



AMPLDev User Guide

OptiRisk Systems

Version: 2.2.5

*Prepared by
Neha Murarka, Victor Zverovich, Christian Valente and Gautam Mitra
OptiRisk Systems*

Copyright © 2011 OptiRisk Systems

DO NOT DUPLICATE WITHOUT PERMISSION

All brand names, product names are trademarks or registered trademarks
of their respective holders.

The material presented in this manual is subject to change without prior notice and is
intended for general information only. The views of the authors expressed in this document
do not represent the views and/or opinions of OptiRisk Systems.



OptiRisk Systems
One Oxford Road
Uxbridge, Middlesex, UB9 4DA
United Kingdom

www.optirisk-systems.com

+44 (0) 1895 256484

Contents

I	Overview of AMPLDev	1
1	Scope and Purpose	2
1.1	What is AMPL?	2
1.2	Who can use AMPLDev?	2
1.3	Why use AMPLDev?	3
2	Installing AMPLDev	4
2.1	AMPLDev stand-alone application	4
2.2	AMPLDev plug-in for Eclipse users	4
	Java Runtime Environment installation	4
	Getting Eclipse	5
	Installing AMPLDev plug-in	5
3	AMPLDev User Interface	7
3.1	The Eclipse Platform	7
3.2	Workbench	8
3.3	Perspectives	9
3.4	Views	10
	Project Explorer	11
	Console	11
	Outline	12
	Solution	12
3.5	Editors	12
II	Modelling with AMPLDev	14
4	Projects within AMPLDev	15
4.1	Concept of a project	15
4.2	Creating a new project	15
	Using the File menu in a non-AMPL perspective	15
	Using the File menu in the AMPL perspective	16
	Using the AMPL perspective toolbar	17
	Using the Project Explorer context menu in the AMPL perspective	17
4.3	Creating and adding files to an AMPL project	17
5	Running Models and Analysing Results	20
5.1	Single file launch	20
	Context menu launch	21
	Launch toolbar	21
5.2	Multiple file launch	21
	The File Selection tab	23
5.3	Default launch	24

5.4	Viewing output and errors	24
5.5	Viewing the solution	25
III	AMPL: A Mathematical Programming Language	27
6	Learning AMPL by Example	28
6.1	A Linear Programming Model	28
6.2	Translating the model into AMPL	30
	Sets	32
	Parameters	32
	Variables	33
	Objective	33
	Constraints	33
7	Databases and Spreadsheets	35
7.1	Example: A Diet Problem	35
7.2	Preparing the Data	36
	Spreadsheets	37
	Databases	38
7.3	Reading Data from Tables	40
	Read Parameters only	42
	Read Sets and Parameters	42
	Establishing Correspondences	43
	Other values	44
7.4	Writing Data into Tables	44
	Rows inferred from the data specifications (<i>data-specs</i>)	45
	Rows inferred from a key specification (<i>key-spec</i>)	47
7.5	Reading and Writing into the <i>same</i> Table	48
	Using two <code>table</code> declarations	48
	Using the same <code>table</code> declaration	48
7.6	Indexed collections of Tables and Columns	49
	Indexed collections of Tables	49
	Indexed collections of Data Columns	51
	Acronyms	53
	References	54
	Index	55

Part I

Overview of AMPLDev

Chapter 1

Scope and Purpose

1.1 What is AMPL?

AMPL^[1] is an algebraic modelling language that is used for formulating and solving optimization problems. AMPL supports linear and non-linear programming optimization models in discrete and continuous variables. Stochastic extension of AMPL (SAMPL) has comprehensive support for stochastic programming; this is an extension of AMPL. You can easily create models and establish connection to various data sources, such as spreadsheets, databases or plain text files, within AMPL. From the model and data, the AMPL translator generates an internal representation, passes it to one of the many solvers supported by AMPL and provides you with the results. The language is simple and generally follows the mathematical notation used both by mathematicians and practitioners in the field of operations research.

AMPLDev is a graphical interface for AMPL. It is based on the popular [Eclipse](#) development platform and is available as a stand-alone application and as a plug-in for Eclipse.

For a general introduction to AMPL see ^[1]; a description SAMPL may be found in (reference).

AMPLDev as a stand-alone application

This version is a complete bundle of the necessary components required to run AMPLDev without the complications of installing any prerequisites. You can think of it as a core Eclipse IDE with the AMPLDev plug-in preinstalled. The only drawback is that this stand-alone version of AMPLDev may have limited capabilities of extending with other Eclipse plug-ins.

If you do not require Eclipse for any other reason, then we recommend installing this version of AMPLDev.

AMPLDev plug-in for Eclipse IDE

The Eclipse IDE combines a variety of extensible tools and frameworks through the use of plug-ins for software development. Due to its extensibility, the Eclipse IDE can be used for different purposes such as programming in Java or Python, creating mobile applications or even using it as a computer algebra system (with Wolfram Workbench).

The AMPLDev plug-in, seamlessly equips Eclipse with AMPL programming capabilities. Once the plug-in is installed, Eclipse can be used as an AMPL IDE with all the features of the stand-alone AMPLDev application and functionality provided by other Eclipse plug-ins.

1.2 Who can use AMPLDev?

AMPLDev can be used by anyone who is considering using an algebraic modelling language and would like a quick and easy way to start off the process of learning AMPL, and even for those who are already using AMPL and would like a modern IDE for this language. The common areas that use

such a tool are distribution, production, scheduling and other areas that have large-scale optimization problems.

To use AMPLDev, you are expected to have a basic knowledge of programming and some optimisation modelling experience. Experience with Eclipse or any other development environment, although not required, will considerably speed up the learning process.

1.3 Why use AMPLDev?

AMPLDev gives AMPL users the benefits of a modern Integrated Development Environment (IDE) which has the following features.

- A smart editor with context-sensitive syntax highlighting.
- Efficient error reporting with the ability to instantly go to the error location.
- A solution view which organises and separates the results from solving the model.
- A project explorer that allows you to organise all your projects and corresponding folders with useful context menus that directly allow you to run AMPL files.
- Built-in interactive AMPL console.
- Outline view that shows the model components: parameters, sets, variables, objectives and constraints.

For users who are new to AMPL such as students who are learning modelling using AMPL for the first time, the carefully designed Graphical User Interface (GUI) helps to easily get started. At the same time AMPLDev is beneficial for advanced users who need such features as built-in interactive console, integration with version control systems provided by Eclipse and projects supporting other programming languages in addition to AMPL.

For the users who wish to install the plug-in version of AMPLDev, Eclipse is absolutely free to use and has minimal installation requirements. Integrating AMPL with Eclipse is a huge advantage if you are already an Eclipse user and can have one software that takes care of all your development needs. In addition, you can even extend the AMPLDev plug-in if you wish to customize it further for your needs.

Chapter 2

Installing AMPLDev

This chapter describes how to set up your computer for using AMPLDev. The first section covers installing the AMPLDev stand-alone application which is the essential step of getting started with AMPLDev. The second section describes how to install the AMPLDev plug-in in Eclipse; primarily for advanced users.

2.1 AMPLDev stand-alone application

This is the fastest way to start using AMPLDev. Download the archive from the website onto your local machine, unless you have already received the archive via other means.

Once you extract all the files from the archive, in the root folder you will find the AMPLDev executable, double-click it and you are good to go!

The stand-alone application does not require any [Java Runtime Environment \(JRE\)](#) or Eclipse installations as these items of software are all bundled in with the executable.

2.2 AMPLDev plug-in for Eclipse users

If you already have Eclipse set up on your machine (a version from 3.3 onwards), then you may skip the next two sections ([2.2.1](#) and [2.2.2](#)).

First, install the [JRE](#) on your machine (see [2.2.1](#)). Following that, [2.2.2](#) describes how to get Eclipse (Eclipse Indigo 3.7 is used throughout this book) and install it. The last subsection explains how to install the AMPLDev plug-in into your Eclipse Platform so that you can program in AMPL using Eclipse.

2.2.1 Java Runtime Environment installation

The [JRE](#) needs to be installed to run Eclipse. You can download the latest version of the [JRE](#) by following the link, <http://www.java.com/en/download/manual.jsp>. It is recommended to use the JRE of version 5 or higher for the Eclipse version (Eclipse Indigo 3.7); our system which is based on this version, is described in this manual.

TIP:

Many computers already have the [JRE](#) installed but if you are not sure, the Java website has a nice tool that checks your computer for you and asks you to download the latest version if you do not have any or if you need an upgrade. You can either search for 'Do I have Java?' or use the following link, <http://www.java.com/en/download/installed.jsp?detect=jre&try=1>.

2.2.2 Getting Eclipse

Once you have the [JRE](#) set up, your computer is now ready for Eclipse. The version used throughout the book is Eclipse Indigo 3.7 and can be freely downloaded from the Eclipse website (<http://www.eclipse.org/downloads/>). For users, who wish to use a previous version, it is recommended to use only versions from Eclipse 3.3 onwards.

On the download page, there are many different types of Eclipse Indigo packages available; we recommend that you download the Classic version. Most of the others focus on a specific development area such as Java or PHP but the Classic version is an all round [Integrated Development Environment \(IDE\)](#) with all the relevant tools needed for our purpose (except at this stage we have not yet installed the AMPLDev plug-in, see section [2.2.3](#)). Eclipse is available for major operating systems (Windows, Linux and Mac OS) and supports 32-bit and 64-bit platforms.

TIP:

Some 64-bit machines may have a 32-bit JRE installed; that is fine as long as the Eclipse is also 32-bit. Hence when downloading Eclipse, it does not depend upon your machine but upon the JRE installed. For example, a 64-bit Eclipse will not work on a 64-bit machine with a 32-bit JRE; either get a 32-bit Eclipse or a 64-bit JRE.

You can choose from numerous mirror sites, available across Europe, Asia, North America, South America and Australia, depending upon your location for better download speeds.

Once the file is downloaded, unzip it at your desired location. As such, there is no setup file that needs to be run for the installation. Then run the eclipse executable (eclipse.exe for Windows' users) which is located inside the extracted 'eclipse' folder. But don't forget to install the AMPLDev plug-in if you would like to use AMPL!

2.2.3 Installing AMPLDev plug-in

There are two ways to install the AMPLDev plug-in; these two ways are described below.

Installing AMPLDev plug-in via update site - recommended

1. Once you have downloaded the AMPL plug-in archive on your computer, start Eclipse and click on **Install New Software...** in the **Help** menu.
2. Click on the **Add...** button.
3. If you have extracted the plug-in file, click on **Local...** and navigate to the AMPL plug-in folder or do the same with **Archive...**, if you have not extracted the AMPL plug-in.
4. Click on OK and the **Install New Software...** window will list the available AMPL plug-in.
5. Select the plug-in and click Next.
6. Follow the default settings and the AMPL plug-in will be installed.
7. Restart Eclipse and you should have AMPL support!

TIP:

If you cannot see the AMPL plug-in listed as an option to install, uncheck the **Group items by category**.

Installing AMPLDev plug-in manually

1. Extract all the files from the AMPL plug-in archive to any location. The location does not matter as you will soon be copying the required folders into Eclipse.
2. Once the archive has been completely extracted, you will find a 'plugins' folder inside it. Copy the contents.
3. Navigate to the folder where you have installed your Eclipse. Inside the 'eclipse' folder, you will see a folder named 'dropins'. Paste the previously copied 'plugins' folder into the 'dropins' folder.
4. Repeat the previous two steps for the 'features' folder in the extracted plug-in archive by copying the contents into Eclipse's dropins' 'features' folder.
5. Simply start Eclipse or restart it (if it was already running) and you should have AMPL support!

Chapter 3

AMPLDev User Interface

In this chapter the [Graphical User Interface](#) for AMPLDev is described. Since the AMPLDev stand-alone system has the same UI as Eclipse with AMPLDev plug-in installed, the following sections are relevant for both types of users: stand-alone and plug-in version. Information regarding both versions are indicated by 'AMPLDev/Eclipse' and the information is generally referencing the type of software installed (stand-alone or plug-in). 'AMPLDev' alone describes information related to AMPLDev, independent of what installation you have.

3.1 The Eclipse Platform

Just as multi-functional organisms are made up of a million cells, the Eclipse Platform is put together with numerous plug-ins. A group of one or more of these plug-ins contribute to the multitude of functions and capabilities that are part of Eclipse. This concept of little sub-systems put together for a larger system also allows for complete extensibility as one can keep adding plug-ins to either

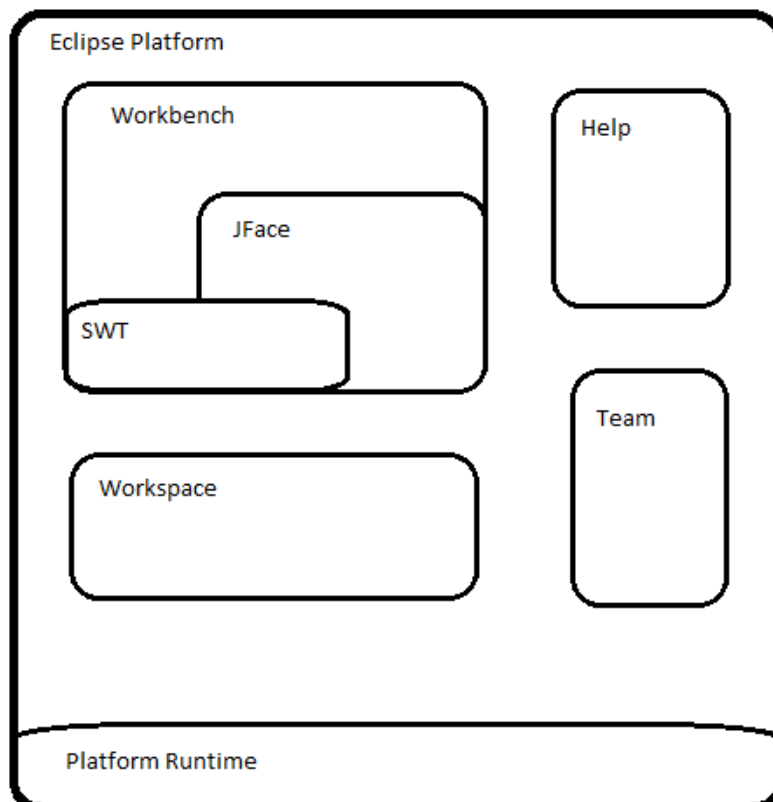


Figure 3.1: Eclipse overview

define their own functions or manipulate the current ones in Eclipse.

The collection of plug-ins is built upon a runtime engine, generally the Java Runtime Environment. Figure 3.1 gives a simple idea of how the architecture and components come together under the Eclipse IDE.

Due to this plug-in extensibility property, we have been able to create a collection of plug-ins and also extend the already available plug-ins, to give Eclipse support for programming in AMPL.

