OptiRisk Systems

AMPL Studio User Manual

Last Update
24 April 2008

OptiRisk SYSTEMS

INVESTOR IN PEOPLE
BS EN ISO 9000 : 2000

OptiRisk Systems

Using AMPL Studio

Contents

Chapter 1: Acknowledgements of Contributions

AMPL Studio and AMPL components have been designed and developed by Dr Mustapha Sadki and are the property of Datumatic Ltd UK.

AMPL Studio and AMPL components have been produced through a business partnership between Datumatic Ltd and UNICOM Consultants, trading as OptiRisk Systems, who are the distributors for AMPL Studio.

We would like to thank Dr Patrick Valente who has worked closely with Dr Mustapha Sadki to design and implement the Stochastic Extensions of AMPL known as SAMPL, which is embedded within AMPL Studio.

We similarly would like to acknowledge Professor Robert Fourer of Northwestern University and Dr David Gay, formerly of Lucent Technologies for their invaluable advice and comments in the realisation of AMPL Studio.

We thank Mr Frank Ellison who is the principal architect of FortMP; he has implemented AMPL driver for FortMP. We acknowledge the help of Dr Bob Bixby, Dr Irv Lusting and Mr Marc Marshall of ILOG for making the business arrangement which enables us to resell CPLEX with AMPL and AMPL Studio.

We extend our thanks to Professor Antonio Alonso Ayuso of the University of Rey Juan Carlos Madrid and Dr Cormac Lucas of Brunel University, for their extensive testing of the system and valuable feedback.

Other Acknowledgements: --

- The Computational Optimisation and Modelling Group is now part of CARISMA: The Centre for the Analysis of Risk and Optimisation Modelling Applications, Brunel University, London (UK).
- AMPL Studio is a trademark of Datumatic Ltd (UK).
- AMPL is a trademark of AMPL Optimization LLC (USA).
- FortMP ™, FortSP™ are trademarks of UNICOM Consultants, trading as OptiRisk Systems.
- CPLEX™ is a trademark of ILOG Inc.

Dr Gautam Mitra, Dr Mustapha Sadki, Dr Kula Kularajan, and Dr Belen Dominguez Ballesteros.

January 2005

Chapter 2: Scope and Purpose

The Scope

This document is designed to serve both as a user guide and as a reference manual.

We assume the user of AMPL Studio has a basic understanding of Linear Programming (LP) and some experience of using AMPL, which is connected to an appropriate solver, such as FortMP, CPLEX or MINOS. In this manual, we first introduce basic concepts of using a graphical user interface (GUI); the GUI incorporates the 'look and feel' as well as a conceptual structure, which closely resembles Microsoft's approach to a 'studio' environment.

The Purpose

The purpose of this manual is to introduce this modelling studio environment to an end user, analyst who can create, maintain and revise AMPL models within the studio environment.

This manual does not provide an introduction to LP modelling. For an introduction to LP modelling, the reader is referred to

(a) …..CARISMA and OptiRisk Systems lecture notes.
(b) …..Text Books by Gautam Mitra (GM), Paul Williams (PW) and Linus Schrage (LS).
(c) AMPL: A Modeling Language for Mathematical Programming prepared by Robert Fouer (Northwestern University), David M Gay (AMPL Optimization LLC), Brian W Kernighan (Princeton University), THOMSONS, BOOKS, COLE, USA.
(d) Stochastic Programming Lecture Notes, Copyright. CARISMA and OptiRisk Systems.

Chapter 3: Directed Reading

The user of AMPL Studio first needs to study the installation and licensing procedure, which is explained in Appendix A. Chapter 4 contains an overview of AMPL Studio; the essential explanation of the main window containing menu bars, tool bars, also workspaces including file view, model view, edit area, status bar are explained. The basic aspects of navigating around and the method of working within the AMPL studio are explained.

A simple outline and explanation of the AMPL modelling language is given in Chapter 5.

In Chapter 6, a step-by-step work through tutorial is provided and the concepts of Workspace, Projects, Model File, simple data connection solution and display of results are illustrated. Chapter 7 explains the connectivity with data and databases; input and output of scalar data items and data table are explained. Chapter 8 describes the advanced features of AMPL language. Chapter 9 outlines scripts, debugging and tracing features of AMPL Studio; the step through debugging is a uniquely attractive feature of the studio.

Chapter 4: Overview of AMPL Studio

AMPL Studio Main Window

When you launch AMPL Studio, the Main window appears. All tasks and commands for using AMPL Studio are carried out from this window. Figure 4.1 shows the Main window with three opened files, **steel.dat**, **steel.mod** and **diet_solution.txt**.

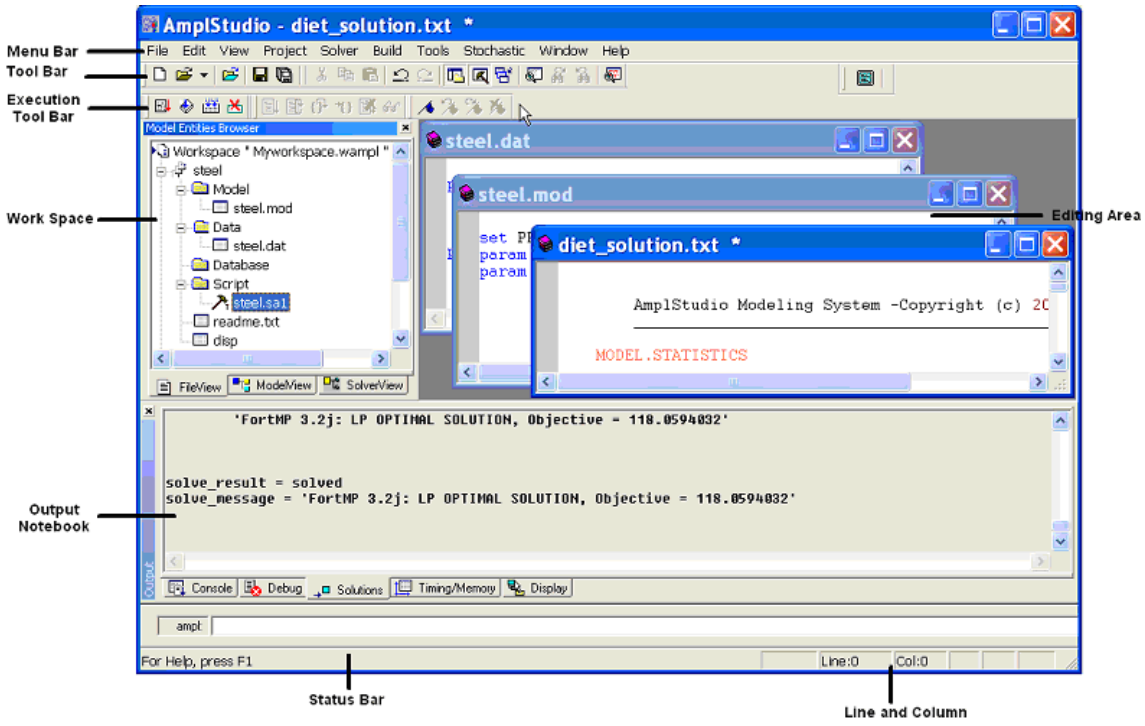


Figure 4.1: AMPL Studio Main Window

Note	If you have a mouse with a wheel between the two buttons, you can use the wheel to scroll up and down.

- **Menu Bar** provides various menu commands to choose from, such as **Save** in the **File menu**, and to display dialog boxes to perform various tasks. Certain menu commands, followed by a ► image on their right hand side, have their own sub-menu commands.

e.g.

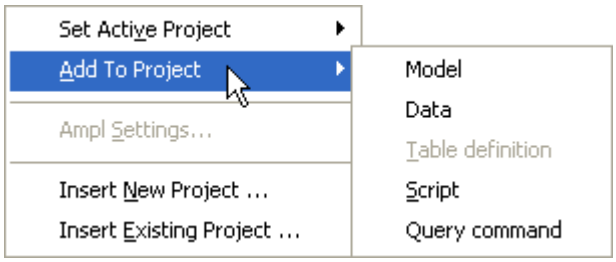


Figure 4.2: AMPL Studio Sub Menus

The **Add To Project** Command menu has five sub command menus, **Model**, **Data**, **Table Definition**, **Script** and **Query Command**.

- **Tool Bar** provides frequently used command buttons.
- **Execution Tool Bar** buttons is used for executing the Models, Projects and Scripts.
- **Work Space** contains a notebook with three pages, **FileView**, **ModelView** and **SolverView**.

  FileView displays project tree structures containing all the files related to each project. The project files are arranged under **Model**, **Data**, **Database** and **Script** containers. It also displays stand-alone models and scripts.

  ModelView displays the various model components, such as **Parameters**, **Sets**, **Variables**, **Constraints**, **Problems** and **Objectives** in separate containers for easy access. Any particular information can be displayed by clicking on it and diverse parts of the solution.

  SolverView is the sane as **ModelView**, with the difference that it displays what solver see after presolve.

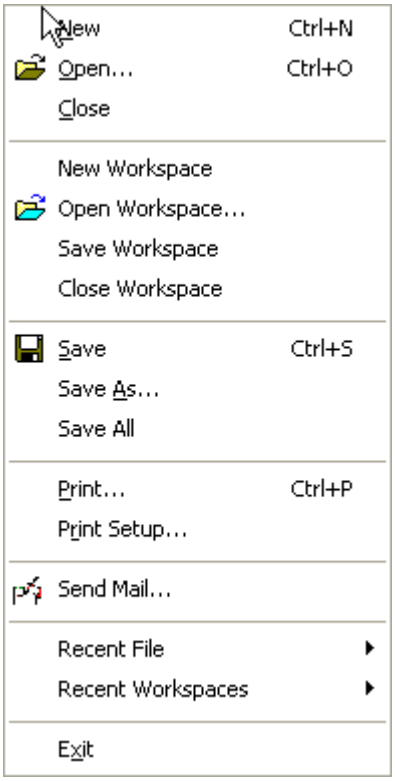
- **Editing Area** displays opened model, data, database, script, and Solution files. New files can be created and any existing files can be edited in this area. You can open more than one file in this space. The opened files are displayed in separate panels with the file name appearing in the title bar.
- **Output Notebook** has five tabs to display AMPL **Console** Messages, **Debug** Information, **Solution** Files, **Timing and Memory** Information, and **Display** all other information. By default AMPL studio will display the most appropriate window for the user action, but the user can switch to another window by clicking on the tab at the bottom of the Output Notebook.

- **Status Bar** displays messages concerning the execution status of AMPL Studio.
- **Line and Column** displays line and column number of the cursor location in the active document in the editing area.

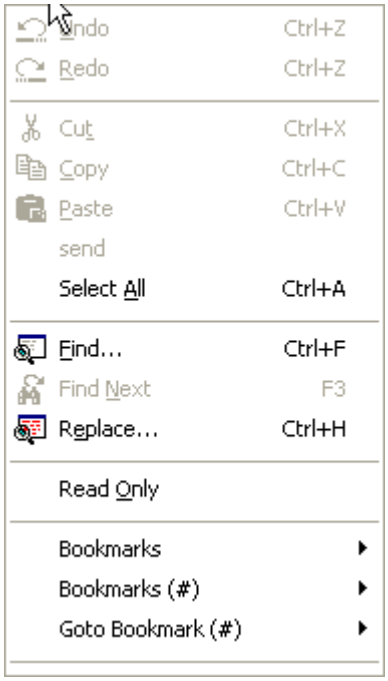
Note	In the graphic interface **Menu Bar**, **Tool Bar**, **Execution Tool Bar**, **Work Space and Output Area** are dockable: A dockable element can be detached from, or floated in its own frame window or it can be attached to, or docked at any side of its parent window.

Menu Bar Commands

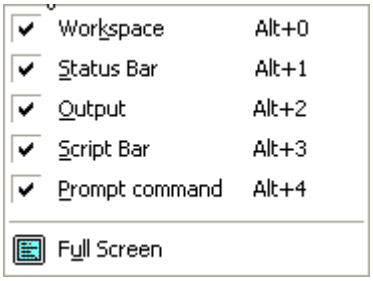
File Menu



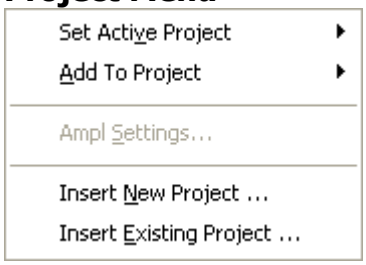
Edit Menu



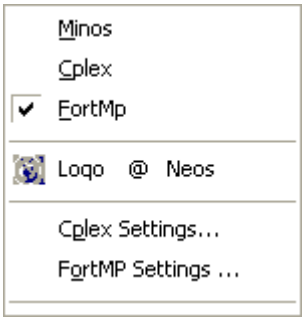
View Menu



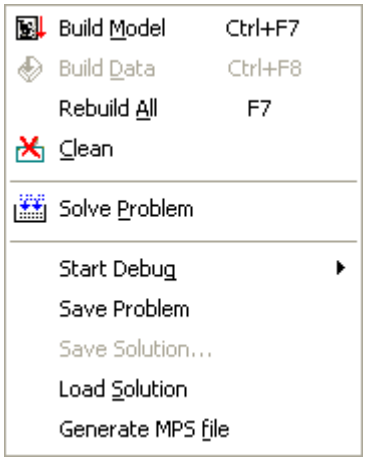
Project Menu



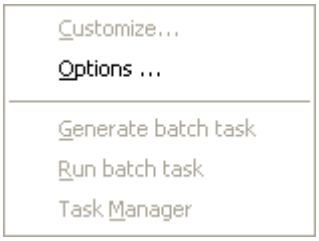
Solver Menu



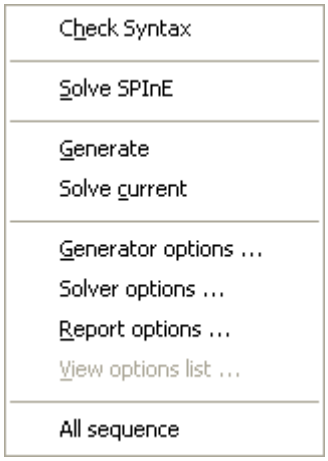
Build Menu



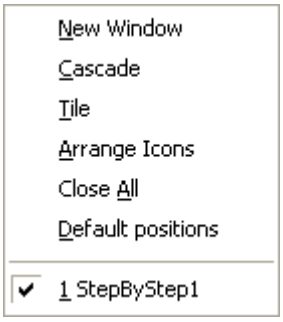
Tools Menu



Stochastic Menu



Window Menu



Help Menu

Figure 4.3: Overview of Commands in the Menu Bar

Some of the menu items have a keyboard shortcut, indicated on the right-hand column of the menu. For example, Keyword Save has the shortcut Ctrl + S, which means that you can save the active document by clicking the Ctrl key and the S key at the same time.

The following tables (Table 4.1 - Table 4.10) list the command found in the menus and provide a description of each command.

Command	Description
New	Creates a new file.
Open	Opens a file.
Close	Closes an opened document.
New Workspace	Creates a new workspace
Open Workspace	Opens an existing workspace
Save Workspace	Saves the current workspace
Close Workspace	Closes the current workspace
Save	Saves the current edited file.
Save As	Saves the current edited file with a new name.
Save All	Saves all the open files.
Print	Prints a document.
Print Setup	Selects a printer and printer connection.
Send Mail	Sends the active document through electronic mail.
Recent File	Displays a list of previously opened documents.
Recent Workspaces	Displays a list of previously opened workspaces
Exit	Exits AMPL Studio.

Table 4.1: File Menu Command descriptions

Command	Description
Undo	Undoes an unlimited number of nested actions in the current editor.
Redo	Redoes previously undone actions in the current editor (unlimited).
Cut	Deletes the selected text from the editor and puts it in the clipboard.
Copy	Copies the selected text, from the editor or output window, to the clipboard.
Paste	Pastes from the clipboard to the current editor.
Send	Email the opened file.
Select All	Selects the entire content of the current editor.
Find	Displays the Find dialog box for specifying search criteria.
Find Next	Finds the next occurrence of the text displayed in the Find box.
Find Previous	Finds the previous occurrence of the text displayed in the Find box.
Replace	Displays the Replace dialog box for specifying search criteria and replacing specified strings.
Read Only	Set the active document as read only file.
Bookmarks	Bookmark a script line.
Bookmarks (#)	Bookmark a line with the number.
Goto Bookmark (#)	Go to the bookmark number.

Table 4.2: Edit Menu Command descriptions

Command	Description
Status Bar	Displays the Status Bar.
Workspace	Displays the Workspace.
Output	Displays the Output Notebook.
Script Bar	Display the Script Execution Toolbar.
Full Screen	Displays the active file in full screen mode.

Prompt Command	AMPL command line

Table 4.3: View Menu Command descriptions

Command	Description
Set Active Project	When several projects are open, remembers the project selected in the Project Tree as the active one.
Add To Project	To insert a model, data, database or script files to the project.
Ampl Settings	To change the AMPL settings.
Insert New Project	To insert a new project into the opened workspace.
Insert Existing Project	To insert an existing project into the opened workspace.

Table 4.4: Project Menu Command descriptions

Command	Description
Minos	To select the Minos Solver as a default Solver.
CPLEX	To select the CPLEX Solver as a default Solver.
FortMP	To select the FortMP Solver as a default Solver.
CPLEX Settings	To Change the CPLEX Solver settings.
FortMP Settings	To Change the FortMP Solver settings.

Table 4.5: Solver Menu Command descriptions

Command	Description
Build Model	To build the model.
Build Data	To build the data
Rebuild All	To build the all models and data
Clean	To clean all read information from the memory.
Solve Problem	Solve the read problem.
Start Debug	Start to debug the script.
Save Problem	Save the current problem.
Save Solution	Save the solution.
Load Solution	Load the solution.

Table 4.6: Build Menu Command descriptions

Command	Description
Customize	[Future Functionality]
Options	Displays the Default Options dialog box that allows changing the AMPL Studio options.

Table 4.7: Tool Menu Command descriptions

Command	Description
Check Syntax	This command performs the syntax check of a model written using SAMPL' s extended AMPL keywords for stochastic programming.
Solve SPInE	The current model is parsed, and then solved using SAMPL' s solver. The solver settings, including the solution types, can be modified using the *Solver options…* command.
Generate	An SMPS representation of the current model instance is generated using this command. By default, SAMPL/SPInE generates Windows/DOS text files. This may not compatible with other UNIX based solvers. The advanced option *UnixOutput* described in the SP Generator options (SPG) section enables the user to change the output text format to UNIX.
Solve Current	This command solves the latest SMPS instance generated for the current model. If such instance is not available, then this command is equivalent to the *Solve SAMPL* command.
Generate Options	This command displays the Generator Options dialog box. Settings for the generator of SMPS instances can be modified using this command.
Solver Options	This command displays the Solver Options dialog box. Settings for SAMPL/SPInE' s solver can be modified using this command.
Report Options	This command displays the Reporting Options dialog box. This dialog box enables the users to change the way

	SAMPL/SPInE exports the solution vectors obtained from the solver.
View Options List	This command displays the current settings of the SAMPL/SPInE system. Advanced users can run this command in order to manually edit the advanced options provided by SAMPL/SPInE.
All Sequence	This command opens a graphic dialog box, which displays the structure of the scenario tree associated with the current model.

Table 4.8: Stochastic Menu Command descriptions

Command	Description
New Window	[Future Functionality]
Cascade	Displays overlapping panels in the editing area.
Tile	Displays panels in the editing area horizontally.
Arrange Icons	To arrange icons.
Close All	Closes all the windows in the Editing Area

Table 4.9: Window Menu Command descriptions

Command	Description
Help Topics	To view AMPL Studio Help Topics
Online	Opens the AMPL Online help window.
www.ampl.com	Go to AMPL web site
About Ampl Studio	Indicates the version of AMPL Studio, the OptiRisk-Systems products used by AMPL Studio, and contains copyright information.

Table 4.10: Help Menu Command descriptions

Tool Bar Buttons

The following buttons appear in the tool bar:

Button	Description
	To create a new blank document
	To open an existing document. AMPL Studio displays an Open File dialog box requesting the file name you wish to open. The

	file is then displayed in the editing area.
	To open an existing Workspace. AMPL Studio displays an Open File dialog box requesting the workspace you wish to open. The workspace and their related projects and files will be displayed in the workspace window.
	To save the active document in editing area.
	To save all the modified files.
	To cut the selection and put it on the clipboard.
	To copy the selection and put it on the clipboard.
	To insert clipboard contents.
	To undo the last action.
	To redo the previously undone action.
	To show or hide the workspace window.
	To show or hide the output window.
	To manage the currently open windows.
	To find the specified text.
	To Repeat the last find text action.
	[Future Functionality]
	To replace specific text with different text.
	To display the active file in full screen mode.

Table 4.11: Toolbar Buttons and Descriptions

Execution and Debugging Tool Bar Buttons

The following buttons appear in the tool bar:

Button	Description
	To build a model.
	To build a data.

	To solve a problem.
	To reset the project.
	To run script.
	Go.
	To step out of a loop in a script and avoid going through all the iterations.
	To go to the next solution of the model or project, or to the next instruction in stepping mode, or to the next choice point in 'stop at choice point' mode.
	Continue running the script without stepping.
	Watch variable
	To set breakpoints/Marker in the AMPL model or script file.
	To go to the previous breakpoint/marker
	To go to the next breakpoint/marker
	Clear all breakpoints/Markers markers

Table 4.12: Execution Toolbar Buttons and Descriptions

Workspace

AMPL Studio Workspace is divided into three sub windows, **FileView**, **ModelView,** and **SolverView**. The user can switch between these windows by clicking on the required tab at the bottom of the Workspace.

The **Fileview** displays the Workspace Files in the Tree structure.

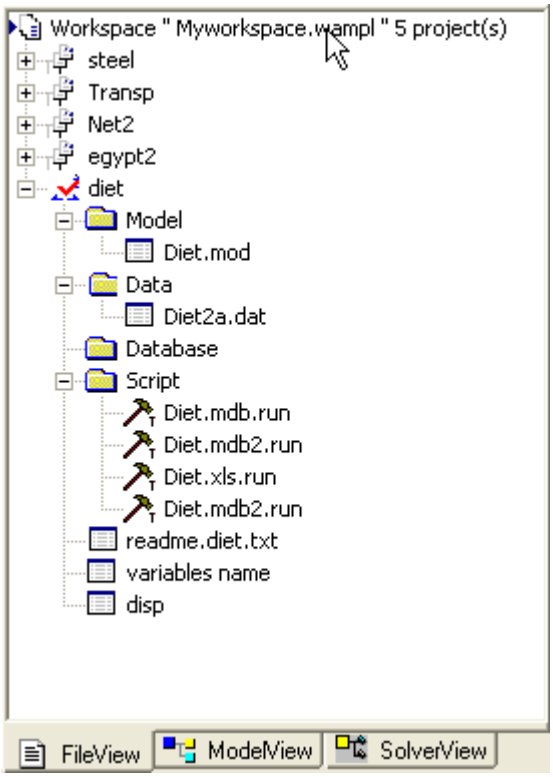


Figure 4.4: Workspace File View

The **ModelView** displays the Model **Parameters**, **Sets**, **Variables**, **Constraints**, **Problems** and **Objectives** information, which becomes available after the models and their associated data files are built.

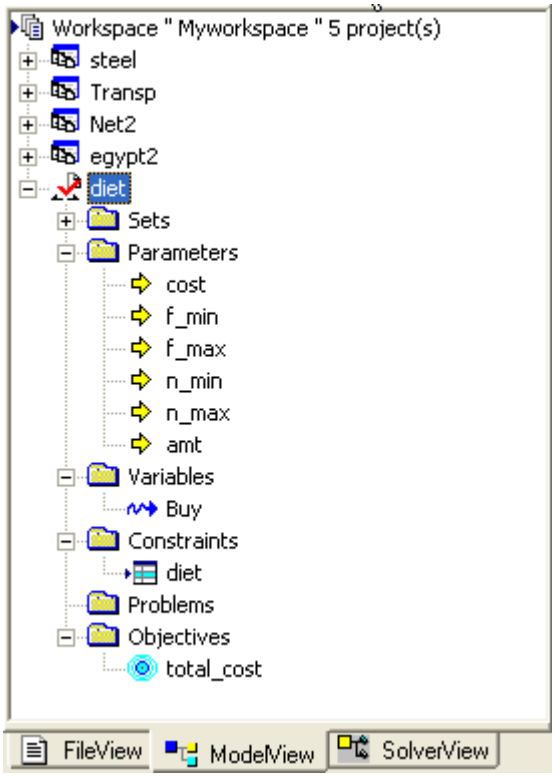


Figure 4.5: Workspace Model View

The **SolverView** displays Solved Model information, which becomes available after the model is solved.

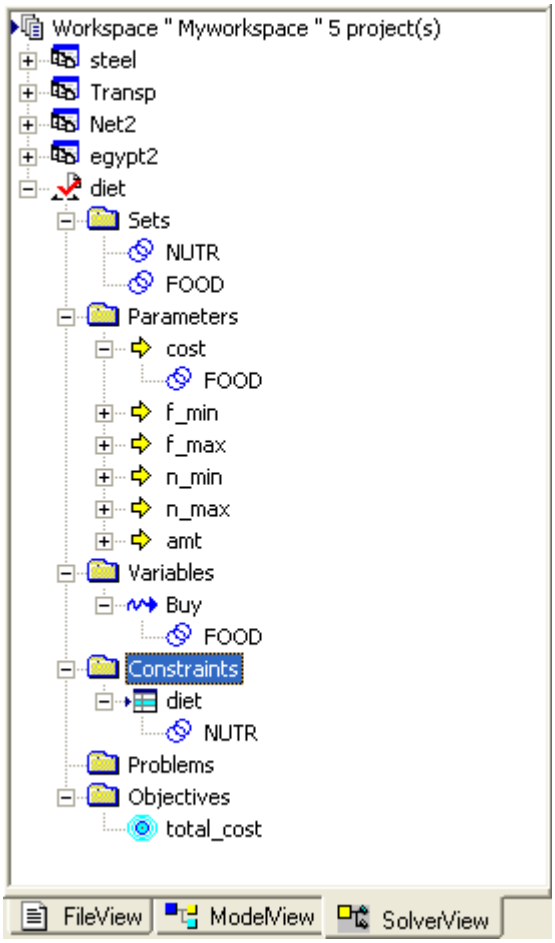


Figure 4.6: Workspace Solver View

Editing Area

AMPL Studio allows the user to open many files into the editing area. One can edit existing files or create new files in the Editable Area using the AMPL Studio's text editor. The user can edit multiple files by switching between the Editor Windows. Also the user can Resize, Minimise, Maximise and close any window.

Console

AMPL studio outputs are divided into Console, Debug, Solution, Timing and Memory, and Display windows. AMPL Studio automatically displays the most appropriate window for the user action. The user can switch between these windows by clicking on the required tab at the bottom of the Output Notebook.

The AMPL console output is displayed in the **Console Window**.

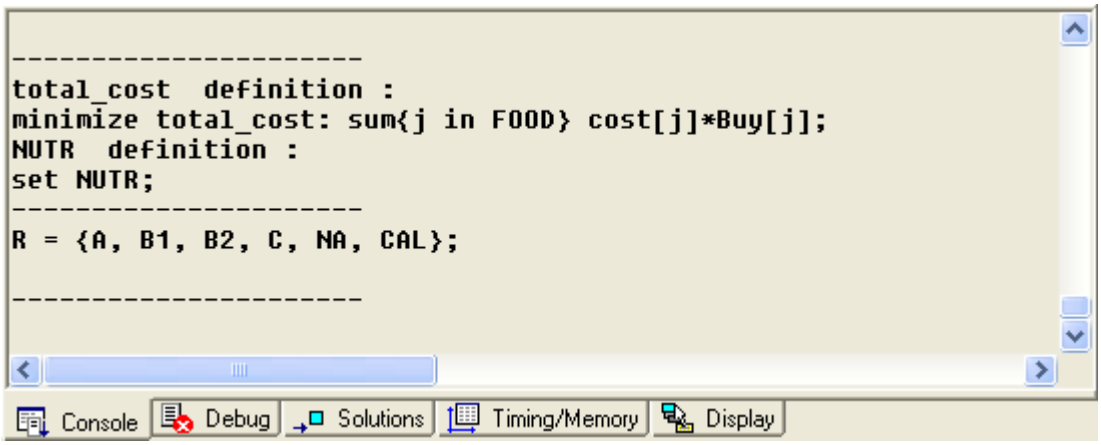


Figure 4.7: Output Console Window

The Debug results are displayed in the **Debug Window**.

