Burglar Problem may then be expressed as:

Problem 1

Maximize: 15**camera* + 100**necklace* + 90**vase* + 60**picture* + 40**tv* + 15**video* +10**chest* + 1**brick*

Subject to: 2**camera* + 20**necklace* + 20**vase* + 30**picture* + 40**tv* + 30**video* + 60**chest* + 10**brick* $\leq$ 102

camera, *necklace*, *vase*, *picture*, *tv*, *video*, *chest*, *brick* $\in \{0, 1\}$

"The decision variables..."

This problem has a number of parts which should already be familiar to you. The variables *camera*, *necklace* etc. are also often known as *decision variables* as it is the value of these that must be decided on during the solution process. In standard linear programming (LP) problems, the decision variables can take any non-negative real values and it was a problem of this kind which you will have solved in the past three chapters. In the case of the Burglar Problem, however, it makes little sense to take half a video, or even five necklaces if only one is there. Instead this may be described as a (mixed) integer programming (MIP) problem, with the decision variables only allowed to be either 0 or 1. They are, in fact, *binary* variables.

"The constraint"

The total weight of all the items taken is given by

2**camera* + 20**necklace* + 20**vase* + 30**picture* + 40**tv* + 30**video* + 60**chest* + 10**brick*

The value of this expression is *constrained* by the total weight of items that Bill can carry, or in other words, this must be less than or equal to 102. LP or MIP problems can have any number of constraints such as this, although for the purposes of this model only one is required.

"The objective function..."

The *objective function* is the value of the items taken. The objective of the modeling exercise is to maximize this function subject to the various constraints.

Entering the Model into Xpress-Mosel

Models specified using the Mosel model programming language are divided up into a number of *blocks*, the outermost being the `model` block. It is here that the problem is given a name, which we will choose to be `burglar` for the purposes of this exercise. All blocks must be explicitly ended using an accompanying `end-`*<blockname>* keyword when the statements contained in them are complete.

The problem name forms the basis for a number of files that are generated by the Optimizer and Mosel.

The second main block that we will use initially is the `declarations` block in which the variables are declared, along with their type. Decision variables are of type `mpvar`. We demonstrate this in Listing 5.1.

Listing 5.1 Declaring decision variables

```
model burglar
  declarations
    camera, necklace, vase, picture: mpvar
    tv, video, chest, brick: mpvar
  end-declarations

end-model
```

model

The model block defines the problem name and contains all statements to be considered part of the model. Any text appearing after the `end-model` is treated as if it were a comment and ignored.

declarations

In the `declarations` block are defined variables and their type, sets and scalars.

The decision variables must also be specified as being binary-valued. This is in some sense another constraint on the problem and so it is stated outside of the `declarations` block along with the other constraints. The syntax for this is simply

variable `is_binary`

Similarly, variables can be described as integral using `is_integer`.

Finally the constraint and objective function must be added. These can be assigned a name and take the form:

name `:=` *linear_expression*

"Continuation lines..."

If the linear expression is particularly long, it may be split over several lines as necessary. There is no character or command which denotes a split line to Mosel, but the line is assumed to continue if it ends in such a way that more input is expected. An example might be if the line ends in a binary operator such as '+', following which another term is expected. This is demonstrated in Listing 5.2 where the full model is given.

Listing 5.2 Adding constraints to the model

```
model burglar
  declarations
    camera, necklace, vase, picture: mpvar
    tv, video, chest, brick: mpvar
  end-declarations

  camera   is_binary
  necklace is_binary
  vase     is_binary
  picture  is_binary
  tv       is_binary
  video    is_binary
  chest    is_binary
  brick    is_binary

  TotalWeight := 2*camera + 20*necklace + 20*vase +
                 30*picture + 40*tv + 30*video +
                 60*chest + 10*brick <= 102
```

Listing 5.2 Adding constraints to the model

```
   TotalValue := 15*camera + 100*necklace + 90*vase +
                 60*picture + 40*tv + 15*video +
                 10*chest + 1*brick
end-model
```

The Optimizer Library Module

With the model constructed, it just remains to solve it to find the maximum value of items that Bill can carry away with him. This can also be done from within Mosel. In addition to the standard Mosel syntax, additional functionality can be loaded into it in the form of library modules allowing interaction with external applications. Using the Optimizer library module, `mmxprs`, Mosel can call the Optimizer to solve the problem and return the solutions for display. It does this with the `uses` keyword. The objective function can then be maximized using the `maximize` command. With both modeling and solution components together, this is perhaps most accurately described as a *model program*. A full model program for the problem is given in Listing 5.3, with additions highlighted in bold face.

Library module names take a `mm` prefix to denote 'Mosel Module'.

Listing 5.3 The full burglar model program

This model is on the Xpress-MP CD as file `burglar1.mos`.

```
model burglar1
  uses "mmxprs"

  declarations
    camera, necklace, vase, picture: mpvar
    tv, video, chest, brick: mpvar
  end-declarations

  camera   is_binary
  necklace is_binary
  vase     is_binary
  picture  is_binary
  tv       is_binary
  video    is_binary
  chest    is_binary
  brick    is_binary
```

Listing 5.3 The full burglar model program

```
  TotalWeight := 2*camera + 20*necklace + 20*vase +
                 30*picture + 40*tv + 30*video +
                 60*chest + 10*brick <= 102

  TotalValue := 15*camera + 100*necklace + 90*vase +
                60*picture + 40*tv + 15*video +
                10*chest + 1*brick

  maximize(TotalValue)

  writeln("Objective value is ", getobjval)

end-model
```

The final part of this model program involves how we deal with the solution once found. Using the `writeln` command, Mosel can be instructed to output information such as the objective function value, obtainable using `getobjval`.

uses

Allows for the loading of library modules into Mosel.

maximize / minimize

Optimization commands calling Mosel to optimize the objective function declared as their argument.

writeln

Instructs Mosel to send information to standard output. It can take any number of arguments, and each such is returned in the stated order.

getobjval

Returns the objective function value following solution.

Exercise *Input the model program of Listing 5.3 and compile, load and run it in Mosel using your chosen interface. What is the maximum value of the haul that Bill can make? Which items does he take to achieve this?*

Modeling Using Arrays

Array Variables and Indexing

Creating models in the form described above is convenient for small numbers of decision variables, but as the size of the problem increases it quickly becomes very cumbersome to work with. Already in our first model specification it was untidy having to explicitly state that each variable was binary and would be equally so if we were to use Mosel to output the values of the decision variables. It is usual in modeling even small problems like this to make use of *array variables*, or *subscripted variables* as they are sometimes called.

Listing 5.4 shows an altered declarations section for the Burglar Problem, introducing the various arrays that will be used.

Listing 5.4 declaring and entering data into arrays

```
declarations
  Items = 1..8
  WEIGHT: array(Items) of real
  VALUE:  array(Items) of real
  x: array(Items) of mpvar
end-declarations

WEIGHT := [  2, 20, 20, 30, 40, 30, 60, 10]
VALUE  := [ 15,100, 90, 60, 40, 15, 10,  1]
```

The first thing defined here is an indexing set for the array elements, called `Items`. `Items` is assigned to hold eight integer values, effectively numbering the elements of the arrays. Following this two more arrays are defined, `WEIGHT` and `VALUE` holding respectively the weights and values for the items to be chosen from. They are declared as arrays indexed by `Items` with entries that are of type `real`. The Mosel type `real` corresponds to double precision floats in C. Mosel also supports types `integer` and `string` which have direct analogs in other programming languages. Outside of the `declarations` block these two arrays have their data entered as a comma-separated list.

The final array defined in the `declarations` block is `x` which will act as our decision variable. It is an array, also indexed by `Items`, with entries of type `mpvar`.

Looping and Summation

Using arrays does not just make the variable declaration tidier. By looping through the array of decision variables we can also declare the variables as binary in a single statement. Mosel supports simple looping with the help of the `forall` statement.

Multiple statements may be executed with a `forall` loop by placing them in a `do` / `end-do` block. For details of this and further details of `forall` structure, see page 165.

forall

Allows looping through an index set. Its structure is:

```
forall(var in set) statement
```

where *var* is a dummy variable, *set* is an indexing set and *statement* is to be executed on each such entry.

Mosel also supports summation notation with the `sum` command, used extensively in constructing linear expressions for constraints and the objective function.

sum

Allows summation over an index set. Its structure is:

```
sum(var in set) expression
```

where *var* is a dummy variable, *set* is an indexing set and *expression* is the generic term in the sum.

Using these to define the decision variables as binary and to set up the constraint and objective function results in the statements in Listing 5.5.

Listing 5.5 Looping and summation in the Burglar Problem

```
forall(i in Items) x(i) is_binary
TotalValue := sum(i in Items) x(i)*VALUE(i)
TotalWeight:= sum(i in Items) x(i)*WEIGHT(i)<=102
```

Comments

With the use of arrays and subscripting, a better model for the Burglar Problem is almost complete. However, as with all programming, it is good practice to ensure that your model is adequately commented to make it easier for both yourself and other to read. It has already been mentioned that any amount of commentary can be added following the `end-model` statement, but what happens when you want to place comments in the body of the model itself? For exactly this purpose, the Mosel model programming language supports two types of comments. Single line comments are introduced with the `!` character. Any text following this character is ignored as comment to the end of the line. For multiline commentary, the pair `(!` and `!)` must be used.