In the rest of this chapter, we describe the various functionalities of the Eclipse IDE and also the additional views and graphical tools added for AMPL. If you already have been using Eclipse, most of the sections can be skipped or at the most skimmed through; the sections relevant to AMPL are clearly stated and you need only review those.

3.2 Workbench

Once the computer has been set up for AMPLDev (see chapter 2), run the eclipse executable (eclipse.exe for Windows' users) to start an Eclipse session or on the ampldev executable for the stand-alone version. When AMPLDev/Eclipse starts, it will ask for a workspace; this is essentially a directory on the computer in which it will save the projects (see figure 3.2).

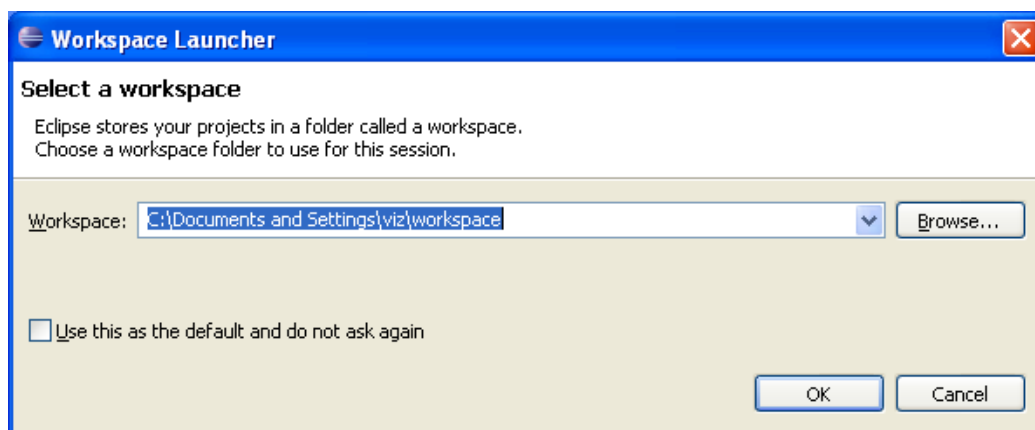


Figure 3.2: Workspace selection

TIP:

You can change the workspace any time by choosing **Switch workspace...** from the **File** menu.

When Eclipse is started for the first time, after choosing the workspace, you are taken to a Welcome screen (see figure 3.3). We will not go into the details of the Welcome screen but you are free to browse through. The AMPLDev stand-alone does not have a Welcome screen, it directly goes to the workbench.

To start using Eclipse, click the 'Workbench' link at the top right hand corner. This starts the Eclipse workbench, basically the development environment (see figure 3.4) to work in.

The workbench is where you can manage the whole project life-cycle, from creation to the final product. A workbench window has menus and toolbars along with perspectives, where a perspective is a customized collection of views and editors based on the purpose of the perspective. Simply put, it is the AMPLDev/Eclipse window that is seen on the desktop!

AMPLDev/Eclipse allows users to open multiple sessions of the workbench. Therefore while a window/workbench is open, by running the eclipse executable file again, another window will open that can be used with a *different* workspace.

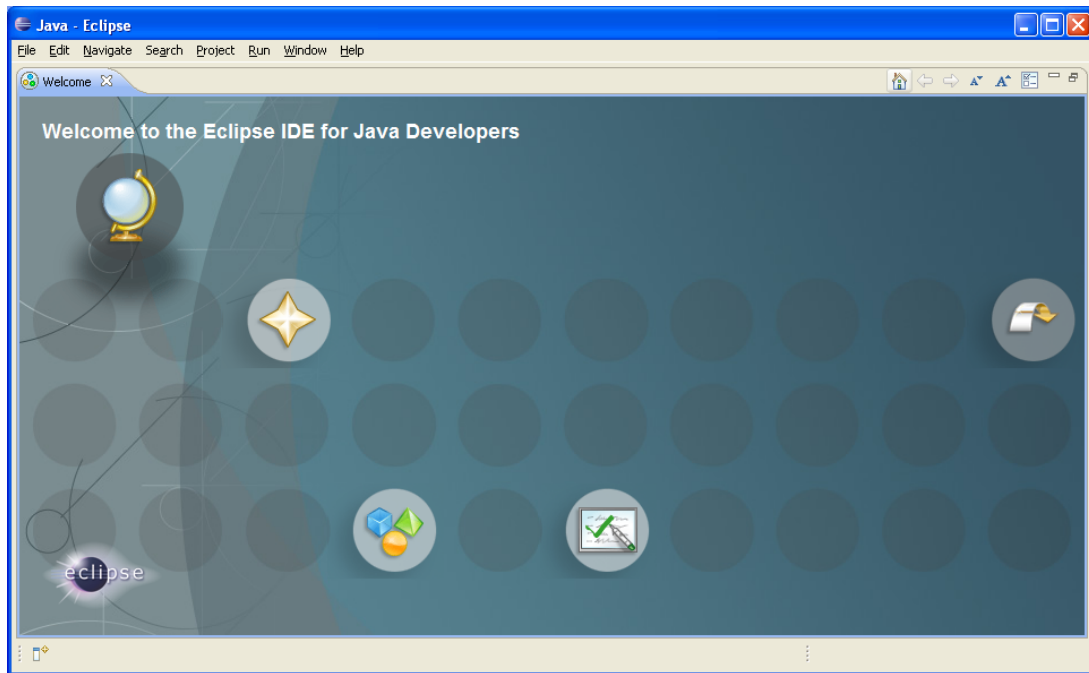


Figure 3.3: Eclipse Welcome screen

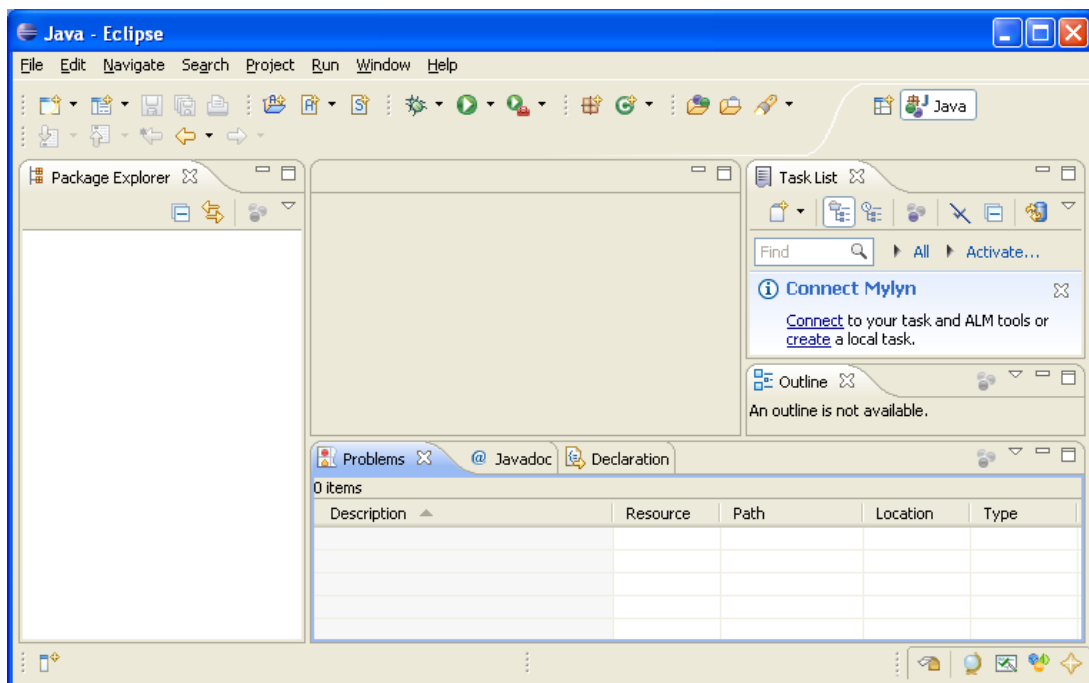


Figure 3.4: Eclipse workbench

3.3 Perspectives

A perspective is a customized layout of views that is set in an efficient and useful way for the purpose it was created. For example, figure 3.4 shows Eclipse with a Java perspective open; it has views such as **Javadoc** and **Package Explorer** which aids a Java based project. Similarly, we have the AMPL perspective which caters to AMPL projects.

Generally, all the views are set around an editor area. You can also open other views that are not part of the current perspective which then gets saved when AMPLDev/Eclipse has exited. Hence, the next time you start AMPLDev/Eclipse and open this perspective, it will look exactly how it was

in the previous session. This allows for customizations of perspectives.

AMPL perspective

There are two ways to get the AMPL perspective. The AMPLDev stand-alone generally starts with the AMPL perspective already but for some systems settings, it might not. Switching to the AMPL perspective can be achieved in one of the following ways:

- Create an AMPL project; Eclipse will ask you to change to the AMPL perspective upon the creation of the project.
- In **Window** menu, select **Open Perspective...**, choose **Other...** and then click on AMPL.

Although it is not necessary to use the AMPL perspective for using AMPL, the layout is what best suits for AMPL modelling (see figure 3.5). AMPLDev/Eclipse will also remember any changes you make, so when you use the AMPL perspective, it will always look like how you set it.

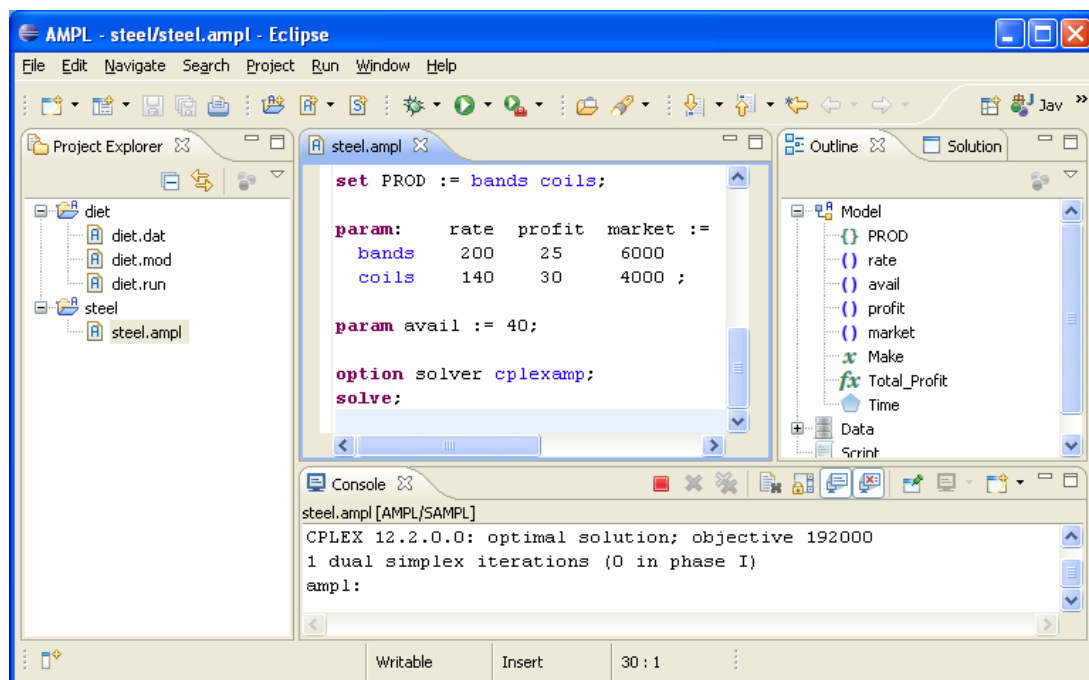


Figure 3.5: AMPL perspective

There are also many AMPL-specific shortcuts that are available easily when in this perspective. They will be discussed in later chapters but one example is the toolbar which has buttons to create AMPL projects and files without going through the **File** menu.

3.4 Views

Views provide for smart ways to represent any kind of information; such as the Project Explorer which shows an easy-to-understand tree structure of all the parts of a project: files, folders etc, collectively known as resources. Views are also handy to navigate through large systems of information.

A perspective contains a set of views that open automatically when the perspective is selected. Although, you can also open views that are not part of the current perspective via the **Window** menu item and then selecting **Show View...** (see figure 3.6). As described previously, these additional views opened in a perspective are saved under the perspective.

Views can be moved around at any time by clicking and holding the left mouse button on its title bar; they can either be left detached or docked around the Workbench window. They can also

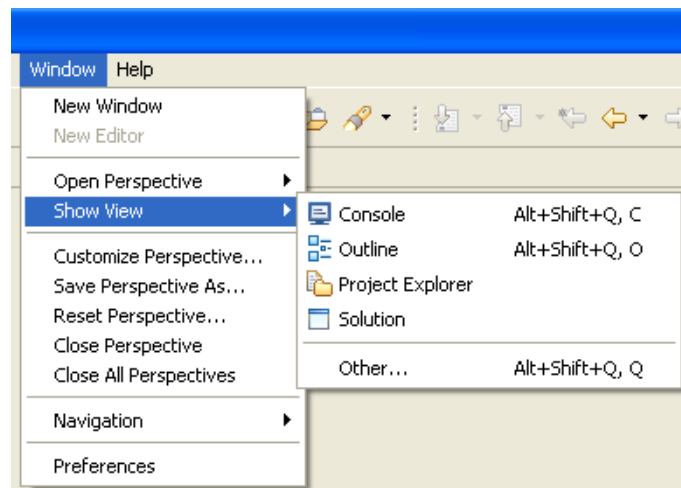


Figure 3.6: Opening an additional view which is not part of the perspective

be maximised by double-clicking on the title bar and doing it a second time will bring it back to its original size and position. To manage the screen space better, views can be stacked on top of each other and viewed by clicking on the tabs above the stacked views. The active view will have a highlighted tab.

Most views also have menus of their own which can be accessed by clicking the drop down arrow in the top right corner of the view. Sometimes they may even have their own toolbars.

The following sections describes the views that form the AMPL perspective.

3.4.1 Project Explorer

As explained in section 3.4, the Project Explorer view (figure 3.7) displays the projects in the workspace in a tree-structure. All the functions are the same as described in the Eclipse manual for the Project Explorer view with the addition of some AMPL specific functions, which are active in the AMPL perspective.

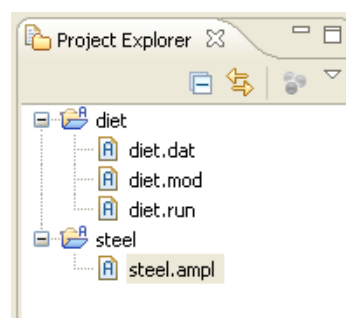


Figure 3.7: The Project Explorer view in the AMPL perspective

The context menu (pop-up menu on right-clicking, figure 3.8) allows you to add AMPL projects and files into the workspace, without going through the File menu wizards. The **Run As...** and the **Debug As...** also have an AMPL quick launch option, if the selection is an AMPL file. On clicking this quick launch, only the selected file is run by AMPL. In order to run multiple files, you must create a launch configuration (see section 5.2)

3.4.2 Console

This view displays all the output and runtime errors from AMPL and SAMPL and allows user to enter commands and input data. For more details, see section 5.4.