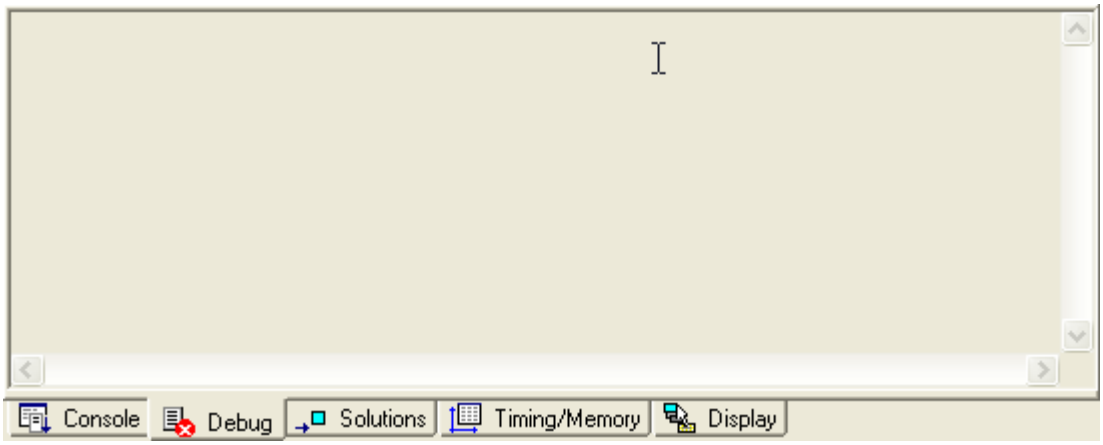


Figure 4.8: Output Debug Window

The Solution files generated by the solvers will be displayed in the **Solution Window**.

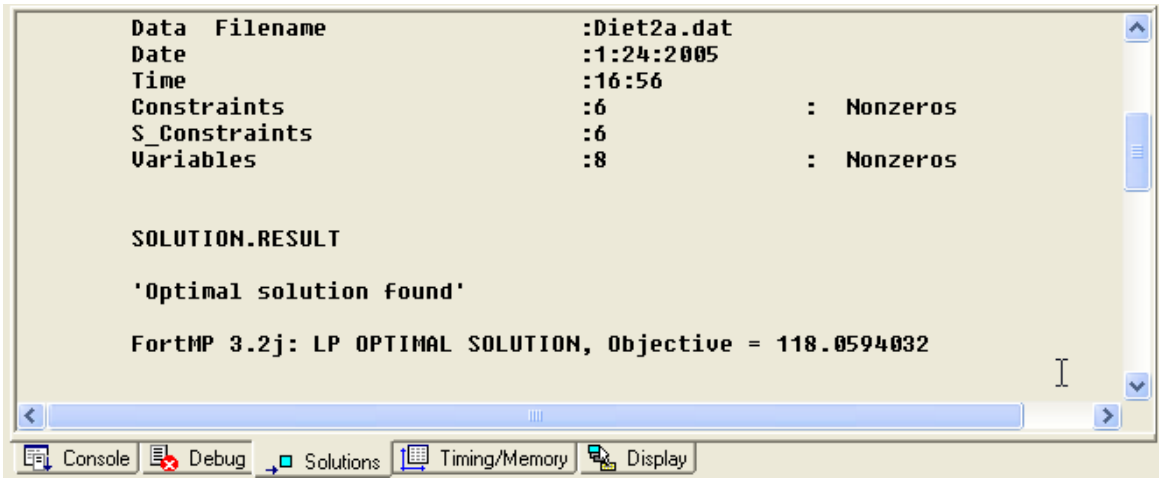
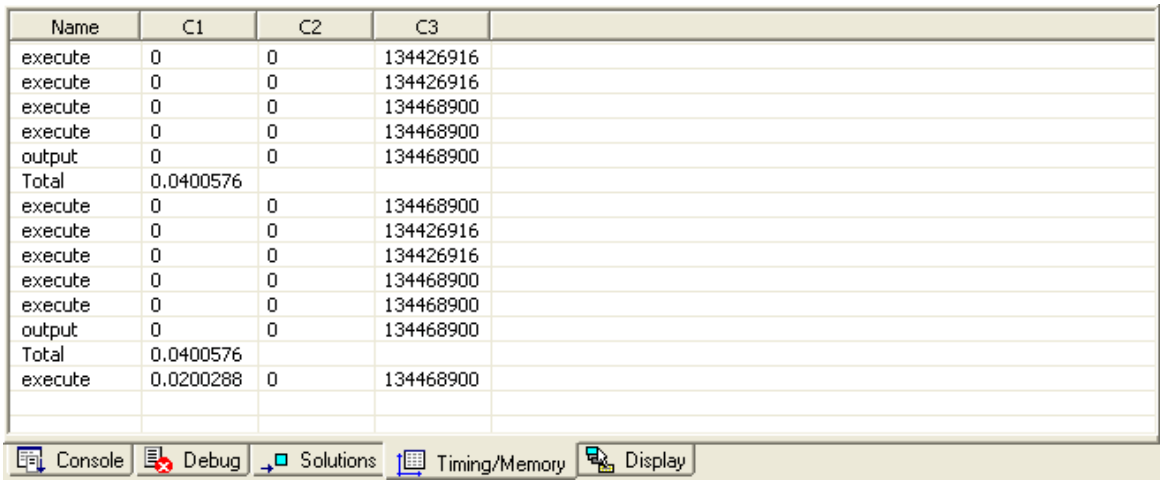


Figure 4.9: Output Solutions Window

The processing Time and Memory usage of AMPL studio are displayed in the **Timing/Memory Window**.



Name	C1	C2	C3
execute	0	0	134426916
execute	0	0	134426916
execute	0	0	134468900
execute	0	0	134468900
output	0	0	134468900
Total	0.0400576		
execute	0	0	134468900
execute	0	0	134426916
execute	0	0	134426916
execute	0	0	134468900
execute	0	0	134468900
output	0	0	134468900
Total	0.0400576		
execute	0.0200288	0	134468900

Figure 4.10: Output Timing and Memory Window

All other AMPL Studio output will be displayed in the **Display Window**.

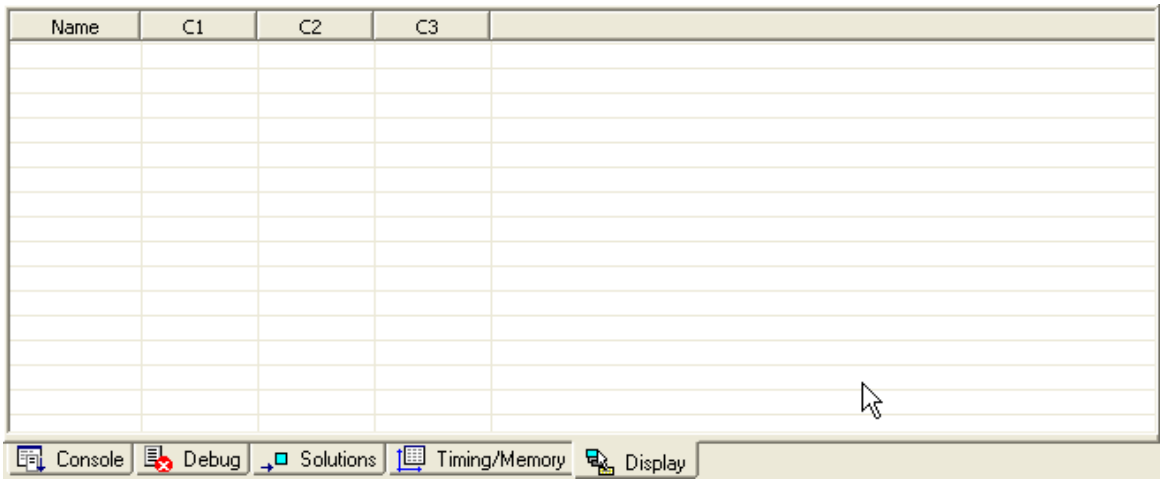


Figure 4.11: Output Display Window

AMPL Studio – Basics

This section describes several basic concepts to consider when using AMPL Studio.

File Types

- **Models**
  Model files contain AMPL statements. A stand-alone model is a model that can be executed in AMPL Studio without any additional requirements.

- **Data files**
  Large problems are better organized by separating the model of the problem from the instance data. The instance data is stored in a data file (or in several data files).

- **Projects**
  AMPL Studio uses the concept of a project to associate a model file with a number of data files. The model file declares the data but does not initialise it. The data files contain the initialisation of each data item declared in the model. The project file then organizes all the related model and data files. A project provides a convenient way to maintain the relationship between related files and runtime options for the environment.

- **Scripts**
  Script files contain AMPL Script, a script language for AMPL. A script handles different models with their data. The models and data files are associated in the script itself.

The following naming conventions are used to indicate these different files:

File Extension	Description
.mod	Used for files containing models.
.dat	Used for files containing data instances.
.sa1 or .run	Used for scripts written in AMPL Script.
.ini	Used for project files.
.wampl	Used for Workspace files

Table 4.12: File extensions and descriptions

In this Chapter and in Chapter 6 we will see how to create project files, associate model and data files with the project, and then find the solution to the problem using the project file.

Working in AMPL Studio

The model and data files used in the examples in this manual are distributed with the product. This way the reader will not have to create these files from scratch, but just open them once AMPL Studio is launched.

Opening an Existing Workspace

To open existing workspaces do the following

Step 1: Choose the **Open Workspace** from the **File** Menu.

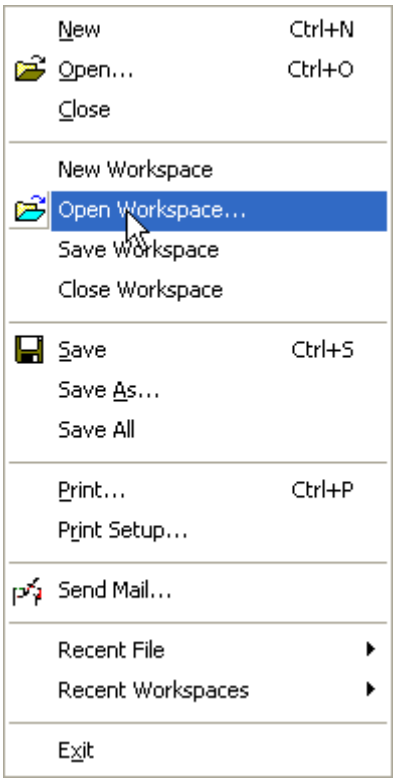


Figure 4.13: Open Workspace from the File Menu

Step 2: AMPL Studio then displays a standard Open File dialog box in order to select the file that corresponds to the workspace we want to open.

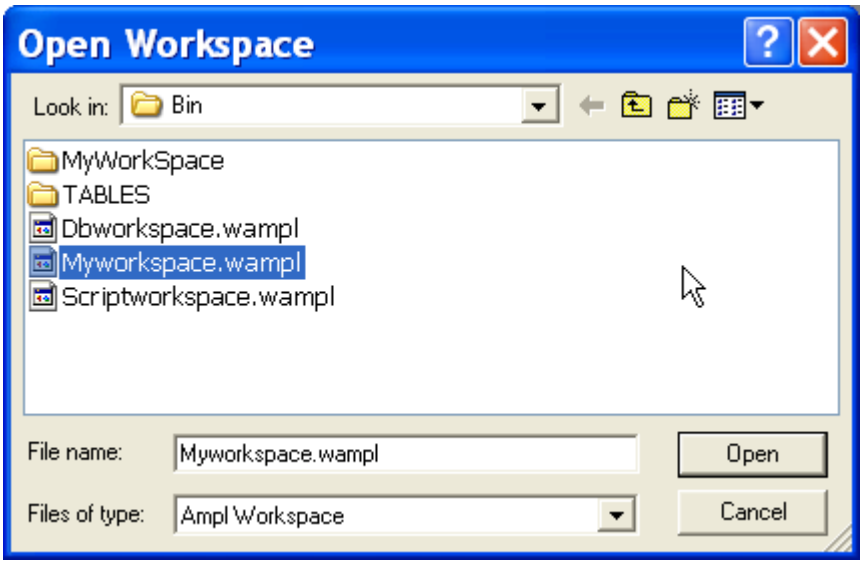


Figure 4.14: Choosing AMPL Workspace File

Select from the directory: **AMPL Studio Installed Directory/bin** and choose the workspace name **Myworkspace.wampl** and click on the **Open** button.

Note	If you have recently used the workspace, you can alternatively select it from the Recent Files submenu.

The AMPL studio will open the workspace and displays it in the workspace window as shown below.

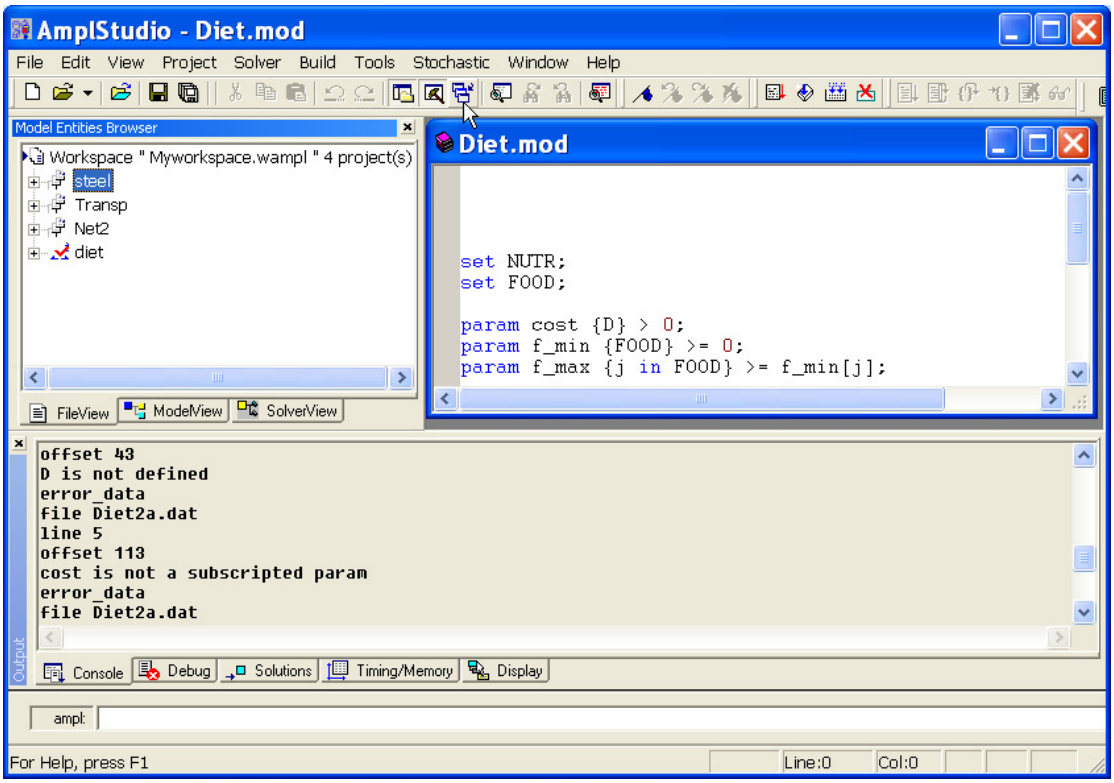


Figure 4.15: Opened Workspace in AMPL Studio

Creating a New Workspace

To create new workspaces do the following

Step 1: Choose the **New Workspace** from the **File** Menu.

File Menu

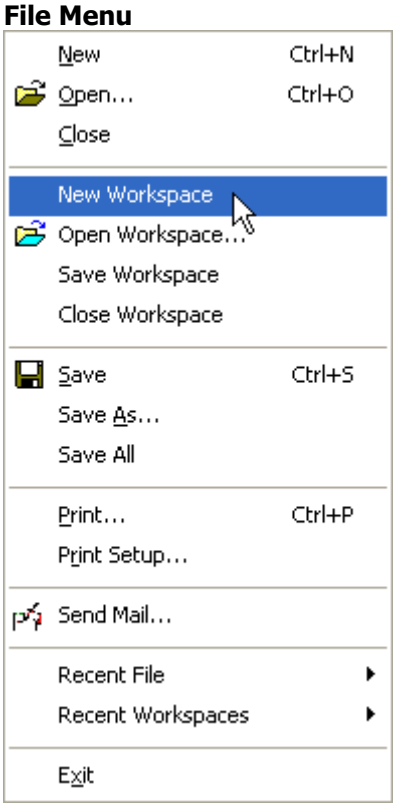


Figure 4.16: New Workspace from the File Menu

Step 2: AMPL Studio then displays a New Workspace dialog box in order to enter the Workspace name and the Folder where the workspace will be created.

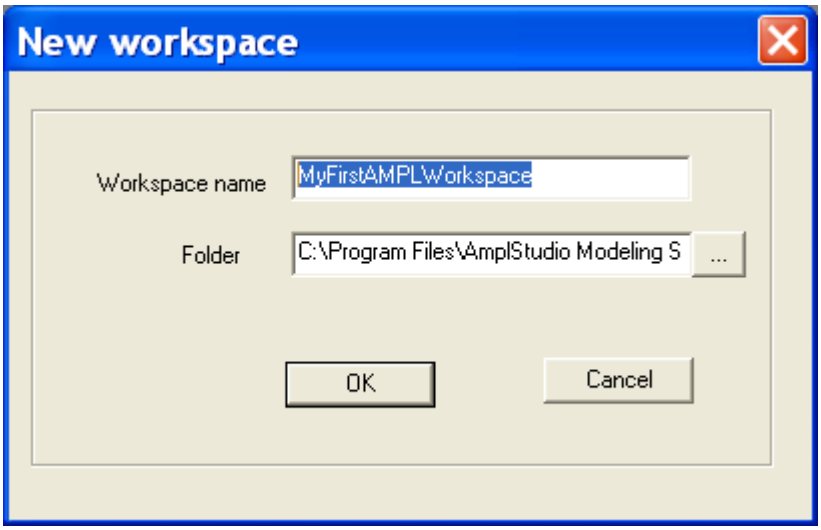


Figure 4.17: New Workspace Dialog box

Enter the workspace name as **`MyFirstAMPLWorkspace`** and choose your preferred folder by clicking on the ... button.

The AMPL studio will open the new empty workspace and display it in the workspace window as shown below.

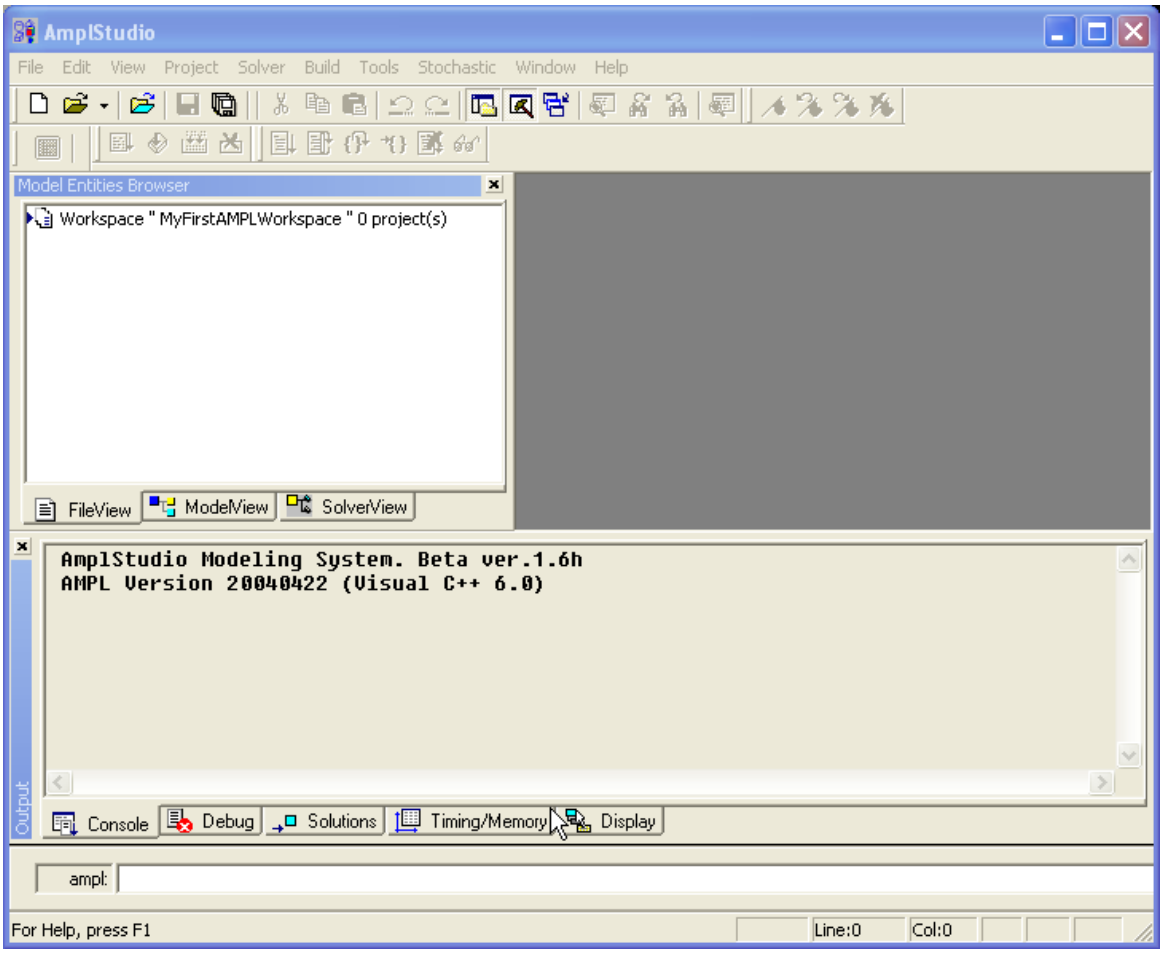


Figure 4.18: New Workspace in AMPL Studio

Inserting an Existing Project into the Workspace

To insert an existing project into the current workspace do the following

Step 1: Choose the **Insert Existing Project** Menu from the **Project** Menu.

Project Menu

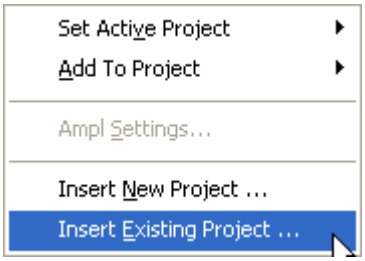


Figure 4.19: Insert Existing Project Menu from the Project Menu

Step 2: AMPL Studio then displays a standard Open File dialog box to select the file that corresponds to the project we want to open.

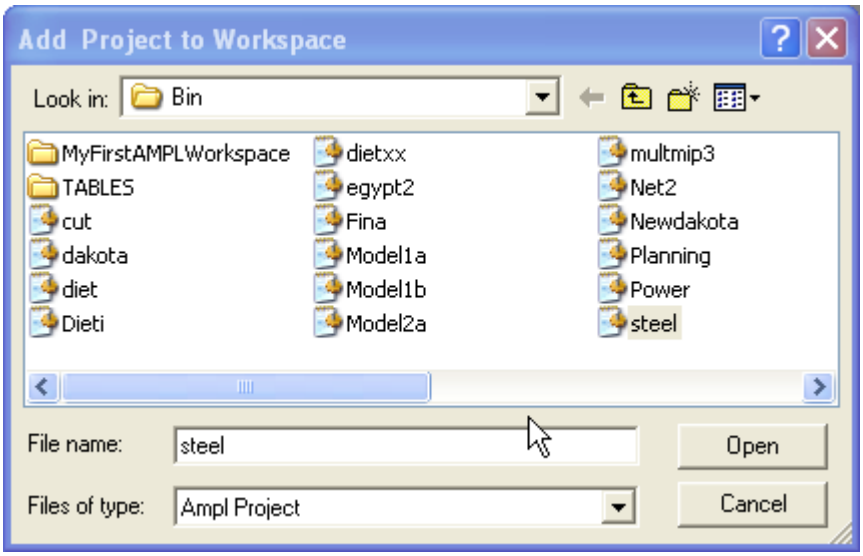


Figure 4.20: Choosing AMPL Project File

Select from the directory: **AMPL Studio Installed Directory/bin** and choose the project file **steel.ini** and click on the **Open** button.

The AMPL studio will insert the project into the workspace and display it in the workspace window as below.

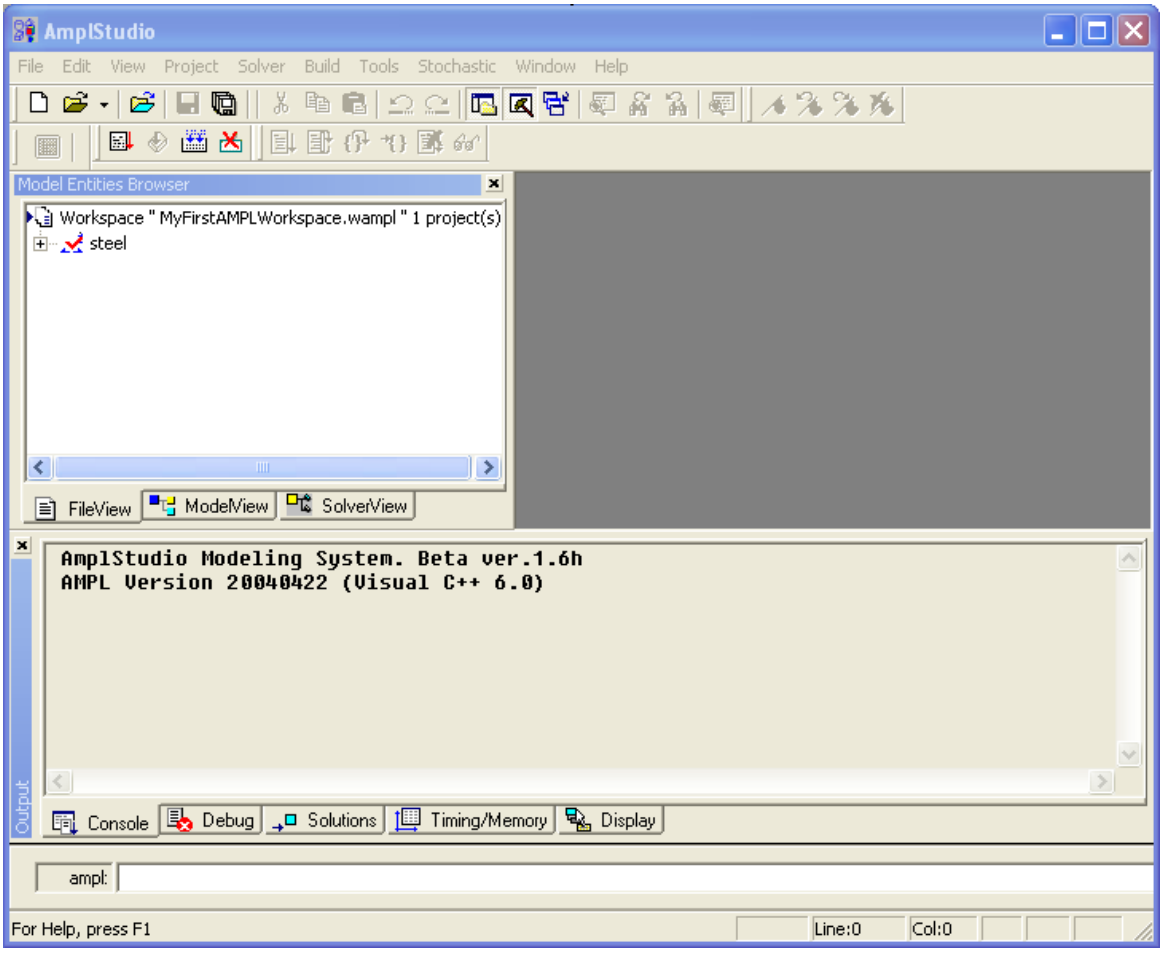


Figure 4.21: Inserted Project in the Workspace

The file can be viewed in the Editing area by clicking on the file in the **workspace**. For example, clicking on the **steel.mod** will display the **steel.mod** in the Editing Area.