With comments added, a model program for the Burglar Problem is given in Listing 5.6.

Listing 5.6 Second attempt at the Burglar Problem

This model is on the Xpress-MP CD as file `burglar2.mos`.

```
model burglar2 (! Modeling with arrays, subscripts,
                  summations, looping and comments !)
  uses "mmxprs"

  declarations
    Items = 1..8                  ! 8 items
    WEIGHT: array(Items) of real
    VALUE:  array(Items) of real
    x: array(Items) of mpvar
  end-declarations

  ! Item:      1,  2,  3,  4,  5,  6,  7,  8
  WEIGHT := [  2, 20, 20, 30, 40, 30, 60, 10]
  VALUE  := [ 15,100, 90, 60, 40, 15, 10,  1]
```

Listing 5.6 Second attempt at the Burglar Problem

```
  ! All x are binary
  forall(i in Items) x(i) is_binary

  ! Objective: maximize the haul
  TotalValue := sum(i in Items) x(i)*VALUE(i)

  ! Constraint: weight restriction
  TotalWeight:= sum(i in Items) x(i)*WEIGHT(i)<=102

  maximize(TotalValue)

  writeln("Objective value is ", getobjval)
  forall(i in Items) writeln(" x(",i,") = ",
    getsol(x(i)))
  writeln
end-model
```

The additional lines at the bottom of the model instruct Mosel to output the values of the decision variables as well as the objective function value. For this another `forall` loop has been used to go through the variable array returning the solution value for each element using `getsol`.

getsol

Returns optimal values of the decision variables following solution of the problem. Its single argument is the variable whose value should be returned.

Exercise *Alter your model program to use array variables, such as in Listing 5.6. Compile, load and run it. Which items are taken by Bill?*

Using String Indices

The model program developed above is considerably simpler to use than our first attempt, but interpreting the solution is made more difficult since we have to manually match up the index number for a variable with the item that it represents. It would be far more useful

if all this information could be presented together. Mosel allows for this with the use of string indices rather than numerical ones.

The use of index sets in the Mosel model programming language is perhaps more simple and natural even than using numerical arrays. Defining `Items` as a string set is the only change necessary and this is demonstrated in Listing 5.7.

Listing 5.7 Using string indices in the Burglar Problem

This model is on the Xpress-MP CD as file `burglar3.mos`.

```
model burglar3
  uses "mmxprs"

  declarations
    Items = {"camera", "necklace", "vase",
             "picture", "tv", "video",
             "chest", "brick"}
    WEIGHT: array(Items) of real
    VALUE:  array(Items) of real
    x: array(Items) of mpvar
  end-declarations

  ! Item:     ca, ne, va, pi, tv, vi, ch, br
  WEIGHT := [  2, 20, 20, 30, 40, 30, 60, 10]
  VALUE  := [ 15,100, 90, 60, 40, 15, 10,  1]

  ! All x are binary
  forall(i in Items) x(i) is_binary

  ! Objective: maximize the haul
  TotalValue := sum(i in Items) x(i)*VALUE(i)

  ! Constraint: weight restriction
  TotalWeight:= sum(i in Items) x(i)*WEIGHT(i)<=102

  maximize(TotalValue)

  writeln("Objective value is ", getobjval)
  forall(i in Items) writeln(" x(",i,") = ",
     getsol(x(i)))
  writeln
end-model
```

Exercise *Alter your model program to make use of string index sets. Compile, load and run it and compare the output with before.*

Versatility in Modeling

Generic and Instantiated Models

The Burglar Problem is just one example of a number of similar modeling problems, known also as knapsack problems. Whilst the above model represents a good first step toward modeling the Burglar Problem, it could still be made easier to generalize should the number of items need to be increased, or even if the problem is changed to a different knapsack problem. With the problem already specified in terms of array variables, the remaining difficulties are largely due to the fact that the data are currently 'hard-wired' into the model — there is no separation between model data and structure. For the current section we shall discuss ways of overcoming these limitations.

Models are generally formulated using symbols to represent the various decision variables, with the relationship between these variables described by a set of equations and inequalities — the constraints. The systematic description of these constraints forms the *generic* model. Combining this with a given set of data produces a particular *numerical* model, the *model instance*, which can then be optimized. Separating off the model structure from the numerical data results in more flexible models which are easier to maintain. For this reason, our aim over the next few pages is to impose such a distinction on the model just created.

Separation also allows you to distribute models as BIM files, protecting intellectual property.

Scalar Declarations

Looking at the model of Listing 5.7, the constraints are already almost in a suitably flexible form. However, the specific information about the maximum weight that Bill can take is still embedded within the body of the model. We can easily get around this by defining a scalar,

MAXWT, assigning this the value 102 and then using that throughout the model instead.

Additional quantities such as this may be assigned values from within the declarations block at the head of the model file. Any quantity which is assigned a value in this way is assumed to be a constant.

Inputting Data From Text Files

The other change we shall make to the model program at this stage is to separate out the model array data and input it from an external data file. Mosel does this through use of the initializations block.

initializations

Enables data to be input from a stated text file. Arrays and constants may be specified in any order and it will be read in at run time.

Suppose that the file burglar4.dat in the Data subdirectory contains the following information:

Listing 5.8 Structure of an external data file

Note that the array values are no longer comma-separated.

```
! Data file for burglar4.mos
WEIGHT: [ 2  20  20  30  40  30  60  10]
VALUE:  [15 100  90  60  40  15  10   1]
```

Then, placing the arrays WEIGHT and VALUE in the initializations block, Mosel will read in the data to fill the arrays at run time. A full listing for the model program implementing this is given in Listing 5.9, with changes highlighted in bold face.

Enter the data of Listing 5.8 into a file burglar4.dat *in the* Data *subdirectory and alter your previous model to input data from this file. Compile, load and run to check this has worked smoothly.*

Listing 5.9 Inputting data from sources files

This model is on the Xpress-MP CD as file `burglar4.mos`.

```
model burglar4
  uses "mmxprs"
  declarations
    Items = {"camera","necklace","vase","picture",
             "tv","video","chest","brick"}
    WEIGHT: array(Items) of real
    VALUE:  array(Items) of real
    MAXWT = 102
    x: array(Items) of mpvar
  end-declarations

  ! Read in data from external file
  initializations from 'Data/burglar4.dat'
    WEIGHT
    VALUE
  end-initializations

  ! All x are binary
  forall(i in Items) x(i) is_binary

  ! Objective: maximize the haul
  TotalValue :=sum(i in Items) x(i)*VALUE(i)

  ! Constraint: weight restriction
  TotalWeight:=sum(i in Items) x(i)*WEIGHT(i)<=MAXWT

  maximize(TotalValue)

  writeln("Objective value is ", getobjval)
  forall(i in Items) writeln(" x(",i,") = ",
     getsol(x(i)))
  writeln
end-model
```

Completing the Burglar Problem

If you have completed the last exercise, the obvious question to ask is 'how much more can we input from the data file?' Certainly the value of the scalar `MAXWT` need not be specified in the model program and can be saved to the data file. The same is actually true of the index set, since the number and names of the items to choose from are really

model data rather than part of the model structure. Suppose, then, that the data file was modified to include the following information:

Listing 5.10 Revised data file for the Burglar Problem

The set is defined as a string array.

```
! Data file for burglar5.mos
Items:["camera" "necklace" "vase" "picture" "tv"
       "video" "chest" "brick"]
MAXWT:102
WEIGHT:[ 2  20  20  30  40  30  60  10]
VALUE:[15 100  90  60  40  15  10   1]
```

If the model program could input all this information at run time, we would finally have separated all the model data from its structure. Listing 5.11 shows the changes to the model program to handle input of this.

Listing 5.11 Completing the Burglar model program

This model is on the Xpress-MP CD as file `burglar5.mos`.

```
model burglar5
  uses "mmxprs"

  parameters
    WeightFile = 'Data/burglar5.dat'
  end-parameters

  declarations
    Items: set of string
    MAXWT: real
    WEIGHT: array(Items) of real
    VALUE: array(Items) of real
    x: array(Items) of mpvar
  end-declarations

  ! Read in data from external file
  initializations from WeightFile
    Items MAXWT WEIGHT VALUE
  end-initializations

  forall(i in Items) create(x(i))
```

Listing 5.11 Completing the Burglar model program

```
  ! All x are binary
  forall(i in Items) x(i) is_binary

  ! Objective: maximize the haul
  TotalValue :=sum(i in Items) x(i)*VALUE(i)

  ! Constraint: weight restriction
  TotalWeight:=sum(i in Items) x(i)*WEIGHT(i)<=MAXWT

  maximize(TotalValue)

  writeln("Objective value is ", getobjval)
  forall(i in Items) writeln(" x(",i,") = ",
    getsol(x(i)))
  writeln
end-model
```

"Understanding Listing 5.11..."