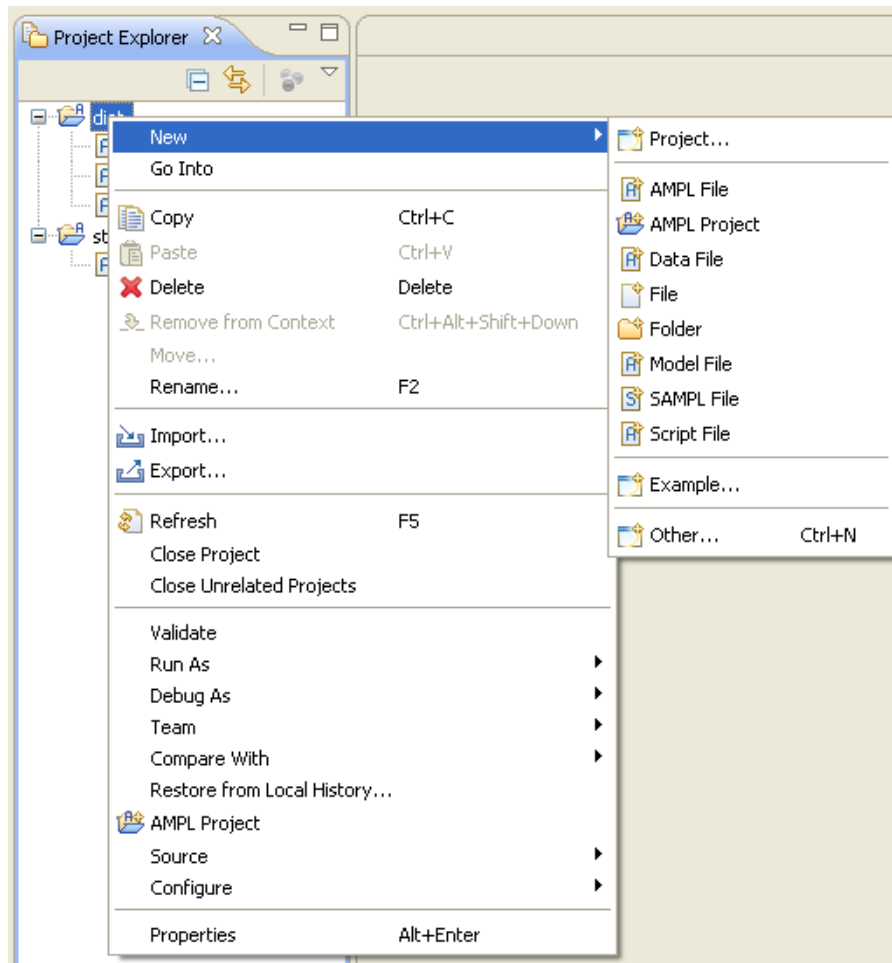


Figure 3.8: Context menu in Project Explorer

3.4.3 Outline

This view shows the list of model components, such as parameters, sets, variables, objectives and constraints, for the AMPL file currently open in the editor.

3.4.4 Solution

This view displays the solution report for the last run. For more details, see section 5.5.

3.5 Editors

The model and data source files are opened, modified and saved using the editor. Different types of files have their own editors associated with them. When you double-click on any file in the views, it will automatically open it in the editor area with its respective editor. If there are multiple editors open at one time, then they are stacked in the editor area but can be separated in the same manner as how views are moved around except that they cannot be detached completely. When a file's name tab has an asterisk (*), it means there are still unsaved changes in that file.

Some perspectives may not require an editor area, but for the ones that do, it is generally in the centre of the workbench and the views are set around it.

TIP:

Eclipse has extensive documentation on the Eclipse Workbench which can be found online on their website at <http://help.eclipse.org/indigo/index.jsp> (Eclipse Indigo 3.7) and can also be accessed via the software's **Help** menu.

Part II

Modelling with AMPLDev

Chapter 4

Projects within AMPLDev

The following sections describe the different ways you can create a project, add files to it and use some of the available templates.

If you are starting Eclipse for the first time, follow the steps described in section 3.2 to start the Eclipse workbench.

4.1 Concept of a project

A project is a collection of source code and any supporting files such as documentation and data that may be arranged using folders in a way similar to organizing files in the filesystem. Projects are contained within workspaces described in section 3.2.

Some projects are associated with one primary language and occasionally some supporting languages; it is not possible to add unsupported language files in such a project. This does not apply to the workspace which can have a mix of various types of projects under it.

You can organise files and folders in any manner as you please under the project root.

4.2 Creating a new project

There are numerous ways to create a project. In general, you do not need to be in the AMPL perspective to create an AMPL project.

4.2.1 Using the File menu in a non-AMPL perspective

This is the most universal method of creating a new project.

1. Go to the **File** menu.
2. Select **New**.
3. Select **Project...**
4. A **New Project** dialog will open which will ask you to select a project wizard. Choose AMPL Project under the AMPL category (see figure 4.1) and then click **Next**.
5. Give the project a name. You can either choose the default workspace location, or browse for another location. The **Create model, data and script directories** option as its name suggest creates the three respective folders in your project, if you check it (figure 4.2). You can always create the folders later (using the context menu of the project, see figure 3.8). These folders are independent of the files you put in them and is only for your personal organization. For example, you can put a .mod file in the data folder. They are not restrictive by type (see figure 3.7).

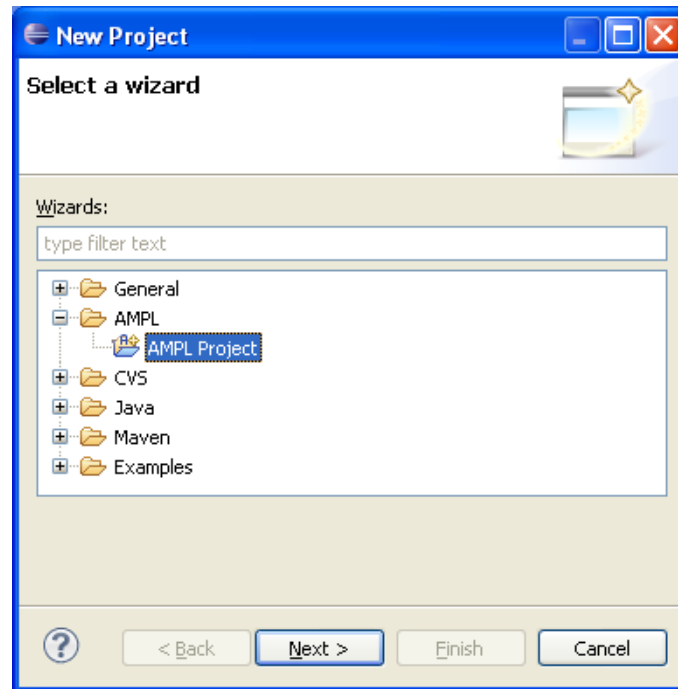


Figure 4.1: Eclipse New Project dialog

6. After clicking **Finish**, Eclipse will ask you if you want to change to the AMPL perspective since we are not in that perspective right now (figure 4.3). Ideally you would but it is a personal choice.
7. The project has been created and you will be able to see it in Project Explorer.

4.2.2 Using the File menu in the AMPL perspective

To create a project in the AMPL perspective is fairly straightforward.

1. Go to **File** menu.
2. Select **New**.

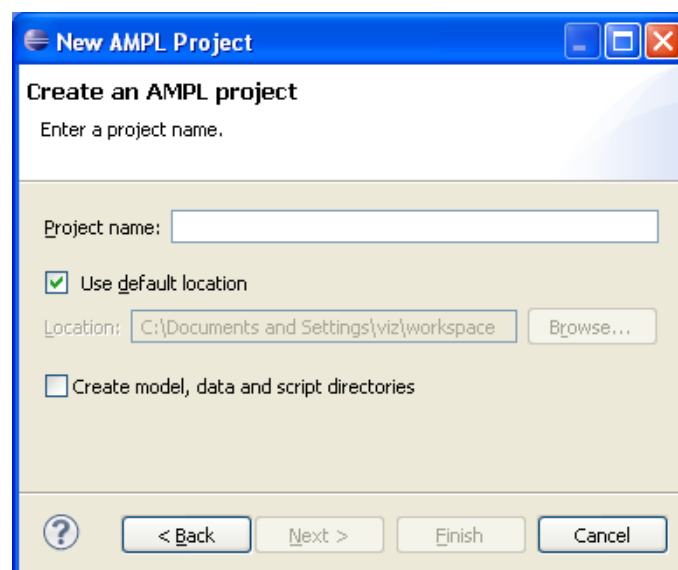


Figure 4.2: AMPL Project wizard

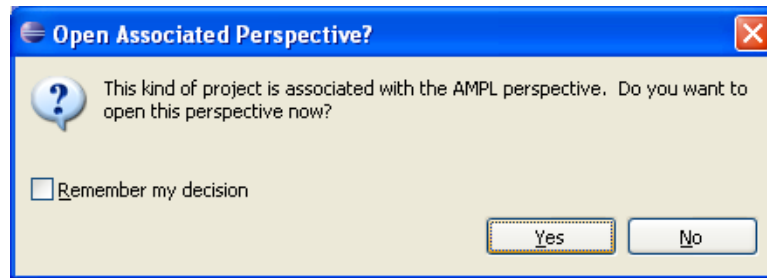


Figure 4.3: Change to AMPL perspective upon creation of AMPL project

3. Choose **AMPL Project** and follow the steps as described in the previous section (section 4.2.1) to create the project.

4.2.3 Using the AMPL perspective toolbar

In the AMPL perspective, there is a toolbar button (📁) to create an AMPL project (see figure 4.4).

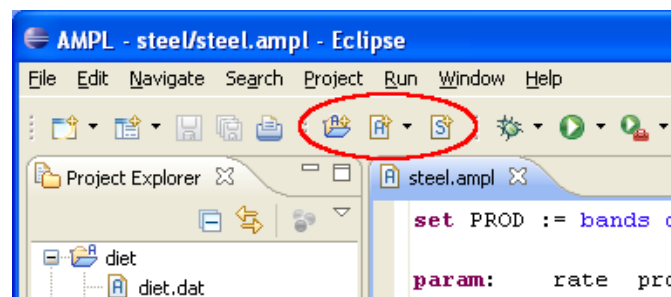


Figure 4.4: AMPL toolbar

After that the same project wizard opens and you can follow the steps as described in section 4.2.1.

TIP:

There is the New button (📁) in the toolbar which is present in all perspectives by default. Clicking this button opens a dialog to create a new file or a project.

4.2.4 Using the Project Explorer context menu in the AMPL perspective

The context menu is the menu that pops up with the right click (or command + click for a Mac). In the AMPL perspective when right-clicking in the Project Explorer view, there are shortcuts to create a project.

4.3 Creating and adding files to an AMPL project

Before describing the steps to create and add files, here is a brief description on the types of files supported by the AMPL plug-in for Eclipse.

Supported AMPL file extensions

The three traditional AMPL file extensions are .mod for model files, .dat for data files and .run for script files. They are all supported by AMPLDev.

In addition we have introduced a new file extension for AMPL files; you can now create an .ampl file which can also be used for AMPL code. Similarly, we have .sampl for SAMPL code. The .ampl

files encompass all aspects of the .mod, .dat and .run file types; for example, you can have Test-Model.ampl, TestData.ampl and TestScript.ampl which can have the same code as TestModel.mod, TestData.dat and TestScript.run. This will not make a difference while executing the files because the AMPL translator treats all the files in the same way regardless of their extension.

We recommend using the .AMPL extension for new files because the .mod and .dat extensions are often associated with other programs. For example, the .mod extension is often recognized as an extension for audio files.

Using the File menu in a non-AMPL perspective

Most of the steps are the same as creating a project except this time we choose to create a new file. This is the most universal method of creating a new file in Eclipse (and thus in AMPLDev).

1. Go to the **File** menu.
2. Select **New**.
3. Select **Other...** (alternatively you could select **File** but in this case you would have to provide the file extension when entering the name of the file).
4. A wizard dialog will open, where you can choose the file you would like to add under the AMPL category (see figure 4.5) and then click **Next**. (You may have noticed that there is the option to add the AMPL project too. Eclipse is full of shortcuts and you will discover many of them along the way.)

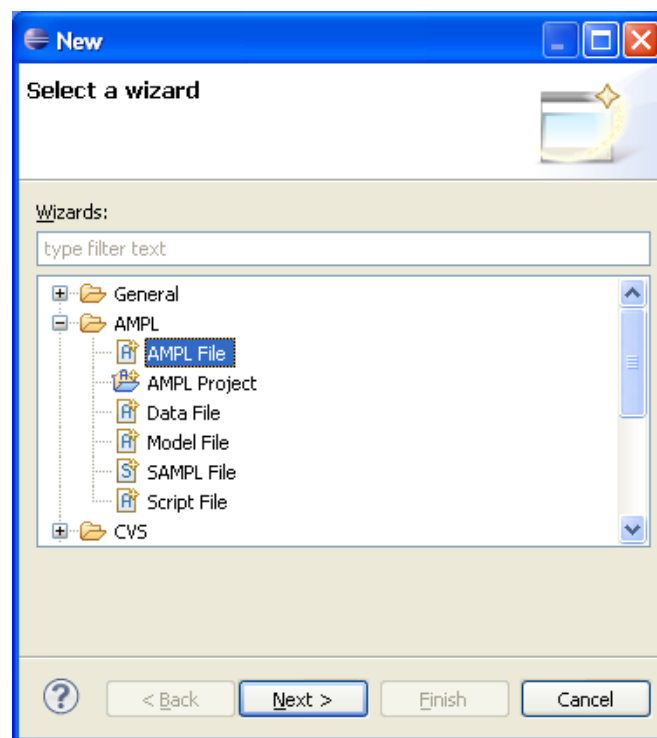


Figure 4.5: The New dialog

5. The first field requires you to choose the project or a folder in the project in which the file must be placed. Then choose a filename. As you can see from figure 4.6, the title of the wizard is an **AMPL File**, so you do not need to enter the extension; this wizard will automatically create a .AMPL file.
6. Once you have filled in the required information, Eclipse will create a new file in the chosen project or project folder and open it in the editor.