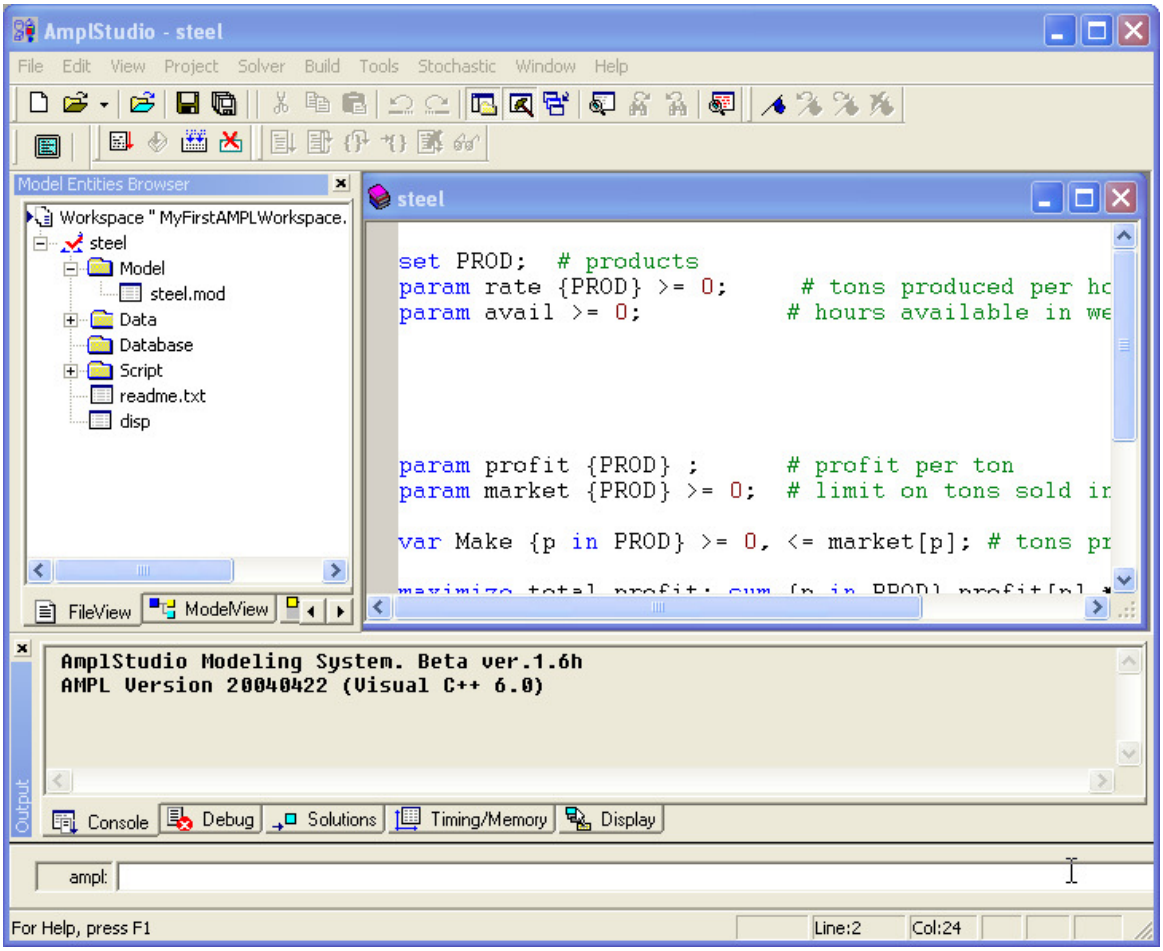


Figure 4.22: Viewing the Model file from the Inserted Project

Adding a New Project into the Workspace

To insert a new project into a workspace do the following

Step 1: Choose the **Insert New Project** Menu from the **Project** Menu.

Project Menu

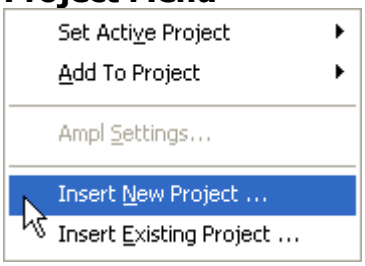


Figure 4.23: Insert New Project Menu from the Project Menu

Step 2: AMPL Studio then displays an Open New Project dialog box in order to specify the new project name and the directory where the project will be created.

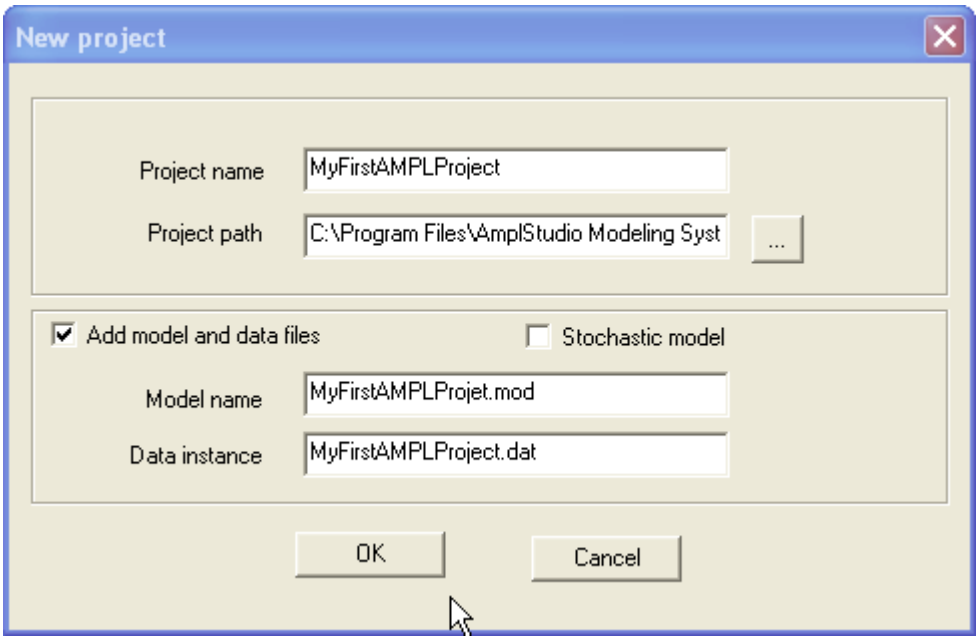


Figure 4.24: Insert New Project Dialog Box

Enter the project name as **`MyFirstAMPLProject`** and choose your preferred folder by clicking on the ... button.

Also you have the option to add the model and data template files by choosing the Add template check box. Type the model and data template files as **`MYFirstAMPLModel.mod`** and **`MyFirstAMPLData.dat`**.

Click the **OK** button to add a new project to the workspace

The AMPL studio will insert the project into the workspace and display it in the workspace window as shown below.

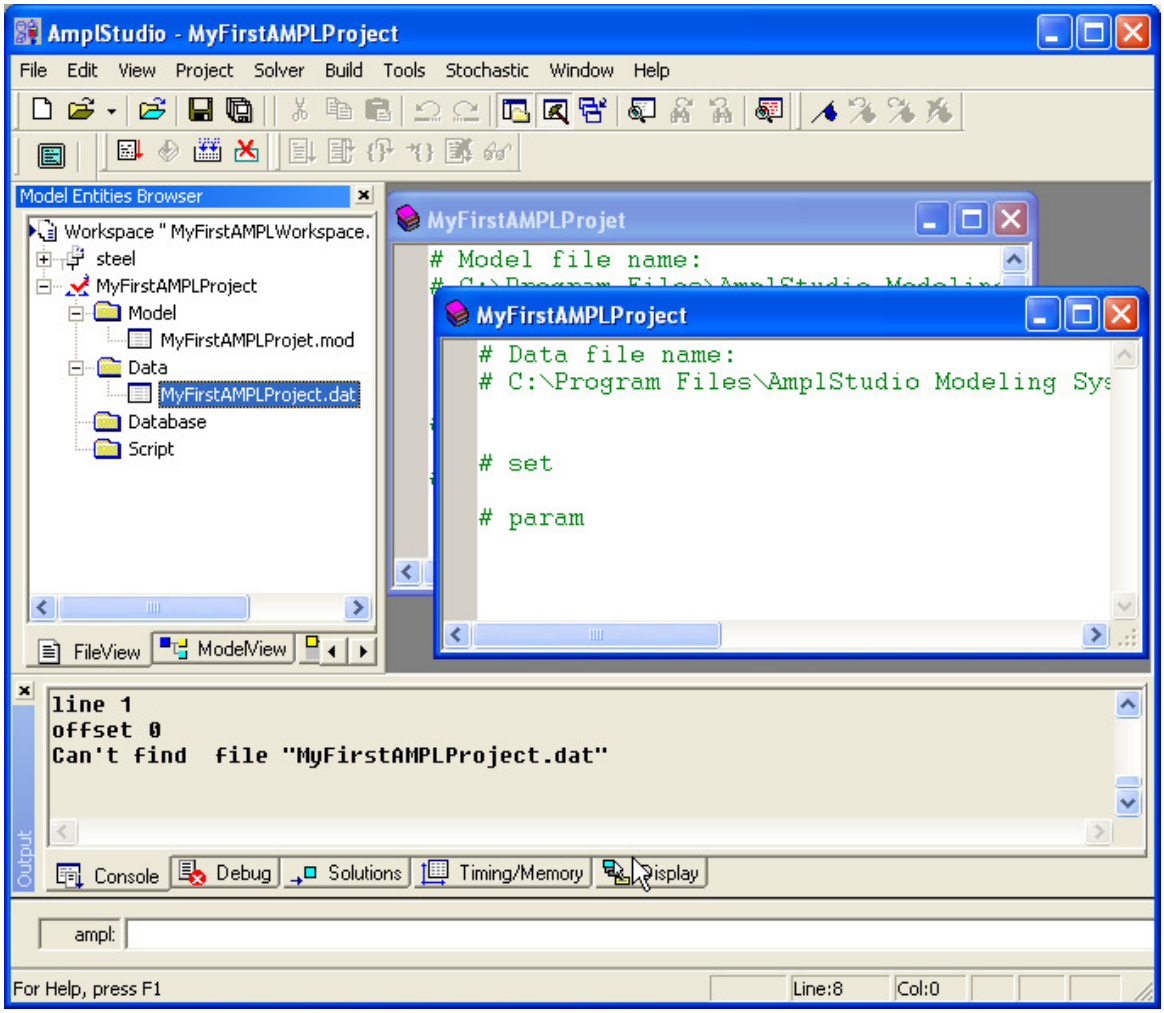


Figure 4.25: New Project View in the AMPL Studio

You can now start to write a new model and data files. Don't worry about writing the model and Data file at this stage. Chapter 5 and 6 will cover this in more detail.

Activating the Project

As you can see the steel project was active ( ) before you add your new project. When you add the new project AMPL studio assumes the new project is going to be your active project and displays it as below

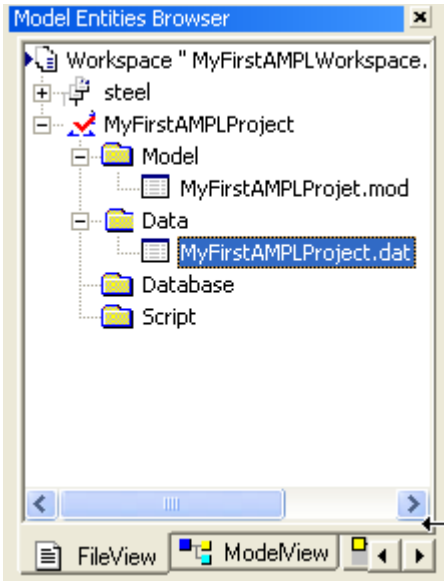


Figure 4.26: New Project Active AMPL Studio

To set the steel project back to active project, do the following.

Step 1: Click on the Steel Project Node.

Step 2: Right clicking the mouse will display the following menu.

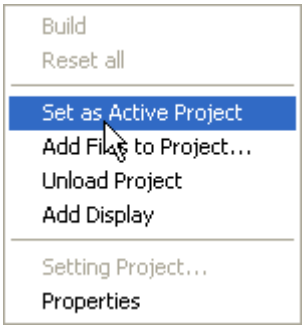


Figure 4.27: Choosing the Set as Active Project Menu

AMPL Studio will change the steel project back to active project as below.

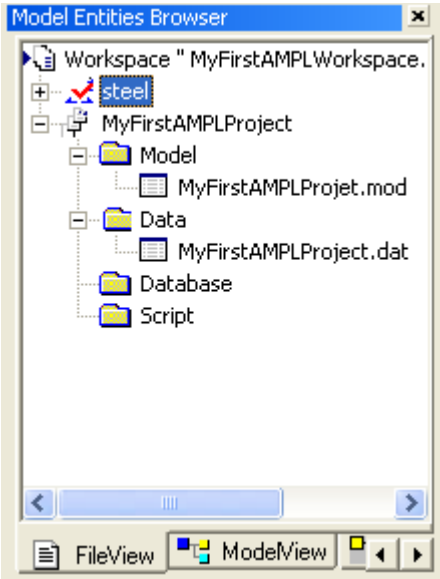


Figure 4.28: Workspace with steel project as active

Note	You can also activate the steel project by selecting the Set Active Project menu from the Project Menu

Selecting the Solver

By default AMPL studio provide three solvers, Minos, CPLEX, and FortMP. You can choose your preferred solver from one of these solvers. In order to choose FortMP as your default Solver select the **FortMP** Menu from the **Solver** Menu.

Solver Menu

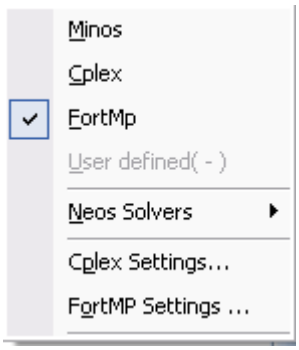


Figure 4.29: Selecting the FortMP Solver as the default solver

AMPL Studio also has the facility to use the solvers provided at the NEOS server. To use one of those solver choose the solver from Neos Solvers dropdown

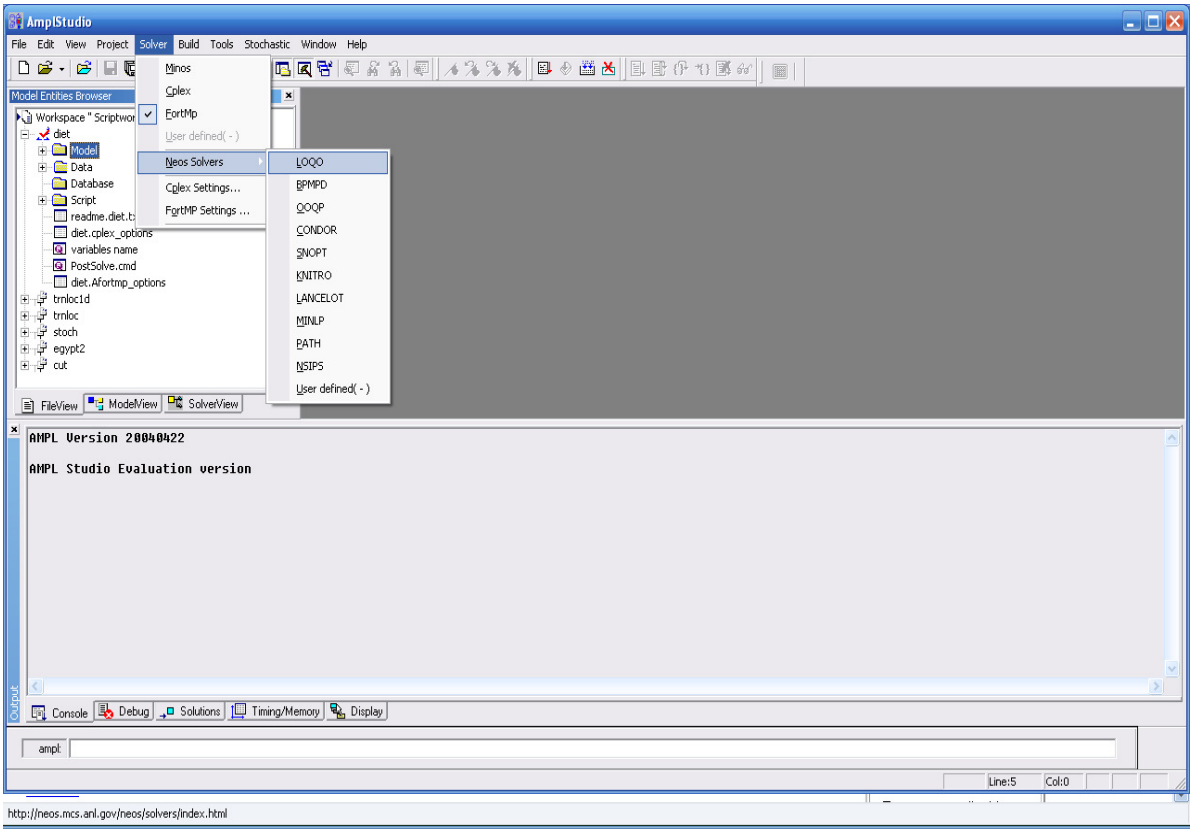


Figure 4.30: Selecting the Neos Solver

Setting the Solver Options

CPLEX and FortMP solvers have their own solver settings. You can change these setting accordingly to suite your project needs. In order to change the FortMP Solver settings choose the **FortMP Settings** menu from the **Solver** Menu.

Solver Menu

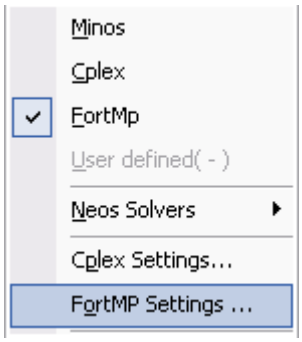


Figure 4.31: Selecting the FortMP Solver Settings

AMPL Studio then displays the following FortMP Solver setting dialog box for your selection.

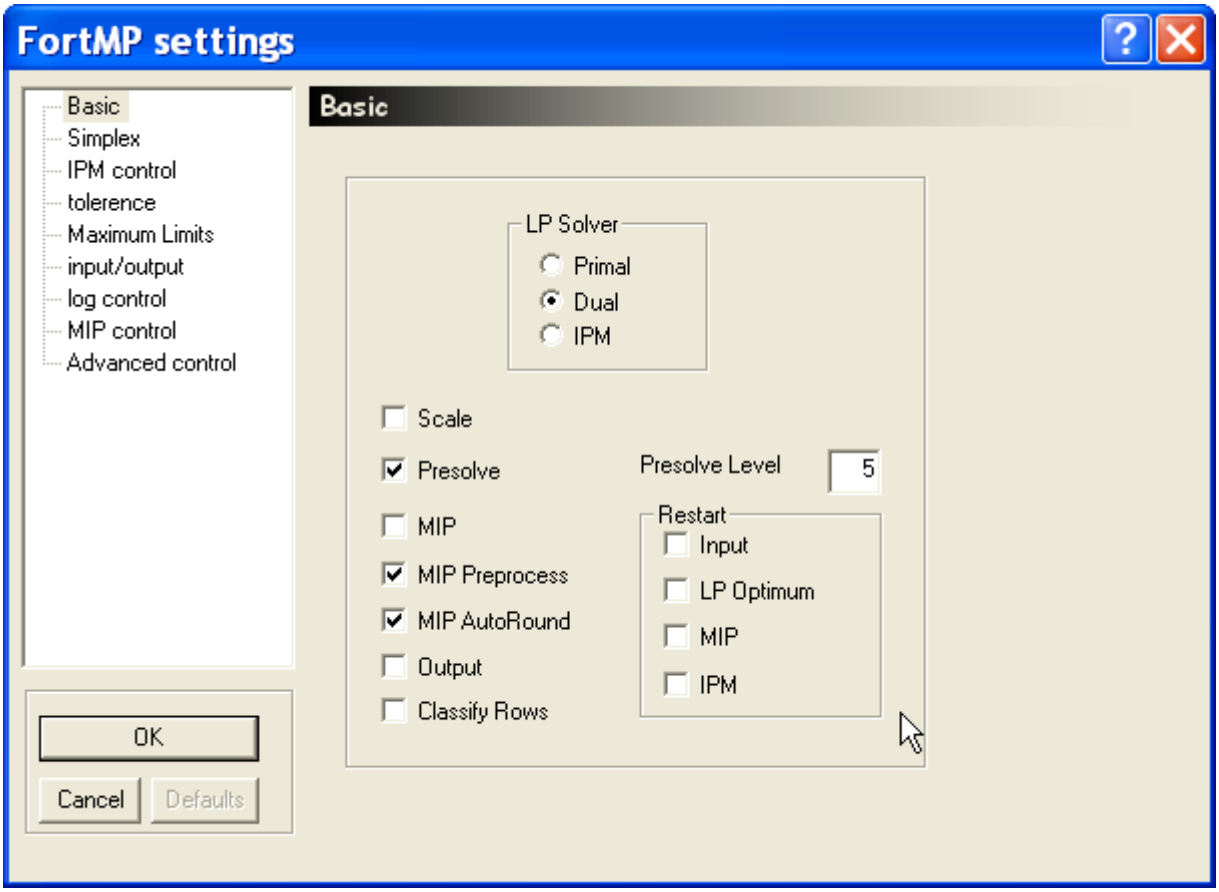


Figure 4.32: FortMP Solver Settings Dialog Box

FortMP Solver settings are divided into **Basic**, **Simplex**, **IPM Control**, **Tolerance**, **Maximum Limits**, **Input/Output**, **Log Control**, **MIP Control** and **Advanced Control**. The detail of these can be found in the FortMP Manual.

Note	CPLEX Solver setting can be done in a similar way.

Some additional options may exist for the solvers, which are not displayed in the solver settings menu. These options can be added in the solver options file. To include the solver options file, first go to Options menu and tick Insert file options in project for additional solver options. A solver options file is then included in the workspace as displayed on the left hand side of AMPL Studio.

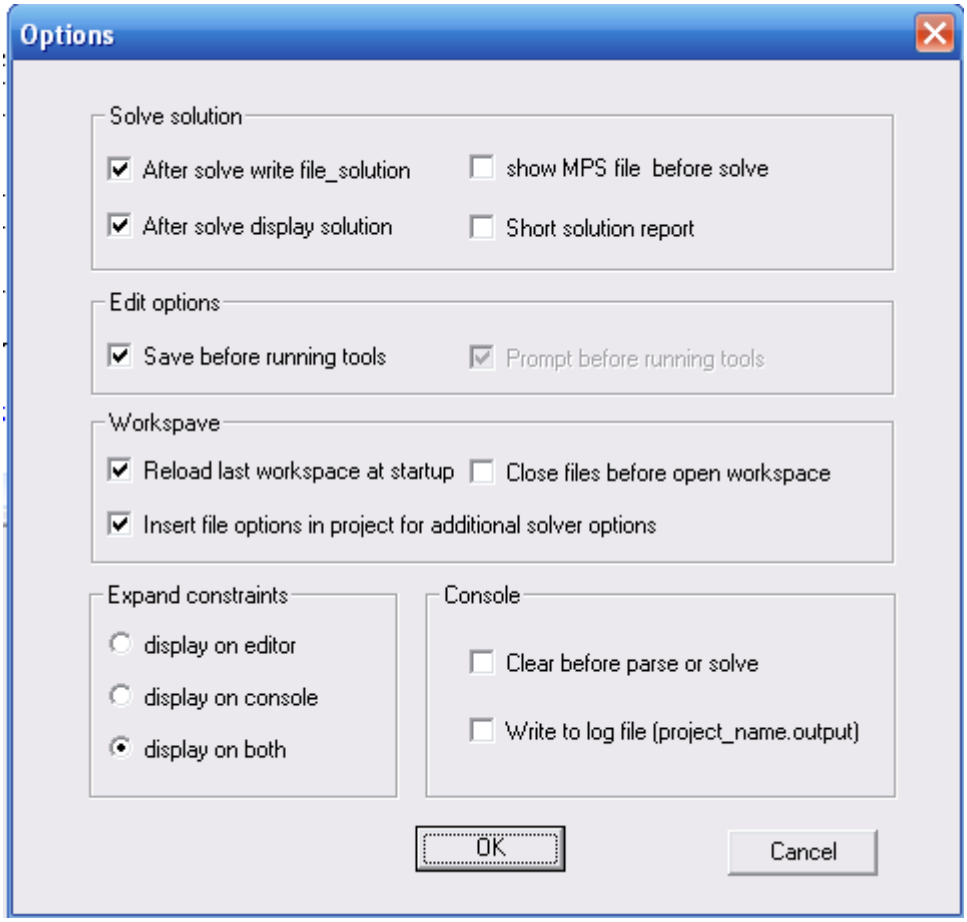


Figure 4.33: Selecting the file option for additional solver settings.

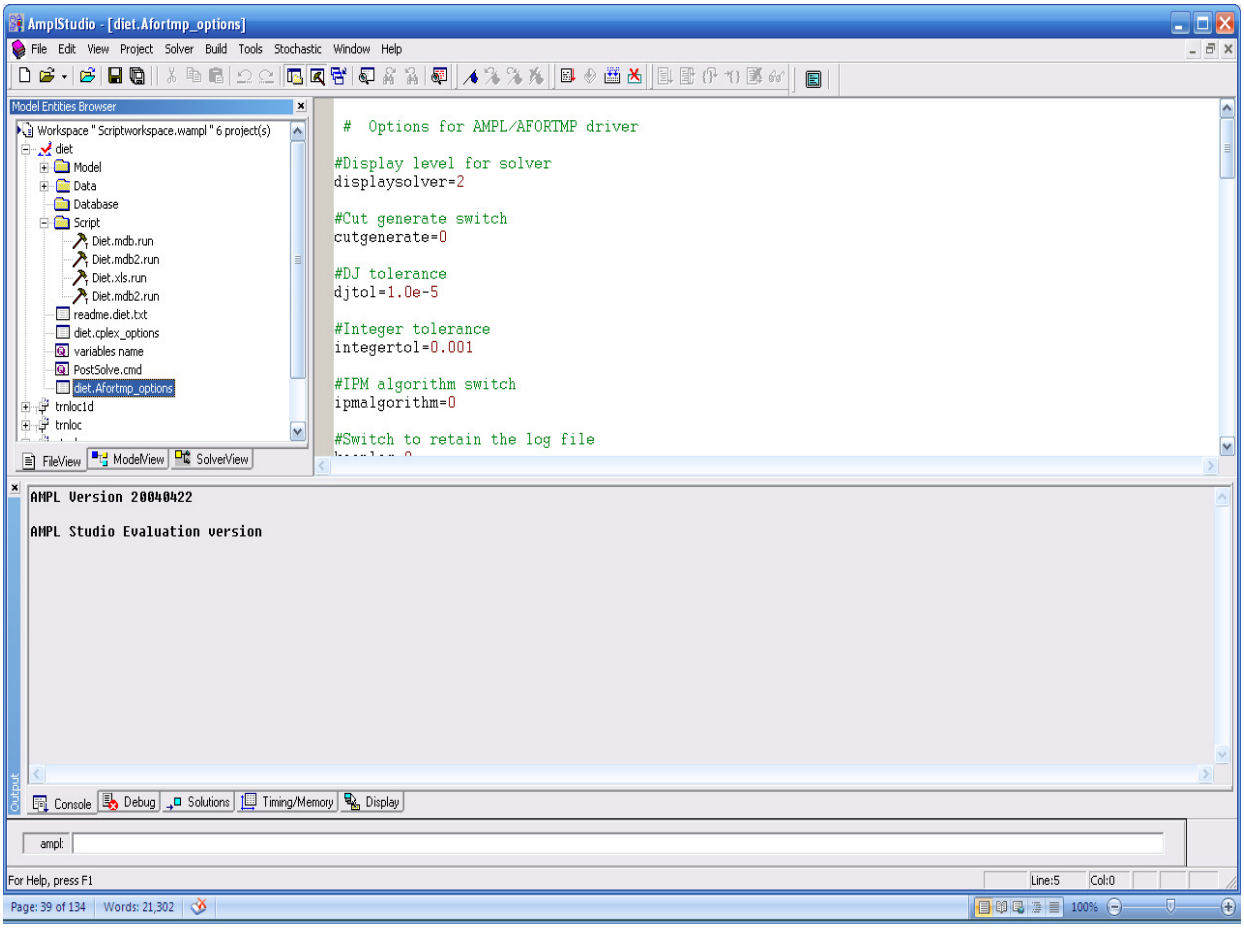


Figure 4.34: Modifying solver settings in solver options file.

Build the Model

In order to solve the problem the model and associated data files need to be built. Do the following steps to build the steel project model and data files.

Step 1: Click on the **`steel.mod`** file.

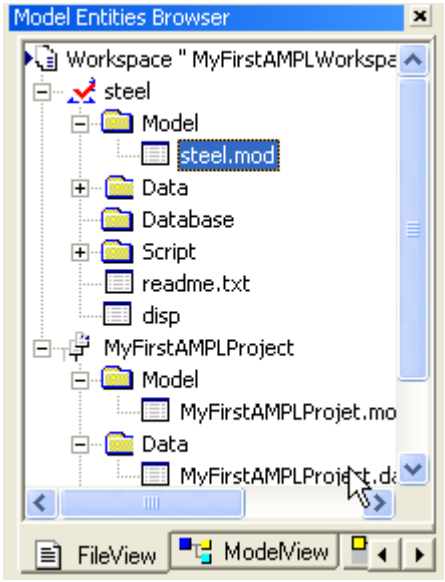


Figure 4.35: Selecting the steel.mod file in the Workspace

Step 2: Click on the button on the Execution Toolbar to Build the Model.

The AMPL Studio reads the model and displays the following Console Message.

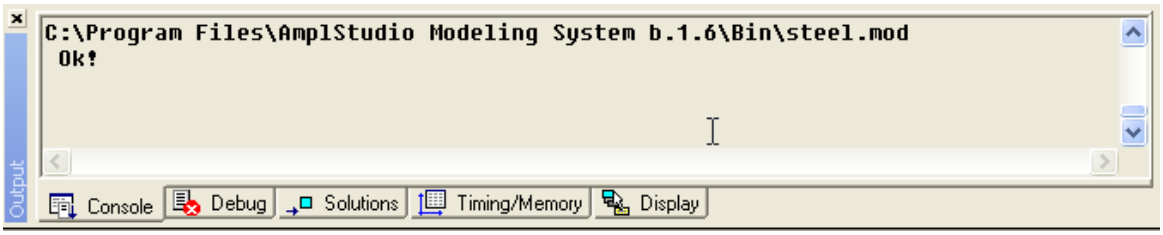


Figure 4.36: AMPL Console message for reading steel.mod file

Step 3: Click on the **`steel.dat`** file.