Jumping to the `declarations` block, the first thing to note is that `Items` and `MAXWT` have to have their type declared if their data is to be read in subsequently. `Items` is a (index) set comprising elements which are strings, and we have defined `MAXWT` to have the same type as the elements of the arrays `VALUE` and `WEIGHT`.

If the model is run with just these changes applied, it will fail at run time, claiming that the model is empty. The problem arises from the fact that our decision variable, `x`, has been declared as indexed by a set of, as yet, undetermined size. Since the indexing set is not input until later, the decision variable array is dynamic and is initialized with no entries. In this case, the array *must* be generated explicitly, which we do here using the `create` command.

create

Explicitly creates a variable that is part of a previously declared dynamic array.

In fact, this is not the only way to get around this problem. A second `declarations` block could be included immediately following the

initializations block and the x variable could be declared here instead. An example of the relevant parts of the model is given in Listing 5.12.

Listing 5.12 Using two declarations sections

```
declarations
  Items: set of string
  MAXWT: real
  WEIGHT: array(Items) of real
  VALUE: array(Items) of real
end-declarations

! Read in data from external file
initializations from WeightFile
  Items MAXWT WEIGHT VALUE
end-initializations

declarations
   x: array(Items) of mpvar
end-declarations
```

The linear nature of the way in which Mosel statements are read and interpreted from a model file means that Items is well-defined for all statements in the second set of declarations, since the size and entries in Items are known immediately following the initializations block. For the current model either of these two methods will work perfectly well, although we will work with the former, keeping the blocks together purely for clarity.

Exercise *Alter your model program to look like Listing 5.11. Compile, load and run it as before to solve the problem.*

Using Parameters with Mosel

The final change in Listing 5.11, which has not yet been discussed, is right at the top of the model program and involves the use of the `parameters` block. Using this, parameters may be defined whose values are then input throughout the model program as they are encountered. The values may be numeric or of string type, so if we had wanted to run the problem with a selection of different maximum weights, `MAXWT` could have been defined here as a parameter rather than a scalar. In this example we show how the data file name can be defined initially and then used in the `initializations` block later to input data to create a model instance.

parameters

This block contains a list of parameter symbols along with their default values to be input into the model program at run time if no alternative values are specified.

We illustrate this point with the introduction of a second example.

The Hiker Problem

Henry decides to go on a hiking holiday carrying all the items that he will need in his rucksack. He has five items from which to choose what he will take and each item that he takes represents a certain saving on his holiday. For example, by taking a tent and sleeping bag, he saves on accommodation costs. The items and their relative savings are as follows:

	Weight	Saving
Tent	60	100
Sbag	28	50
Camera	20	35
Clock	8	10
Food	40	50

However, he can only carry items up to a maximum weight of 100 pounds. What items should Henry take to maximize his saving?

Solving the Hiker Problem

This problem is another example of a knapsack problem and as such the model structure is identical to that which we have been creating in this chapter. To solve the problem, therefore, we only need run the same model program again with a different set of model data. This is illustrated in Listing 5.13, where the saving will be loaded into the array `VALUE`.

Listing 5.13 Data file for the Hiker Problem

```
! hiker.dat - Data file for Hiker Problem
Items:["Tent" "Sbag" "Camera" "Clock" "Food"]
MAXWT: 100
WEIGHT: [ 60  28  20   8  40]
VALUE:  [100  50  35  10  50]
```

Exercise *Enter the data of Listing 5.13 into a file* `hiker.dat` *in the* `Data` *subdirectory. The compiled file* `burglar5.bim` *can be loaded into Mosel and run, passing the* `WeightFile='Data/hiker.dat'` *parameter as an argument to* `run`*. Which items should Henry take and what is his maximum saving?*

A full listing demonstrating this using Mosel as a Console application is given in Listing 5.14.

Listing 5.14 Solving the Hiker Problem

```
C:\Mosel Files>mosel
** Xpress-Mosel **
(c) Copyright Dash Associates 1998-zzzz
>load burglar5
>run WeightFile='Data/hiker.dat'
Objective value is 160
 x(Tent) = 1
 x(Sbag) = 1
 x(Camera) = 0
 x(Clock) = 1
 x(Food) = 0

Returned value: 0
>
```

Getting Help

Over the course of this chapter we have developed a relatively simple knapsack problem to introduce some of the basic elements of the Mosel model programming language. Over successive iterations of this process we saw the benefits of separating model structure from its data in terms of versatility in modeling. However a presentation such as this can only really scratch the surface and there are many features of the Mosel language which have not been touched on. In the following chapter a few of these will be briefly discussed to provide a feel for what else is possible, but for full details you should consult the Mosel Reference Manual. This gives a detailed description of all the possibilities in the language as well as describing the functions in some of the standard library modules accompanying the software. The various models described in this chapter can all be found on the installation CD-ROM along with many other examples of using the Mosel language.

Summary

In this chapter we have learnt how to:

- declare variables, arrays, constants and indexing sets;
- enter constraints and the objective function;
- employ the Optimizer library module to solve problems and to use `writeln` to return solution information;
- use subscripted variables, summation and simple looping constructs;
- input data from external files, separating model data from the structure.

Chapter 6
Further Mosel Topics

Overview

In this chapter you will:

- learn about using sets and arrays;
- learn how to import and export data using data files and spreadsheets;
- learn about conditional variables and constraints;
- meet the basic programming structures of the Mosel language;
- learn about writing subroutines in your models.

Introduction

In the previous chapter we saw how models can be described in a number of different ways using the Mosel model programming language, going beyond the trivial examples which had been used as a basis for learning the interfaces in Chapters 2 – 4. Despite the variety of ideas covered there, a number of topics related to the Mosel language are worth a brief discussion. Each topic described here is a self-contained unit and can be read independently of the others if you have a particular interest for your own modeling.

The topics that will be described here are the following:

- constant sets, dynamic sets and set operations;
- multi-dimensional, dynamic and sparse arrays;
- importing data from files and ODBC-enabled products;
- conditional generation of variables and constraints;
- selection and looping constructs;
- writing your own procedures and functions.

Further details of these may be found in the Mosel Reference Manual.

Working With Sets

Sets are collections of objects of the same type, where an ordering is not imposed on their elements. Mosel sets may be defined for any of the elementary types: the basic types (`integer`, `real`, `string` and `boolean`) and the Xpress-MP types (`mpvar` and `linctr`). They can be initialized in three different ways, which we briefly recall.

Constant Sets: Sets whose elements have been declared within a `declarations` section in a model are *constant* sets — their contents cannot subsequently be changed. An example of this would be:

```
declarations
  SnowGear = {'hat','coat',scarf','gloves','boots'}
end-declarations
```

Data importing is covered in detail in "Importing and Exporting Data" on page 148.

File Initialization: Elements of sets may also be stored in data files and imported into a model using an `initializations` block:

```
declarations
  FIB: set of integer
end-declarations

initializations from 'datafile.dat'
  FIB
end-initializations
```

where the file `datafile.dat` contains data such as:

```
FIB: [1 1 2 3 5 8 13 21]
```

Implicit File Initialization: Sets used to index arrays may also be initialized indirectly during initialization of the array. For a model

```
declarations
  REGION: set of string
  DEMAND: array(REGION) of real
end-declarations

initializations from 'transport.dat'
  DEMAND
end-initializations
```

where the file `transport.dat` might contain data such as:

```
DEMAND: [(North) 240 (South) 159 (East) 52 (West) 67]
```

the set `REGION` will be initialized as:

```
{'North','South','East','West'}
```

Dynamic, Fixed and Finalized Sets

Sets which are not constant are considered by Mosel to be *dynamic*, that is, elements may be added to or removed from the set at any point. Once a set has been used to index an array, it becomes *fixed* and elements may no longer be deleted, although further elements can still be added. In many cases, however, the contents of a set do not change once it has been initialized and in such circumstances there is little reason to prefer dynamic sets over constant ones. Rather the opposite is true, for the following reason: Arrays indexed by dynamic sets are themselves created dynamic in Mosel. Since Mosel handles static arrays (those indexed by constant sets) slightly more efficiently than dynamic arrays, it is preferable to employ static arrays in models where possible. For this reason, Mosel provides the `finalize` statement allowing you to turn dynamic sets into constant ones.

In a continuation of the previous example, this might be used as follows:

```
finalize(FIB)

declarations
  x: array(FIB) of mpvar
end-declarations
```

Set Operations

In all examples so far in this manual, sets have been employed purely for the indexing of other modeling objects. However, this need not be the case and we provide details here of a few of the other possibilities available. Some of these are demonstrated in Listing 6.1.

Listing 6.1 Using set operators

```
model Sets
  declarations
    Cits  = {"Rome","Bristol","London","Paris",
     "Liverpool"}
    Ports = {"Plymouth","Bristol","Glasgow",
     "London","Calais","Liverpool"}
    Caps  = {"Rome","London","Paris","Madrid"}
  end-declarations

  writeln("Union of all places: ", Cits+Ports+Caps)
  writeln("Intersection of all three: ",
     Cits*Ports*Caps)
  writeln("Cities that are not capitals: ",Cits-Caps)

end-model
```

This example illustrates the use of the following three set operations:

- set union ('+');
- set intersection ('*');
- set difference ('-').