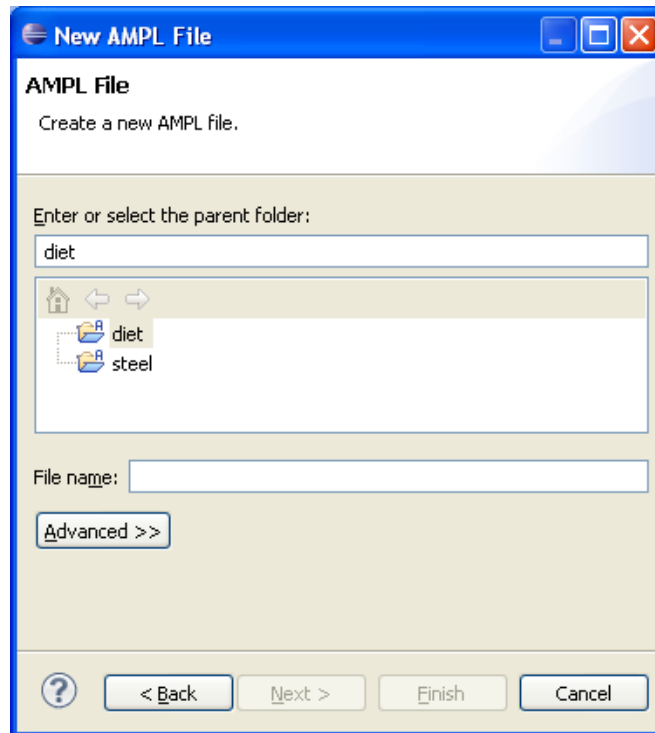


Figure 4.6: New AMPL File wizard

Using other means

To create and add a file via the **File** menu and the Project Explorer context menu in the AMPL perspective, and the AMPL perspective toolbar is exactly the same as the project creation described in section 4.2. The new file shortcuts are in the same locations and you can follow the same steps for the file wizard as described in section 4.3.

Chapter 5

Running Models and Analysing Results

Running a model written in AMPL requires the use of AMPL compiler/interpreter which translates the model and combines the model and data into a machine-readable and solver-specific form, passes it to one of the many solvers supported by AMPL. It then gets the results and executes any script commands. In Eclipse, these runs are referred to as *launches*. It is possible to run a single file or multiple files in the order you want. There is an additional default launch available for projects that contain only one model and one data file.

Note: At this stage, debug functionalities are not implemented, so the **Run** and **Debug** buttons have the same effect for AMPL projects.

5.1 Single file launch

The previous chapter tells you how to create a project and add files to it. When you do a single run, there is no relationship between the consecutive launches. So for example, you cannot run a model file first and then a corresponding data file, each launch will be independent and Eclipse will not be able to associate the model with the data. (For doing that, refer to section 5.2).

The single file launch allows you to quickly run a file without creating a custom launch configuration. A simple AMPL model that will work in a single file launch, is given below:

```
set PROD; # products

param rate {PROD} > 0; # tons produced per hour
param avail >= 0; # hours available in week

param profit {PROD}; # profit per ton
param market {PROD} >= 0; # limit on tons sold in week

var Make {p in PROD} >= 0, <= market[p]; # tons produced

# Objective: total profits from all products
maximize Total_Profit: sum {p in PROD} profit[p] * Make[p];

# Constraint: total of hours used by all
# products may not exceed hours available
subject to Time: sum {p in PROD} (1/rate[p]) * Make[p] <= avail
    ↔;

data;

set PROD := bands coils;
```

```

param:      rate  profit  market :=
  bands    200    25      6000
  coils    140    30      4000 ;

param avail := 40;

option solver cplexamp;
solve;

```

You may want to change the solver based on the ones installed on your machine, changing the statement 'option solver cplexamp;' accordingly.

There are two ways to run this code: context menu or launch toolbar. Both of them can be used in any perspective.

5.1.1 Context menu launch

This method is fairly straightforward.

1. In the Project Explorer, select the file you wish to execute. For this example, we have placed the code in the file steel.ampl.
2. Right click (or command + click for Mac) to open the context menu.
3. Select **Run As** or **Debug As** and you will see another menu pop-up (figure 5.1).
4. This pop-up will have an option saying **1 AMPL**. By selecting this, AMPLDev will automatically create a launch configuration for this file and the file will be executed. The launch configuration will have the same name as the file.

Note: in the Project Explorer, if you right click on a project folder, the **Run As** or **Debug As** will show you an AMPL option, but this is only valid for a default launch, refer to section 5.3.

5.1.2 Launch toolbar

Once you have created a default configuration like in the previous defined steps, you can see these launches in the dropdown menus accessible with the **Run** (▶) and **Debug** (⚙️) toolbar buttons (figure 5.2).

When you click the down arrow next to these buttons, you will see the previous launched configuration for steel.ampl. You will also see the other method of launching the file which is the **Run As** or **Debug As** option, see figure 5.3.

This shortcut creates a launch configuration for the current selection which means if there is a file selected in the editor or in the Project Explorer, it will launch that. In the case of steel.ampl, since there already is a launch configuration it will not create a new one. You can even choose **1 steel.ampl** if you would like to launch that file again.

As mentioned before, the launches are independent of the file extension; so whether you run an .ampl file, a .mod file or a .run file, AMPLDev will treat them the same way. The only exception is for the files with the .dat extensions which will be run in the data mode (as if the files contain the data statement at the beginning).

5.2 Multiple file launch

Having a single file execution is not always desirable; you may like to separate your model from the data or have multiple models for some data and it is not ideal to put all the code in one file. For this reason, there is a multiple file launch.

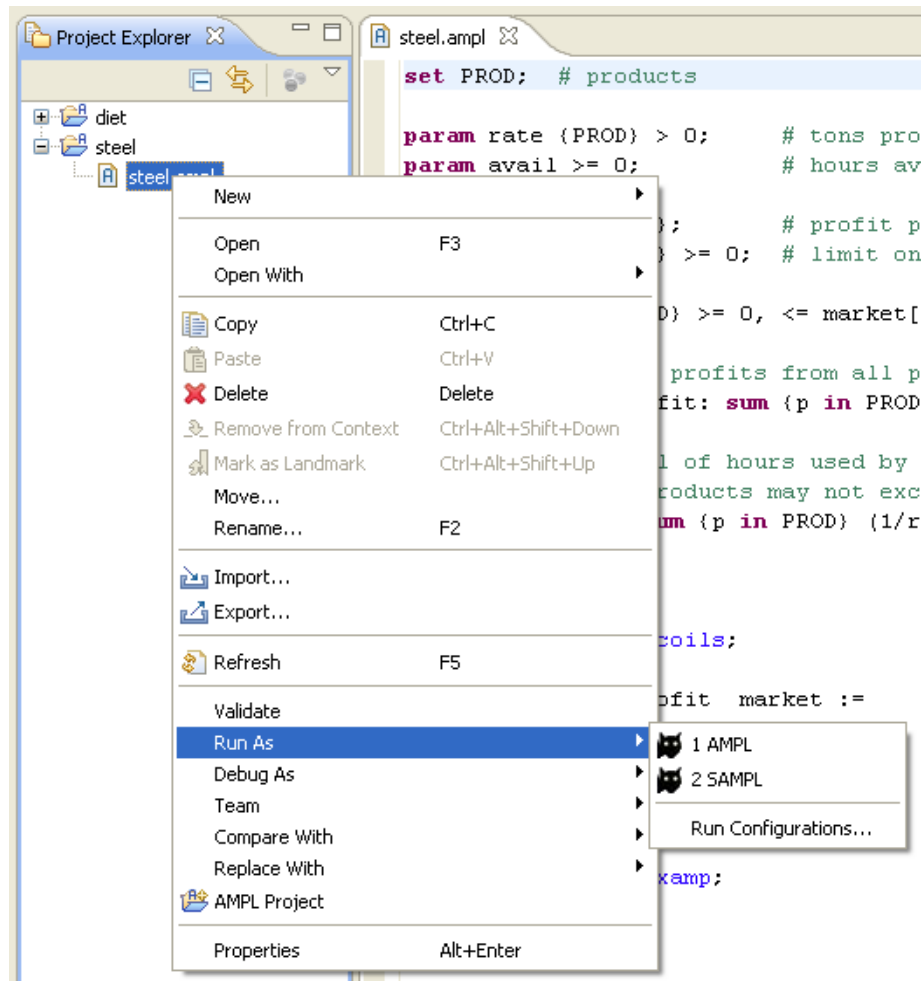


Figure 5.1: Context menu launch

In order to run multiple files, you will need to use the launch configuration dialog. This can be found in either the context menu or launch toolbar shortcuts mentioned in the single file launch (section 5.1). Looking at figure 5.1 and 5.3, you will see the option **Run Configurations...** (or **Debug Configurations...**). by selecting that you will open the launch configuration dialog (figure 5.4).

Double click **AMPL/SAMPL** in the tree on the left, or right click it and select **New**. This will create a new configuration and will open a clean **File Selection** tab on the right. You can give the configuration a name to help differentiate between various configurations.

TIP:

You will notice that the configuration created earlier **steel.ampl** is also present on the left hand side. By selecting that, you can add more files to it later on, if you wish to expand that code into



Figure 5.2: Launch toolbar

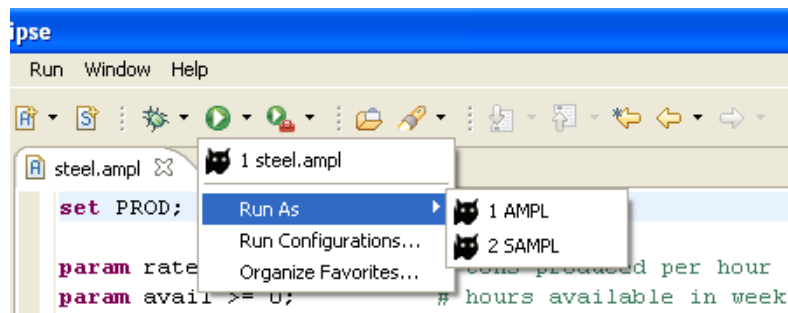


Figure 5.3: Launch toolbar Run menu: **1 steel.ampl** is the previously created launch configuration

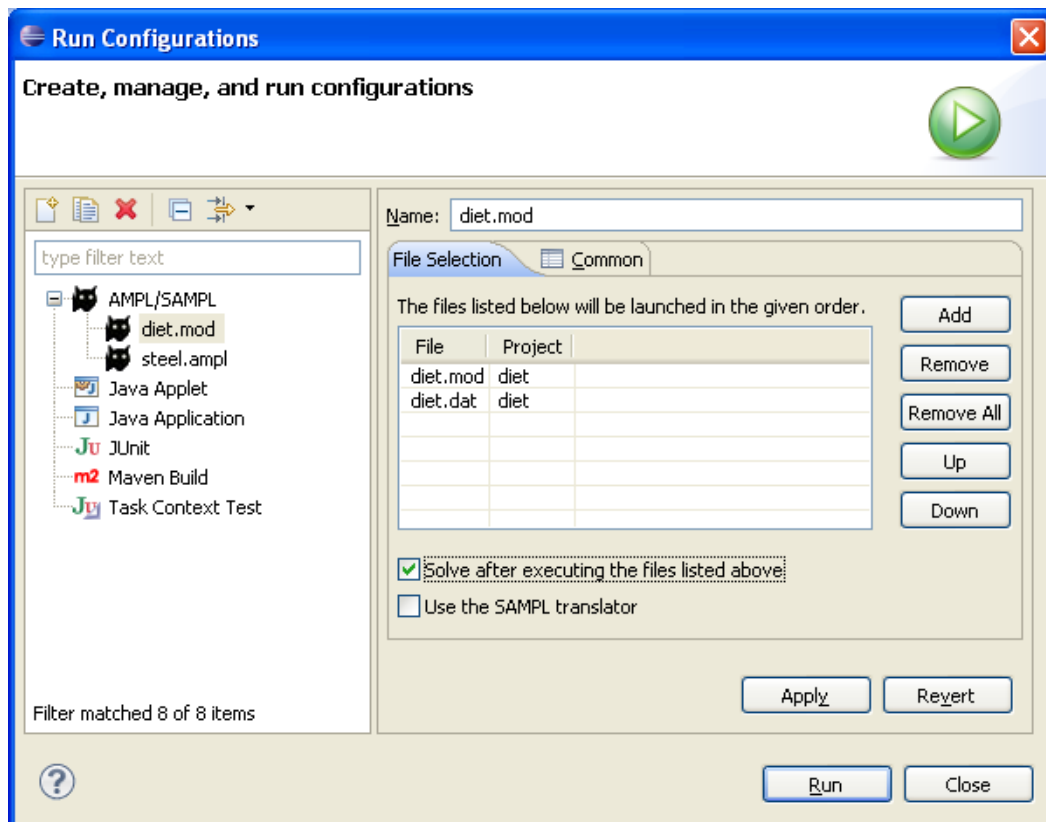


Figure 5.4: Launch configuration dialog

multiple files. So the single file launch can later be adapted for multiple files.

5.2.1 The File Selection tab

In this tab, you can add the files you wish to run together. Start by clicking the **Add** button to add some files. This will open a window with a list of all the files in their respective projects and folders (figure 5.5). If you click the folder, all the files in it will be selected.

After selecting the files, they will be automatically added to the table so you can now select the order in which you would like to execute them by selecting the file in the table and clicking the **Up** and **Down** buttons to move the selection (figure 5.4). Clicking **Apply** just saves the changes, although Eclipse will always ask you to save if there are any unsaved changes before you launch.

You will also notice two checkboxes under the table. The first one (**Solve after executing the files listed above**) is the equivalent of writing 'solve;' at the end of your AMPL code. If checked AMPL will automatically solve after running the files selected in the table. This will also populate the Solution view (see section 5.5). The second one (**Use the SAMPL translator**) specifies whether

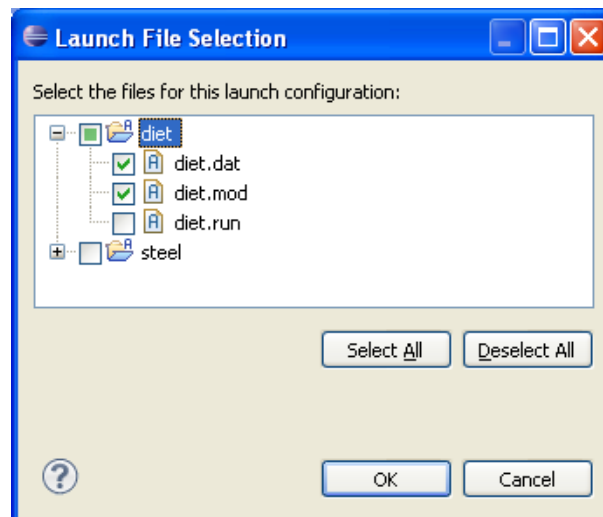


Figure 5.5: Selecting the files for the launch configuration

to use the SAMPL translator instead of the AMPL one.