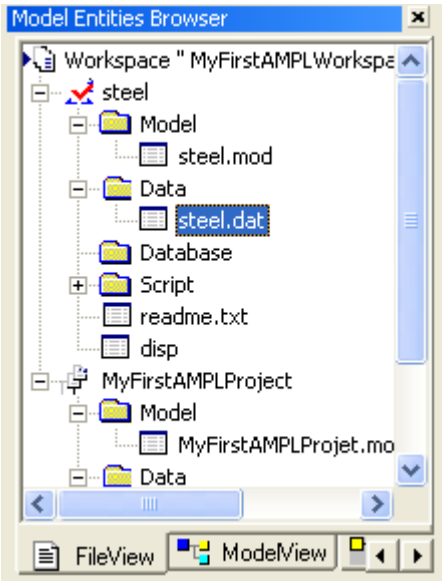


Figure 4.37: Selecting the steel.dat file in the Workspace

Step 4: Click on the button on the Execution Toolbar to Build the Data.

The AMPL Studio reads the model and displays the following Console Message.

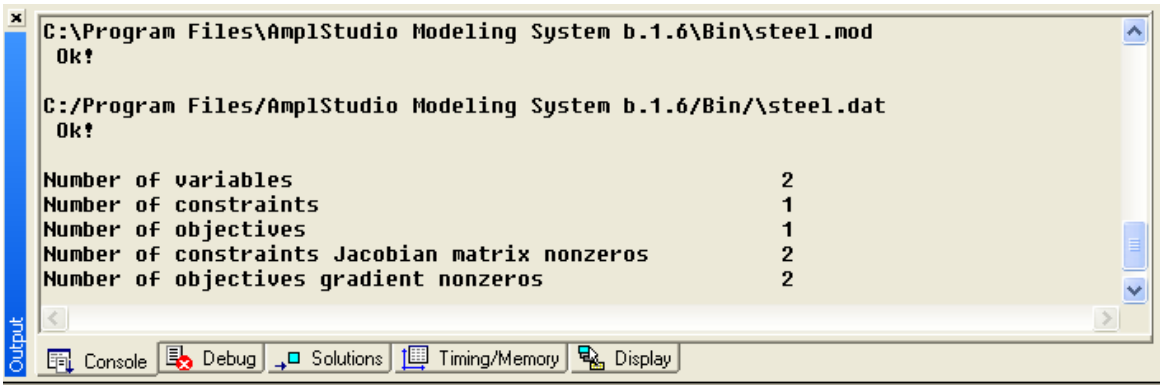
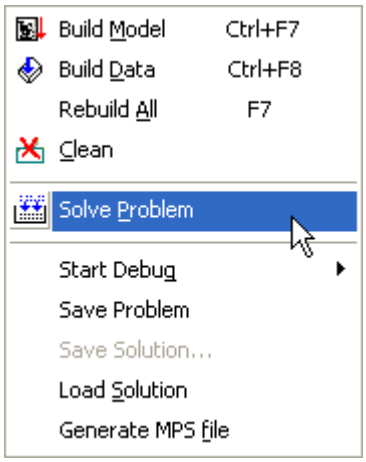


Figure 4.38: AMPL Console message for reading steel.mod and steel.dat file

Solving the Problem

Now the Model and Data files are read and the Solver is selected. In order to solve the problem do the following steps.

Step 1: Click on the button on the Execution Toolbar to solve the problem.

Note	You can also select the Solve Problem Menu from the Build Menu. 

The AMPL Studio will solve the steel problem using FortMP Solver and display the solution file in the editing area.

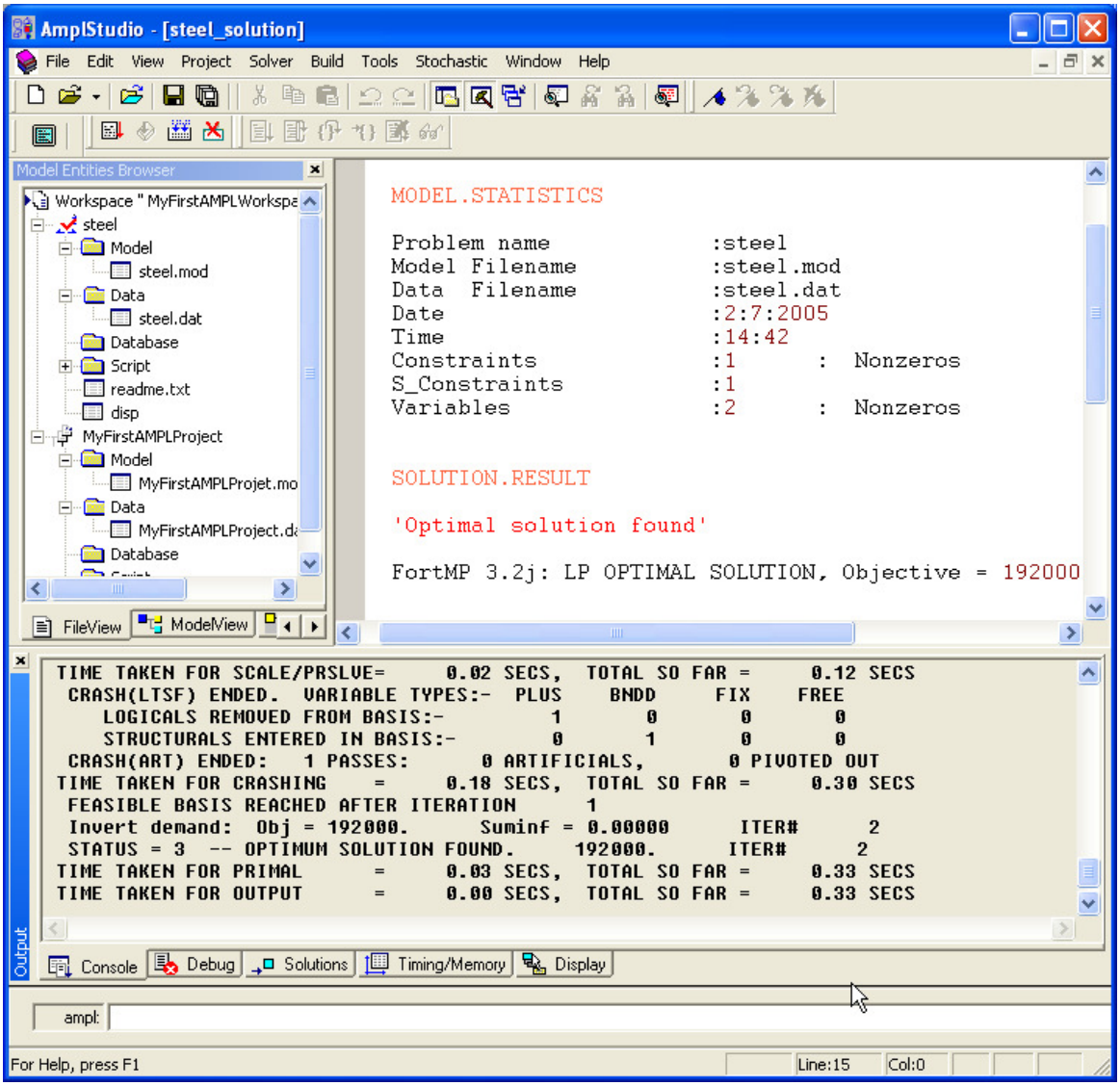


Figure 4.39: AMPL Studio Solver Console message and Solution Display

Viewing Results

The user can view various parts of the model and the solution from the Workspace and their information will be displayed on the display window as shown below.

Step 1: Click on the **ModelView** tab on the **Workspace**.

Step 2: Expand the Parameters node and Double Click on the `rate` **Parameters**.

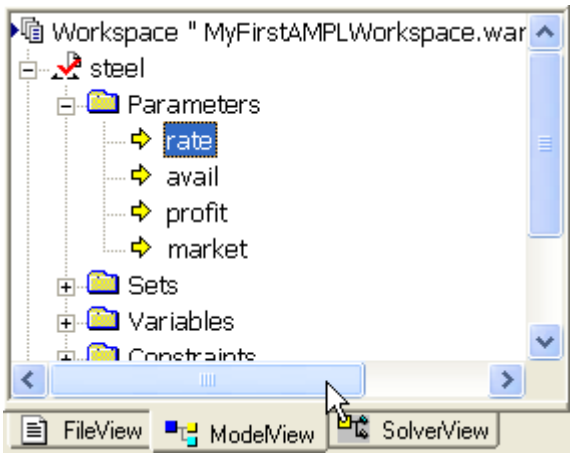


Figure 4.40: Choosing the rate Parameter for Viewing

The AMPL Studio Display Window displays the rate parameters as below.

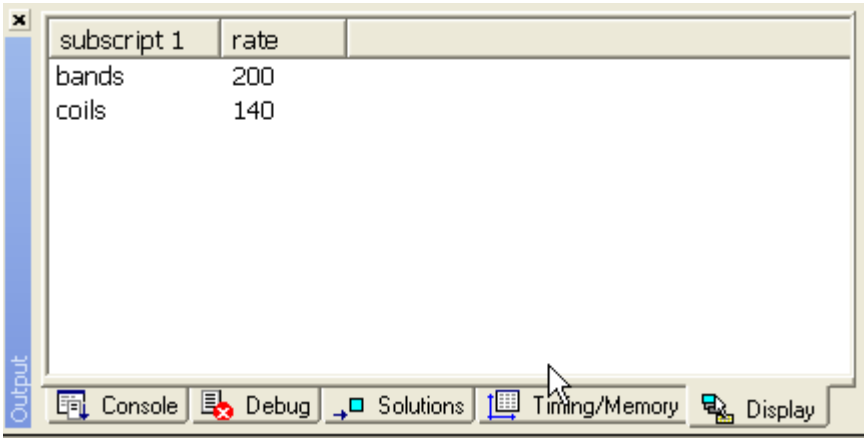


Figure 4.41: Displaying the rate Parameter

Step 3: Now Expand the Variable node and Double Click on the **Make Variable.**

The AMPL Studio Display Window displays the Make variable value as shown below.

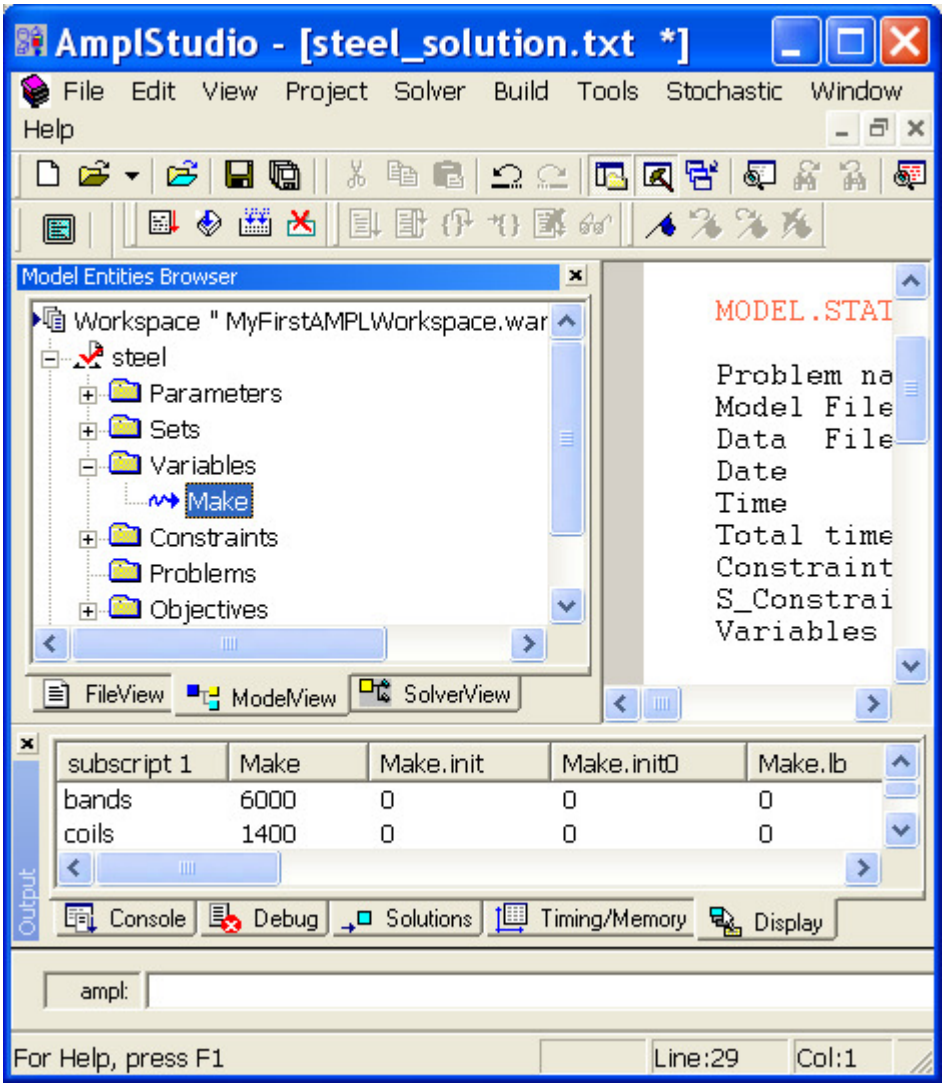


Figure 4.42: Displaying the Make Variable

Solving the project with the Script

In the previous sections you have solved the steel project. During the Solution process you have gone through a number of steps like **Build Model**, **Build Data**, **Selecting the Solver**, etc., to generate the solution. This process can be automated by creating a script file.

The following steel.sa1 script file was written to automate what we have done during the previous section. In this case we use CPLEX solver to solve the steel problem.

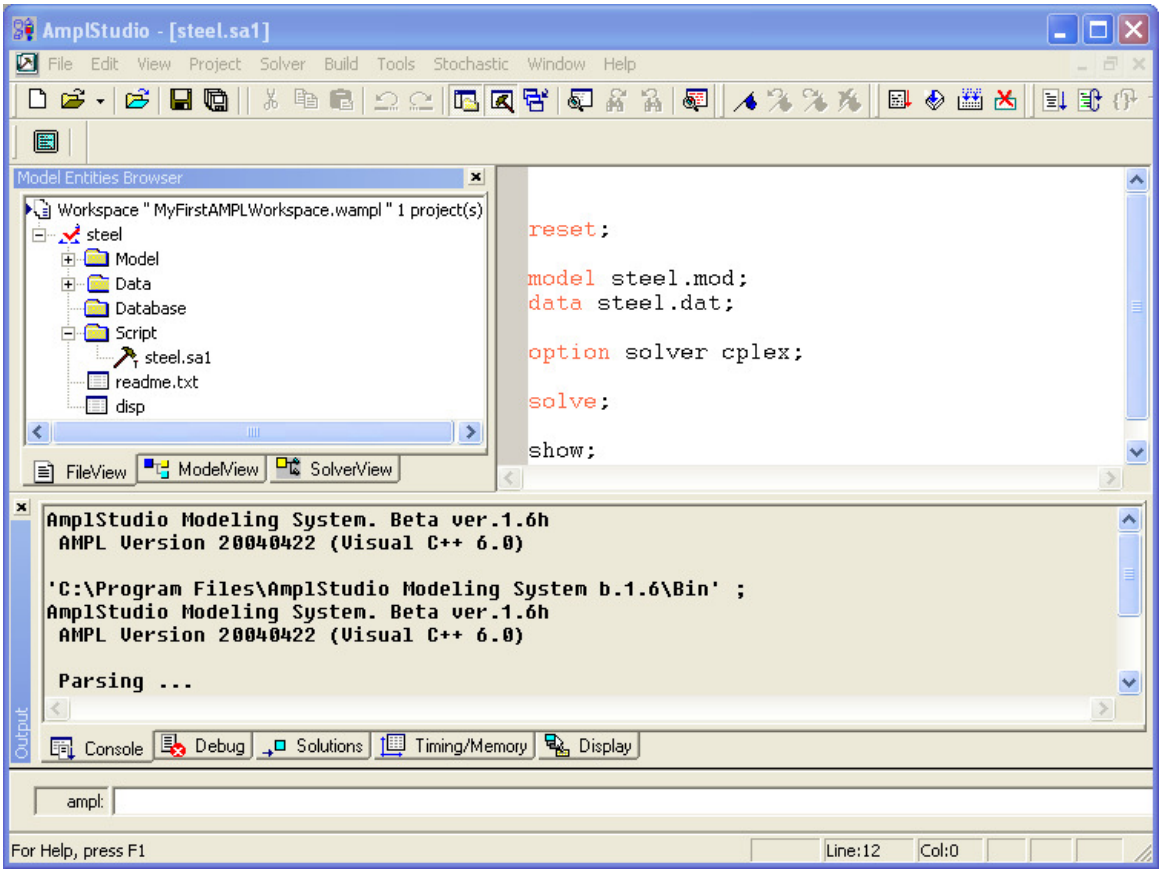


Figure 4.43: Writing Script File

Clicking on the button will execute all the AMPL Statements in the script file and display the results.

Setting the AMPL Studio Options

AMPL Studio has various options for you to choose from. In order to update the AMPL studio options choose the **Options** Menu from the **Tools Menu**.

Tools Menu

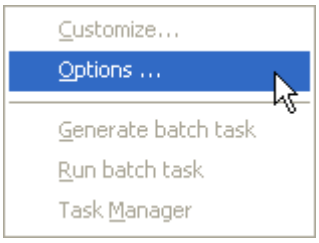


Figure 4.44: Choosing AMPL Studio Options

AMPL Studio then displays an AMPL Studio options in the following dialog box for your selection.

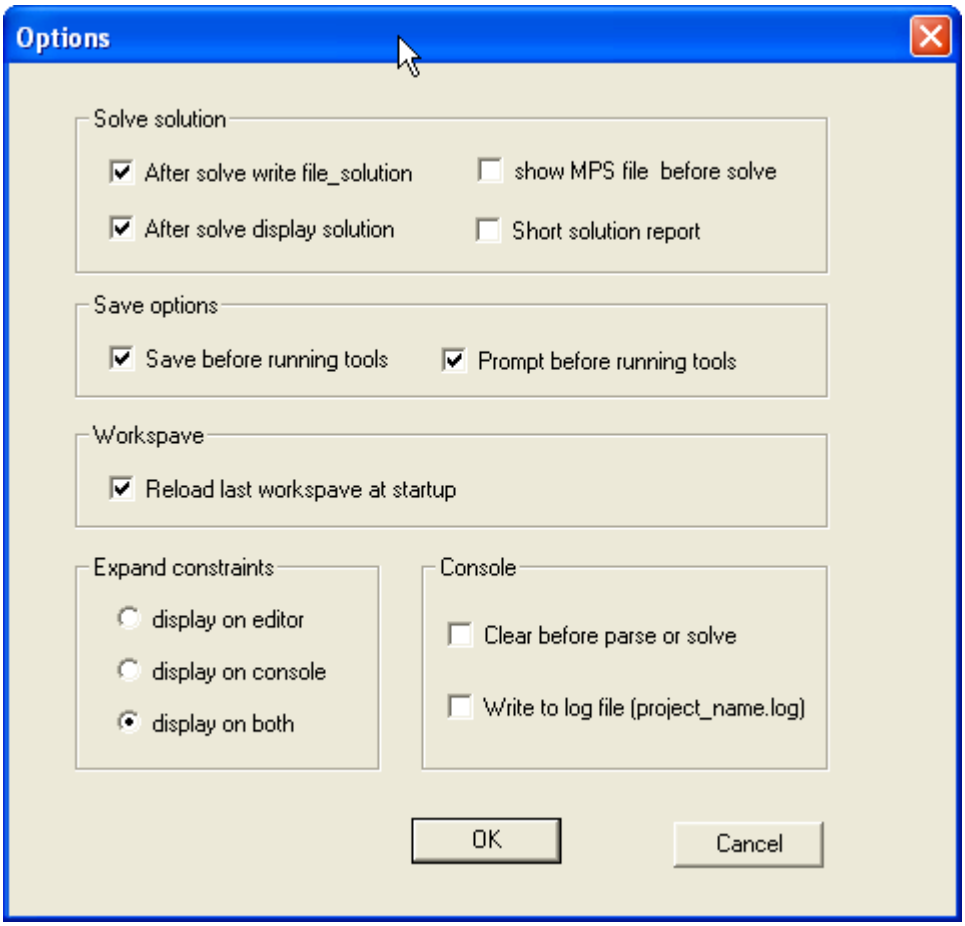


Figure 4.45: AMPL Studio Options Dialog Box

AMPL Studio options are divided into **Solve Solution**, **Save Options**, **Workspace** and **Expand Constraints** categories.

Using online help

Online help can be accessed from the Help Menu. You need an Internet connection to access **www.ampl.com** Menu from **Help** Menu.

Terminating the AMPL Studio

Selecting the **Exit** menu from the **File** Menu will terminate the AMPL Studio session.

Chapter 5: Introducing AMPL through AMPL Studio

Introduction to Models for Linear programming

In order to suitably represent the linear programs we make use of mathematical notations. We call the compact description of the general form of the problem, as a 'model'. The fundamental components of a model are:

- Sets
- Parameters
- Variables, whose values the solver is to determine
- An Objective, to be maximized or minimized
- Constraints, that the solution must satisfy

The example below shows a symbolic model:

Given: P, a set of products
a_j = Tons per hour of product j, for each j ∈ P
b = hours available at the mill
c_j = profit per ton of product j, for each j ∈ P
u_j = maximum tons of product j, for each j ∈ P

Define variables: X_j = tons of product to be made, for each j ∈ P

Maximize: $\sum_{j \in P} c_j X_j$

Subject to: $\sum_{j \in P} (1/a_j) X_j \le b$

$0 \le X_j \le u_j$, for each $j \in P$

Figure 5.1: A symbolic production model in algebraic form

Fundamental Components of AMPL linear programming Model

Sets

Unordered Sets

The most elementary kind of AMPL set is an unordered collection of character strings. Usually all of the strings in a set are intended to represent instances of the same kind of entity.

The declaration of a set need only contain the keyword 'set' and a name. For example a model may declare

```
set PROD;
```

to indicate that a certain set will be referred to by the name PROD in the rest of the model. A name may be any sequence of letters, numerals, and underscore (_) characters that is not a legal number. A few names have special meanings in AMPL and may only be used for specific purposes, while a large number of names have predefined names that can be changed if they are used in some other way.

A declared set's membership is normally specified as part of the data for the model. Occasionally, however, it is desirable to refer to a particular set of strings within a model. A literal set of this kind is specified by listing its members within braces:

```
set PROD = {"bands", "coils", "plate"};
```

This sort of declaration is best limited to cases where a set's membership is small, is a fundamental aspect of the model, or is not expected to change often.

Sets of numbers

Set members may also be numbers. In fact a set's members may be mixture of numbers and strings, though this is seldom the case. In an AMPL model, a literal number is written in the customary way as a sequence of digits, optionally preceded by a sign, containing an optional decimal point, and optionally followed by an exponent; the exponent consists of a d, D, e or E, optionally a sign, and a sequence of digits.

A set of numbers is often a sequence that corresponds to some progression in the situation being modeled, such as a series of weeks or years. Just as for strings, the numbers in a set can be specified as part of the data, or can be specified within a model as a list between braces, such as {1, 2, 3, 4, 5, 6}. This sort of set can be described more concisely by notation 1..6. An addition 'by' clause can be used to specify an interval more than 1 between the numbers; for instance,

```
1990.. 2020  by    5
```

Represents the set

```
{1990, 1995, 2000, 2005, 2010, 2015, 2020}
```

This kind of expression can be used anywhere that a set is appropriate.

The members of a set of numbers have the same properties as any other numbers, and hence can be used in arithmetic expressions.

Set Operations

AMPL has four operators that construct new sets from existing ones:

```
A union B          union: in either A or B
A inter B          intersection: in both A and B
A diff B           difference: in A but not B
A symdiff B        symmetric difference: in A or B but not both
```

The following example shows how this work:

```
ampl:set Y1 = 1990 .. 2020 by 5;
ampl:set Y2 = 2000 .. 2025 by 5;
ampl: display Y1 union Y2, Y1 inter Y2;
set Y1 union Y2 := 1990 1995  2000  2005  2010  2015  2020  2025;
set Y1 inter Y2 := 2000 2005  2010  2015  2020;
ampl: display Y1 diff Y2, Y1 symdiff Y2;
set Y1 diff Y2 := 1990  1995;
set Y1 symdiff Y2 := 1990 1995 2025;
```

Set membership operations and functions

Two other AMPL operators, 'in' and 'within', test the membership of sets. As an example the expression

```
"B2" in NUTR
```

Is true if and only if the string "B2" is a member of the set NUTR. The expression

```
MINREQ within NUTR
```

is true if all members of the set MINREQ are also members of NUTR – that is, if MINREQ is a subset of(or is same as) NUTR.

AMPL also provides 'not in' and 'not within', which reverses the truth value of their results.

The built in function 'card' computes the number of members in (or cardinality of) a set; for example, `card(NUTR)`, is the number of the members in `NUTR`.

Indexing Expressions

In algebraic notation, the use of sets is indicated informally by phrases such as "for all i ∈ P" or "for t=1,…,T" or "for all j ∈ R such that $c_j > 0$." The AMPL counterpart is the indexing expression that appears within braces { … }. An indexing expression is used whenever we specify the set over which a model component is indexed, or the set over which a summation runs. Since an indexing expression defines a set, it can be used in any place where a set is appropriate.

The simplest form of indexing expression is just a set name or expression within braces. For example:

```
param rate {PROD} > 0 ;
param avail {1..T} > = 0;
```

References to these parameters are subscripted with a single set member, in expression such as `avail[t]` and `rate[p]`.

The names such as `t` or `i` that appear in subscripts and other expressions in our models are examples of *dummy indices* that have been defined by indexing expressions. In fact, any indexing expression may optionally define a dummy index that runs over the specified set.

An indexing expression consists of an index name, the keyword 'in', and a set expression as before. Although a name defined by a model component's declaration is known throughout all subsequent statements in the model, the definition of dummy index name is effective only within the scope of the defining indexing expression. Once an indexing expression's scope has ended, its dummy index becomes undefined. Thus the same index name can be defined again and again in the model.

As a final option, the set in an indexing expression may be followed by a colon(:) and a logical condition. The indexing expression then represents only the subset of members that satisfy the condition. For example:

```
{j in FOOD: f_max [j] - f_min[j] < 1}
```

describes the set of all foods whose minimum and maximum amounts are nearly the same.

Ordered Sets

Any set of numbers has a natural ordering, so numbers are often used to represent entities, like time periods, whose ordering is essential to the specification of a model. To describe the difference between this week's inventory and the previous week's inventory, for example, we need the weeks to be ordered so that the "previous" week is always well defined.

An AMPL model can also define its own ordering for any set of numbers or strings, by adding the keyword 'ordered' or 'circular' to the set's declaration. The order in which we give the set's members, in either the model or data, is the order in which AMPL works with them. In a set declared 'circular', the first member is considered to follow the last one, and the last to precede the first; in an ordered set, the first member has no predecessor and the last member has no successor.

There are many functions on ordered sets to retrieve some specific members from the set. Users are referred to AMPL manual or AMPL textbook for further details.

Parameters

In AMPL a single named numerical value is called *parameter*. Although some parameters are defined as individual scalar values, most occur in vectors or matrices or other collections of numerical values indexed over sets. Parameters and other numerical values are the building blocks of the expressions that make up a model's objective and constraints.

Parameter declarations have a list of optional attributes, optionally separated by commas:

parameter declaration:

```
param name alias_opt indexing_opt attributes_opt ;
```

The attributes may be any of the following:

```
attribute:
      binary
      integer
      symbolic
      relop expr
      In sexpr
      = expr
      Default expr
      relop:
      < <= = == != <> > >=
```

The keyword *integer* restricts the parameter to be an integer; *binary* restricts it to 0 or 1. If *symbolic* is specified, then the parameter may assume any literal or

numeric value, and the attributes involving <.<=,>= and > are disallowed; otherwise the parameter is numeric and can only assume a numeric value.

The attributes involving comparison operators specify that the parameter must obey the given relation. The = and *default* attributes are analogous to the corresponding ones in set declarations and are mutually exclusive.
Recursive definitions of indexed parameters are allowed, so long as the assigned values can be computed in a sequence that only references previously computed values. For example:

```
param comb 'n choose k' {n in 0..N, k in 0..n}
      = if k = 0 or k = n then 1 else comb [n-1,k-1] + comb[n-1,k];

Computes the number of ways of choosing n things k at a
time.
```

Variables

The variables of a linear program have much in common with its numerical parameters. Both are symbols that stand for numbers, and that may be used in arithmetic expressions. Parameter values are supplied by the modeler or computed from other values, while the values of variables are determined by an optimizing algorithm. Syntactically, variable declarations are the same as the parameter declaration defined earlier, except that they begin with the keyword 'var' rather than 'param'. The meaning of qualifying phrases within the declaration may be different, however when these phrases are applied to variables rather than to parameters.