The first and last of these have associated operators '+=' and '-=' which act on sets in much the same way as they do on numbers, modifying a set subject to the elements of another.

Exercise *Type in and run the example of Listing 6.1 to find the union, intersection and difference of the sets of places. Alter your example to use the operators '+=' and '-='.*

Mosel supports a number of other operators for use with sets. The `in` operator has already been seen in several examples of the `forall` structure, allowing us to loop over all elements of a set. Comparison operators may also be used and include: subset ('<='), superset ('>='), difference ('<>') and equality ('='), returning boolean expressions. Their use is illustrated in Listing 6.2.

Listing 6.2 Using set comparison operators

```
model "Set Comparisons"

  declarations
    RAINBOW = {"red","yellow","orange","green",
     "blue","indigo","violet"}
    BRIGHT = {"orange","yellow"}
    DARK = {"blue","brown","black"}
  end-declarations

  writeln("BRIGHT is included in RAINBOW: ",
     BRIGHT<=RAINBOW)
  writeln("BRIGHT is not the same as DARK: ",
     BRIGHT<>DARK)
  writeln("RAINBOW contains DARK: ", RAINBOW>=DARK)

end-model
```

Exercise *Enter the model of Listing 6.2, altering it to include the other comparison operators in output statements. What is the output produced?*

Working with Arrays

In contrast to sets, arrays are *ordered* collections of objects of the same type and may be defined for any of the elementary types: the basic types (`integer`, `real`, `string` and `boolean`) and the Xpress-MP types (`mpvar` and `linctr`). Arrays may be indexed by a set, examples of which have been seen in the last section, or more simply using the natural numbers without reference to a defined set.

Multi-Dimensional Arrays

Initializing and entering data into arrays in one dimension may be simply achieved in the following way.

```
declarations
  OneDim: array(1..10) of real
end-declarations

OneDim:= [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

We have already seen many examples of setting up one-dimensional arrays in Chapter 5. Initializing multi-dimensional arrays can be achieved in exactly the same manner, although in this case formatting can be used advantageously to make the result more intuitive. In two dimensions, this could be done as follows:

```
declarations
  TwoDim: array(1..2,1..3) of real
end-declarations

TwoDim:= [11, 12, 13,
          21, 22, 23]
```

which is, of course, syntactically the same as

```
TwoDim:= [11, 12, 13, 21, 22, 23]
```

These statements are interpreted by Mosel by placing the values into the array `TwoDim` row-wise — it is the last subscript which varies most rapidly. Thus, the values in the arrays are the following:

```
TwoDim[1][1] = 11
TwoDim[1][2] = 12
TwoDim[1][3] = 13
TwoDim[2][1] = 21
TwoDim[2][2] = 22
TwoDim[2][3] = 23
```

In higher dimensions, the same principle may be applied. Listing 6.3 describes a small model which enters data into a three-dimensional array and then prints out the array elements along with their values.

Listing 6.3 Initializing a three-dimensional array

```
model ThreeDimEx
  declarations
    ThreeDim: array(1..2,1..3,1..4) of integer
  end-declarations

  ThreeDim:= [111, 112, 113, 114,
              121, 122, 123, 124,
              131, 132, 133, 134,
              211, 212, 213, 214,
              221, 222, 223, 224,
              231, 232, 233, 234]

  forall(i in 1..2, j in 1..3, k in 1..4) do
    writeln("ThreeDim(",i,",",j,",",k,") = ",
            ThreeDim(i,j,k) )
  end-do
end-model
```

See page 165 for details of the 'forall' structure.

Exercise *Enter the model of Listing 6.3 and compile, load and run it. Check that the array element indices correspond to the element values.*

Fixed and Dynamic Arrays

Arrays in the Mosel language may have either a fixed size, or be dynamically sized as the model program is run. By default an array is created of fixed size if all its indexing sets are of fixed size, in other words if they are constant, or have had their sizes *finalized*. Fixed arrays have space for all their cells created at the point of declaration with uninitialized cells given the value `0` or `''` (the empty string) depending on the array type.

For details of the possible set types, see "Working With Sets" previously.

By contrast, if the size of an array is not known at the point of declaration, it is created *dynamic*. This might occur if one of its index sets does not have a fixed size, possibly because it is yet to be read in from an external file. Dynamic arrays are created empty and have cells added as they are assigned values, allowing the array to grow as necessary. An array may further be forced to be dynamic by using the `dynamic` qualifier.

If an array of type `mpvar` is either declared as dynamic, or becomes implicitly so, with the size of at least one of its indexing sets unknown at declaration, the corresponding variables are not created. In such circumstances, the individual elements must all be created by hand using the `create` command. If this is not done, the array will have zero size, no decision variables will exist and the problem will be empty. An example of this was seen in Listing 5.11 in the previous chapter and we shall encounter it again in the section "Conditional Variables and Constraints" on page 159.

Sparsity

Almost all large scale LP and MIP problems have a property known as *sparsity*, that is each variable appears with a nonzero coefficient in a very small fraction of the total set of constraints. Often this property is reflected in the data arrays used in the model, with many zero values. When this happens, it is more convenient to provide just the nonzero values of the data array rather than listing all the values. This is also the easiest way to input data into data arrays with more than two dimensions. An added advantage is that less memory is used.

Suppose that we have a data file, `SparseEx.dat`, which contains the information in Listing 6.4.

Listing 6.4 A sparse data format array example

```
! SparseEx.dat - Data for SparseEx.mos
COST: [(Depot1 Customer1) 11
       (Depot2 Customer1) 21
       (Depot3 Customer2) 32
       (Depot2 Customer2) 22
       (Depot1 Customer3) 13]
```

In this file format, the bracketed terms denote the place in the array where the value is to be stored.

In this example an element of the array `COST` has a nonzero value assigned for sending a product from `Depot1` to `Customer1`, from `Depot2` to `Customer1` and so on. However, since no cost is defined for sending a product from `Depot2` to `Customer3`, the array is considered sparse. In Listing 6.5, the model reads in the data contained in this file and prints a list of all possible `DEPOT-CUSTOMER` pairs with their associated costs. Those elements which had no cost assigned are given the value `0`.

Listing 6.5 Inputting data in sparse data format

```
model SparseEx
  declarations
    DEPOT, CUSTOMER: set of string
    COST: array(DEPOT,CUSTOMER) of integer
  end-declarations

  initializations from 'SparseEx.dat'
    COST
  end-initializations

  forall(d in DEPOT, c in CUSTOMER)
    writeln(d," -> ",c,": ",COST(d,c))
end-model
```

In this example, the array `COST` is dynamically sized since the sizes of neither of its indexing sets were known when it was declared. The only space that is used to hold data corresponds to the number of entries added — space for the extra zero elements is not created. This can be seen by using the command `display COST` at the console.

Exercise *Enter the above model and run it in Mosel. By displaying the 'value' of the array* `COST`*, check that this array is dynamic.*

Importing and Exporting Data

There are obvious benefits to be gained from separating the form, or structure, of a model from the particular data that make up a model instance. The Mosel model programming language encourages this modeling principle and incorporates a powerful set of facilities for importing and exporting data.

It is possible to enter data into Mosel model program arrays in four main ways:

- directly in the model program file;
- by use of the `initializations` block;
- by use of ODBC;
- by use of the `readln` command.

The latter three ways also provide corresponding methods for the export of data following solution. In this section we cover each of these and discuss their relative benefits.

Model Data Entry

Perhaps the simplest method of entering data in Mosel arrays involves embedding data directly in the model itself. We have seen a number of examples of this in the previous chapter, but a further example of how data may be entered in this way is provided in Listing 6.6.

In this example a one-dimensional array of size five is created and data is entered: `A(1)` is assigned the value `1`, `A(2)` the value `2.5` and so on. Finally the array is displayed to the console using a `writeln` statement.

Listing 6.6 Native data entry into arrays

```
model NativeEx
  declarations
    A: array(1..5) of real
  end-declarations

  A:=[1, 2.5, -6.1, 10, 77]

  writeln("A is: ",A)
end-model
```

Whilst quick and easy to use, the downside to this method is that it does not in any way separate model structure from its data. Its use is therefore perhaps limited to the creation of small test examples or the development of prototype models.

Data Transfer Using the `initializations` Block

The `initializations` block may be used to initialize basic objects such as scalars, arrays or sets from external data files and can be used to export solution data later in the model program.

Importing Data from ASCII Files: An initialization data file must contain one or more records of the form:

```
label: value
```

where `label` is a text string and `value` is either a constant, or a collection of values separated by spaces and enclosed in square brackets. Collections of values are used to initialize sets or arrays. During data input, each object to be initialized is associated to a `label` in the external file. This is typically the same label as the object name, but may be different if the modifier `as` is used. When an `initializations` block is executed, the given file is opened and the requested labels are searched for in this file to initialize the corresponding objects.

No particular formatting is required in the file: spaces, tabs and line breaks are all normal separators. Moreover, single line comments are

also supported within the file. An example of such a file is given in Listing 6.7.

Listing 6.7 ASCII file format to use with `initializations`