Once you are ready to run the set of files, hit the **Run** or **Debug** button and your files will be executed.

5.3 Default launch

This launch is available for projects that contain only one model and one data file. This is to avoid the hassle of creating a launch configuration (as it comes under the multiple file launch), for a simple project structure.

You do not need to do anything special for using this type of launch; AMPLDev automatically checks whether the launch conditions of one model and one data file are satisfied in the project and creates a default configuration. You use the same shortcuts as described in single file launch (section 5.1).

The main difference is that when you have a one model and one data project you can even launch it from the project folder's context menu via **Run As** or **Debug As**. If these conditions are not satisfied then, AMPLDev will ask you to create a launch configuration.

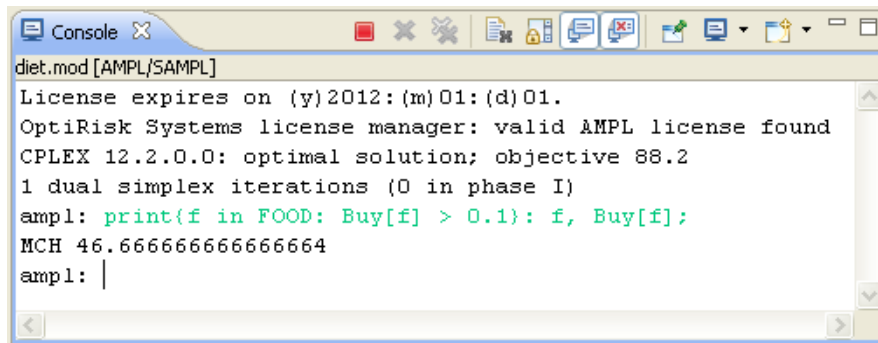
5.4 Viewing output and errors

While a run is in progress the output and errors if there are any will show up in the Console view. This view is generally located in the bottom panel of AMPLDev/Eclipse in the AMPL perspective. If it is not visible you can view it via the **Window** menu, selecting **Show View** and then choose the **Console** view.

Console is interactive meaning that you can enter and execute arbitrary commands there (see figure 5.6). The text entered by the user is colored differently from the output text.

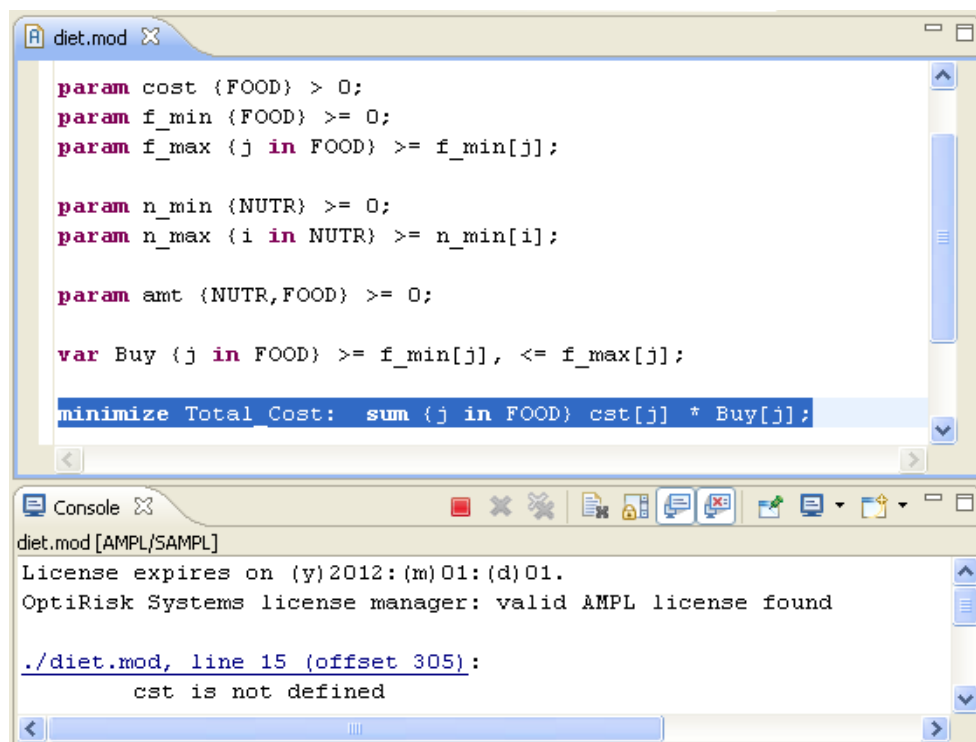
In the case of an error the detailed information and a hyperlink to the error location if available is shown in the console (figure 5.7). In this example, there is a syntax error because 'cost' has been mistyped. The location hyperlink contains the name of the file, the line number and the offset from the beginning of the file. Clicking the hyperlink opens the file in the editor and highlights the line presumably containing the error. Note that in some cases the error may be located in the code above this line, for example in the case of missing semicolon.

You can use the 'display' and 'print' commands to output various information from your AMPL code to the console.



```
diet.mod [AMPL/SAMPL]
License expires on (y)2012:(m)01:(d)01.
OptiRisk Systems license manager: valid AMPL license found
CPLEX 12.2.0.0: optimal solution; objective 88.2
1 dual simplex iterations (0 in phase I)
AMPL: print(f in FOOD: Buy[f] > 0.1): f, Buy[f];
MCH 46.666666666666664
AMPL: |
```

Figure 5.6: Console view



```
diet.mod
param cost {FOOD} > 0;
param f_min {FOOD} >= 0;
param f_max {j in FOOD} >= f_min[j];

param n_min {NUTR} >= 0;
param n_max {i in NUTR} >= n_min[i];

param amt {NUTR,FOOD} >= 0;

var Buy {j in FOOD} >= f_min[j], <= f_max[j];

minimize Total_Cost: sum (j in FOOD) cst[j] * Buy[j];
```

```
diet.mod [AMPL/SAMPL]
License expires on (y)2012:(m)01:(d)01.
OptiRisk Systems license manager: valid AMPL license found

./diet.mod, line 15 (offset 305):
cst is not defined
```

Figure 5.7: Error shown in the Console view

5.5 Viewing the solution

The solution report will be displayed in the Solution view (generally located in the right panel of the AMPL perspective. If you cannot see it then you can open it via **Window** → **Show View...**). This view is only populated if the **Solve after executing the files listed above** option is checked for the launch configuration (see figure 5.4).

There are three sections in the Solution view, refer to figure 5.8. The first part states the **Objective** value and the name of the objective function in brackets.

The first table describes the variables and their corresponding values. **Reduced Cost** is the amount by which the objective value would improve if the value of the corresponding variable is increased by one. **Slack** is the distance from the bound.

The second table describes the constraints where **Body** is the body of the constraint, **Dual Value** is an indication of how much the objective value can be increased if the constraint is relaxed by one unit and lastly **Slack** is the distance between the body of the constraint and its bound.

Objective: 88.19999999999975 (Total_Cost)

Variable	Value	Reduced Cost	Slack
Buy['BEEF']	0.0	1.30000000...	0.0
Buy['CHK']	0.0	0.06999999...	0.0
Buy['FISH']	0.0	1.03000000...	0.0
Buy['HAM']	0.0	1.63000000...	0.0
Buy['MCH']	46.6...	6.74410527...	46....
Buy['MTL']	0.0	0.10000000...	0.0
Buy['SPG']	0.0	0.10000000...	0.0
Buy['TUR']	0.0	1.23000000...	0.0

Constraint	Body	Dual Value	Slack
Diet['A']	699...	-9.82987174...	-1.9...
Diet['B1']	699...	0.0	-1.9...
Diet['B2']	699...	0.12600000...	-1.9...
Diet['C']	163...	-1.29247406...	933....

Figure 5.8: Solution view

Part III

AMPL: A Mathematical Programming Language

Chapter 6

Learning AMPL by Example

This chapter gives an overview of the basic syntax required to start modelling in AMPL. We use a simple real world example to introduce this syntax.

6.1 A Linear Programming Model

National Insurance Associates (NIA) carries an investment portfolio of stocks, bonds and other investment alternatives. Currently £200,000 of funds are available and must be considered for new investment opportunities. The four stock options National is considering and the relevant financial data are as follows:

	Stock			
	A	B	C	D
Price per share	£100	£50	£80	£40
Annual rate of return	0.12	0.08	0.06	0.10
Risk measure per £ invested	0.10	0.07	0.05	0.08

Table 6.1: Financial Data

The risk measure quantifies the risk associated with the stock. It could be simply the mean absolute deviation and it indicates uncertainty in respect of it realising the projected annual return: higher values indicate greater risk.

NIA's top management has stipulated the following investment guidelines,

1. The annual rate of return for the portfolio must be 9%
2. No one stock can account for more than 50% of the total sterling investment

For the NIA problem, the [Linear Programming \(LP\)](#) model needs to estimate the number of stocks NIA should buy with minimum risk and complying with the provided guidelines. These are referred to as the objective and the constraints, respectively.

Objective: Minimise the risk as much as possible while buying the shares. Based on table 6.1, we risk 0.10 on each pound invested in stock A, 0.07 for stock B, 0.05 for stock C and 0.08 for stock D. Therefore, if you are buying x_1 shares for stock A, then our risk exposure is $0.10 \times 100 \times x_1$, since each share of stock A costs £100. Along the same lines, we can formulate our risk to be,

$$Risk = (0.10 \times 100 \times x_1) + (0.07 \times 50 \times x_2) + (0.05 \times 80 \times x_3) + (0.08 \times 40 \times x_4)$$

Next, we formulate the constraints. Some constraints are explicitly given such as the investment guidelines in our problem but some need a little more logic reasoning such as constraint 1, described below.

Constraint 1: The NIA has given a budget (£200,000) on the amount we have available to invest in these four stocks. So, we must comply with the budget.

$$(100 \times x_1) + (50 \times x_2) + (80 \times x_3) + (40 \times x_4) \leq 200000$$

Constraint 2: The first guideline says that the annual rate of return must be 9%. Table 6.1 contains the annual rate of return for each stock. From that, we have the following,

$$(0.12 \times 100 \times x_1) + (0.08 \times 50 \times x_2) + (0.06 \times 80 \times x_3) + (0.10 \times 40 \times x_4) \geq 200000 \times 0.09$$

Constraint 3: The second guideline says that no single stock can be more than 50% of the total sterling investment. So each stock can only be invested for a maximum of £100,000.

$$100 \times x_1 \leq 100000$$

$$50 \times x_2 \leq 100000$$

$$80 \times x_3 \leq 100000$$

$$40 \times x_4 \leq 100000$$

This linear program is summarised in listing 6.3.

An algebraic formulation of the complete model is given below along with the corresponding data for the NIA problem after,

Listing 6.1: Algebraic Formulation of NIA Model

Given:	S	a set of stocks
	a_i	$i \in S$, price per share of stock i
	b_i	$i \in S$, annual rate of return for stock i
	c_i	$i \in S$, risk per pound sterling invested in stock i
	R	required annual rate of return for portfolio
	M	maximum fraction of total pound sterling investment a stock can account for
	F	total funds available
Define:	x_i	$i \in S$, number of shares bought of stock i
Minimise:	$\sum_{i \in S} c_i a_i x_i$	risk over all shares bought for all stocks
subject to:	$\sum_{i \in S} (a_i x_i) \leq F$	amount invested must be less than or equal to available funds
	$\sum_{i \in S} (b_i a_i x_i) \geq RF$	total return must be greater than or equal to required return of portfolio
	$a_i x_i \leq MF$	$i \in S$, investment in stock i must be less than the maximum a stock can account for in total sterling investment

Listing 6.2: Data for the NIA Model

$$S = \{A, B, C, D\}$$

$$R = 0.09$$

$$M = 0.5$$

$$F = 200000$$

i	A	B	C	D
a_i	100	50	80	40
b_i	0.12	0.08	0.06	0.10
c_i	0.10	0.07	0.05	0.08

6.2 Translating the model into AMPL

Listing 6.1 describes a generalised model which does not have to be specific to the NIA problem; it is simply a minimisation model based on some constraints. Only when the data in Listing 6.2 is applied to the model, are we addressing a particular problem. The NIA problem can now be written out explicitly as shown in Listing 6.3.

Listing 6.3: Linear Program of NIA problem

Minimise: $10x_1 + 3.5x_2 + 4x_3 + 3.2x_4$

subject to: $100x_1 + 50x_2 + 80x_3 + 40x_4 \leq 200000$

$12x_1 + 4x_2 + 4.8x_3 + 4x_4 \geq 18000$

$100x_1 \leq 100000$

$50x_2 \leq 100000$

$80x_3 \leq 100000$

$40x_4 \leq 100000$

Such a formulation is manageable when the data set is small enough to be written explicitly, but as the data set increases, it is harder to do this. AMPL is a language that encourages separation of model and data; it allows you to define your model and data as their corresponding algebraic formulations described in Listing 6.1 and 6.2. Thereby, making the model independent of the data and letting the language manage the formulation of the optimisation problem corresponding to Listing 6.3.

The equivalent AMPL representation of the NIA problem is specified in Listing 6.4 and 6.5

Listing 6.4: AMPL code for Model

```
#Sets
set stocks;

#Parameters
param reqReturn;
param maxAllow;
param totalFunds;
param price {stocks};
```

```

param return {stocks};
param risk {stocks};

#Variables
var buyAmount{stocks} >= 0;

#Objective
minimize riskObj:
    sum{i in stocks}(risk[i] * price[i] * buyAmount[i]);

#Constraints
subject to investment:
    sum{i in stocks}(price[i] * buyAmount[i]) <= totalFunds;
subject to ret:
    sum{i in stocks}(return[i]*price[i]*buyAmount[i]) >=
        ↪reqReturn * totalFunds;
subject to invest{i in stocks}:
    price[i] * buyAmount[i] <= maxAllow * totalFunds;

```

Listing 6.5: AMPL code for Data

```

data;

#Sets
set stocks := A B C D;

#Parameters
param reqReturn := 0.09;
param maxAllow := 0.5;
param totFunds := 200000;

param price := A 100
               B 50
               C 80
               D 40;

param return := A 0.12
                B 0.08
                C 0.06
                D 0.10;

param risk := A 0.10
              B 0.07
              C 0.05
              D 0.08;

```

By having the ability to define a model separately from the data, it is possible to re-use this model for different situations. For example, when the total funds has increased or the portfolio can include more than the four described stocks.

The AMPL syntax very closely follows conventional mathematical terms so it's easy to take the algebraic model and convert it into AMPL. The basic components of AMPL are,

- Sets

- Parameters
- Variables (whose values the solver is to determine)
- Objective (to be maximised or minimised)
- Constraints (the solution must satisfy)

Comments can appear anywhere in the AMPL code. They start with a `#` symbol and are active till the end of the line.