Phrases beginning with >= or <= are by far the most common in declarations of variables for linear programs. For example:

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

The declaration creates an indexed collection of variables `Make[p]`, one for each member `p` of the set `PROD`; the rules in this respect are exactly the same as for parameters. The effect of the two qualifying phrases is to impose a restriction, or constraint, on the permissible values of the variables. Specifically, >= 0 implies that all of the variables `Make[p]` must be assigned non negative values by the optimizing algorithm, while the phrase `<=market[p]` says that, for each product `p`, the value given to `Make[p]` may not exceed the value of the parameter `market[p]`. In general, either >= or <= may be followed by an arithmetic expression in previously defined sets and parameters and currently defined dummy indices. The values following >= and <= are lower and upper bounds on the variables.

An = phrase in a variable declaration gives rise to a definition, as in parameter declaration. Because a variable is being declared, however, the expression to the

right of = operator may contain previously declared variables as well as sets and parameters.

A := or 'default' phrase in a variable declaration gives initial values to the indicated variables. Variables are not assigned an initial value by := can also be assigned initial values from a data file.

Finally, variables can be defined as 'integer' or 'binary'.

Linear Expressions

An arithmetic expression is *linear* in a given variable if, for every unit increase or decrease in the variable, the value of expression increases or decreases by some fixed amount. An expression that is linear in all its variables, is called a linear expression.

AMPL recognizes as a linear expression any sum of terms of the form:

```
constant-expr
variable-ref
(constant-expr) * variable ref
```

Provided that each `constant-expr` is an arithmetic expression that contains no variables, while `var-ref` is a reference to a variable. The parentheses around the constant-expr may be omitted if the result is the same according to the rules of operator precedence.

Objectives

The declaration of an objective function consist of one of the keywords `minimize` or `maximize`, a name, a colon, and a linear expression in previously defined sets, parameters and variables. For example:

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

and

```
maximize Total_Profit:
      sum {p in PROD, t in 1..T}
            (sum   {a   in   AREA[p]   revenue[p,a,t]   *   Sell[p,a,t]   -
prodcost[p] * Make[p,t] - invcost[p] * Inv[p,t]);
```

Within AMPL commands, the objective's name refers to its value.

Although a particular linear program must have one objective function, a model may contain more than one objective declaration. Moreover, any `minimize` or `maximize` declaration may define an indexed collection of objective functions, by including an indexing expression after the objective name. In these cases, we

may issue an `objective` command, before typing `solve`, to indicate which objective is to be optimized.

Constraints

The simplest kinds of constraint declaration begins with the keywords `subject to`, a name, and a colon. Even the `subject to` is optional; AMPL assumes that any declaration not beginning with a keyword is a constraint. Following the colon in an algebraic description of the constraint, in terms of previously defined sets, parameters and variables. For example:

```
subject to  Time:
     sum{p in PROD} (1/rate[p])* Make[p] <= avail;
```

The name of a constraint, like the name of an objective, is not used anywhere else in an algebraic model, though it figures in alternative "columnwise" formulations and is used in AMPL command environment to specify the constraint's dual value and other associated quantities.

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

The constraint `Time`, for example, is generalized in the subsequent example to say that the production time may not exceed the time available in each processing stage `s`.

```
subject to Time{s in STAGE}:
     sum {p in PROD} (1/rate[p,s])* Make[p] <= avail[s];
```

The indexing expression in a constraint declaration should specify a dummy index for each dimension of the indexing set.

AMPL's algebraic description of a constraint may consist of any two linear expressions separated by an equality or inequality operator:

```
linear-expr <= linear-expr
linear-expr = linear-expr
linear-expr >= linear-expr
```

While it is customary in mathematical descriptions of linear programming to place all terms containing variables to the left of the operator and all other terms to the right, AMPL imposes no such requirement. AMPL also allows double inequality constraints. The permissible forms for a constraint of this kind are:

```
const-expr <= linear-expr <= const-expr
const-expr <= linear-expr <= const-expr
```

`where each const-expr` must contain no variables. The effect is to give upper and lower bounds on the value of the `linear-expr`.

The example below gives the AMPL model and data files for the symbolic algebraic model considered in the beginning of this chapter.

```
set P;
param a {j in P};
param b;
param c {j in P};
param u {j in P};
var x {j in P};
maximize Total_Profit: sum {j in P} c[j]* X[j];
subject to Time: sum {j in P} (1/a[j]) * X[j] <= b;
subject to Limit {j in P}: 0 <= X[j] <= u[j] ;
```

Figure 5.2: Basic production model in AMPL

```
set P := bands coils;
param:      a     c     u     :=
 bands      200   25    6000
 coils      140   30    4000  ;
param b := 40;
```

Figure 5.2: Production model data file in AMPL

Stochastic Extension to AMPL: SAMPL

In addition to supporting AMPL language syntax for deterministic problems, AMPL Studio has an extension for stochastic programming called SAMPL, available as a separate package.

SAMPL has additional syntax and commands. Users are referred to SAMPL manual for more details.

Chapter 6: A Step-By-Step Walk Through Example

Now you know the basics of AMPL studio. Now we will go through the steps involved in solving a simple real world problem of National Insurance Associate's (NIA) investment problem using AMPL studio. Before we open the AMPL studio the problem needs to be analysed and translated into mathematical notation, and then into an AMPL model. The following steps go into detail.

A Simple Real World Problem

National Insurance Associates carries an investment portfolio of stocks, bonds and other investment alternatives. Currently £200,000 of funds is 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: Financial Data

The risk measure indicates the relative uncertainty associated with the stock in terms of it realising the projected annual return: higher values indicate greater risk.

National'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

They request you to find the investment decisions.

Formulating the Problem into Mathematical Form

In this problem we need to find the number of stocks A, B, C and D need to be bought with the provided guidelines and with minimum risk.

Now this problem needs to be presented in the mathematical form. This will involve three steps

(1) Formulate an LP that minimises risk

(2) Identifying the Decision Variables
The decision that National faces is to decide how much of each stock to buy.

Let x_1 be the number of shares of stock A bought
x_2 be the number of shares of stock B bought
x_3 be the number of shares of stock C bought
x_4 be the number of shares of stock D bought

(3) Determine the values of these four variables in order to minimise National's risk

Identify the Objective Function

In our example we wish to minimise risk. We risk £0.10 on each pound invested in stock A, similarly for stock B the risk is 0.07 per pound, for stock C it is 0.05, and for stock D the corresponding risk is 0.08.

Thus if we buy x_1 shares of stock A, we have a risk exposure of $0.10*100*x_1$ since each share costs £100. Similarly, if we buy x_2 shares of stock B we risk $0.07*50*x_2$, while for stocks C and D the risk measures are $0.05*80*x_2$ and $0.10*40*x_2$. Therefore this leads to the following quantity that we wish to minimise

$$\text{Risk} = 0.10*100\, x_1 + 0.07*50\, x_2 + 0.05*80\, x_1 + 0.10*40\, x_2$$

Identifying the Constraints

The first constraint concerns the budget. That is we can't invest more than the money we have available. This leads to the following constraint

$$100*x_1 + 50*x_2 + 80*x_3 + 40*x_4 \leq 200000$$

The second constraint concerns the rate of return of the portfolio and is as follows

$$100*0.12*x_1 + 50*0.08*x_2 + 80*0.06*x_3 + 40*0.10*x_4 \geq 200000*.09$$

Finally, the cash investment in any one stock cannot exceed 50% of the total investment

$$100*x_1 \leq 100000$$

$50*x_2 \leq 100000$
$80*x_3 \leq 100000$
$40*x_4 \leq 100000$ and $x_1 \geq 0, x_2 \geq 0, x_3 \geq 0, x_4 \geq 0$

Translating the Mathematical Problem into AMPL Model

AMPL is mainly an algebraic language. That means it follows the algebraic syntax used in the mathematical representation of the problems. AMPL's main keyword declarations are **set, param**, **var** and **maximize**/**minimize**

Since AMPL deal with plain text files the above problem can be rewrite as the following AMPL model as follows. Where x_1, x_2, x_3 and x_4 are replaced with the most suitable variable names StockA, StockB, StockC and StockD.

Minimize

```
Risk = 10*StockA + 3.5*StockB + 4*StockC + 4*StockD
```

Variables

```
StockA ≤ 1000
StockB ≤ 2000
StockC ≤ 1250
StockD ≤ 2500
```

Subject to

```
100*StockA + 50*StockB + 80*StockC + 40*StockD ≤ 200000
12*StockA + 4*StockB + 4.8*StockC + 4*StockD ≥ 18000
```

Using AMPL Studio to Solve the Problem.

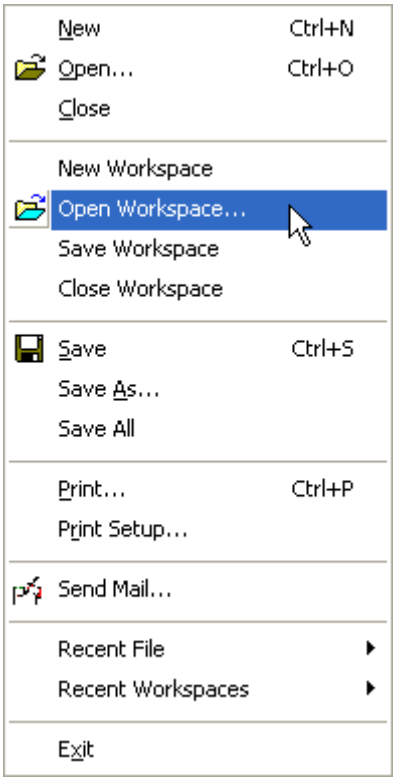
Now the AMPL model is ready for the problem. Now you open the AMPL studio.

Create Workspace

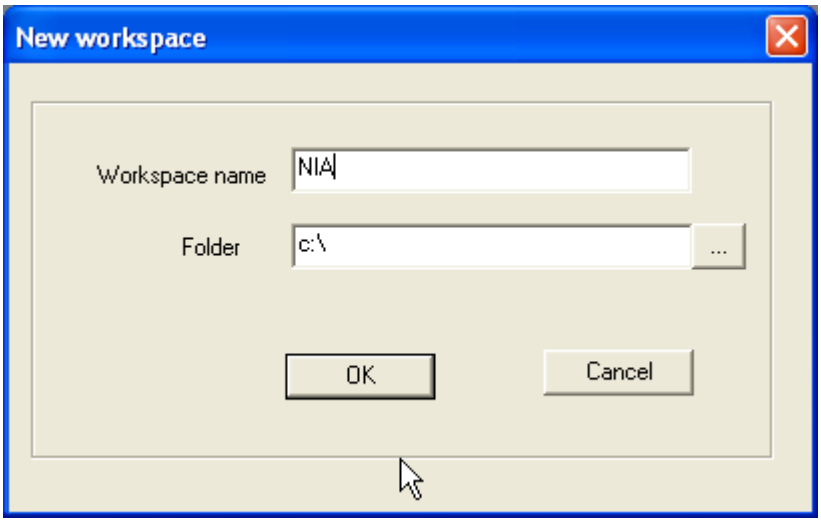
In order to create a new workspace for NIA's problem create a new workspace with the following steps.

Step 1: Choose **New Workspace** from the **File** menu.

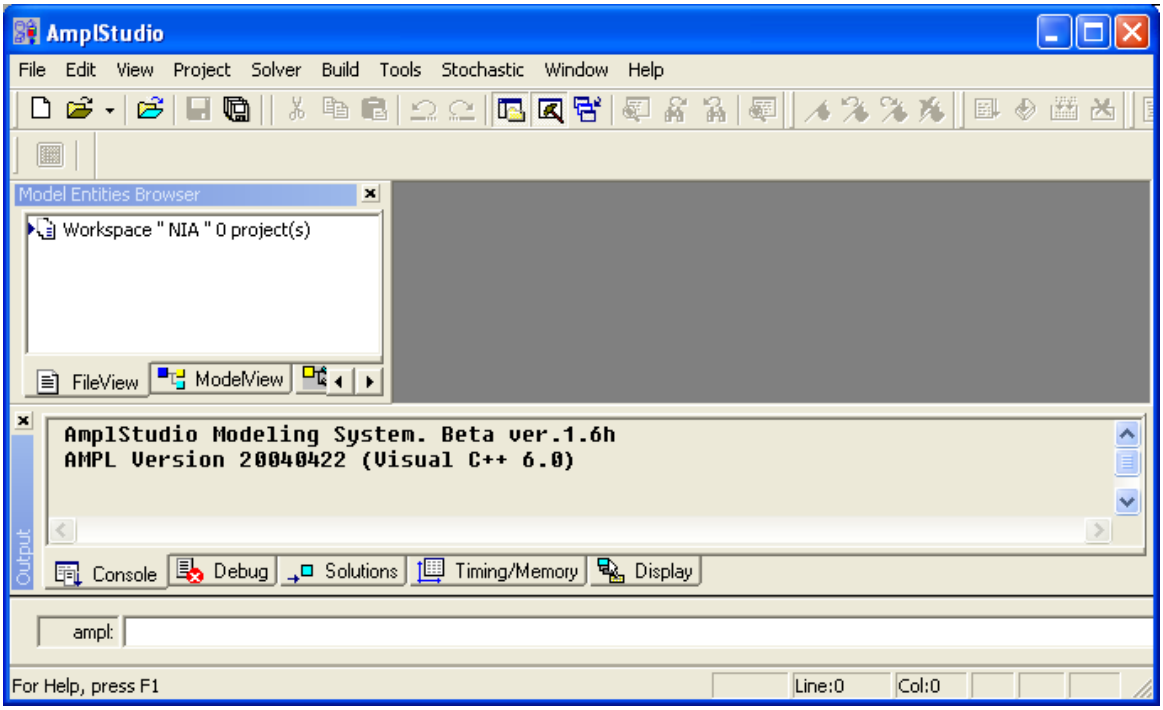
File



Step 2: Write workspace name as NIA and choose your appropriate folder (C:\) by clicking the ellipsis (…) button where you want to create your workspace.

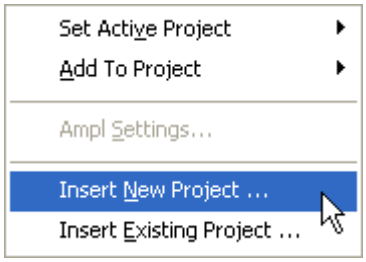


Click OK to create the workspace at your chosen folder.



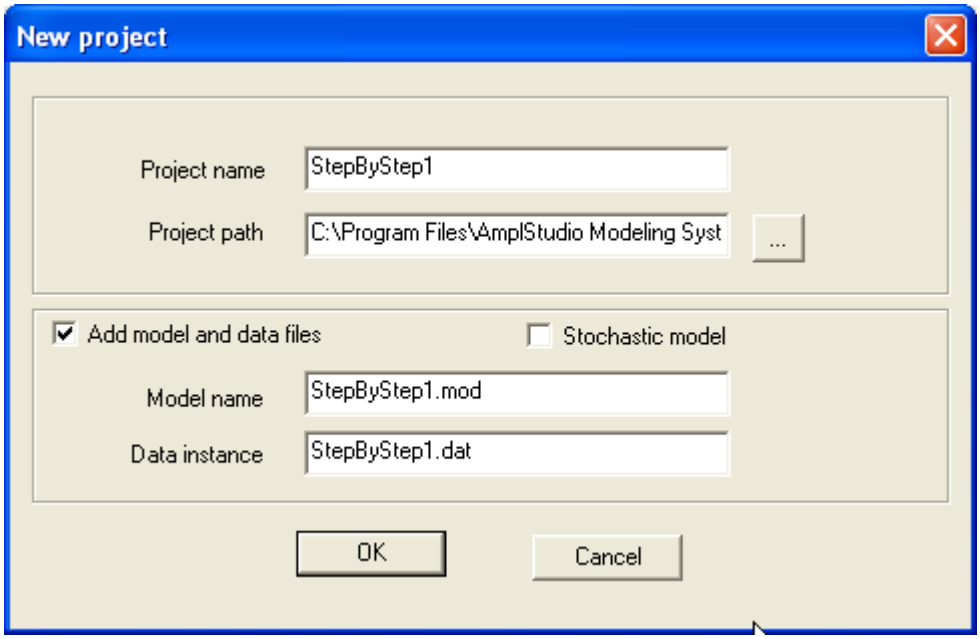
Create a Project

Step 1: Having created a workspace, we now define a new project by selecting **Insert New Project** from the **Project** menu.



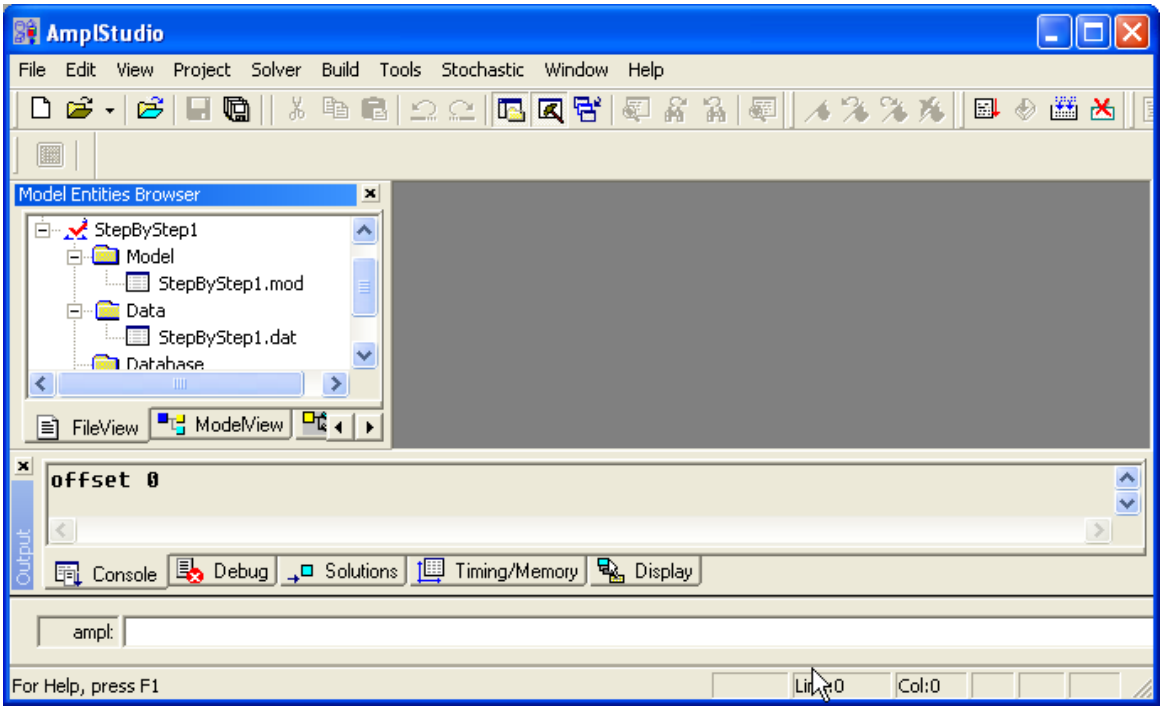
Step 2: Enter the Project name as "**`StepByStep1`**" and choose your preferred folder by clicking the ellipsis (…) button.

Check the ☑ **Add templates** checkbox and write the **Model name** as "**`StepByStep1.mod`**" and the **Data instance** as "**`StepByStep1.dat`**".



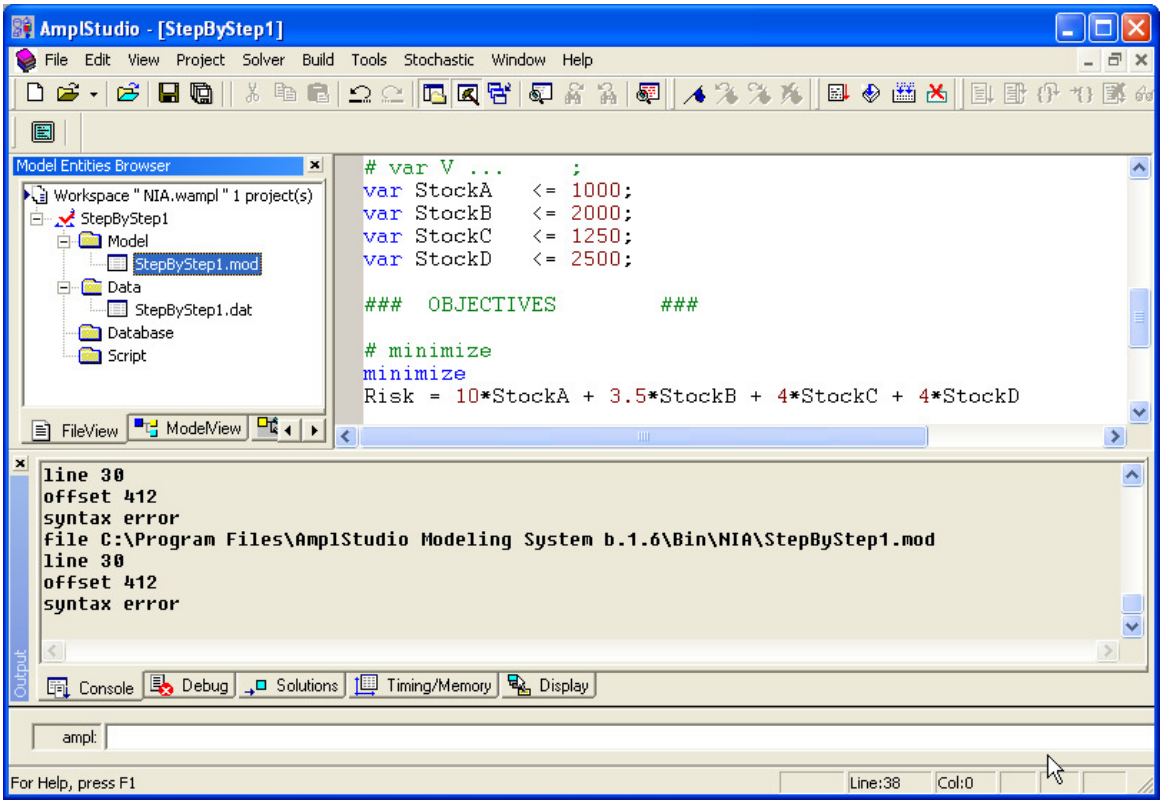
Clicking OK will create a new project with the model and data template files within the created workspace.

Create an AMPL Model file



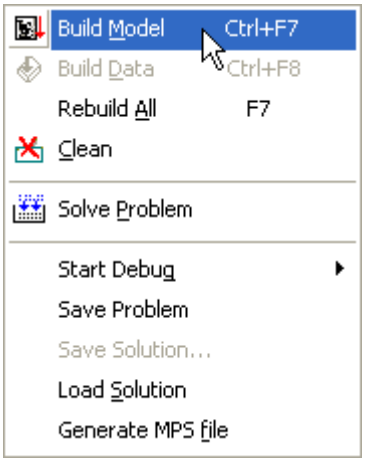
Double clicking on the model file will open the model template file. The lines with # at the beginning are comment lines. The AMPL key words will be in blue and the numbers in red.

Step 1: Now write your AMPL model in this window.

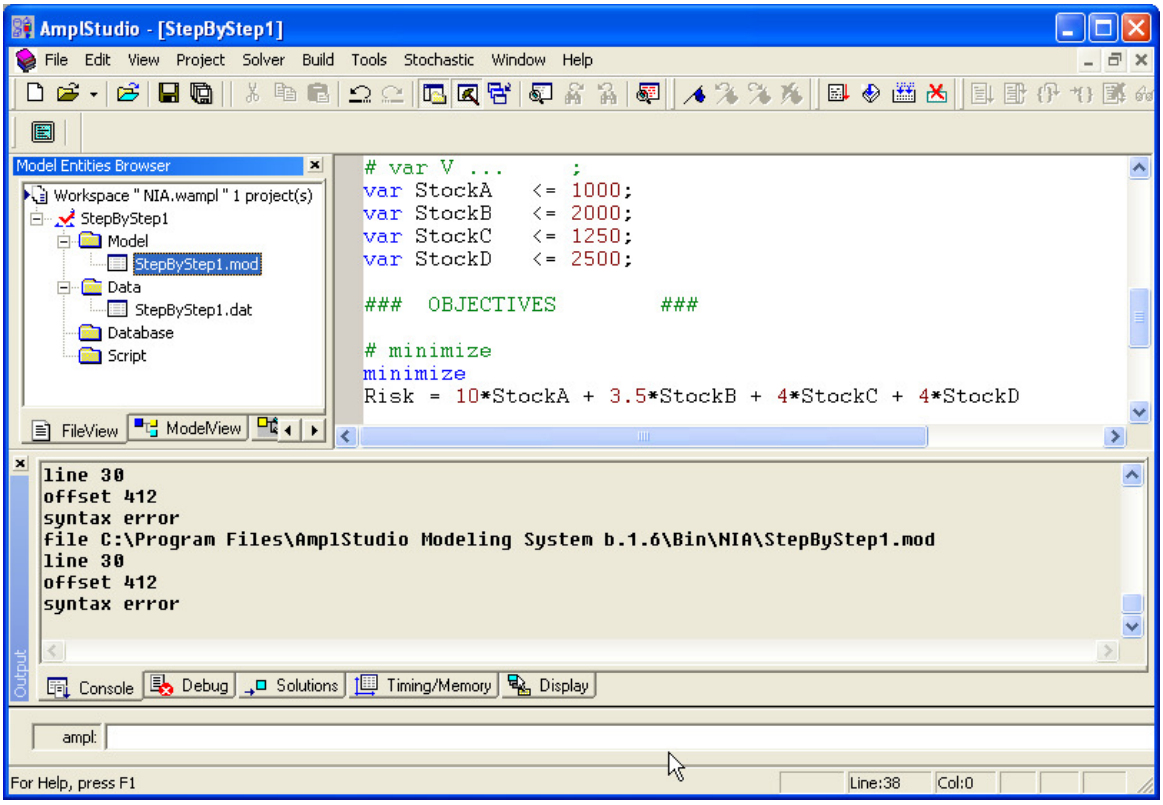


Step 2: To check the syntax of your model choose Build Model menu from Build menu.

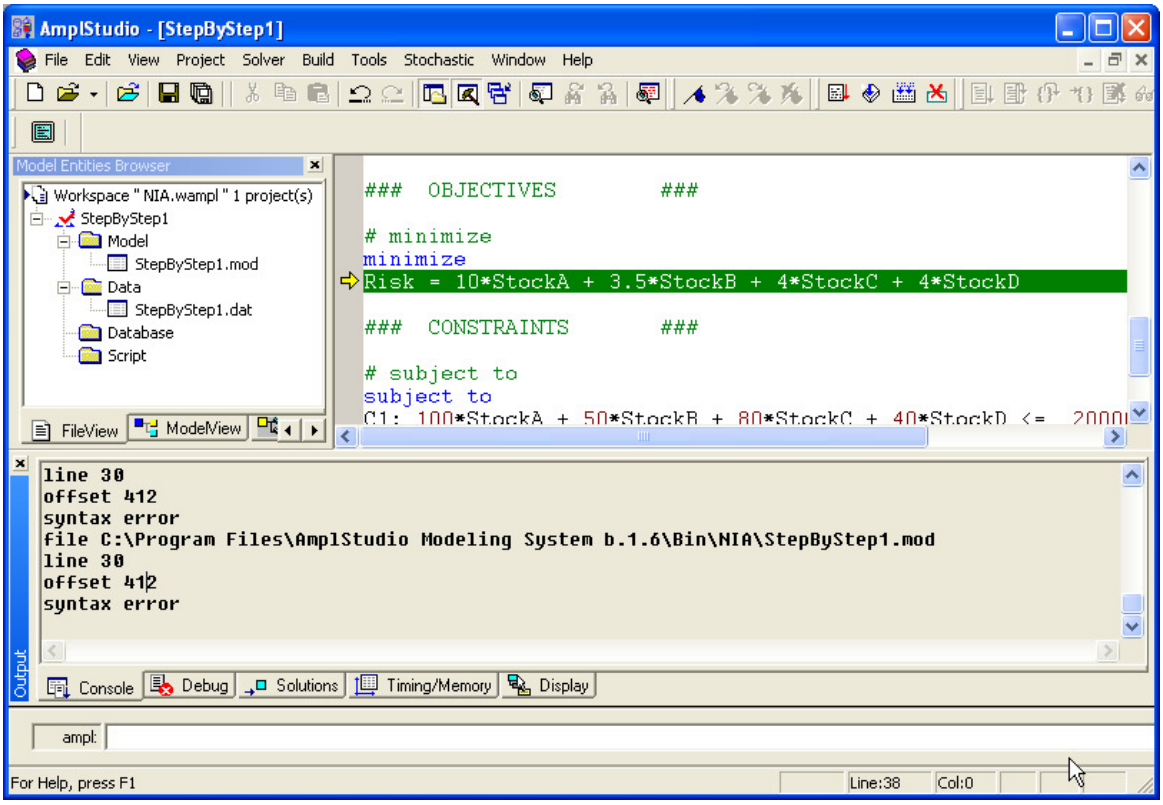
Build Menu



If any syntax errors occurred then the appropriate error messages will be displayed in the Console Window. In the above model displays the following syntax error.