```
! InitEx.dat: Using the initialization block
Item: 25
albatross: [12.9 76 1.55 0.99]
A1: [23 15 43 29 90 180]
```

In this file one integer (`Item`) is defined along with two arrays (`A1` and `albatross`). Listing 6.8 provides an example model file which reads these values into memory.

Listing 6.8 Model file to demonstrate input from ASCII files

```
model InitEx
  declarations
    Item: integer
    A1: array(1..6) of integer
    A2: array(1..4) of real
  end-declarations

  initializations from 'InitEx.dat'
    Item A1
    A2 as "albatross"
  end-initializations

  writeln("Item is: ",Item)
  writeln("A1 is: ",A1)
  writeln("A2 is: ",A2)
end-model
```

Several things are worth noting in this example. Firstly, the name of the data file should appear on the same line as the beginning of the `initializations` block, quoted and following the word `from`. Since the `initializations` block can also be used for data export,

the `from` modifier specifies the sense of the data transport. In this case we are obtaining information from an external source.

Second, since items in the `initializations` block are separated by spaces, several of these may be placed on the same line. In the example, we have done this with `Item` and `A1`. It should also be noted that objects here need not be in the same order as the data in the accompanying data file, although it is often sensible to maintain the order, if only for the sake of clarity.

Finally, the array data associated with the label `albatross` in the data file is to be read into the array `A2`. This is achieved in the model file by use of the `as` modifier. Since the label in the data file must be quoted in the model file when this construct is used, labels in the model file may consist of more than one word if necessary.

Exercise *Type in the example above, or create your own to read in and display data. Your example should demonstrate all of the possibilities described above.*

Exporting Data to ASCII Files: An equivalent method may also be used to export data from Mosel to external data files following solution of the problem, with the format for the `initializations` block being:

```
initializations to filename
  identifier [as label]
end-initializations
```

When this form is executed, the values of all provided labels in the file are updated with the current value of the corresponding identifier. If a label cannot be found, a new record is appended to the end of the file and the file is created if it does not exist.

Exercise *Alter your model from the exercise above to export the data to a separate file using this format. Consult the new file with a text editor or similar to see how data has been entered.*

Data Transfer Using ODBC

Creating and maintaining data in text files is quite a simple process, but for many people a considerably more efficient and useful format is provided by spreadsheets and databases. A facility exists in the Mosel language whereby data may be imported from, and exported to, ODBC-enabled spreadsheet and database programs using the structured query language, SQL. To do so requires the use of the library module `mmodbc`, in addition to an extra authorization in your Xpress-MP license.

If you are going to work through the examples here, you will need access to Microsoft Excel.

Setting Up ODBC: Suppose that in a spreadsheet called `Myss.xls` the following data has been entered into the stated cells and the

	A	B	C
1			
2		First	Second
3		6.2	1.3
4		-1.0	16.2
5		2.0	-17.9

range `B2:C5` has been called `MyRange`. When data is imported into Xpress-MP from a spreadsheet, the first row in the marked range is always assumed to contain column headings and is consequently ignored. In this example, data from `MyRange` would fill an array of size `3*2` and in the following, ODBC will be used to extract this data into an array `A(3,2)`.

A spreadsheet range must have a top row with column titles.

Using ODBC

In Windows' Control Panel, select *32-bit ODBC* and set up a User Data Source called `MyExcel` by clicking *Add*, selecting *Microsoft Excel Driver (*.xls)* and filling in the ODBC Microsoft Excel Setup dialog. Click *Options* >> and clear the *Read Only* check box.

Exercise *Set up a user data source for Excel, as described in the box 'Using ODBC'.*

Importing Data From ODBC-Enabled Products: In Listing 6.9 we demonstrate how data may be read into Mosel from an Excel spreadsheet such as this. Notable here is that SQL syntax must be used to obtain the required data from the spreadsheet and the statement '`SELECT * FROM MyRange`' says, quite simply, that everything in the range `MyRange` is to be returned. This is then placed into the array `A` as required.

Listing 6.9 Reading in data from an Excel spreadsheet

```
model ODBCImpEx
  uses "mmodbc"

  declarations
    A: array(1..3,1..2) of real
    CSTR: string
  end-declarations

  CSTR:= 'DSN=MyExcel; DBQ=Myss.xls'

  setparam("SQLndxcol",false)
  SQLconnect(CSTR)
  SQLexecute("SELECT * FROM MyRange",[A])
  SQLdisconnect

  forall(i in 1..3)do
    writeln("Row(",i,"): ",A(i,1)," ",A(i,2))
  end-do
end-model
```

Exercise *Construct a model (or alter one of your previous ones) which obtains its data from Excel in this way.*

Exporting Data to ODBC-Enabled Products: Exporting data back to spreadsheets can be achieved in much the same manner. An example of this is provided in Listing 6.10, where a new sheet is created, `New`, into which data from the array `A` is entered.

Listing 6.10 Writing data to an Excel spreadsheet

```
model ODBCExpEx
  uses "mmodbc"

  declarations
    A: array(1..3,1..2) of real
  end-declarations

  forall(i in 1..3,j in 1..2) A(i,j) := i*j

  setparam("SQLndxcol",false)
  SQLconnect('DSN=MyExcel; DBQ=Myss.xls')
  SQLexecute("CREATE TABLE New(Col1 integer, Col2
     integer)")
  SQLexecute("INSERT INTO New(Col1,Col2) VALUES
     (?,?)",A)
  SQLdisconnect

end-model
```

Since SQL is used for any communication between Mosel and ODBC-enabled applications, far more complicated statements than this are possible. You should consult any standard text on SQL for details of the possibilities.

Exercise *Alter your previous model to output data back into Excel following solution of a problem, or alter one of the previous examples to do this. If you encounter problems, consult the following section.*

Opening and Using Microsoft Excel Tables: A number of points should be considered when writing data to Microsoft Excel from Mosel. Since Excel is a spreadsheet application and ODBC was primarily designed for databases, special rules have to be followed to read and write Excel data using ODBC:

- named ranges must be used in an Excel worksheet to refer to tables of data;
- column names must be used as field names;

- the data type of each field should be defined using a row of specimen data below the column headings.

When using ODBC with Excel, it is important that the top row of each range contains the column headings — otherwise errors will occur and data will not be transferred correctly to and from the worksheet. A row of specimen data is also required in the row below the column headings to identify the data type of each column. This specimen row must also be included in the range.

Users should also be aware that when writing to database tables specified by a named range in Excel, the range will increase in size if new data is added. Now suppose that we wish to write further data over the top of data that has already been written to a range using ODBC. Within Excel it is not sufficient to delete the previous data by selecting it and hitting the Delete key. If this is done, further data will be added after a blank rectangle where the deleted data used to reside. Instead, it is important to use *Edit*, *Delete*, *Shift cells up* within Excel, which will eliminate all traces of the previous data, and the enlarged range.

Microsoft Excel tables can be created and opened by only one user at a time. However, the 'Read Only' option available in the Excel driver options allows multiple users to read from the same `.xls` files.

Sizing Arrays for Spreadsheet Data: Since Mosel evaluates objects that it encounters in the order in which they are encountered, the sizes of tables may be adjusted dynamically. In practice, this allows for the possibility that, having written a model, the size of the region in the spreadsheet may actually change if additional data is entered. Since this is a particularly useful trick, we will briefly describe its usage here.

Suppose that we are continuing to work with the spreadsheet `Myss.xls` and we are concerned that the size of `MyRange` may change as more data are added. This can be dealt with by constructing an additional region of the spreadsheet, which we will name `Sizes`. Into this we will put the numbers that characterize the problem as follows:

Reducing the number of rows by 1 allows for the row containing just the column names.

Number of Rows	Number of Columns

=ROWS(MyRange)-1	=COLUMNS(MyRange)

Naming the range formed by the two cells in the first column as `NRows` and the range formed by the two cells in the second column as `NCols`, the commands in Listing 6.11 may then form the introductory part of a model.

Listing 6.11 Dynamic table sizing with spreadsheets

```
model SizingEx
  uses "mmodbc"

  declarations
    NRows, NCols: integer
  end-declarations

  SQLconnect('DSN=MyExcel;DBQ=Myss.xls')
  NRows:=SQLreadinteger("select NRows from Sizes")
  NCols:=SQLreadinteger("select NCols from Sizes")

  declarations
    A: array(1..NRows,1..NCols) of real
  end-declarations

  SQLexecute("select * from MyRange",[A])
  SQLdisconnect

  forall(i in 1..NRows,j in 1..NCols) do
    writeln("A(",i,",",j,") = ", A(i,j))
  end-do
end-model
```

Exercise *Alter your previous model to allow for dynamic resizing of the tables. Now change the number of rows with data to be read in and run the new model in Mosel. Check that the extra data is imported.*

Data Transfer Using `readln` and `writeln` Commands

A considerably more general method for reading data from ASCII files is provided by the `readln` and `read` commands. These commands

assign the data read in from the active input stream to given symbols, or attempt to match a given expression with what is read in. The command `readln` expects all symbols to be recognized to be contained in a single line, whereas `read` allows this to flow over multiple lines.

Use of these commands is perhaps best illustrated by way of an example. Suppose we have an input data file containing information such as that described in Listing 6.12.

Listing 6.12 File format for use with `readln`