6.2.1 Sets

A set can be defined as a collection of well-defined objects. In our example, we have one fundamental set: the stocks. When defining a set, AMPL uses the `set` keyword before the unique name to identify the set.

```
set stocks;
```

The data associated with this set is represented in Listing 6.5.

Note: All declarations in AMPL must end with a semi-colon(`;`).

6.2.2 Parameters

Parameters are the numerical values that are associated with the model. This is what is generally used to define the available data. In AMPL, we use the `param` keyword to declare a parameter. We have three parameters that take scalars,

```
param reqReturn;
param maxAllow;
param totalFunds;
```

They represent the required annual rate of return for the portfolio, the maximum amount of a single stock that can be accounted for in the investment and the total funds available for the investment, respectively. The scalars are defined in the data (Listing 6.5)

Parameters do not necessarily represent a single value; they can be vectors or matrices of numerical values, as well. The following parameters are a group of values that have been indexed over the set of stocks; implying that for each parameter, there is one numerical value for each object in the set. The indexing set is written after the unique identifier and within curly brackets.

```
param price {stocks};
param return {stocks};
param risk {stocks};
```

The parameter's data assignment demonstrates its ability to take more than one numerical value,

```
param price := A 100
               B 50
               C 80
               D 40;
```

Here, the parameter `price` assigns a value to each stock defined in the set `stocks`.

6.2.3 Variables

These are the decision variables that are determined by the optimising algorithm, unlike parameters whose values are given by the modeller. Apart from that, variables declarations are the same as parameters. In AMPL, a variable is declared using the keyword `var` and they can also be indexed over sets.

```
var buyAmount{stocks} >= 0;
```

In our example, there is only one decision variable which represents the amount of each stock to buy. Hence, it is indexed over the set `stocks`.

Variables and parameters can also apply restrictions on their numerical values. For the variable `buyAmount`, we have put a non-negative restriction i.e. for every value estimated for every stock in the variable, the value must be greater than equal to 0.

6.2.4 Objective

The objective function to be optimised is defined by a linear or non-linear expression. The expression is generally defined using the sets, parameters and variables. Depending on the model, we add the keywords `minimize` and `maximize` before the name of the function.

```
minimize riskObj:
    sum{i in stocks}(risk[i]*price[i]*buyAmount[i]);
```

Please take special care with the spellings as AMPL uses American English.

Here, there are two new concepts: the dummy index `i` and the keyword `sum`. The dummy index is declared by associating it with a set using the `in` keyword: `{i in stocks}`. They are used to obtain a member using the subscript expression (`[]`) on the parameter or variable who have been indexed over this set. For example, in the declaration of parameter `risk`, (`param risk {stocks};`), it has the simplest form of an indexing expression: `{stocks}`. When we wish to index into a particular member, we use the dummy index: `risk[i]`.

The keyword `sum` is used with an indexing expression to sum over all the values defined by the index. In our example, we sum the products of the risk, price and amount to buy for each stock. The scope of the dummy index extends only till the end of this linear expression: the scope of the `sum`.

6.2.5 Constraints

Constraints generally start with the keywords `subject to` but even this is optional; AMPL assumes that any declaration not beginning with a keyword is a constraint. The algebraic description of a constraint may be an equality or inequality composed of the parameters and variables. The simplest constraint imposes a limitation; for example the investment constraint.

```
investment: sum{i in stocks}(price[i]*buyAmount[i]) <=
    ↪totalFunds;
```

Here, we are making sure that the amount of stocks bought is less than or equal to the available funds: an upper limit. Similarly the `ret` constraint places a lower limit on the amount of return expected from our investment.

Most of the constraints in large linear programming models are defined as indexed collections by giving an indexing expression after the constraint name.

```
invest{i in stocks}: price[i]*buyAmount[i] <= maxAllow*
    ↪totalFunds;
```

The constraint `invest` imposes a limit on each stock, represented by the dummy index `i`, by not letting any one particular stock be bought for more than `maxAllow` of the total sterling investment.

Chapter 7

Databases and Spreadsheets

AMPL's structure of indexed data is very similar to the structure of relational tables commonly used in database applications. Users can take advantage of this similarity using AMPL's `table` declaration. This chapter illustrates the use of databases and spreadsheets in AMPL; a simple model (the diet problem) will be used for that purpose and is introduced in the first section. Then, it describes how to prepare the data for AMPL to use, how to establish correspondences between AMPL and data entities and how to perform the actual read/write operations.

7.1 Example: A Diet Problem

In the diet problem, we must choose a mix of prepared foods that will satisfy the daily required nutrients. They must be chosen in such a way that these requirements are met and the cost is minimal for a week's worth of food i.e. at least 700% of each required nutrient. Following (table 7.1) is the list of prepared food items with their corresponding costs,

FOOD	Food Description	Cost	f_min	f_max
BEEF	Beef	3.19	2	10
CHK	Chicken	2.59	2	10
FISH	Fish	2.29	2	10
HAM	Ham	2.89	2	10
MCH	Mac & Cheese	1.89	2	10
MTL	Meat Loaf	1.99	2	10
SPG	Spaghetti	1.99	2	10
TUR	Turkey	2.49	2	10

Table 7.1: Prepared Food Data

`f_min` and `f_max` impose a minimum and maximum limit on the amount of food chosen. If we try to solve the diet problem without any limit on the chosen foods, then the solution tends to have too much of specific food types and none at all of others which is not ideal in a real world situation.

In the same manner we put minimum and maximum limits on the amount of each nutrient (table 7.3): 700% and 20000% respectively, 0 *mg* and 50000 *mg* for sodium (NA), and 16000 *Cal* and 24000 *Cal* for total calories (CAL).

We have two sets that contains the food and nutrient items: FOOD and NUTR. For the already described data, we require 6 parameters: `cost`, `f_min`, `f_max`, `amt`, `n_min` and `n_max`. To represent the data shown in table 7.2, we use the parameter `amt`. It stands for the amount of each nutrient present in each type of food and is therefore indexed over both the sets. The AMPL model is shown in listing 7.1.

FOOD	A(%)	C(%)	B1(%)	B2(%)	NA(mg)	CAL(Cal)
BEEF	60	20	10	15	938	295
CHK	8	0	20	20	2180	770
FISH	8	10	15	10	945	440
HAM	40	40	35	10	278	430
MCH	15	35	15	15	1182	315
MTL	70	30	15	15	896	400
SPG	25	50	25	15	1329	370
TUR	60	20	15	10	1397	450

Table 7.2: Nutrient Data

NUTR	n_min	n_max
A	700	10000
C	700	10000
B1	700	10000
B2	700	10000
NA	0	50000
CAL	16000	24000

Table 7.3: Nutrient Limits

Listing 7.1: Model file for Diet problem

```

set FOOD;
set NUTR;

param cost {FOOD} > 0;
param f_min {FOOD} >= 0;
param f_max {j in FOOD} >= f_min[j];

param n_min {NUTR} >= 0;
param n_max {i in NUTR} >= n_min[i];

param amt {NUTR,FOOD} >= 0;

var Buy {j in FOOD} >= f_min[j], <= f_max[j];

minimize total_cost: sum {j in FOOD} cost[j] * Buy[j];

subject to diet {i in NUTR}:
    n_min[i] <= sum {j in FOOD} amt[i,j] * Buy[j] <= n_max[i];

```

7.2 Preparing the Data

There are various ways to pass the data introduced in the previous section to AMPL. The most obvious is writing it directly as AMPL data; a possible implementation of that is given in listing 7.2.

Listing 7.2: Data file for Diet problem

```

set NUTR := A C B1 B2;
set FOOD := BEEF CHK FISH HAM MCH MTL SPG TUR ;

param:    cost    f_min    f_max :=
  BEEF    3.19    2        10
  CHK     2.59    2        10
  FISH    2.29    2        10
  HAM     2.89    2        10
  MCH     1.89    2        10
  MTL     1.99    2        10
  SPG     1.99    2        10
  TUR     2.49    2        10 ;

param:    n_min    n_max :=
  A        700    20000
  C        700    20000
  B1       700    20000
  B2       700    20000
  NA        0     50000
  CAL    16000    24000 ;

param amt (tr):
      A    C    B1    B2    NA    CAL :=
  BEEF   60   20   10   15   938   295
  CHK    8    0   20   20  2180   770
  FISH   8   10   15   10   945   440
  HAM    40   40   35   10   278   430
  MCH    15   35   15   15  1182   315
  MTL    70   30   15   15   896   400
  SPG    25   50   25   15  1329   370
  TUR    60   20   15   10  1397   450 ;

```

Realistically, we may want to store the data in an external database or spreadsheet; this may be because the data can be very large and cumbersome to type out in AMPL or because we already have the data stored externally. AMPL provides commands which allow you to import the data from an external database or spreadsheet into AMPL's indexed data structure. This is primarily done by the `table` declaration,

```

table table-name inoutopt string-listopt :
    key-spec, data-spec, data-spec, ...;

```

and by the corresponding read table command,

```

read table table-name;

```

The various parts of the above declaration will be described in the following sections.

7.2.1 Spreadsheets

In the following syntax and examples, we use Microsoft Excel to describe connections to an external spreadsheet.

In an Excel file called `diet.xls`, we create a range, called `Foods` with column names: `cost`, `f_min` and `f_max`. Since these parameters are indexed over the set, `FOOD`, we add a column to

represent it (see AMPL code listing 7.1). It is important to create a range and give it a suitable unique name as this is what we will be using when reading and writing the data between the spreadsheet and AMPL (see figure 7.1(a)).

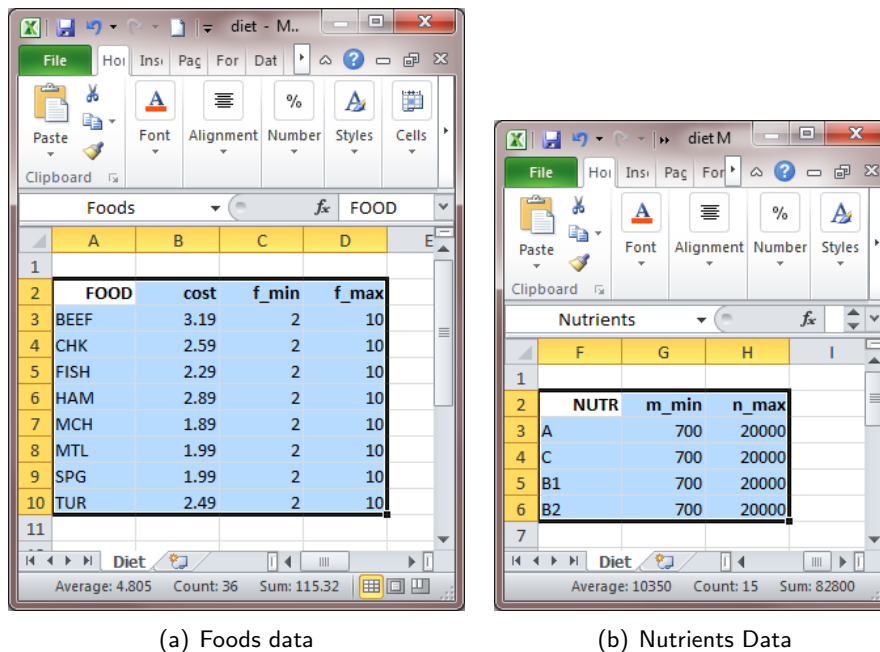


Figure 7.1: Excel Spreadsheet for Foods and Nutrients data

In the Foods range, the key column is FOOD. The key column is indicated by the sets a parameter is indexed over; moreover it can be inferred from the table (figure 7.1(a)) as the minimum amount of column data needed to identify one row without ambiguity.

Similarly, we create a range called Nutrients for n_{\min} and n_{\max} indexed over NUTR and a table Amounts for the third parameter amt. There is a slight difference in the latter table i.e. the parameter is indexed over two sets: NUTR and FOOD. To handle this, there is a column for each set which goes over the set values in such a manner that it creates a list of unique pairs between the two sets (see figure 7.2).

In order to connect to the spreadsheet, the optional *string-list* which is part of the `table` declaration is used. For spreadsheets, the table *string-list* consists of the table handler name, the filename and the optional range name. The range name is only necessary to specify if the AMPL *table-name* in the `table` declaration is different from it.

```
table Foods IN "ODBC" "diet.xls": ... ;
```

The table handler being used is called [Open Database Connection \(ODBC\)](#) and as the name suggests, it provides access to databases and spreadsheets through an open database connection. The Excel filename is "diet.xls". The filename may also contain the directory if it is at a different location. If the AMPL *table-name* was different, then the declaration will change to,

```
table FoodInput IN "ODBC" "diet.xls" "Foods": ... ;
```

Since the AMPL table name is now FoodInput, we must specify the range name in the Excel file that contains the data: "Foods".

7.2.2 Databases

AMPL allows connections to any SQL database as long as the appropriate ODBC handler is installed. For the purposes of this example we have used MySQL.

	J	K	L	M
2	NUTR	FOOD	amt	
3	A	BEEF	60	
4	B1	BEEF	10	
5	B2	BEEF	15	
6	C	BEEF	20	
7	NA	BEEF	938	
8	CAL	BEEF	295	
9	A	CHK	8	
10	B1	CHK	20	
11	B2	CHK	20	
12	C	CHK	0	
13	NA	CHK	2180	
14	CAL	CHK	770	
15	A	FISH	8	
16	B1	FISH	15	
17	B2	FISH	10	
18	C	FISH	10	
19	NA	FISH	945	
20	CAL	FISH	440	
21	A	HAM	40	
22	B1	HAM	35	

Figure 7.2: The amount of nutrients in each food. (The list is not completely visible.)

We create an SQL database called `diet`, in which there is a table called `Foods` with column names that match the parameter and set names. Similarly, we create table `Nutrients` and `Amounts` with their columns corresponding to the appropriate parameters and sets.

The main difference between spreadsheets and databases occurs in the connection strings. There are two ways to connect to a database: [Data Source Name \(DSN\)](#) or the standard connection string.

Standard Connection String

The *string-list* component of the `table` declaration is similar to Excel; there are three typical parts: table handler name (which will be "ODBC" again), a connection string and a relational table name (again, if this is omitted then the relational table name will be taken as the AMPL *table-name*). The interesting part is in the connection string; a typical example of a MySQL connection string is,

```
"DRIVER={MySQL ODBC 5.1 Driver};SERVER=localhost;DATABASE=diet;
↔USER=myUsername;PASSWORD=myPassword;OPTION=3;INITSTMT=SET
↔sql_mode='ANSI_QUOTES';"
```

The connection parameters are:

- **DRIVER:** This is the name of the driver which allows the ODBC handler to connect AMPL and the SQL database.
- **SERVER:** Name of the server that is hosting the database. If the database is on the machine you are using, a local database, then the server name is `localhost`.
- **DATABASE:** Name of the database. In this case, it is `diet`.