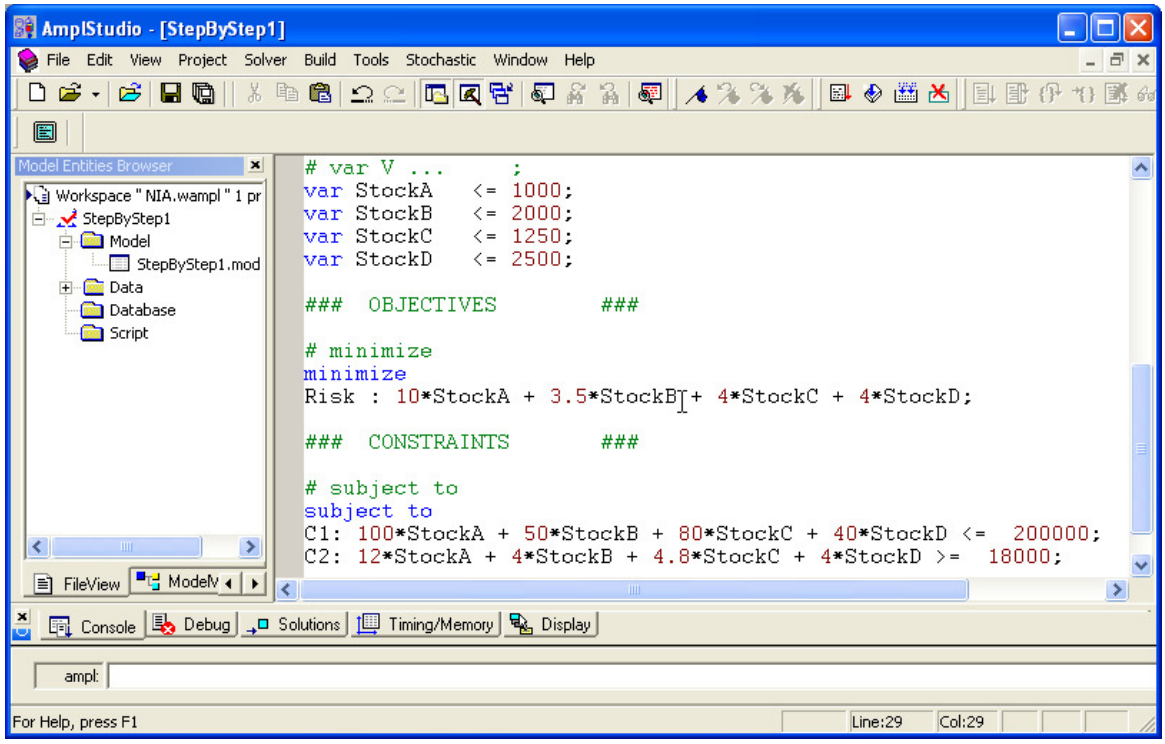


Step 3: Double click on the error line (line 30) will display the following screen.

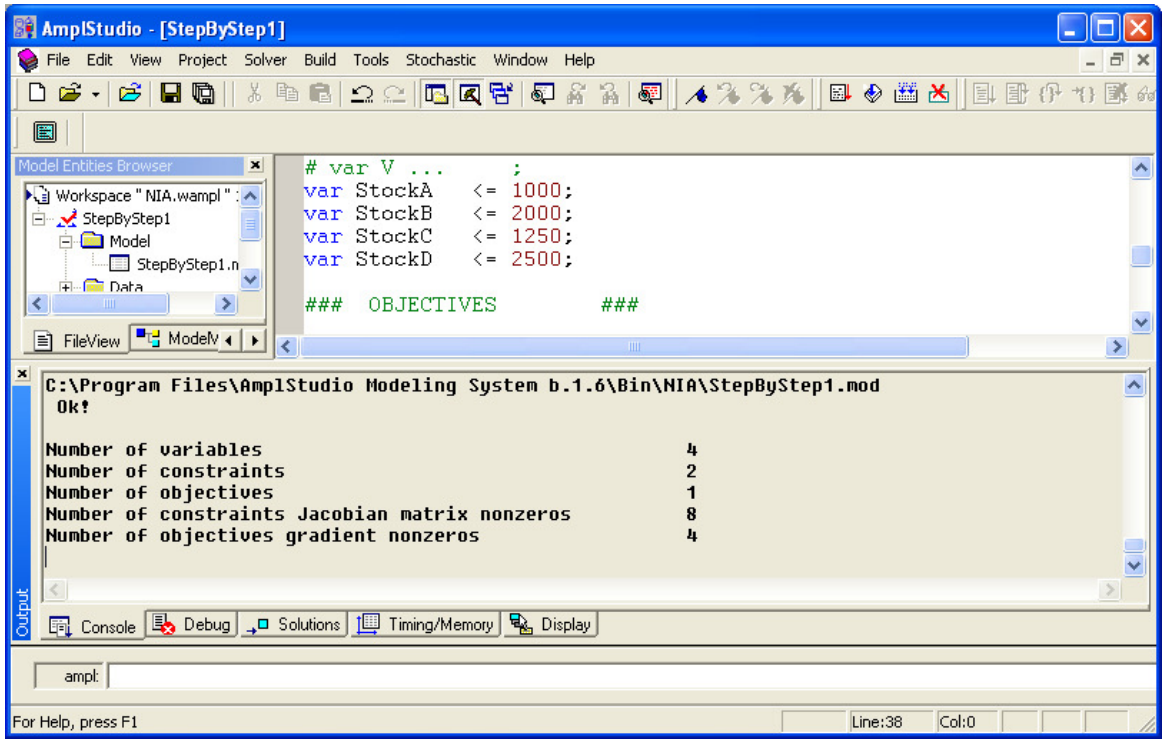


Step 4: The line has two errors.

1. Risk = should be replaced by Risk :
2. Semicolon is missing at the end of line.



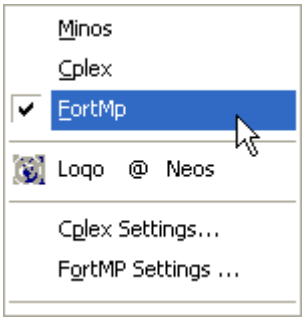
Step 5: Make these corrections and compiling it again will show the following.



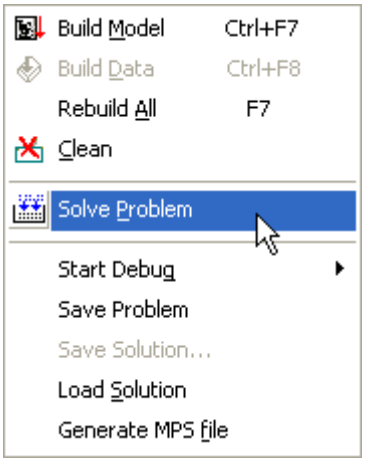
Solve and Display Results

Step 1: In order to solve the model you need to select the solver. By default you will receive FortMP solver with your AMPL studio distribution. FortMP is a powerful solver and capable to handle this simple problem.

Solver

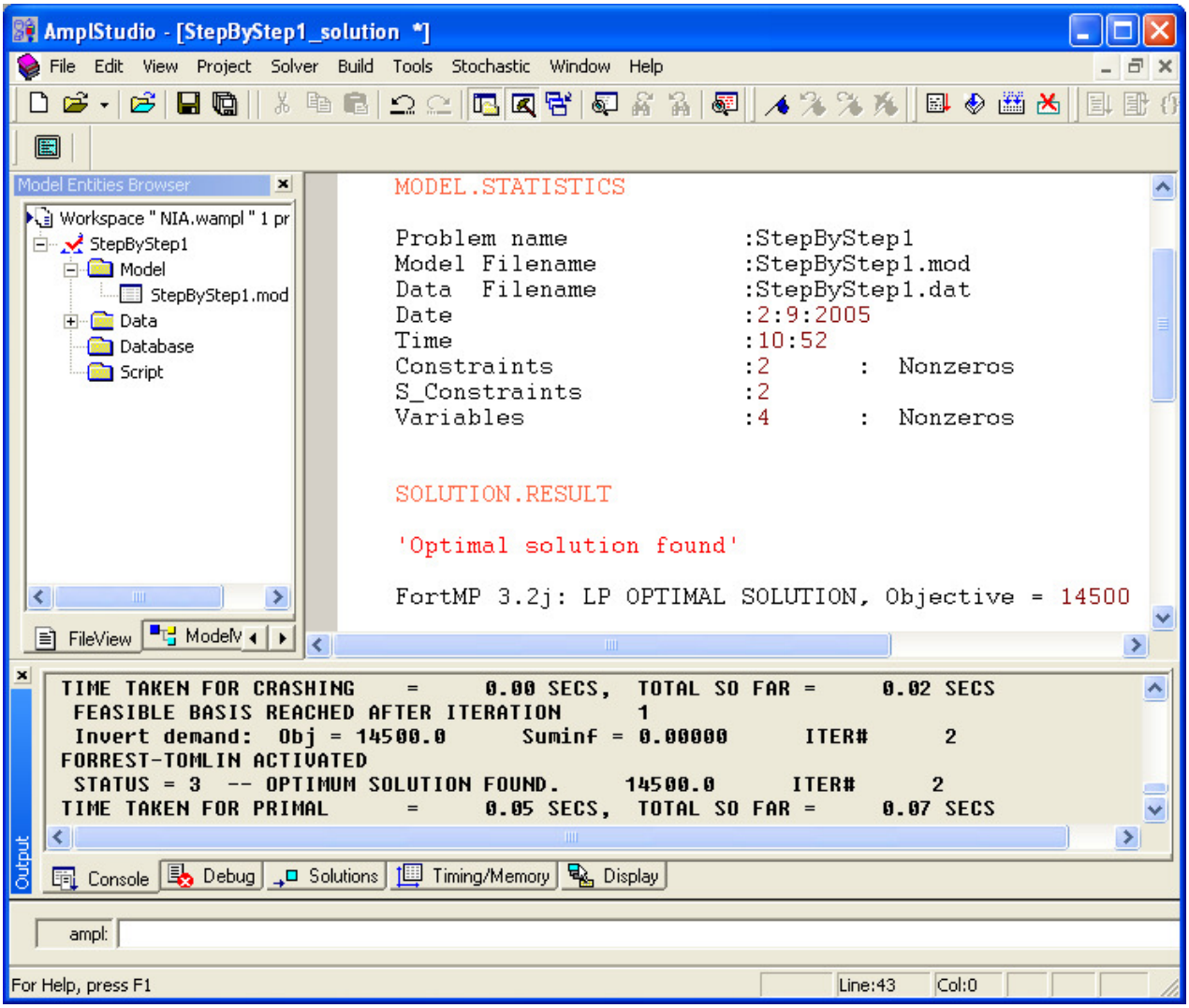


Step 2: Now you can run this problem by choosing the **Solve Problem** menu from the **Build** Menu.



Immediately the problem will be solved and the results will be displayed in the **Editing Area**.

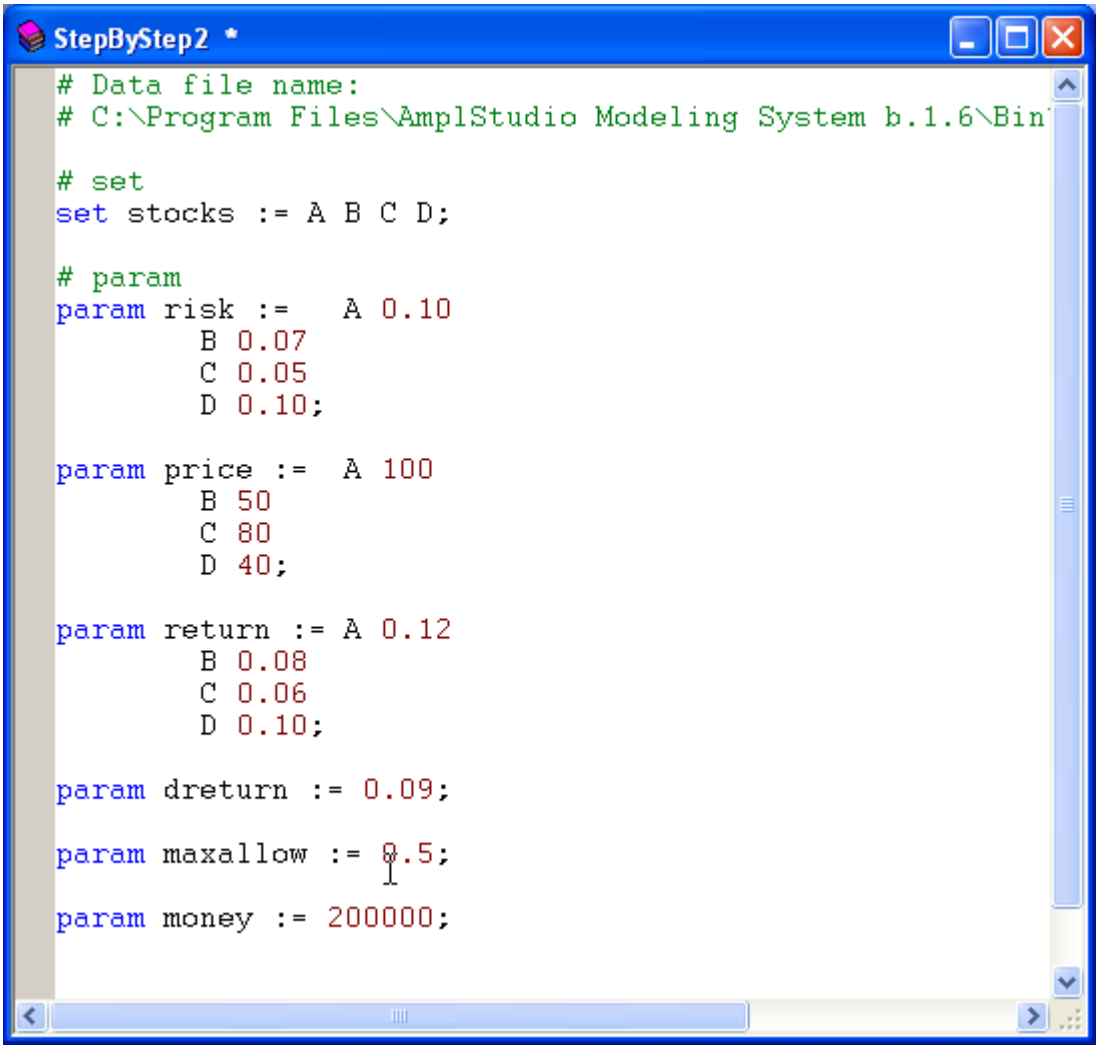
X

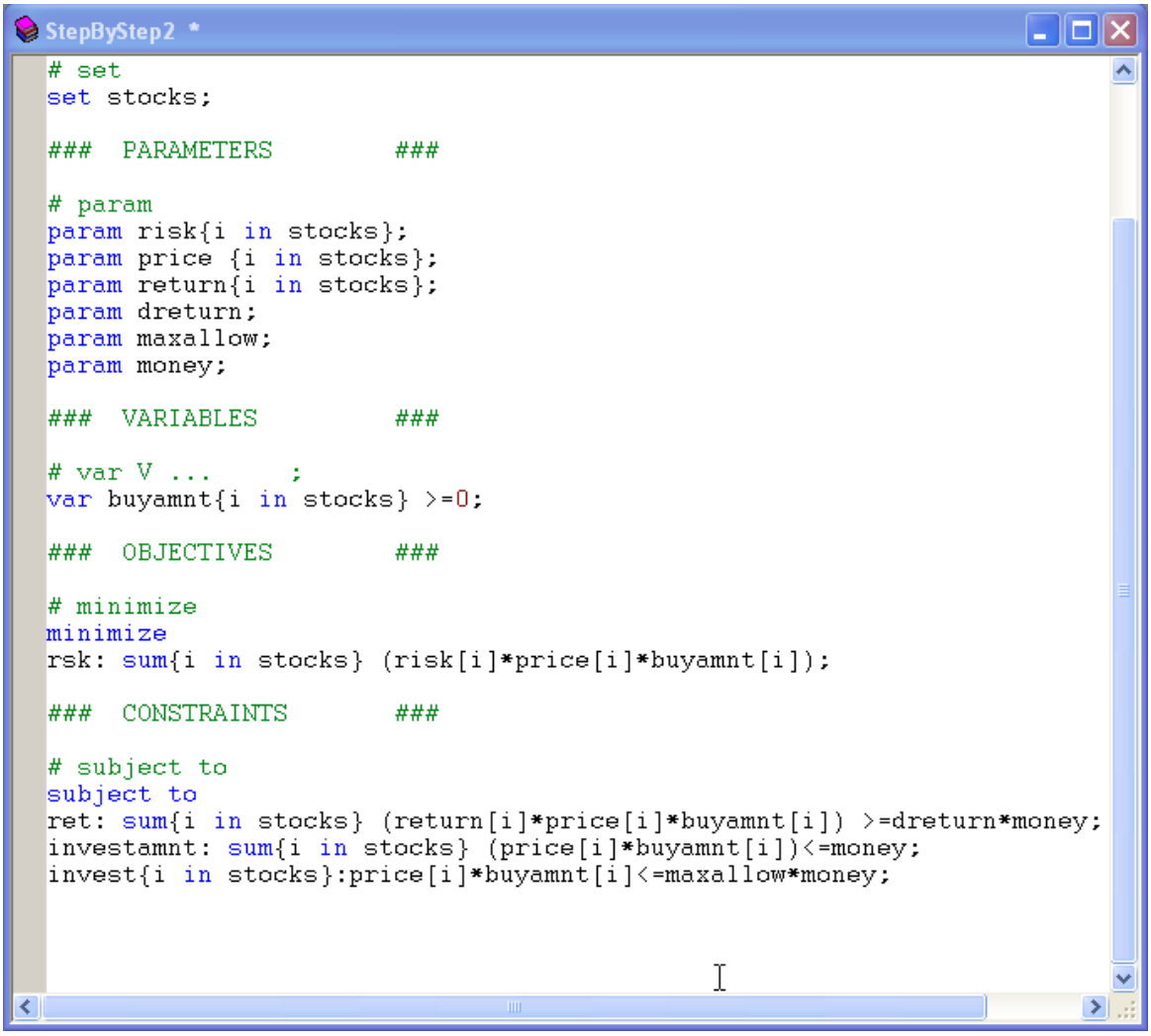


Enhance to Data Separated project

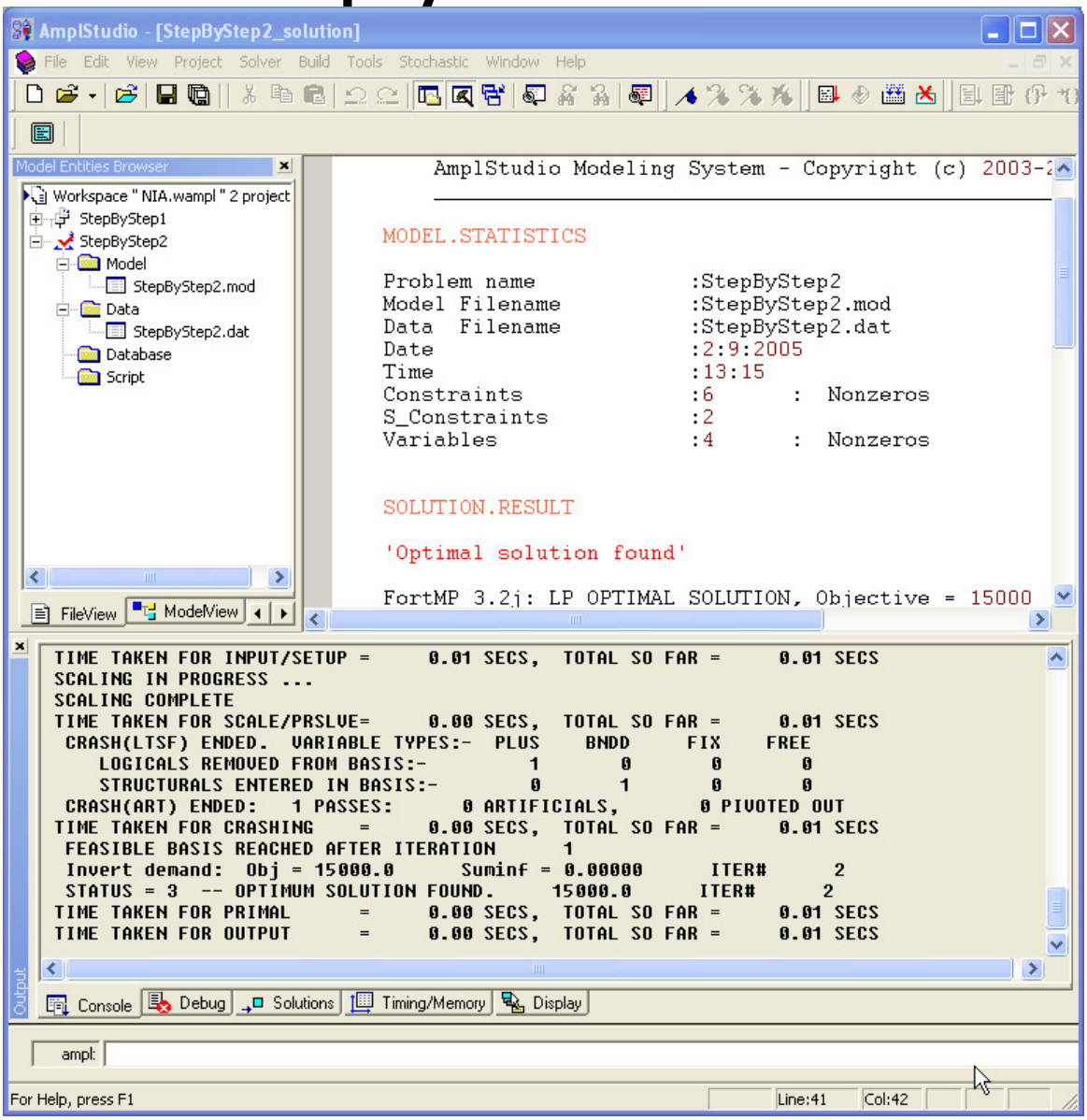
Creating Data and Model Files

The following is the investment problem exploiting structure.


StepByStep2 *
# Data file name:
# C:\Program Files\AmplStudio Modeling System b.1.6\Bin
# set
set stocks := A B C D;
# param
param risk := A 0.10
        B 0.07
        C 0.05
        D 0.10;
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 dreturn := 0.09;
param maxallow := 0.5;
param money := 200000;


StepByStep2 *
set
set stocks;
PARAMETERS
param
param risk{i in stocks};
param price {i in stocks};
param return{i in stocks};
param dreturn;
param maxallow;
param money;
VARIABLES
# var V ... ;
var buyamnt{i in stocks} >=0;
OBJECTIVES
minimize
minimize
rsk: sum{i in stocks} (risk[i]*price[i]*buyamnt[i]);
CONSTRAINTS
subject to
subject to
ret: sum{i in stocks} (return[i]*price[i]*buyamnt[i]) >=dreturn*money;
investamnt: sum{i in stocks} (price[i]*buyamnt[i])<=money;
invest{i in stocks}:price[i]*buyamnt[i]<=maxallow*money;

Solve and Display Results



Chapter 7: Connecting to a Database; Importing and Exporting

AMPL allows taking advantage of the structure of indexed data, which is closely related to the structure of relational tables commonly found in database applications. In AMPL Studio the user is able to exploit such feature and connect the models and/or projects to a database in order to work with relational data. In this chapter we will see how to create a database, how to import and export data, and how to solve and display the results using the created database.

Creating the Database

A relational database that exploits the structure of the algebraic model for our problem at hand must be composed of relational tables that reflect the model's indexing structure.

To go through the steps we will use as an example the "diet problem", which seeks to find the optimum mix of foods that satisfies some vitamins requirements. The algebraic representation for the diet problem using the AMPL syntax is shown below.

```
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];
```

The first set we find in our example is **FOOD**. Three parameters **cost**, **f_min**, and **f_max** are indexed over the set **FOOD**. Using this indexed structure we create a relational table, in which the key column will be the column corresponding to the values for the set **FOOD**.

FOOD	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

We can use an Excel spreadsheet to store such relational table, by just creating a range that includes the column names; in our example the range is called "**Foods**" (see Figure 7.1). The name of the range will be used subsequently when reading the data from the spreadsheet into the AMPL Studio model.

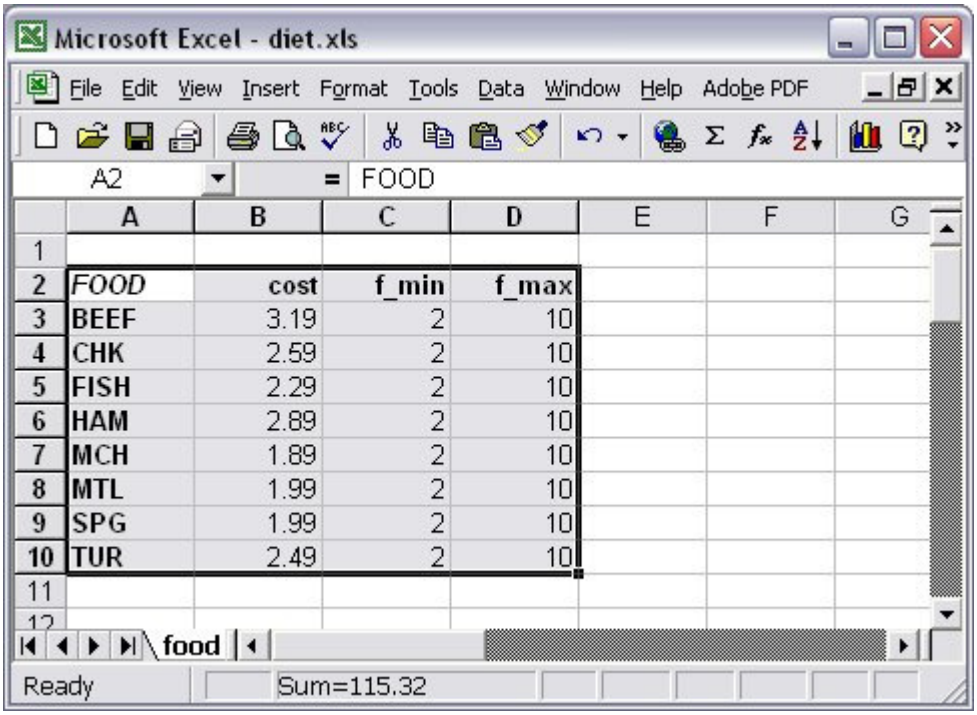


Figure 7.1: Excel range as relational table

In the same way we can create a second relational table with the set NUTR, which will be the key column, and the two parameters, n_min and n_max, which are indexed over the set NUTR.

NUTR	n_min	n_max
A	700	20000
C	700	20000
B1	700	20000
B2	700	20000
NA	0	50000
CAL	16000	24000

In the Excel spreadsheet we would then create a range, "Nutrients", that corresponds to this relational table (Figure 7.2).

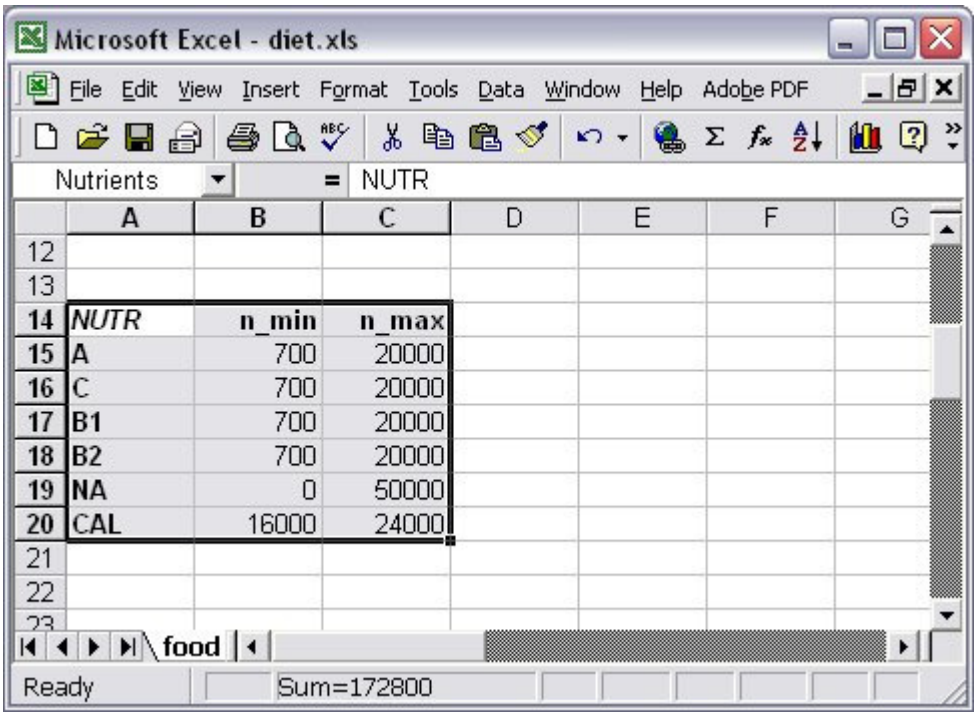
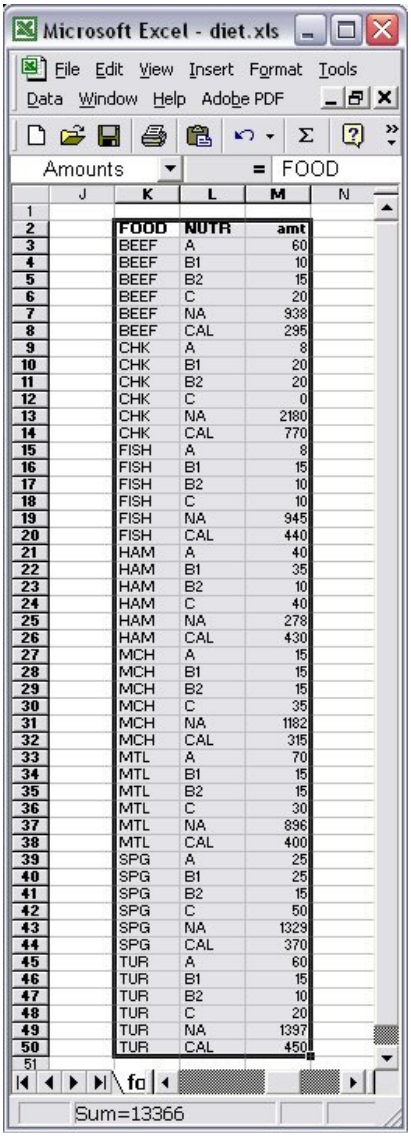


Figure 7.2: Excel range "Nutrients" as relational table

In a similar fashion a third relational table is created for the parameter **amt**, which is indexed over the two sets **NUTR** and **FOOD**. The following table has as key the two columns corresponding to the values for the sets **FOOD** and **NUTR**.