```
! File ReadlnEx.dat
read_format( 1 and 1)= 12.5
read_format( 2 and 3)= 5.6
read_format(10 and 9)= -7.1
```

This file, `ReadlnEx.dat`, is then read in by the model program which is described in Listing 6.13. The model begins in the usual way by declaring variables necessary for the problem. The data in the file are to be read into the array `A`, which is sparse and to be sized dynamically.

Listing 6.13 Reading in data with `readln`

```
model ReadlnEx
  declarations
    A: array(range,range) of real
    i, j: integer
  end-declarations

  fopen("ReadlnEx.dat",F_INPUT)
  readln("!")
  repeat
    readln("read_format(",i,"and",j,")=",A(i,j))
  until getparam("nbread") < 6
  fclose(F_INPUT)

  writeln("A is: ",A)
end-model
```

Note here that the dynamic array `A` is indexed by a dynamic set, `range`.

See page 167 for details of the '`repeat`' structure.

Using the `fopen` command, the data file is opened and assigned to the active input stream, `F_INPUT`. We then read a line containing the `!` character, since the first line of our file will contain a comment. Following this, the repeat loop reads in as many lines as it can in the format described in Listing 6.12, assigning the value after the = sign to an array element indexed by the numerical values spanning the word 'and'. Finally the input stream is closed and the array is printed to the console.

During a run of the program, the commands `read` and `readln` set a control parameter, `nbread`, to the number of items actually recognized in a line. By consulting this, it becomes evident when information has been read in which does not match a given string. In our example, after the first line we might expect six items per line to be recognized by the parser. This can be used to end the `repeat` loop when there is no further data to be read in.

Exercise *Create your own model which uses* `readln` *and* `read` *to input data from an external source file, displaying the data that it has read.*

Exporting Data With `write` and `writeln`: Array and solution data may be written to file in much the same way using the `writeln` and `write` commands. By opening a file and assigning it the active output stream, `F_OUTPUT`, any subsequent `write` or `writeln` commands will direct output to the file rather than to the console. An example of this is given in Listing 6.14 where the simple problem of Chapters 2 – 4 is again solved and solution values exported to a data file.

It should be noted that when information is exported in this manner, any information currently in the file being written to is lost and the file is completely overwritten with the new data. When this is undesirable, files may instead be opened for appending using:

```
fopen(Filename, F_APPEND)
```

Exercise *Enter the model of Listing 6.14 and run it, writing data to the file* `WritelnEx.dat`. *Now alter the model to append data and run the program again. Consult the output file with a text editor.*

Listing 6.14 Outputting solution data with `writeln`

```
model WritelnEx
  uses "mmxprs"

  declarations
    a,b: mpvar
  end-declarations

  first:=  3*a + 2*b <= 400
  second:=   a + 3*b <= 200
  profit:=   a + 2*b

  maximize(profit)

  fopen("WritelnEx.dat",F_OUTPUT)
  writeln("Profit = ",getobjval)
  writeln("a=",getsol(a),": b=",getsol(b))
  fclose(F_OUTPUT)
end-model
```

Conditional Variables and Constraints

Conditional Bounds

Suppose that we wish to apply an upper bound to some, but not all members of a set of N variables, x_i. The upper bound that will be applied varies for each variable and these are given by the set U_i. However, they should only be applied if the entry in the data table $COND_i$ is greater than some other amount, say 20. If the bound *did not* depend on the value of $COND_i$, as has been usual up to now, then this might have been expressed using:

```
forall(i in 1..N) x(i) <= U(i)
```

For the conditional bound, however, this must be altered slightly:

```
forall(i in 1..N | COND(i) > 20) x(i) <= U(i)
```

| The vertical bar (|) character, followed by a logical expression denotes a conditional operator and should be read as "to be done whenever the following expression is true", or "such that".

Thus the line in the second of these examples reads "for `i=1` to `N`, the variable `x(i)` must be less than or equal to `U(i)` whenever `COND(i)` is greater than `20`".

Conditional Variables

The existence of variables can also be made conditional by declaring a dynamic array of variables and then creating only those which satisfy a certain condition. An example of this is given in Listing 6.15. The only variables which will actually be defined here are `x(1)`, `x(2)`, `x(3)`, `x(6)`, `x(8)` and `x(10)`. By constructing a small model and outputting it to the console, this is evident.

We have already seen an example of this in Chapter 5, where the indexing set was also input from an external file. For details, see Listing 5.11 on page 132.

Listing 6.15 Conditional generation of variables

```
model ConditionalEx
  declarations
    Elements = 1..10
    COND: array(Elements) of integer
    x: dynamic array(Elements) of mpvar
  end-declarations

  COND:= [1, 2, 3, 0, -1, 6, -7, 8, -9, 10]
```

Listing 6.15 Conditional generation of variables

```
  forall(i in Elements | (COND(i) = i)) create(x(i))

  ! build a little model to show what's there
  obj:= sum(i in Elements) x(i)
  c:= sum(i in Elements) i*x(i) >= 10
  exportprob(0,"",obj)
end-model
```

Exercise *Create and run the model of Listing 6.15. Check that only six variables exist by checking the LP format matrix of the constructed problem.*

Use of `exportprob` in this way provides a nice way of seeing directly the problem that has just been created. Without specifying a file name for output, the matrix is displayed on screen.

Basic Programming Structures

The Mosel model programming language provides all the possibilities of a high-level programming language in addition to commands and procedures for model specification. This provides powerful additions to the modeling environment, some of which we shall now discuss.

`if` Statements

The general form of the `if` statement is:

```
if expression_1
then commands_1
[ elif expression_2
then commands_2 ]
[ else commands_3 ]
end-if
```

The selection is executed by evaluating the boolean *expression_1*. If this is `true`, then *commands_1* are executed and control passes to after the `end-if` statement. If optional `elif` sections are included, then the boolean *expression_2* is evaluated and if this is `true`, then *commands_2* are executed, before control passes to after the `end-if` statement. If none of the expressions prefixed with either `if` or `elif` evaluate to `true`, then any *commands_3* following an optional `else` statement are executed. The program continues from after the `end-if` statement. An example of this is given in Listing 6.16.

Listing 6.16 Using `if` statements in modeling

The `parameters` block lists quantities which may be changed at run time, along with their default values.

```
model IfEx
  parameters
    DEBUG=0
    File='InitEx.dat'
  end-parameters

  declarations
    A: array(1..6) of integer
    Item: integer
  end-declarations

  initializations from File
    Item A as "A1"
  end-initializations

  if(DEBUG=1) then
    writeln("Item is ",Item,"\nA is ",A)
  else
    writeln("Data read in from ",File,"...")
  end-if
end-model
```

The command sequence `\n` is interpreted as a newline character only when enclosed in double quotes.

In this example an integer `DEBUG` is used to control output during a run of a model. For normal use, `DEBUG` is set to `0`, so the only statement output by Mosel is to tell us when and from where the data have been read in. However, since `DEBUG` is a parameter, its value can be changed at run time and if it is set to `1`, the data read in will also be displayed to confirm that everything is set correctly. Listing 6.17 shows an example session with Mosel in which this model program is

run first as normal and then subsequently with the DEBUG parameter set to 1.

Listing 6.17 Using the model debugging feature

```
C:\Mosel Files>mosel
** Mosel **
(c) Copyright Dash Associates 1998-zzzz
>cload IfEx
Compiling `IfEx'...
>run
Data read in from InitEx.dat...
Returned value: 0
>run DEBUG=1
Item is 25
A is [23,15,43,29,90,180]
Returned value: 0
>
```

Another example like this can be found in Listing 6.22 on page 170.

Using this facility provides a particularly convenient way of debugging long models. It is a common trick to use the writeln command during debugging to output values of variables when trying to detect the cause of an error. If any statements added for this purpose are enclosed in an if statement of this form, then they can be left in the model when debugging has been completed, rather than having to identify and remove the extra statements. If further details are to be subsequently added to the model, then additional debugging may be required. The next time around, your debugging statements will already be in place to help.

Exercise *Enter and run the example of Listing 6.16 both as normal and with the* DEBUG *feature enabled. Now alter the model program using an* elif *statement to add an extra* DEBUG *level of* 2 *which additionally prints out the file name from which the data is collected.*

`case` Statements

The general form of the `case` statement is:

```
case Expression_0 of
  Expression_1 : Statement_1
  or
  Expression_1 : do Statement_list_1 end-do
  [
    Expression_2 : Statement_2
    or
    Expression_2 : do Statement_list_2 end-do
  ...]
  [ else Statement_list_3]
end-case
```

The selection is executed by evaluating the boolean *Expression_0* and sequentially comparing it with each *Expression_i* until a match is found. At this point either *Statement_i* or *Statement_list_i* is executed (depending on the form of the construction) and control passes to after the `end-case` statement. If none of the expressions match and the `else` statement is present, then the statement(s) *Statement_list_3* are executed and control passes to after the `end-case`. A simple example demonstrating this is given in Listing 6.18.

In this example the entries in an array `A` are searched through and categorized according to a (somewhat bizarre) rule: it is interesting to know whether they are either `0`, in the range `1` to `3` inclusive, or else either `8` or `10`. Anything falling outside of this is declared as 'not interesting' to us. Each element in `A` is categorized and a statement about its value is printed to the screen.

Exercise *Construct your own model program which makes use of the* `case` *construction. Alternatively, adapt your previous program to use the* `case` *construction to handle multiple debugging levels in code.*

Listing 6.18 A simple `case` example