- **USER:** The username of the account being used. If the username is the root account then use `root`.
- **PASSWORD:** The password associated with the user account.
- **OPTION:** This is specific to MySQL and is used to make the server behave in a specific manner (Please refer to MySQL reference for details). We always set `OPTION=3`; for the scope of this manual.
- **INITSTMT:** This is also specific to MySQL and is used in this case to switch the ANSI mode (`SET sql_mode='ANSI_QUOTES'`). When connecting to a MySQL database through AMPL, the quotes must be in ANSI mode or else AMPL throws a table not found error.

The connection string changes based on the MySQL ODBC connector version and the SQL application used. Information regarding connection strings is easily available online (Example: <http://www.connectionstrings.com/>).

Hence, the complete table declaration for Foods is,

```
table dietFoods IN "ODBC" "DRIVER={MySQL ODBC 5.1 Driver};
  ↪SERVER=localhost;DATABASE=diet;USER=myUsername;PASSWORD=
  ↪myPassword;OPTION=3;INITSTMT=SET sql_mode='ANSI_QUOTES';"
  ↪"foods": ... ;
```

If instead of `dietFoods`, we use `foods`, then the third string in the above declaration is not required.

Data Source Name (DSN)

An alternative to using a complete connection string is to use the ODBC configuration utility and create a [DSN](#).

For Windows users, go to Control Panel → Administrative Tools → Data Sources (ODBC). The ODBC Data Source Administrator dialog will pop up; then select Add under User DSN or System DSN. It will ask you to select a driver; in our case, we choose MySQL ODBC 5.1 Driver and click Finish.

Once Finish is clicked, MySQL will open its Data Source Configuration utility (see figure 7.3). The connection parameters are the same as described in the previous section. The 'Initial Statement' corresponds to `INITSTMT` and can be found under Details and then the Connection tab. It contains the SQL statement: `SET sql_mode='ANSI_QUOTES';`. With the help of the utility, you can even 'Test' the connection to the database to make sure it is working.

The table declaration for using a DSN is given below,

```
table dietFoods IN "ODBC" "Diet DSN" "foods": ... ;
```

Here the second string will contain the data source name, which is "Diet DSN" (figure 7.3). Again, if the AMPL *table-name* is the same as the SQL table name, then the third string ("foods") is not required.

7.3 Reading Data from Tables

In order to use an external relational table, we employ a `table` declaration that specifies a read/write status of `IN`. The general form of this kind of declaration is,

```
table table-name IN string-listopt :
  key-spec, data-spec, data-spec, ... ;
```

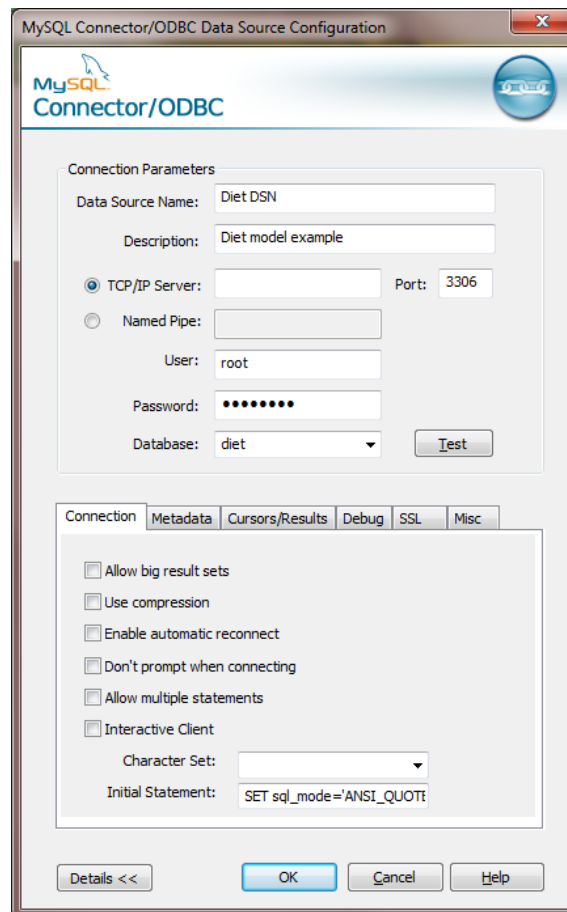


Figure 7.3: MySQL Data Source Configuration Utility

Each table declaration has two parts divided by the colon. Before the colon, the declaration provides general information: *table-name* - the name by which the table is known within AMPL, the keyword **IN** - states that the default for all non-key table columns will be read-only (AMPL will use the columns as only input and will not write out to them) and the optional *string-list* - provides the information to locate the table in an external database file, and is specific to the database type and access method being used (Described in section 7.2).

After the colon, the declaration gives the details of the correspondence between AMPL entities and relational table columns. The *key-spec* - names the key columns and is surrounded by square brackets, [...] and *data-spec* - gives the data columns. Data values are subsequently read from the table into the AMPL entities by the following command,

The **table** declaration only defines a correspondence. To read values from columns of a relational table into the AMPL sets and parameters, we must use an explicit **read table** command. The *table-name* must be the same in both the declarations.

```
read table table-name;
```

In the Diet problem, to read data from the table Foods in the spreadsheet, we would use the following declaration followed by the **read** command,

```
table Foods IN "ODBC" "diet.xls":
  FOOD <- [FOOD], cost, f_min, f_max;
read table Foods;
```

The *string-list*, "ODBC" "diet.xls" specifies that we are connecting to the external relational database through an Open Database Connection (ODBC). **For the sake of simplicity, we will not**

keep writing out the *string-list*; it is the only thing that varies for the database but the rest of the syntax remains the same. In the second part, the expression `FOOD <- [FOOD]` indicates that the entries in the key column `FOOD` will populate the AMPL set `FOOD`. `cost`, `f_min` and `f_max` are the names of the other three columns in the relational table from which we will read the values into the corresponding parameters `cost`, `f_min` and `f_max`.

7.3.1 Read Parameters only

Values from the data columns are assigned to like-named parameters in AMPL. It is sufficient to give a square-bracketed list of key columns and then a list of data columns. The simplest case is when there is only one key column,

```
table Foods IN : [FOOD], cost, f_min, f_max;
```

Here, the columns `cost`, `f_min` and `f_max` are associated with their corresponding parameters in the current AMPL model. Note that we are not using the `FOOD <- [FOOD]` expression as in this case we will not be reading the key column values, only the parameters.

When the following command is executed,

```
read table Foods;
```

the relational table is read one row at a time. The key column entry is used as a subscript to the parameters; for example, referring to table 7.1, `cost[HAM]` will be assigned with 2.89.

Reading multidimensional parameters are done in the same manner as for a single indexing set. The name of each data column must match an AMPL parameter and the dimension of the parameter's indexing set must equal the number of key columns.

```
table Amounts IN : [NUTR, FOOD], amt;
read table Amounts;
```

Values of unindexed or scalar parameters may also be supplied by a relational table; the table will have only one row and no key columns i.e. the data column contains a single value. The `table` declaration will have an empty *key-spec*, `[]`. For example, if we have a parameter that takes in a value for the number of time periods (`param T;`), the `table` declaration can be written as,

```
table TimePeriods IN : [], T;
read table TimePeriods;
```

7.3.2 Read Sets and Parameters

As mentioned in section 7.3, we can read the members of a set from the data columns at the same time that parameters indexed over that set are read from the table by using the following expression for the *key-spec*,

```
set-name <- [key-col-spec, key-col-spec, ...]
```

The simplest case involves reading a one-dimensional set and the parameters indexed over it, as in the diet problem, we have,

```
table Foods IN :
  FOOD <- [FOOD], cost, f_min, f_max;
```

In this particular case, where the key column `FOOD` has the same name as the AMPL set `FOOD`, the `table` declaration can have an abbreviated *key-spec*,

```
table Foods IN :
  [FOOD] IN, cost, f_min, f_max;
```

We can also write the `table` declaration in the following manner if the key column's name in the relational table is different from the AMPL set,

```
table Foods IN :
  FOOD <- [FoodKey], cost, f_min, f_max;
```

Here, the entries of the key column `FoodKey` from the relational table are read into AMPL's set, `FOOD`.

A similar syntax is used for the multidimensional case. In the case of reading the parameter `amt` which is indexed over two sets, `NUTR` and `FOOD`, we can use the following declaration,

```
table Amounts IN :
  PAIR <- [NUTR, FOOD], amt;
```

When the `read table Amounts;` declaration is executed, each row of the relational table provides a pair of entries from the key columns: `NUTR` and `FOOD`. These members are read into AMPL as pairs into the two-dimensional set, `PAIR`. Hence, we have to update the code of the diet model from listing 7.1 to the following,

```
set FOOD;
set NUTR;
set PAIR within {NUTR, FOOD};

...
param amt {PAIR} >= 0;
...
```

We have added a new set, `PAIR`, that is made up of the sets `NUTR` and `FOOD`, and we have updated the `amt` parameter's indexing set.

7.3.3 Establishing Correspondences

There are times when the AMPL model's set and parameter declarations do not necessarily conform in all respects to the organization of tables in the external databases. The most common difference is when the names of the parameters differ from their corresponding data columns in the relational tables. To resolve such an issue, we can use the following form in the *data-spec*,

param-name ~ *data-col-name*

Thus, in the diet problem, if `Foods` is defined as,

```
table Foods IN:
  [FOOD], cost, f_min ~ lowerlim, f_max ~ upperlim;
```

then the AMPL parameter, `f_min` and `f_max` would be read from data columns `lowerlim` and `upperlim` in the external relational table.

Similarly, *index* ~ *key-col-name* can be used in the *key-spec* to associate a dummy index for subscripting the *param-name* in the *data-specs*. There are three common cases to benefit from such a correspondence,

Case 1: When the numbering in a parameter index is different between the AMPL parameter and relational table. For example, if the relational table started counting time periods from 0 but in the model, the time periods start at 1, the `table` declaration can be as follows,

```
table BondPrices IN:
    [b ~ BOND, t ~ TIME], price[b,t+1] ~ price;
```

Case 2: When the order of multidimensional parameters in AMPL do not match each other. For example, if we have another amount parameter in the diet model called `amt2` which is declared as,

```
param amt {NUTR,FOOD} >= 0;
param amt2 {FOOD,NUTR} >= 0;
```

The `table` declaration for these two parameters would be as follows,

```
table Amounts IN:
    [n ~ NUTR, f ~ FOOD], amt, amt2[f,n] ~ amt2;
```

Case 3: When the values of an AMPL parameter are divided among several data columns. For example, in the parameter `amt`, if the nutritional data for BEEF and CHK are given in different data columns, `amtBeef` and `amtChk` respectively. Then, the `table` declaration is,

```
table Amounts IN: [n ~ NUTR],
    amt[n, "BEEF"] ~ amtBeef, amt[n, "CHK"] ~ amtChk;
```

In all these cases, when using dummy indices, it is important to use a correspondence (\sim *data-col-name*) even though the parameter names match in AMPL and the relational table. This is because, `amt2[f,n]` does not exist in the relational table and hence we must associate it with `amt2`. In general, wherever the AMPL expression for the recipient is not a valid data column in the relational table, a \sim *data-col-name* must be used.

7.3.4 Other values

A `table` declaration used for reading data can have assignable expressions anywhere a parameter is permitted. In AMPL, an assignable expression is one which can have a value assigned to it. Hence variables and constraint names can be assigned with values while reading from a relational table. This is primarily done when evaluating a previously stored solution or to provide initial values to the solver. For example,

```
table Foods IN:
    [FOOD] IN, cost, f_min, f_max, Buy;
read table Foods;
```

We read the set members of `FOOD`, parameters values and also the initial values for the variable, `Buy`.

7.4 Writing Data into Tables

When writing data to a relational table, the `table` declaration uses a read/write status of `OUT`.

```
table table-name OUT string-listopt :
    key-spec, data-spec, data-spec, ...;
```

The optional *string-list* is the same as the one used when reading from a table and provides the information about the external database file, and is specific to the database type and access method being used. Once again, we will not be including this *string-list* in the following examples where irrelevant.

Just as when reading from the table, the actual writing only takes places when the

```
write table table-name;
```

command is executed. Generally, the external file specified in the **table** declaration is either created if it does not exist or overwritten if it does. The same applies for a relational table that is specified in the *string-list*. The **table** declaration can also specify appending or overwriting columns in an already existing table, which is described in section 7.5.

TIP:

Generally, the **write** command should be called after the model is solved, if there are entities in the *data-specs* that are populated while solving the model. Otherwise, those data columns will be empty and the output table may not be useful.

Although, the *key-specs* and *data-specs* are similar to the ones used for reading from a table, there are some differences. When writing to a table, the syntax allows for a broader range of AMPL expressions to be used especially because when reading, the data is already existing in the table but in the case of writing, the data needs to be determined by AMPL. The next two sections describe how writing rows can be inferred using either the *data-specs* or the *key-spec*.

7.4.1 Rows inferred from the data specifications (*data-specs*)

When only the key column names for the relational table are specified in a bracketed list, i.e. without specifying any indexing AMPL sets,

```
[key-col-name, key-col-name, ...]
```

then the tables rows are determined using the union of the indexed sets implied by the AMPL entities stated in the *data-specs*: inference is implicit. For this reason, all the items listed in the *data-specs* must have the same dimension.

For example, in the case of entities that index over one set,

```
table Foods OUT "ODBC", "diet.xls" "FoodsOut":  
  [FoodName], f_min, Buy, f_max;
```

the *string-list* here specifies the connection type and the external filename, plus the relational table name "FoodsOut". When **write table** Foods; is executed the following columns are created in the table: FoodName, f_min, Buy and f_max. Here, the implicit set among the AMPL entities (f_min, Buy and f_max) is FOOD. Therefore, each row of the column FoodName gets a member of the set FOOD and the rest of the columns get the values of f_min, Buy and f_max subscripted by that member.

Tables with more than one dimension are managed in a similar manner. Therefore, the following command,

```
table Amounts OUT : [NUTR, FOOD], amt;  
write table Amounts;
```

will produce an output similar to figure 7.2. The rows will be indexed over the union of NUTR and FOOD.

Using Correspondences

We can also export a relational table with suffixed variables or constraints such as the dual and slack values related to the constraint diet.

```
table Nutrs OUT: [Nutrient],
    diet.lslack, diet.ldual, diet.uslack, diet.udual;
```

But this will produce an error since most database software do not allow a 'dot' in their column names. Hence we establish a correspondence (\sim).

```
table Nutrs OUT: [Nutrient],
    diet.lslack ~ lb_slack, diet.ldual ~ lb_dual,
    diet.uslack ~ ub_slack, diet.udual ~ ub_dual;
```

This will assign the data columns with the name on the right of \sim . It can also be used with unsuffixed names when you wish to have a data column name different from the AMPL entity because sometimes AMPL entities do not have valid column names.

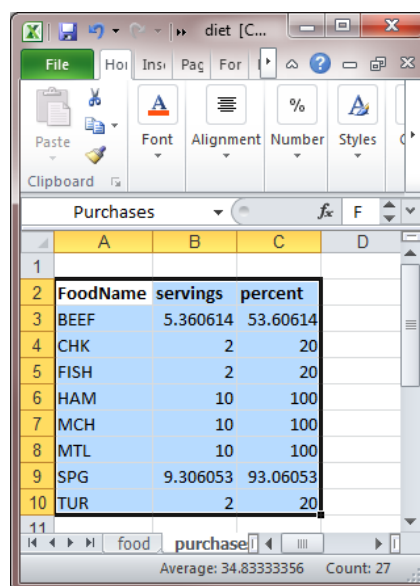
Correspondences can also be used with an AMPL expression instead of only an entity. These will generally require indexing over the corresponding set using a dummy index. The following statements are examples of unsuffixed names and indexing over a dummy index for calculating the data.

```
table Purchases OUT: [FoodName],
    Buy ~ servings, {f in FOOD} 100*Buy[f]/f_max[f] ~ percent;
```

or

```
table Purchases OUT: [FoodName],
    {f in FOOD} (Buy[f] ~ servings,
        100*Buy[f]/f_max[f] ~ percent);
```

Figure 7.4 shows a spreadsheet with the output of any of the previous `table` declarations.



	A	B	C	D
1				
2	FoodName	servings	percent	
3	BEEF	5.360614	53.60614	
4	CHK	2	20	
5	FISH	2	20	
6	HAM	10	100	
7	MCH	10	100	
8	MTL	10	100	
9	SPG	9.306053	93.06053	
10	TUR	2	20	
11				

Figure 7.4: Output for table Purchases

The *data-spec* can also contain expression with operators like `sum` that need to define their own dummy indices. For example, the following declaration calculates the total of each nutrient over all the food products,

```
table Totals OUT: [NUTR],
  {n in NUTR}(sum {f in FOOD} amt[n,f] ~ TotNut);
```

7.4.2 Rows inferred from a key specification (*key-spec*)

The other option for writing to a relational table is when the table rows are determined by explicitly specifying the AMPL sets.

```
set-spec -> [key-col-spec, key-col-spec, ...]
```

set-spec can be the name of an AMPL set or a set-expression enclosed in { } and *key-col-specs* are the names of the corresponding key columns for the relational table. In section 7.3.2, the arrow <- points in the other direction, indicating values that need to be read into the set; this declaration uses the opposite, (->), to indicate information is to be written from the set into the key columns.

In the case of a one-dimensional set, the following command,

```
table Foods OUT:
  FOOD -> [FoodName], f_min, Buy, f_max;
```

will create a table row for each member of the AMPL set, FOOD, when the `write table Foods;` command is executed. The key column can have the same name as the AMPL set,

```
table Foods OUT:
  FOOD -> [FOOD], f_min, Buy, f_max;
```

In this special case, `FOOD -> [FOOD]` can also be written as `[FOOD] OUT`.

Table with more than one dimension are managed by surrounding the list of sets within curly brackets and the number of *key-col-specs* must be equal to the dimension of the *key-spec*. Therefore, the `table` declaration will be written as,

```
table Amounts OUT :
  {NUTR, FOOD} -> [Nutrient, FoodName], amt;
```

Using Correspondences

The use of ~ is the same as the previous section's 'Using Correspondences' (section 7.4.1). The only time the syntax differs is when using dummy indices. Since the rows are now determined from the *key-spec*, the definition of the dummy index can appear either in the *set-spec* or the *key-col-spec*, shown below respectively,

```
table Purchases OUT: {f in FOOD} -> [FoodName],
  Buy[f] ~ servings, 100*Buy[f]/f_max[f] ~ percent;
```

and

```
table Purchases OUT: FOOD -> [f ~ FoodName],
  Buy[f] ~ servings, 100*Buy[f]/f_max[f] ~ percent;
```

Figure 7.4 represents the output of both the above mentioned declarations.

7.5 Reading and Writing into the same Table

The previous sections describe how to import data from an external relational table and how to export data into a different relational table. There could be cases in which we would want to use the same external relational table for both actions: importing and exporting. There are two ways to handle this: either by using two separate `table` declarations or combining them into one declaration which will specify which columns are to be read and which are to be written into.

7.5.1 Using two `table` declarations

By using the declarations described in the previous sections, one `table` declaration can be used for reading the data and a second one can be used for writing into the same table. However, the outcome of the `write` command will overwrite the entire relational table. This is not always ideal because usually, when writing to an already existing table, one would prefer either adding or rewriting certain columns but not re-writing the whole table.

To distinguish between columns meant for reading versus columns for writing, each column can be given a read/write status. For example, if the diet problem has an external table, called `Foods`, with the columns `cost`, `f_min` and `f_max` for reading and `Buy` for writing, then the `table` declaration to *read* the data will be,

```
table FoodInput IN "ODBC" "diet.xls" "Foods":
  FOOD <- [FoodName], cost, f_min, f_max;
```

In order to write the results into the *same* table without overwriting the table, the declaration can be as follows:

```
table FoodOutput "ODBC" "diet.xls" "Foods":
  [FoodName], cost IN, f_min IN, Buy OUT, f_max IN;
```

First, `read table FoodInput;` is executed wherein the three columns `cost`, `f_min` and `f_max` are read into AMPL; if the `Buy` column exists in the relational table, it will be ignored. Once the model is solved, `write table FoodOutput;` is executed. The only column that will be written is the column with an `OUT` status, `Buy`; the other columns are left as-is.

If the AMPL table for output is declared in the following manner,

```
table FoodOutput "ODBC" "diet.xls" "Foods":
  [FoodName], Buy OUT;
```

where *all* the columns have an `OUT` status, then most database software assumes that the entire table must be re-written. Hence the above declaration will delete all the previously read data columns and overwrite it with the `Buy` column.

Although, if we use the following declaration instead,

```
table FoodOutput "ODBC" "diet.xls" "Foods":
  [FoodName], Buy;
```

then only the `Buy` columns will be overwritten; the default status of the columns in such a situation is taken to be `INOUT`.

7.5.2 Using the same `table` declaration

When using the same declaration to perform reading and writing, the columns in the *data-spec* can contain the following read/write statuses:

- `IN`: for a column that is only for reading

- **OUT**: for a column that is only for writing
- **INOUT**: for a column which will be for both, reading and writing

The *key-spec* may use the following arrows:

- **<-** : reads the data in the key columns into an AMPL set
- **->** : writes the data from the AMPL set into the key columns
- **<->** : does both, reads and writes data between the AMPL set and key columns

In the diet example, a single **table** declaration can be written as,

```
table Foods "ODBC" "diet.xls" "Foods":
    FOOD <- [FoodName], cost IN, f_min IN, Buy OUT, f_max IN;
```

When **read table** Foods; is executed, the data from FoodName is read into the AMPL set FOOD and the data columns cost, f_min and f_max are read into their corresponding parameters. When the **write table** Foods; command is executed then data is written into the table's Buy column only.

7.6 Indexed collections of Tables and Columns

Sometimes declaring an indexed collection of tables or defining an indexed collection of data columns within a table is more convenient for the purposes of the problem. This can be done using the **table** declaration.

7.6.1 Indexed collections of Tables

Just the way sets, parameters and other AMPL components can have an indexing expression, in the same manner, tables can also have an indexing expression.

```
table table-name {indexing-expr}opt string-listopt : ...;
```

Each member of the indexing expression defines an AMPL table which are denoted by appending a bracketed subscript or subscripts of the *table-name*.

In the diet problem, the following declaration defines one table for each member of the set FOOD,

```
table DietSens {j in FOOD}
    OUT "ODBC" "diet.xls" ("Sens" & j):
    [Food], f_min, Buy, f_max;
```

When **write table** DietSens; is executed, diet.xls will now contain a range for every member of FOOD in their corresponding sheets (see figure 7.5). Each of these ranges will be named using the third item (which defines the table/range name) in the *string-list*: ("Sens" & j), where j represents the index for the set FOOD. Therefore, AMPL table DietSens["BEEF"] will have a corresponding range called SensBEEF and so on.

If instead of the third item in the *string-list*, the second item (which defines the filename) contains a string expression with the index,

```
table DietSens {j in FOOD}
    OUT "ODBC" ("DietSens" & j & ".xls"):
    [Food], f_min, Buy, f_max;
```

	A	B	C	D
1	Food	f_min	Buy	f_max
2	BEEF	2	5.360614	10
3	CHK	2	2	10
4	FISH	2	2	10
5	HAM	2	10	10
6	MCH	2	10	10
7	MTL	2	10	10
8	SPG	2	9.306053	10
9	TUR	2	2	10

Figure 7.5: Table with indexed ranges (Shown above only for BEEF)

then on execution of the `write` command, there will be an Excel file created for each item in `FOOD` whose names will be `DietSensBEEF.xls` and so on. Each of these files will contain a single table whose default name will be `DietSens`.

Similarly, the table indexing expression can be used in a string expression to create different *data-col-names* but in the same relational table,

```
table DietSens {j in FOOD} "ODBC" "diet.xls":
  [Food], Buy ~ ("Buy" & j);
```

After the `write` command, figure 7.6 illustrates how the `table` declaration has created a column for each member of `FOOD` and given them the names corresponding to the string expression: `("Buy" & j)`. This declaration does not have a read/write status of `OUT`; if it did then each column would have overwritten the previous one. Instead the *data-spec* `Buy` has been left without a read/write status, hence in this case the default will be `INOUT`.

	A	B	C	D	E	F	G	H	I
1	Food	BuyBEEF	BuyCHK	BuyFISH	BuyHAM	BuyMCH	BuyMTL	BuySPG	BuyTUR
2	BEEF	5.360614	5.360614	5.360614	5.360614	5.360614	5.360614	5.360614	5.360614
3	CHK	2	2	2	2	2	2	2	2
4	FISH	2	2	2	2	2	2	2	2
5	HAM	10	10	10	10	10	10	10	10
6	MCH	10	10	10	10	10	10	10	10
7	MTL	10	10	10	10	10	10	10	10
8	SPG	9.306053	9.306053	9.306053	9.306053	9.306053	9.306053	9.306053	9.306053
9	TUR	2	2	2	2	2	2	2	2

Figure 7.6: Table with indexed data columns

7.6.2 Indexed collections of Data Columns

In a table declaration, each *data-spec* generally refers to a different AMPL parameter, variable or expression. However, there are times when data values that correspond to a single AMPL entity are split into multiple data columns, one for each member of a specified indexing set. This is most common when reading or writing two-dimensional tables. For example, the parameter,

```
param amt {NUTR,FOOD} >= 0;
```

can also be represented as a two-dimensional table (see figure 7.7) instead of a list of unique pairs generated by {NUTR,FOOD} (as in figure 7.2)

The screenshot shows a Microsoft Excel spreadsheet titled 'diet2 [Compatibility Mode] - Microsoft Excel'. The spreadsheet displays a table with the following data:

	A	B	C	D	E	F	G	H	I
1	NUTR	BEEF	CHK	FISH	HAM	MCH	MTL	SPG	TUR
2	A	60	8	8	40	15	70	25	60
3	C	20	0	10	40	35	30	50	20
4	B1	10	20	15	35	15	15	25	15
5	B2	15	20	10	10	15	15	15	10
6	NA	938	2180	945	278	1182	896	1329	1397
7	CAL	295	770	440	430	315	400	370	450

Figure 7.7: Two-dimensional data table in Excel

The general form for specifying an indexed collection of data columns is,

$$\{indexing-expr\} < data-spec, data-spec, \dots >$$

Each *data-spec* has any of the forms previously seen and the *indexing-expr* defines one or more dummy indices that run over the indexing set. These indices are used in expressions within the *data-specs* and also appear in string expressions that give the names of the columns in the external database.

In the Diet problem, such a **table** declaration can be written as,

```
table dietAmts IN "ODBC" "diet2.xls":
  [i ~ NUTR], {j in FOOD} <amt[i,j] ~ (j)>;
```

From figure 7.7, there is one key column, NUTR and the rest are data columns headed by the members of the set FOOD. The key column is represented by [i ~ NUTR] and associates the first table column with the set NUTR and the index i. The *data-specs* are generated by {j in FOOD} \leftrightarrow <...> for each FOOD member. The specific *data-specs* are represented by amt[i,j] ~ (j) where amt[i,j] denotes the AMPL parameter to which the data must be read into and is subscripted with the dummy indices for the two sets it represents (NUTR and FOOD) and (j) is a string expression for the name of the data column. If there were no parentheses around (j), then it would have denoted the single character j as the column name.

When *writing* to a two-dimensional table, the following declaration is used to get a result as shown in figure 7.7

```
table AmountsOutput OUT "ODBC" "diet2.xls":
  {i in NUTR} -> [NUTR], {j in FOOD} <amt[i,j] ~ (j)>;
```


Acronyms

DSN Data Source Name. [38](#), [39](#)

GUI Graphical User Interface. [7](#)

IDE Integrated Development Environment. [5](#)

JRE Java Runtime Environment. [4](#), [5](#)

LP Linear Programming. [28](#)

ODBC Open Database Connection. [37](#)

References

- [1] David M. Gay Robert Fourer and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, 2 edition, 11 2002. <http://www.ampl.com/>.

Index

- AMPL
 - file extensions, [17](#)
 - perspective, [10](#)
 - views, [11](#)
- AMPLDev plug-in
 - installation, [4](#)
- AMPLDev plug-in (advanced users), [2](#)
- AMPLDev stand-alone, [2](#)
 - installation, [4](#)
- Eclipse
 - editors, [12](#)
 - installation, [5](#)
 - perspective, [9](#)
 - views, [10](#)
 - welcome screen, [8](#)
 - workbench, [8](#)
- Files
 - file extensions, [17](#)
 - new
 - context menu (AMPL perspective), [19](#)
 - File menu, [18](#)
 - File menu (AMPL perspective), [19](#)
 - toolbar commands (AMPL perspective), [19](#)
- Launch
 - default launch, [24](#)
 - File Selection tab, [23](#)
 - multiple file, [22](#)
 - single file, [20](#)
 - context menu, [21](#)
 - launch toolbar, [22](#)
- Projects, [15](#)
 - new, [15](#)
 - context menu (AMPL perspective), [17](#)
 - File menu, [15](#)
 - File menu (AMPL perspective), [16](#)
 - toolbar commands (AMPL perspective), [17](#)
- Views
 - Console, [11](#), [24](#)
 - Outline, [12](#)
 - Project Explorer, [11](#)
 - Solution, [12](#), [25](#)
- Wizard, [15](#)