```
model CaseEx
  declarations
    A: array(1..10) of integer
  end-declarations

  A:=[1,2,3,0,-1,1,3,8,2,10]

  forall(i in 1..10) do
    case A(i) of
      0:    writeln('A(',i,') is 0')
      1..3: writeln('A(',i,') is between 1 and 3')
      8,10: writeln('A(',i,') is either 8 or 10')
      else  writeln('A(',i,') is not interesting')
    end-case
  end-do
end-model
```

`forall` Loops

The general form of the `forall` statement is:

`forall` (*Iterator_list*) *Statement*
or
`forall` (*Iterator_list*) `do` *Statement_list* `end-do`

Here the *Statement* or *Statement_list* is repeatedly executed for each possible index tuple generated by the *Iterator_list*. An example of this is provided in Listing 6.19.

In this example, Mosel fills the array `F` with the first 20 numbers in the Fibonacci sequence, printing them to the console as it goes. In this sequence the first two numbers are `1` and subsequent numbers are generated as the sum of the two preceding. By looping through the indexing set `Elems`, a simple `if` statement determines if we are initializing the first two terms and, if not, applies the algorithm to determine the next term in the sequence.

Listing 6.19 A simple `forall` example

```
model ForallEx
  declarations
    Elems=1..20
    F: array(Elems) of integer
  end-declarations

  forall(i in Elems) do
    if(i=1 or i=2) then
      F(i):=1
    else F(i):= F(i-1) + F(i-2)
    end-if
    writeln("F(",i,")\t= ",F(i))
  end-do
end-model
```

The command sequence `\t` is interpreted as a tab character only when enclosed in double quotes.

Other examples that we have seen involving the `forall` loop include setting decision variables as binary in Listing 5.6 and explicitly creating variables in Listing 5.11, both in the previous chapter.

Exercise *Create your own model program making use of the* `forall` *loop, or enter and run the example given here.*

`while` Loops

The general form of the `while` statement is:

`while` (*Expression*) *Statement*
or
`while` (*Expression*) `do` *Statement_list* `end-do`

With this construction, the boolean *Expression* is evaluated and the *Statement* or *Statement_list* is executed as long as *Expression* is `true`. If *Expression* evaluates to `false`, the `while` statement is completely skipped. An example is given in Listing 6.20.

The `while` statement is used here to construct a 'times table', printing the product of the row and column numbers. At the end of each row

Listing 6.20 A simple `while` example

```
model WhileEx
  declarations
    i,j: integer
  end-declarations

  i:=1
  j:=1

  while(i <= 10) do
    while(j <= 10) do
      write(i*j)
      if(j=10) then writeln
      else write("\t")
      end-if
      j+=1
    end-do
    j:=1
    i+=1
  end-do
end-model
```

a newline character is printed, whilst between numbers in a row the tab character is used.

Exercise *Either enter the example in the listing or create your own example making use of the* `while` *statement. Compile, load and run it to check output.*

`repeat` Loops

The final looping structure is the `repeat` loop, which has the following general form:

`repeat` *Statement_1*
[Statement_2...]
`until` *Expression*

Here *Statement_1* (and any further statements) are repeatedly executed until the boolean *Expression* evaluates to `false`. By contrast with the `while` loop, statements in a `repeat` loop are guaranteed to be executed at least once, since the execution takes place before the *Expression* is evaluated. An example is given in Listing 6.21.

Listing 6.21 A simple `repeat` example

```
model RepeatEx
  declarations
    i: integer
    number: integer
  end-declarations

  i:=1
  writeln("Input a positive integer:")
  readln(number)

  repeat i+=1
  until ((number mod i =0) or i>sqrt(number))

  if(i>sqrt(number)) then writeln(number," is prime!")
  else writeln(number," is divisible by ",i)
  end-if
end-model
```

This example provides a short program testing numbers to see if they are prime. When run, the user is prompted to enter a number which is then repeatedly tested until either a divisor is found or it is found to be prime.

Exercise *Enter the example of Listing 6.21 and run it. Now alter it to test numbers input to make sure they are integral and bigger than zero. You may need to consider separately how to cope with the input of* `1`.

Procedures and Functions

It is possible to group together sets of statements and declarations in the form of subroutines which can be called several times during the execution of a model. Mosel supports two kinds of subroutines: procedures and functions. Both of these can take parameters, define local data and call themselves recursively.

Procedures

Procedures consist of a collection of statements which should be run together in a number of places. On execution the statements in the body are run and no value is returned.

procedure

The procedure block takes the form:

```
procedure proc_name [ (param_list) ]
  proc_body
end-procedure
```

where *proc_name* is the procedure's name and *param_list* its list of formal parameters, separated by commas. When the procedure is called, statements in *proc_body* are executed.

An example of a procedure is given in Listing 6.22, printing a banner at the beginning of a model run. For models using parameters, an example such as this can provide a useful way of classifying output, particularly if used for information sent to file rather than to the screen. Listing 6.23 shows the use of this program from Console Xpress.

Create a simple procedure outputting a banner at the beginning of a model run. If your model makes use of parameter or other settings read in from external files, you could have these output as well.

Listing 6.22 A simple procedure

```
model ProcEx
  parameters
    DEBUG=0
    File='InitEx.dat'
  end-parameters

  procedure banner(DEBUG:integer,File:string)
    writeln("This is the ProcEx model, release 1")
    if(DEBUG = 1) then
      writeln("\tDebugging on...")
    end-if
    if(File <> 'InitEx.dat') then
      writeln("\tInput file ",File," in use...")
    end-if
    writeln
  end-procedure

  banner(DEBUG,File)
end-model
```

Procedures may be placed together at the end of the model as long as they are declared before use using the `forward` keyword. See the following sections for details.

Listing 6.23 Running the procedure with parameter input

```
C:\Mosel Files>mosel
** Mosel **
(c) Copyright Dash Associates 1998-zzzz
>cload ProcEx
Compiling `ProcEx'...
>run
This is the ProcEx model, release 1

Returned value: 0
>run 'DEBUG=1,File=data.dat'
This is the ProcEx model, release 1
        Debugging on...
        Input file data.dat in use...

Returned value: 0
>
```

Extra debugging information is also made available by using the `g` flag with `compile` or `XPRMcompmod`.

Functions

A function is a collection of statements which should be run together in one or more places, and returns a value.

function

The function block takes the form:

```
function func_name [ (param_list) ]: type
  func_body
end-function
```

where *func_name* is the function's name, *param_list* its list of formal parameters, separated by commas and *type* is the basic type of the return value. When the procedure is called, statements in *func_body* are executed and a value returned.

Listing 6.24 provides an example of using three functions to find *perfect numbers*. A perfect number is one for which the sum of its divisors is equal to itself. The first function, `isPrime`, checks the primality or otherwise of a number, whilst the second calculates positive integral powers of a number. Finally, the third calls these to generate *Mersenne Primes*, from which it calculates perfect numbers using Euclid's formula.

Notable in this example is the keyword `returned` which must be present in any function description. Its value is sent back to the caller when the function exits, so it should be set to a value of basic type *type*, as declared at the top of the function. In our example, integer values are assigned to `returned` either by each possibility from an `if` statement, or from the power calculation.

Exercise *Enter the example of Listing 6.24 and run it to calculate the first four or five perfect numbers. Adapt the code to make the functions more general, rejecting invalid arguments.*

Listing 6.24 Calculating perfect numbers with functions

Forward declaration of functions is also permitted. See the following sections for details.

```
model PerfectEx
  function isPrime(number: integer): boolean
    i := 1
    repeat i+= 1
    until ((number mod i = 0) or i > sqrt(number))
    if(i > sqrt(number)) then returned := true
    else returned := false
    end-if
  end-function

  function power(a,b: integer): integer
    pow := 1
    while(b > 0) do
      pow := pow*a
      b-=1
    end-do
    returned := pow
  end-function

  function Perfect(n: integer): integer
    Mn := power(2,n)-1
    if(isPrime(Mn)) then
      returned := power(2,n-1)*Mn
    else returned := 0
    end-if
  end-function

  i := 1; k := 1
  while(k<5) do
    i+=1
    if(Perfect(i) > 1) then
      write(Perfect(i)," = ")
      forall(j in 0..i-1) write(power(2,j),"+")
      write(power(2,i)-1)
      forall(j in 1..i-2)
        write("+",(power(2,i)-1)*power(2,j))
      writeln; k+=1
    end-if
  end-do
end-model
```

Recursion

Both functions and procedures may be used recursively, either calling themselves directly, or indirectly. This can be particularly useful, for which reason we provide an example. In Listing 6.25 the function `hcf` is called recursively to determine the highest common factor of two numbers.

Listing 6.25 Recursion of functions

```
model HcfEx
  function hcf(a,b: integer): integer
    if(a=b) then
      returned:=a
    elif(a>b) then
      returned:=hcf(b,a-b)
    else
      returned:=hcf(a,b-a)
    end-if
  end-function

  declarations
    A,B: integer
  end-declarations

  write("Enter two integer numbers:\n A: ")
  readln(A)
  write(" B: ")
  readln(B)

  writeln("Highest common factor: ",hcf(A,B))
end-model
```

Exercise *Type in the example of Listing 6.25 and experiment with it on a few examples.*

Forward Declaration

We have already hinted that Mosel allows not just for recursion, but also *cross recursion*, where two subroutines alternately call each other. However, since all functions and procedures must be declared before they can be called, the subroutine defined first will of necessity call another which has not been defined. The solution is to use the `forward` keyword to *declare* one or both of the subroutines before their commands are *defined*.

"Declaration v. definition..."

An important distinction should be made at this point between the various vocabulary used. The *declaration* of a subroutine states its name, the parameters (type and name) and, in the case of a function, the type of the return value. The *definition* of the subroutine that will follow later in the model program contains the body of the subroutine, that is, the commands that will be executed when the subroutine is called.

This difference is perhaps best illustrated by way of an example. In Listing 6.26 we implement a quick sort algorithm for sorting a randomly-generated array of numbers into ascending order. The procedure `start` that starts the sorting algorithm is defined at the very end of the program, so it needs to be declared at the beginning before it is called.

Listing 6.26 Forward declaration of subroutines

```
model "Quick Sort"
  parameters
    LIM=50
  end-parameters

  forward procedure start(L:array(range) of integer)

  declarations
    T: array(1..LIM) of integer
  end-declarations

  forall(i in 1..LIM) T(i):=round(.5+random*LIM)
  writeln(T)
  start(T)
  writeln(T)
```

Listing 6.26 Forward declaration of subroutines

```
  ! swap the positions of two numbers in an array
  procedure swap(L:array(range) of integer,
      i,j:integer)
    k:=L(i)
    L(i):=L(j)
    L(j):=k
  end-procedure

  ! main sorting routine
  procedure qsort(L:array(range) of integer,
      s,e:integer)
  ! Determine partitioning value
    V:=L((s+e) div 2)
    i:=s; j:=e
    repeat      ! Partition into two subarrays
      while(L(i)<V) i+=1
      while(L(j)>V) j-=1
      if(i<j) then
        swap(L,i,j)
        i+=1; j-=1
      end-if
    until i>=j

                ! Recursively sort the two subarrays:
    if(j<e and s<j) then
      qsort(L,s,j)
    end-if

    if(i>s and i<e) then
      qsort(L,i,e)
    end-if
  end-procedure

  ! start of the sorting process
  procedure start(L:array(r:range) of integer)
    qsort(L,getfirst(r),getlast(r))
  end-procedure

end-model
```

The idea of the quick sort algorithm is to partition the array that is to be sorted into two parts. One of these contains all values smaller than the partitioning value and the other all the values that are larger than this value. The partitioning is then applied to the two subarrays recursively until all the values are sorted.

Exercise *Type in the model of Listing 6.26, or adapt your previous examples to place all subroutine definitions at the end of the program.*

Use of the `forward` keyword is particularly useful in providing structure to your model files and making them easier to follow. By placing all detail of a model into subroutines and providing the descriptions of these at the end, the main flow of the program is evident.

Having reached this point, you have encountered enough features of the Mosel model programming language to enable you to create your own, more sophisticated models and solve them using the interface of your choice. As you develop these, you may need to use further features of the Mosel language, all of which may be found in the Mosel Reference Manual. The following (and final) chapter of this introduction provides a glossary of some of the terms that have been used.

Summary

In this chapter we have learnt how to:

- initialize fixed-size and dynamic array in several dimensions;
- import and export model data using external sources and applications;
- create conditional expressions and variables;
- create selection and looping structures;
- write functions and procedures.

Chapter 7
Glossary of Terms

binary variable

A decision variable that may take the values 0 or 1 in a solution. Problems that contain binary variables are mixed integer programming problems.

bound

A simple constraint, used to set simple upper or lower bounds on a decision variable, or to fix a decision variable to a given value.

constraint

A linear inequality (or equation) that must be satisfied by the value of the decision variables at the optimal solution.

constraint type

There are five types of constraint: $\geq$ (greater than or equal to), $\leq$ (less than or equal to), $=$ (equal to), ranged and unconstraining, e.g. an objective function.

continuous variable

A decision variable that may take any value between 0 and infinity (or between some lower and upper bound in general). Also called a linear variable.

control

A parameter in the Optimizer whose value influences the solution process.

decision variable

An object (an 'unknown') whose value is to be determined by optimization. In linear programming (LP) problems, all decision variables are linear (or continuous), that is, they may take any value between 0 and infinity (or between a lower and upper bound in general). In mixed integer programming (MIP) problems, some variables may be binary, integer etc., or form a special ordered set. Sometimes referred to simply as 'variables'.

dj

See reduced cost.

dual value

The change in the objective value per unit change in the right hand side of a constraint. Sometimes referred to as the shadow price — the 'price' of the resource being rationed by the constraint. The dual value is a measure of how tightly a constraint is acting. If we are hard up against a $\leq$ constraint, then a better objective value might be expected if the constraint is relaxed a little. The dual value gives a numerical measure to this. Generally, if the right hand side of a $\leq$ row is increased by 1, then the objective value will increase by the dual value of the row. More specifically, since the dual value is a marginal concept, if the right hand side increases by a sufficiently small δ then the objective value will increase by $\delta \times$ dual value.

global entity

Any integer, binary, partial integer or semi-continuous variable or a special ordered set.

input cost

The original objective coefficient of a variable.

integer programming (IP) problem

An alternative name for a mixed integer programming (MIP) problem, in particular, one which doesn't contain any linear variables.

integer variable

A decision variable that may take the values 0, 1, 2,… up to some small upper limit in a solution. Problems that contain integer variables are mixed integer programming problems.

keyword (in Mosel)

A keyword in an Xpress-MP model is either a model command or else introduces / terminates a block. It is an element of the Mosel model programming language, or from a loaded library module. Typical keywords are `declarations` and `uses`.

linear expression

A term (or sum of terms) of the form *constant* × *decision variable*.

linear inequality or equation

A linear expression that must be 'greater than or equal to', 'less than or equal to', or 'equal to' a constant term (the right hand side).

linear programming (LP) problem

A decision problem made up of linear decision variables, an objective function and constraints. The objective function and constraints are linear functions of the decision variables. The data values (i.e. the coefficients of the variables and the constant terms in constraints) are all known values.

linear variable

A decision variable that may take any value between 0 and infinity (or between some lower and upper bound in general). Also called a continuous variable.

mixed integer programming (MIP) problem

A linear programming (LP) problem which contains some global entities, e.g. some of the variables may be binary variables, integer variables, etc. or form a special ordered set.

model

The set of decision variables, objective function, constraints and data that define the problem. Often used to mean the Xpress-MP representation of the problem.

model instance

A model structure complete with explicit data, i.e. values for the data and parameters.

model program

A model combined with imperative statements possibly related to solving it. These are run by Mosel and written in the Mosel model programming language.

model structure

The definition of the decision variables, data tables and constraints, and the algebraic relationship between them, without specifying any explicit data or parameters.

objective function

A linear or quadratic expression which is to be optimized (maximized or minimized) by choosing values for the decision variables.

optimal solution

A solution that achieves the best possible (maximum or minimum) value of the objective function.

partial integer variable

A decision variable that, in a solution, may take the values 0, 1, 2,… up to some small upper limit, and then any continuous value over that limit. Problems that contain partial integer variables are mixed integer programming problems.

problem attribute

A value set by the Optimizer during the solution process, which may be retrieved by library users. It is a particular property of the problem being solved or its solution.

reduced cost

The amount by which the objective coefficient of a decision variable would have to change for the variable to move from the bound it is at. Sometimes referred to as the 'dj'.

right hand side (RHS)

The constant term in a constraint. By convention the value is written on the right hand side of the constraint, although this is not necessary in Xpress-MP.

semi-continuous variable

A decision variable that, in a solution, may take the value 0 or any continuous value in a range from a lower limit to an upper limit. Problems that contain semi-continuous variables are mixed integer programming problems.

shadow price

See dual value.

slack value

The amount by which a constraint differs from its right hand side.

solution

A set of values for the decision variables that satisfy the constraints. See also optimal solution.

special ordered set (SOS)

An ordered set of variables that must fulfil a special condition. In a Special Ordered Set of type 1 (an 'SOS1' or 'S1 set') at most one variable may be nonzero. In a Special Ordered Set of type 2 (an 'SOS2'

or 'S2 set') at most two variables may be nonzero, and if two variables are nonzero, they must be next to each other in the ordering. In Xpress-MP the ordering is supplied in a 'reference row', which may be an ordinary constraint or a separate unconstraining constraint. There is no requirement that the variables must take integer values.

variable

See decision variable.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X