FOOD	NUTR	amt
BEEF	A	60
BEEF	B1	10
BEEF	B2	15
BEEF	C	20
BEEF	NA	938
BEEF	CAL	295
CHK	A	8
CHK	B1	20
CHK	B2	20
CHK	C	0
CHK	NA	945
CHK	CAL	770
FISH	A	8
FISH	B1	15
FISH	B2	10
…	…	…

The corresponding Excel range, "Amounts", would look like Figure 7.3.



	J	K	L	M	N
1					
2		FOOD	NUTR	amt	
3		BEEF	A	60	
4		BEEF	B1	10	
5		BEEF	B2	15	
6		BEEF	C	20	
7		BEEF	NA	938	
8		BEEF	CAL	295	
9		CHK	A	8	
10		CHK	B1	20	
11		CHK	B2	20	
12		CHK	C	0	
13		CHK	NA	2180	
14		CHK	CAL	770	
15		FISH	A	8	
16		FISH	B1	15	
17		FISH	B2	10	
18		FISH	C	10	
19		FISH	NA	945	
20		FISH	CAL	440	
21		HAM	A	40	
22		HAM	B1	35	
23		HAM	B2	10	
24		HAM	C	40	
25		HAM	NA	278	
26		HAM	CAL	430	
27		MCH	A	15	
28		MCH	B1	15	
29		MCH	B2	15	
30		MCH	C	35	
31		MCH	NA	1182	
32		MCH	CAL	315	
33		MTL	A	70	
34		MTL	B1	15	
35		MTL	B2	15	
36		MTL	C	30	
37		MTL	NA	896	
38		MTL	CAL	400	
39		SPG	A	25	
40		SPG	B1	25	
41		SPG	B2	15	
42		SPG	C	50	
43		SPG	NA	1329	
44		SPG	CAL	370	
45		TUR	A	60	
46		TUR	B1	15	
47		TUR	B2	10	
48		TUR	C	20	
49		TUR	NA	1397	
50		TUR	CAL	450	
51					

Figure 7.3: Excel range "Amounts" as relational table

In our **Diet.mod** model there are other entities indexed over the set **FOOD**, such as the variables:

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

Therefore, some assorted result expressions such as **Buy**, **Buy.rc**, **{j in FOOD} Buy[j]/f_max[j]**, can be included as output columns in our relational tables. In this case, we can include three new columns to the "**Foods**" range in our Excel spreadsheet, as in Figure 7.4. The last three columns **Buy**, **BuyRc**, and **BuyFrac**, will be then output columns that will be populated once the model is solved.

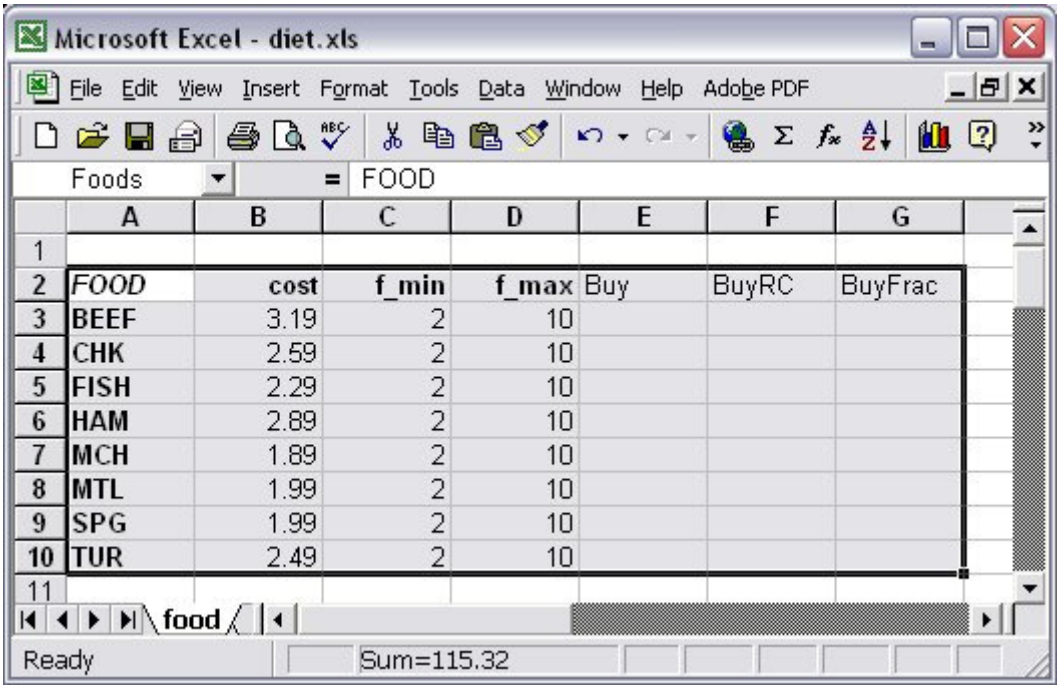


Figure 7.4: Excel range "Foods" with input and output columns

If we used an Access database to store our relational tables, the relational database for our example might look like Figure 7.5.

Figure 7.5: Access database for the Diet problem

As in the Excel spreadsheet case, we have three relational tables, Foods, Nutrients, and Amounts. The design of the Access relational tables is shown in Figure 7.6.

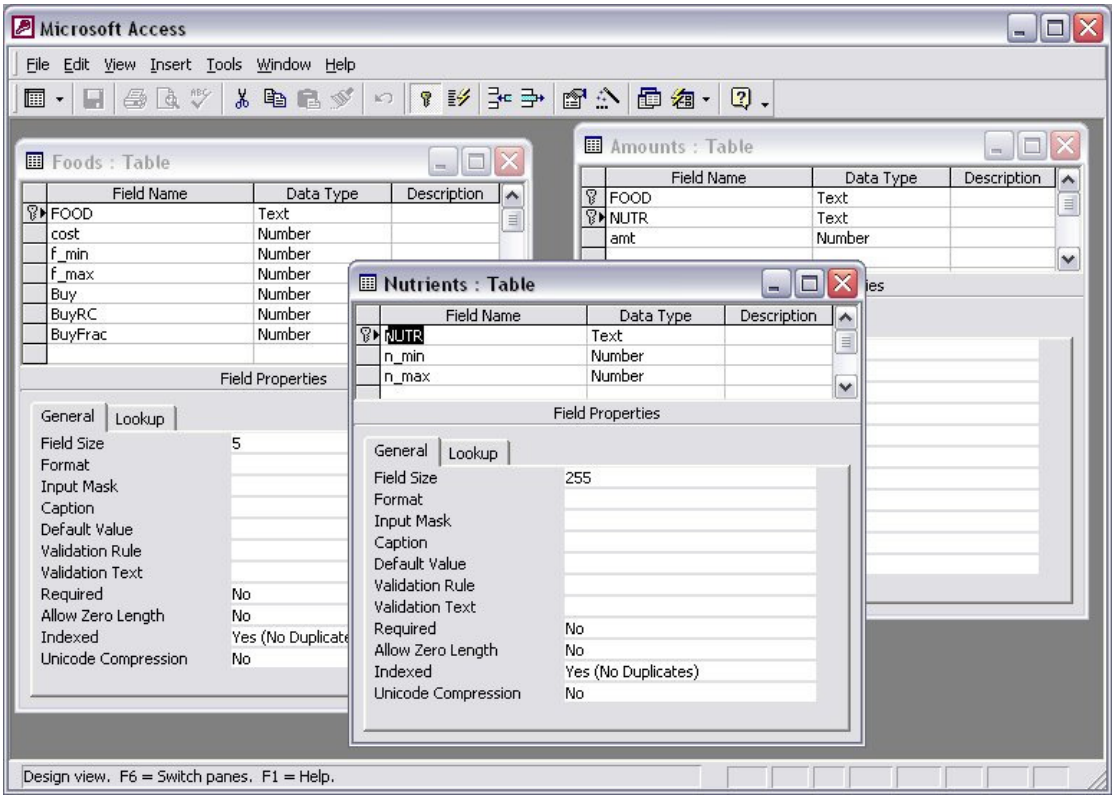
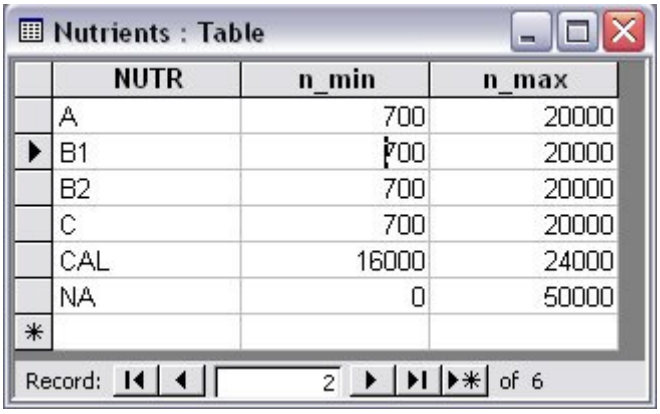


Figure 7.6: Access Data Tables Design for the Diet problem

In this case the relational data would be as below.

Figure 7.7: Access Relational Data in Foods table

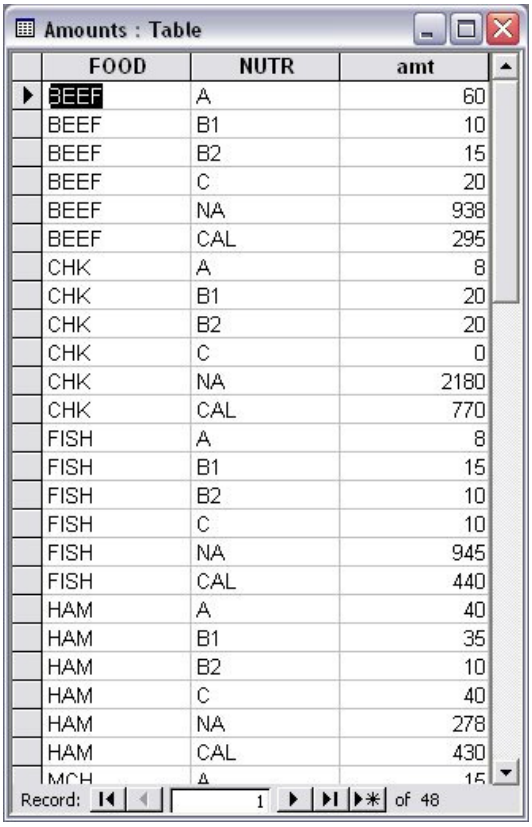


Nutrients : Table

	NUTR	n_min	n_max
	A	700	20000
▶	B1	700	20000
	B2	700	20000
	C	700	20000
	CAL	16000	24000
	NA	0	50000
*			

Record: 2 of 6

Figure 7.8: Access Relational Data in Nutrients table



Amounts : Table

	FOOD	NUTR	amt
▶	BEEF	A	60
	BEEF	B1	10
	BEEF	B2	15
	BEEF	C	20
	BEEF	NA	938
	BEEF	CAL	295
	CHK	A	8
	CHK	B1	20
	CHK	B2	20
	CHK	C	0
	CHK	NA	2180
	CHK	CAL	770
	FISH	A	8
	FISH	B1	15
	FISH	B2	10
	FISH	C	10
	FISH	NA	945
	FISH	CAL	440
	HAM	A	40
	HAM	B1	35
	HAM	B2	10
	HAM	C	40
	HAM	NA	278
	HAM	CAL	430
	MCH	A	15

Record: 1 of 48

Figure 7.9: Access Relational Data in Amounts table

Now that we have created the relational database, we will see how the relational tables are linked to the AMPL Studio model in order to import and export data from and to the database.

Importing data from tables

In order to use an external relational table, such as the tables created in the section above, for reading only, you should 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-list_opt :
      key-spec, data-spec, data-spec, … ;
```

Each table declaration has two parts. Before the colon, the declaration provides general information. The table-name is 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, i.e., AMPL will use these columns as input columns and will not write out to them. The optional string-list is specific to the database type and access method being used, and we will look into it in more detail in a later section.

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, which are surrounded by brackets […]. The data-spec gives the data columns. Data values are subsequently read from the table into AMPL entities by the command

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

The `table` declaration only defines a correspondence. To read values from columns of a relational table into AMPL sets and parameters, it is necessary to give an explicit `read table` command.

For instance, in our Diet problem example, when we want to read the data from the table "Nutrients", we would use the following declaration followed by the `read` command:

```
table dietNutrs IN "ODBC" "TABLES/diet.xls" "Nutrients":
      NUTR <- [NUTR], n_min, n_max;

read table dietNutrs;
```

In our example the string-list "**ODBC**" "**TABLES/diet.xls**" "**Nutrients**" specifies that we are connecting to the external relational database through an Open Database Connection (ODBC). It also specifies the external file, in this case an Excel spreadsheet "**diet.xls**" located in the directory "**TABLES**". The string "**Nutrients**" gives the name of the relational table we are declaring. In the second part of the declaration we find the expression **NUTR <- [NUTR]**, which indicates that the entries in the key column **NUTR** are to be copied into AMPL to define the members of the set **NUTR**. The expressions **n_min** and **n_max** are the names of the other two columns in the relational table from which we will read the values into the parameters **n_min** and **n_max**.

The table-name may be different from the name of the corresponding table within the external relational database. In any case, the table-name should be the same in both, the table declaration and the read command.

In a similar way we can read the data from the "**Amounts**" relational table

```
table dietAmts IN "ODBC" "TABLES/diet.xls" "Amounts":
      [NUTR, FOOD], amt;
```

```
read table dietAmts;
```

Reading parameters only

To assign values from data columns to like-named AMPL parameters, it suffices to give a bracketed list of key columns and hen a list of data columns. In our Diet problem example, in the simplest case where there is only one key column we could write

```
table Foods IN "ODBC" "TABLES/diet.xls":
      [FOOD], cost, f_min, f_max;

read table Foods;
```

In the same way, when we want to read multidimensional parameters, the name of each data column must also be the name of an AMPL parameter, and the dimension of the parameter's indexing set must equal the number of key columns.

```
table Amounts IN "ODBC" "TABLES/diet.xls":
      [NUTR, FOOD], amt;

read table Amounts;
```

	The subscripts given by the key column entries must be valid for the parameters when the values of these parameters are first needed by AMPL, but the parameters need not be declared over sets named as the key columns

Values of unindexed (scalar) parameters may be supplied by a relational table that has one row and no key columns, so that each data column contains exactly one value. The corresponding table declaration has an empty key-spec, [].

Reading a set and parameters

We can read the members of a set form a table's key column or columns, at the same time that parameters indexed over that set are read from the data columns. To indicate that a set should be read from a table, the key-spec in the table declaration is written in the form

```
Set-name <- [key-col-spec, key-col-spec,…]
```

The simplest case involves reading a one-dimensional set and the parameters indexed over it. In our Diet problem example we have

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

In this particular case, since the key column [**FOOD**] is named like the AMPL set **FOOD**, the table declaration could be abbreviated to

```
table Foods IN "ODBC" "TABLES/diet.xls":
       [FOOD] IN , cost, f_min, f_max;
```

For the multidimensional case, an analogous syntax is used fir reading a multidimensional set along with parameters indexed over it.

Let's suppose we had in our **Diet.mod** the following sets and parameters:

```
set FOOD;
set NUTR;
set PAIR within {FOOD, NUTR};
…
param amt {PAIR} >=0;
```

In this case we would have a table declaration that might look like

```
table Amounts IN "ODBC" "TABLES/diet.xls":
      PAIR <- [NUTR, FOOD], amt;
```

Establishing correspondences

Sometimes the AMPL model's set and parameter declarations do not necessarily correspond in all respects to the organization of tables in the external relational databases.

One of the most common differences appears in the names amongst the sets and parameters and the corresponding columns in the relational tables. A table declaration can associate a data column with a differently named AMPL parameter by use of a data-spec of the form

```
param-name ~ data-col-name
```

In our Diet problem example, if we had the following `table` declaration

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

We would be saying that the AMPL parameters **f_min** and **f_max** would be read from the data columns **lowerlim** and **upperlim** in the relational table respectively.

In a similar way, when the AMPL index is not named as the corresponding column in the relational table, we would have

```
index ~ key-col-name
```

This index may then be used in a subscript to the optional param-name in one or more data-specs.

Three common cases where we can benefit from this correspondence are as follow.

Case 1: as an example, the time periods are counted from 0 in the relational table, but in the model the time periods start counting from 1:

```
table tableName IN:
      [p ~ PROD, t ~ TIME],
      market[p,t+1] ~ market, revenue[p,t+1] ~ revenue;
```

Case 2: the AMPL parameters have subscripts from the same sets but in different orders. In this case key column indexes must be used to provide a correct index order:

For example, we have in the AMPL model

```
param market {PROD, 1..T};
param revenue {1..T, PROD};
...
```

we could have a `table` declaration as follows

```
table tableName IN:
      [p ~ PROD, t ~ TIME],
      market, revenue[t, p] ~ revenue;
```

Case 3: the values for an AMPL parameter are divided among several database columns. In this case key column indexes can be used to describe the values to be found in each column:

For example, if we have the revenue values given in two columns, one for "p1" and in another column for "p2", the `table` declaration would be as follows

```
table tableName IN:
      [t ~ TIME],
      revenue["p1", t] ~ revenuep1,
      revenue["p2", t] ~ revenuep2;
```

Reading other values

Any assignable expression, such as a variable name, a constraint name, a variable or constraint qualified by an assignable suffix, may appear anywhere that a parameter name would be allowed. Therefore, any assignable expression can appear in a table declaration.

Note	An expression is *assignable* if it can be assigned a value, such as by placing it on the left hand side of `:=` in a `let` command.

In our Diet problem example we could have the following `table` declaration

```
table Foods IN:
      FOOD IN, cost, f_min, f_max, Buy, Buy.priority ~ prior;
```

where we are reading from the table Foods the initial values for the Buy variables, as well as their branching priorities.

Exporting data into tables

In order to use an external relational table for writing only, you should employ a `table` declaration that specifies a read/write status of `OUT`. The general form of this kind of declaration is

```
table table-name OUT string-list_opt :
      key-spec, data-spec, data-spec, … ;
```

As for the case in which we read data from the table, each table declaration has two parts. Before the colon, the declaration provides general information. The table-name is the name by which the table is known within AMPL. The keyword `OUT` states that the default for all non-key table columns will be write-only, i.e., AMPL will use these columns as output columns and will not read from them. The optional string-list is specific to the database type and access method being used, and we will look into it in more detail in a later section.

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, which are surrounded by brackets […]. The data-spec gives the data columns. Data values are subsequently written to the table by the command

`write table` table-name;

Note	Depending on the circumstances, the **write table** command may create a new external file or table, overwrite an existing table, overwrite certain columns within an existing table, or append columns to an existing table.

This way the **write table** command allows writing meaningful results back to the external relational database once the model has been solved.

The key-specs and data-specs in the `table` declaration for writing external tables resemble those for reading. Nevertheless, the range of AMPL expressions

allowed when writing is much broader, including essentially all set-valued and numeric-valued expressions. Moreover, whereas the table rows to be read are those of some existing table, the rows to be written must be determined from AMPL expressions in some part of a `table` declaration. Specifically, rows to be written can be inferred either from the data-specs, or from the key-spec. Each of these alternatives uses a different syntax.

Writing rows inferred from the data specifications

If the key-spec is simply a bracketed list of the names of key columns,

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

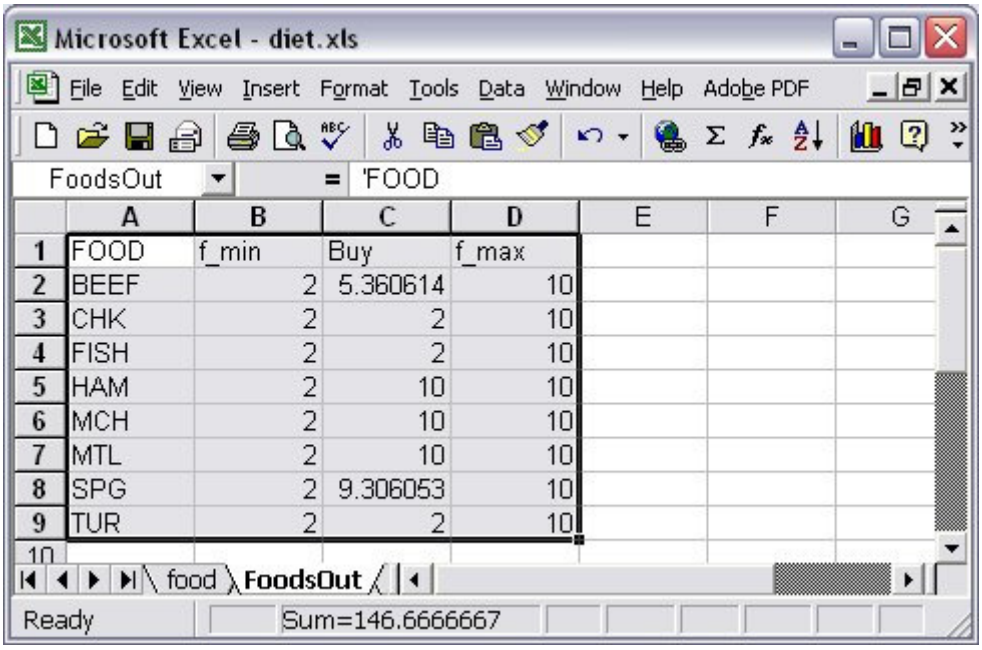
then the `table` declaration works similar to the `display` command, except that all the items listed in the data-specs must have the same dimension.

In the simplest case, the data-specs are the names of model components indexed over the same set.

For instance, in our Diet problem example, the `table` declaration and the `write table` command

```
table Foods OUT "ODBC" "TABLES/diet.xls" "FoodsOut":
[FOOD], f_min, Buy, f_max;
…
write table Foods;
```

would have as a result a new range named "FoodsOut" as shown in Figure 7.10.



	A	B	C	D	E	F	G
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			
10							

Figure 7.10: Output table range "FoodsOut" in Excel

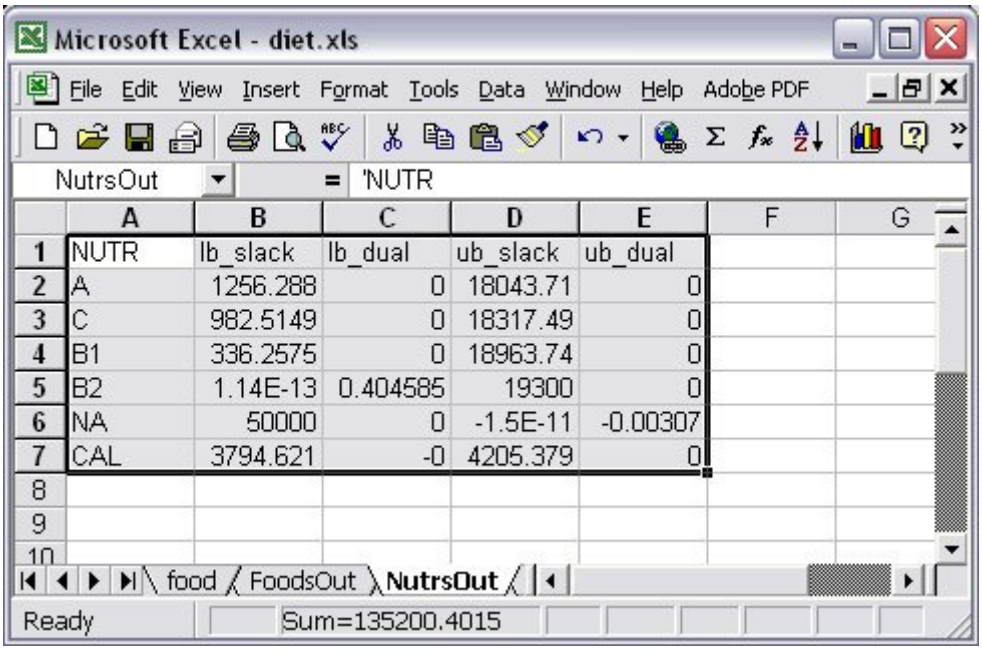
Tables of higher-dimensional sets are handled in the same way, with the number of bracketed key-column names listed in the key-spec being equal to the dimensions of the items in the data-spec.

We could also write out to a relational table suffixed variables or constraint names, such as the dual and slack values.

In our Diet problem example, we could for instance write out the dual and slack values related to the constraint "**diet**":

```
table Nutrients OUT "ODBC" "TABLES/diet.xls" "NutrsOut":
      [NUTR],
      diet.lslack ~ lb_slack, diet.ldual ~ lb_dual,
      diet.uslack ~ ub_slack, diet.udual ~ ub_dual;
…
write table Nutrients;
```

which would have as a result a new relational table "**NutrsOut**" in our Excel Spreadsheet **diet.xls**, as shown in Figure 7.11.



	A	B	C	D	E	F	G
1	NUTR	lb_slack	lb_dual	ub_slack	ub_dual		
2	A	1256.288	0	18043.71	0		
3	C	982.5149	0	18317.49	0		
4	B1	336.2575	0	18963.74	0		
5	B2	1.14E-13	0.404585	19300	0		
6	NA	50000	0	-1.5E-11	-0.00307		
7	CAL	3794.621	-0	4205.379	0		
8							
9							
10							

Figure 7.11: Output table range "NutrsOut" in Excel

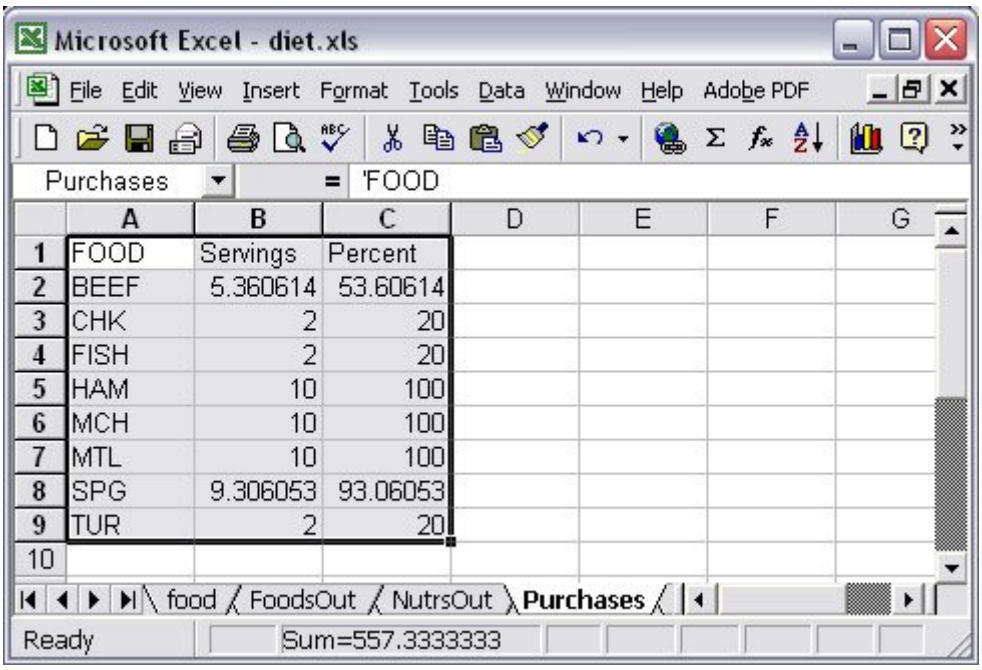
More general expressions for the values in data columns can also be used. Since indexed AMPL expressions are rarely valid column names for a database, they should generally be followed by ~ data-col-name to provide a valid name for the corresponding data table column.

For instance, we could have in our Diet problem example the following table declaration:

```
table Purchases OUT "ODBC" "TABLES/diet.xls":
      [FOOD],
```

```
        Buy ~ Servings,
        {j in FOOD} 100*Buy[j]/f_max[j] ~ Percent;
…
write table Purchases;
```

The resulting relational table is displayed in Figure 7.12.



	A	B	C	D	E	F	G
1	FOOD	Servings	Percent				
2	BEEF	5.360614	53.60614				
3	CHK	2	20				
4	FISH	2	20				
5	HAM	10	100				
6	MCH	10	100				
7	MTL	10	100				
8	SPG	9.306053	93.06053				
9	TUR	2	20				
10							

Figure 7.12: Output table range "Purchases" in Excel

Note	The expression in a data-spec may also use operators like `sum` that define their own dummy indices.

Writing rows inferred from a key specification

We can also use table declarations to write one table row for each member of an explicit specified AMPL set. In this case the key-spec must be of the form

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

This form uses an arrow pointing from left to right, i.e., pointing from an AMPL set to a key column list, indicating that the information will be written from the set into the key columns.

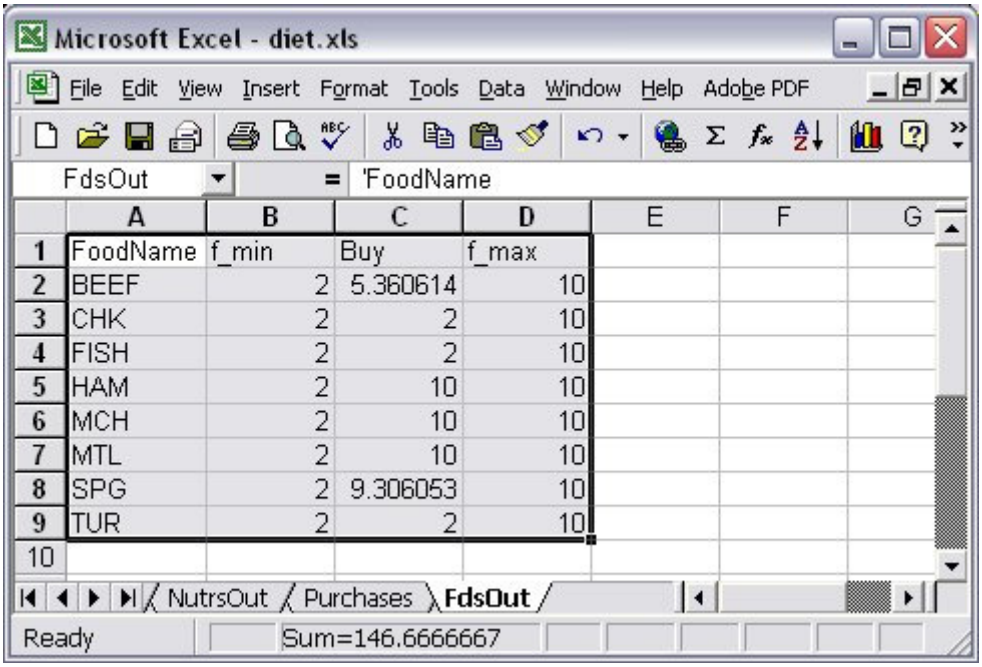
The set-spec is composed of an explicit expression, such as the name of an AMPL set, or any other AMPL set-expression enclosed in braces { }. The key-col-spec gives the names of the corresponding key columns in the database.

The simplest case of this form would be writing database columns for model components indexed over the same one-dimensional set.

In our Diet problem example, we could have

```
table FdsOut OUT "ODBC" "TABLES/diet.xls":
      FOOD -> [FoodName], f_min, Buy, f_max;
…
write table FdsOut;
```

giving the relational table shown in Figure 7.13.



	A	B	C	D	E	F	G
1	FoodName	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			
10							

Figure 7.13: Output table range "FdsOut" in Excel

or in case we wanted the same name for the table as for the set, we could have written the declaration as

```
table FdsOut OUT "ODBC" "TABLES/diet.xls":
      [FOOD] OUT, f_min, buy, f_max;
```

Importing From and Exporting To the Same Table

In the previous sections you have learnt 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 you want to use the same external relational table for both actions, import and export data. In this case you could use two separate `table` declarations, one to read data, and a second declaration to write data.

You may also combine these two declarations into one that specifies some columns to be read and some columns to be written into.

Importing and exporting data using two table declarations

The same external relational table can be read by one `table` declaration and a `read table` command, and later on it can be written by another `table` declaration and a `write table` command. These two `table` declarations follow the syntax and rules described in the previous sections.

Note	Even though you can use two different `table` declarations, one to read and another one to write the same external relational table, the AMPL table-name should be different in both `table` declarations.

In our Diet problem example, we can have an external relational table "**Foods**" with some columns that contain input for the model, and other columns that will contain results.

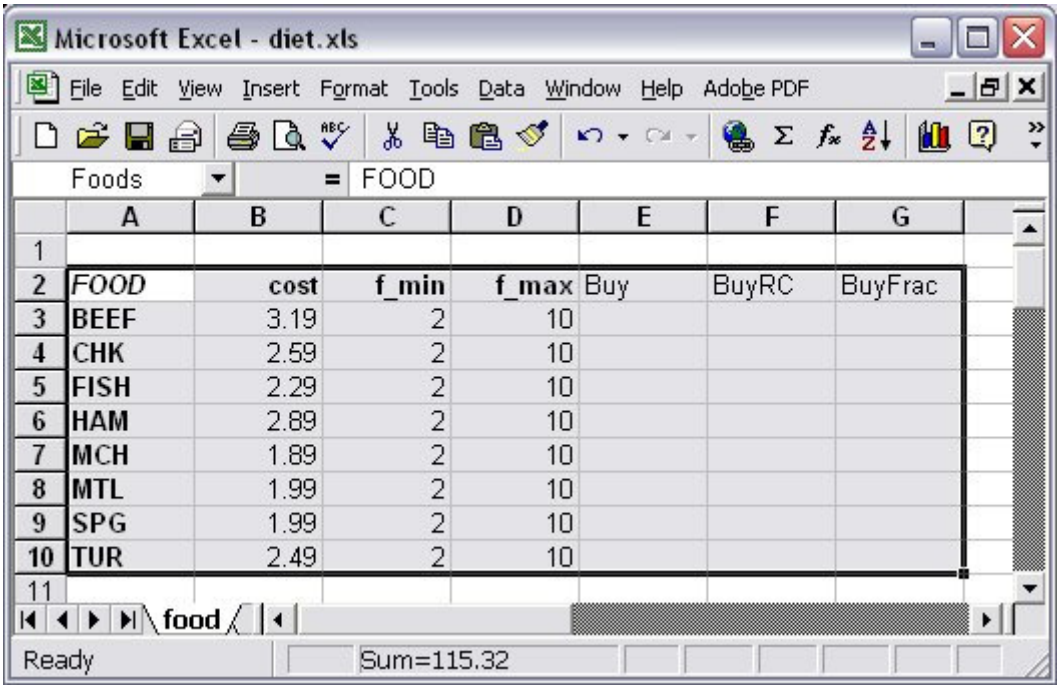
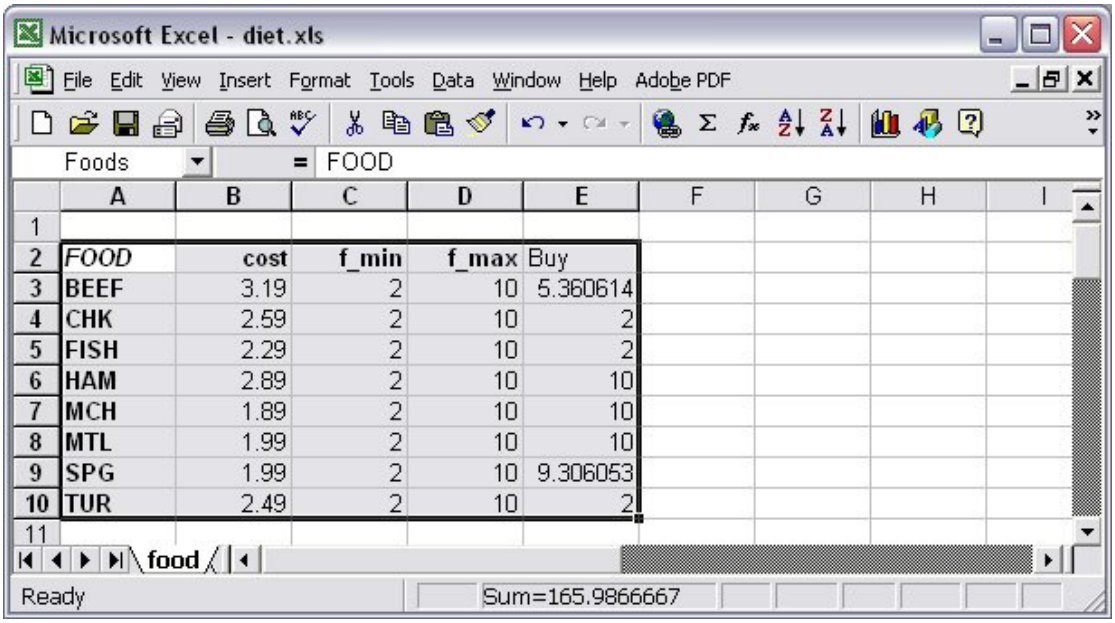


Figure 7.14: Excel range "Foods" with input and output columns

For instance, in Figure 7.14 we have the columns **cost**, **f_min**, and **f_max** as input columns, whereas the columns **Buy**, **BuyRC**, and **BuyFrac** are output columns. This relational table would correspond to the following `table` declarations:

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

table outputFoods "ODBC" "TABLES/diet.xls" "Foods":
      [FOOD], Buy;
```



	A	B	C	D	E
1					
2	*FOOD*	cost	f_min	f_max	Buy
3	BEEF	3.19	2	10	5.360614
4	CHK	2.59	2	10	2
5	FISH	2.29	2	10	2
6	HAM	2.89	2	10	10
7	MCH	1.89	2	10	10
8	MTL	1.99	2	10	10
9	SPG	1.99	2	10	9.306053
10	TUR	2.49	2	10	2
11					

Figure 7.15: Input/Output table range "Foods" in Excel

The user should be careful when using two separate `table` declarations for input and output from the same table:

We could have also used the following table declarations:

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

table outputFoods OUT "ODBC" "TABLES/diet.xls" "Foods":
      [FOOD], Buy;
```

or similarly

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

table outputFoods "ODBC" "TABLES/diet.xls" "Foods":
            [FOOD], Buy OUT;
```

In this case all the data columns in the external relational table "Foods" would have been deleted by the write table outputFoods command, and you would only find the columns specified in the outputFoods table declaration, i.e., the "**FOOD**" and "**Buy**" columns:

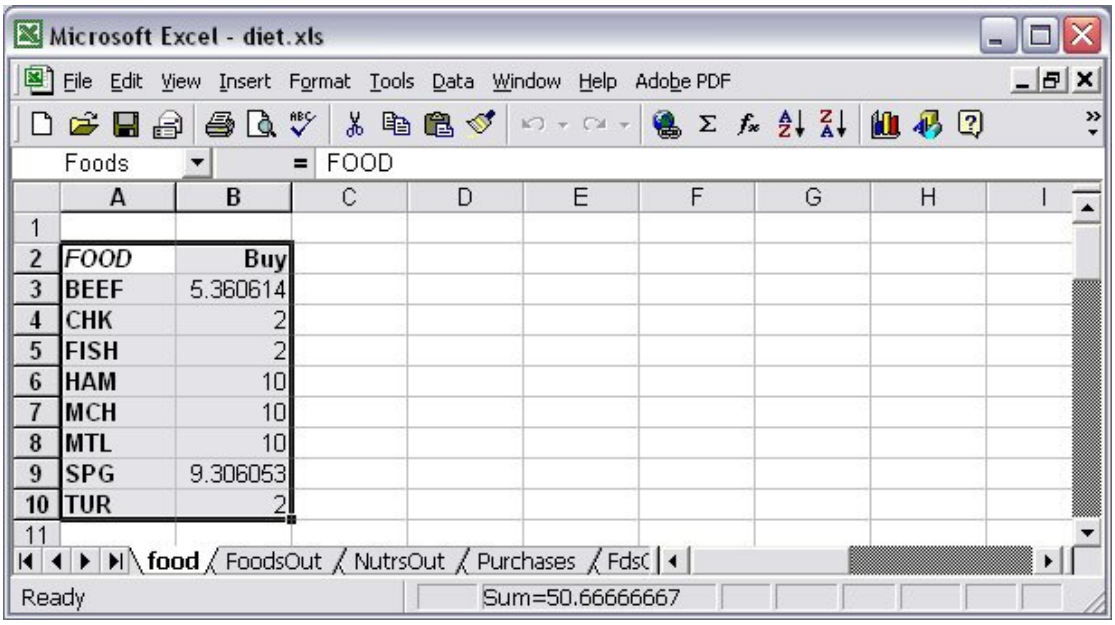


Figure 7.16: Input/Output table "Foods" if rewriting all columns

Note	The general convention is that overwriting of an entire existing table or file is intended only when all the data columns in the `table` declaration have read/write status `OUT`. Selective rewriting or addition of columns is intended otherwise.

Reading and writing using the same table declaration

In many cases a single `table` declaration suffices to read and write the same external relational table.

The key-spec may use the arrow <- to read contents of the key columns into an AMPL set, or use the arrow -> to write members of an AMPL set into the key columns, or even <-> to do both.

A data-spec may specify read/write status `IN` for the columns that will only be read into AMPL, status `OUT` for the columns that will only be written out from AMPL, or status INOUT for the columns that will be both read and written.

Note	The default read/write status for a column in a `table` declaration is `INOUT`.

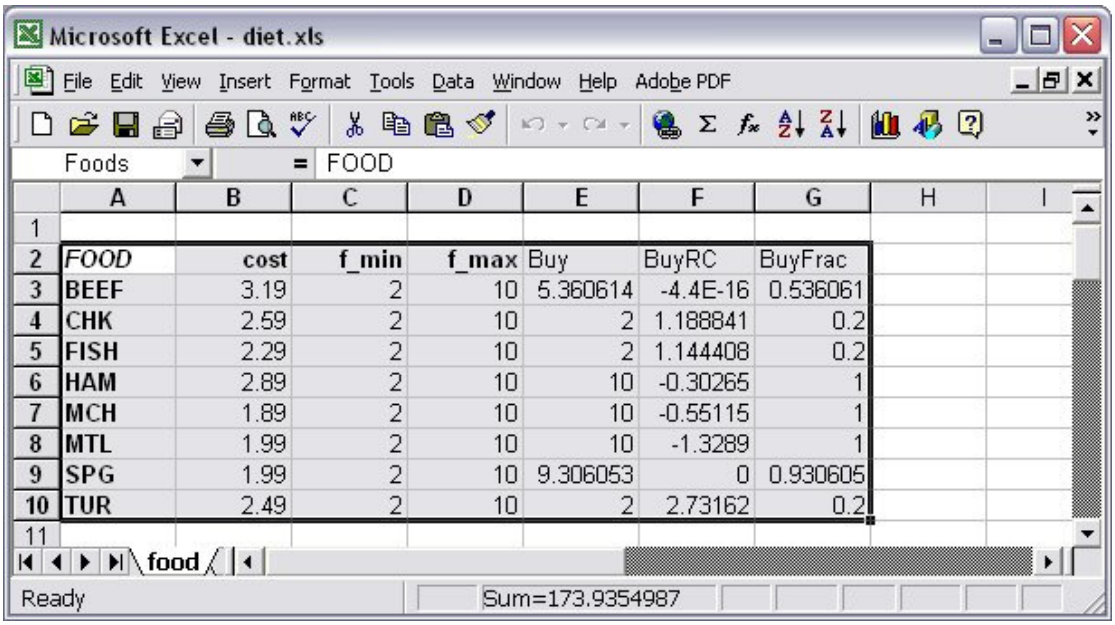
The **read table** command related to such combined table declaration will read only the keys or data columns that are specified in the table declaration with **IN** or **INOUT** read/write status.

The **write table** command related to such combined table declaration will write only the keys or data columns that are specified in the table declaration with **OUT** or **INOUT** read/write status.

In our Diet problem example, we could use the following `table` declaration to read and write the Foods table:

```
table dietFoods "ODBC" "TABLES/diet.xls" "Foods":
      FOOD <- [FOOD],
      cost IN, f_min IN, f_max IN,
      Buy OUT,
      Buy.rc ~ BuyRC OUT,
      {j in FOOD} Buy[j]/f_max[j] ~ BuyFrac;
…
read table dietFoods;
…
write table dietFoods;
```

and we would obtain the table as in Figure 7.17.



	A	B	C	D	E	F	G
1							
2	*FOOD*	**cost**	**f_min**	**f_max**	Buy	BuyRC	BuyFrac
3	BEEF	3.19	2	10	5.360614	-4.4E-16	0.536061
4	CHK	2.59	2	10	2	1.188841	0.2
5	FISH	2.29	2	10	2	1.144408	0.2
6	HAM	2.89	2	10	10	-0.30265	1
7	MCH	1.89	2	10	10	-0.55115	1
8	MTL	1.99	2	10	10	-1.3289	1
9	SPG	1.99	2	10	9.306053	0	0.930605
10	TUR	2.49	2	10	2	2.73162	0.2
11							

Figure 7.17: Input/Output table "Foods" using one table declaration

Index Collections of Tables and Columns

Sometimes it is convenient to declare an indexed collection of tables, or to define an indexed collection of data columns within a table. This can be done with the `table` declaration.

Indexed collections of tables

The table declarations can be indexed by following the table-name by an optional {indexing-expr}:

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

In this case one table is defined for each member of the set specified by the indexing-expr. Individual tables in this collection are denoted by appending a bracketed subscript or subscripts to the table-name.

For instance, in our **Diet** problem example, we could create one different table in our external relational database for each value of the set **FOOD**:

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

Which will have as a result the creation of one table per j in FOOD:

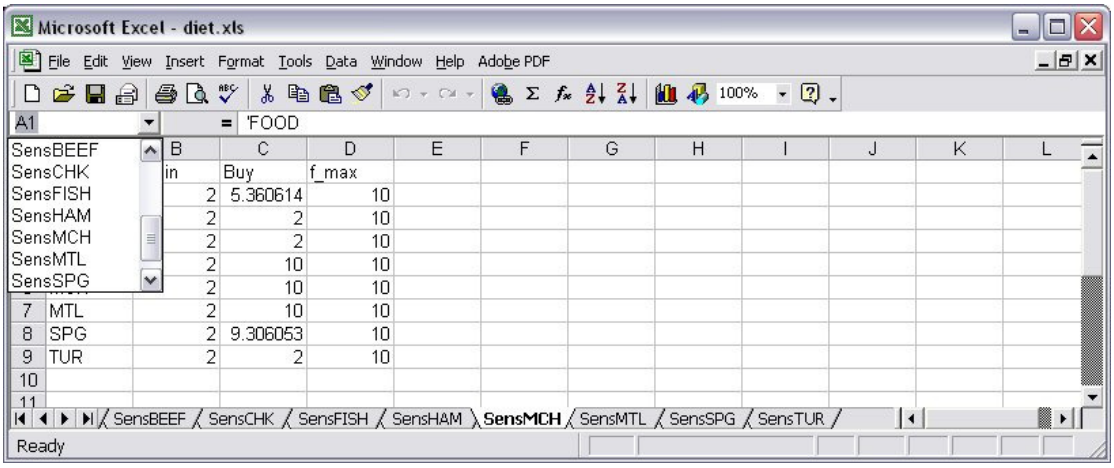


Figure 7.18: Tables collection

Note	You could also create a collection of databases if the `table` declaration were to give a string expression for the second string in the string-list.
	e.g.,
	`table DietSens {j in FOOD}`
	`OUT "ODBC" ("TABLES/diet" & j & ".xls"):`
	`[FOOD], f_min, Buy, f_max;`
	
	This `table` declaration would create a different Excel spreadsheet for each value in the set FOOD.
	
	In the same way you could make correspond every member of an indexed collection of AMPL tables to a different data-col-name within the same external database, and same relational table:

	table DietSens {j in FOOD} “ODBC” "TABLES/diet.xls": [FOOD], Buy ~ (“Buy” & j); This `table` declaration would create a different column for each member of the set FOOD, within the same table DietSens.

Indexed collections of data columns

Due to the natural correspondence between data columns of a relational table and indexed collections of entities in an AMPL model, each data-spec in a `table` declaration normally refers to a different AMPL parameter, variable or expression. However, occasionally the values for one AMPL entity are split among multiple data columns. In this case we can define a collection of data columns, one for each member of a specified indexing set.

The general form for specifying an indexed collection of table columns is the following

```
{indexing-expr} < data-spec, data-spec, … >
```

Each data-spec has any of the forms previously seen.

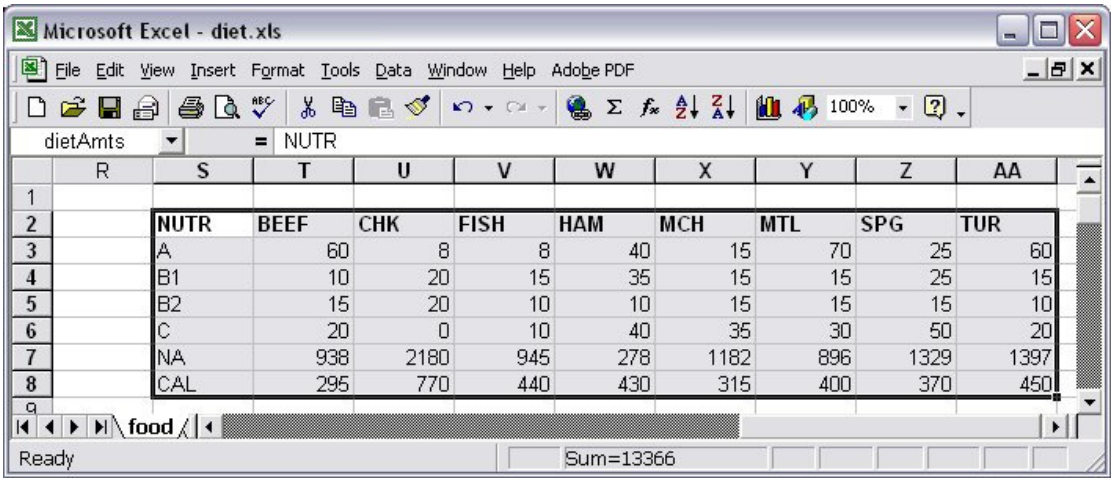
For each member of the set specified by the indexing-expr, AMPL generates one copy of each data-spec within the angle brackets <…>.

The indexing-expr also defines one or more dummy indices that run over the index set. These indices are used in expressions within the data-specs, and also appear in string expressions that give the names of columns in the external database.

In our Diet problem example, if we have the following table declaration:

```
table dietAmts IN “ODBC” “TABLES/diet.xls”:
      [i ~ NUTR], {j in FOOD} < amt[i,j] ~ (j) >;
```

The key-spec [i ~ NUTR] associates the first table column with the set NUTR. The data-spec {j in FOOD} <…> causes AMPL to generate an individual data-spec for each member of the set FOOD. The result would be as displayed in Figure 7.19.



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

Figure 7.19: Indexed collection of data columns

A similar approach works for writing two-dimensional tables.

Standard and Built-in Table Handlers

To work with external database files, AMPL relies on table handlers. These are add-ons, usually in the form of shared or dynamic link libraries that can be loaded as needed. AMPL Studio is distributed with a "standard" table handler that runs under Microsoft Windows and communicates via the Open Database Connectivity (ODBC) application programming interface; it recognizes relational tables in the formats used by Access, Excel, and any other application for which and ODBC driver exists on your computer.

As you have seen in the previous examples, AMPL communicates with handlers through the string-list in the `table` declaration. The form and interpretation of the string-list are specific to each handler.

The general form of the string-list in a `table` declaration for the standard ODBC table handler is

$$\text{“ODBC” “connection-spec” “external-table-spec”}_{\text{opt}}\ \text{“verbose”}_{\text{opt}}$$

The string "ODBC" indicates that data transfers using this table should employ the standard ODBC handler. The connection-spec identifies the database file name that will be read or written.

	If the connection-spec is a filename of the form name.ext, where ext is a 3-letter extension associated with an installed ODBC driver, then the named file is the database file. Other forms of connection-spec are more specific to ODBC.

The external-table-spec normally gives the name of the relational table, within the specified file that is to be read or written. As we have seen previously, if the table name is omitted, then the name of the relational table is taken to be the same as the table-name of the containing `table` declaration.

The string verbose is used to request diagnostic messages, such as the DSN= string that ODBC reports using.

	The external-table-spec could have the special form "SQL=sql-query" In such case, the table declaration applies to the relational table that is temporarily created by a statement in the Structure Query Language (SQL). All the columns specified in the table declaration should have the read/write status IN, since it does not make sense to write to a temporary table.

Using our Diet problem example, three common table-handling statements would be as follows:

Case 1: For a Microsoft Access table in a database file **diet.mdb** located in the TABLES directory:

```
Table Foods IN "ODBC" "TABLES/diet.mdb" :
      FOOD <- [FOOD], cost, f_min, f_max;
```

Case 2: For a Microsoft Excel table in a database file **diet.xls** located in the TABLES directory:

```
Table Foods IN "ODBC" "TABLES/diet.xls" :
      FOOD <- [FOOD], cost, f_min, f_max;
```

Case 3: For an ASCII text table in a file **Foods.dat** located in the TABLES directory:

```
Table Foods IN "TABLES/Foods.dat":
      FOOD <- [FOOD], cost, f_min, f_max;
```

Note	Where no details are given, the table is read by default from the ASCII text file using AMPL's built-in text table handler. For these built-in table handlers for text and binary files, the table declaration's string-list contains at most one string identifying the external file that contains the relational table. If the string has the form "filename.tab" the file is considered to be an ASCII text file. If the string has the form "filename.bit" the file is considered to be a binary text file. If no string-list is given, a text file table-name.tab is assumed.

Solve and Display Results

After solving our Diet problem example we obtain the following solution file.

```
AmplStudio Modeling System - Copyright (c) 2003-2004, Datumatic Ltd
_____________________________________________________________________
MODEL.STATISTICS

      Problem name                  :diet
      Model Filename                :Diet.mod
      Data  Filename                :Diet2a.dat
      Date                          :1:9:2005
      Time                          :20:5
      Constraints                   :6          :  Nonzeros
      S_Constraints                 :6
      Variables                     :8          :  Nonzeros

SOLUTION.RESULT

      'Optimal solution found'
      FortMP 3.2j: LP OPTIMAL SOLUTION, Objective = 118.0594032

DECISION.VARIABLES
```

```
 Name              Activity          .uc        Reduced Cost
------------------------------------------------------------
Buy['BEEF']         5.3606         10.0000          -0.0000
Buy['CHK']          2.0000         10.0000           1.1888
Buy['FISH']         2.0000         10.0000           1.1444
Buy['HAM']         10.0000         10.0000          -0.3027
Buy['MCH']         10.0000         10.0000          -0.5512
Buy['MTL']         10.0000         10.0000          -1.3289
Buy['SPG']          9.3061         10.0000           0.0000
Buy['TUR']          2.0000         10.0000           2.7316
------------------------------------------------------------

CONSTRAINTS

Name           Slack                 body                 dual
------------------------------------------------------------
diet['A']     1256.2882          1956.2882               0.0000
diet['B1']     336.2575          1036.2575               0.0000
diet['B2']       0.0000           700.0000               0.4046
diet['C']      982.5149          1682.5149               0.0000
diet['NA']      -0.0000         50000.0000              -0.0031
diet['CAL']   3794.6206         19794.6206               0.0000
END
```

We have also seen along the chapter that by using the table declarations and write table commands we can also display the results in an external relational database.

Chapter 8: Advanced Features of AMPL

AMPL provides a variety of commands like model, solve and display that tell the AMPL modeling system what to do with models and data. These commands are not part of AMPL modeling language itself but are intended to be used in an environment where you give a command, wait for the system to display a response, then decide what command to give next. In AMPL studio, these commands can be given from the command prompt window.

Fig 8.1. Command prompt window in AMPL Studio.

Modelling Commands

Options

The behavior of AMPL commands depends on a variety of options. For example: Controlling the display of results, Choosing alternative solvers etc.
The option command displays and sets option values. Each option has a name and a value that may be a number or a character string. For example, the options prompt1 and prompt2 are strings that specify formats. The option display_width has a numeric value, which says how many characters wide the output produced by the display command maybe.
An option command can be issued at the command prompt.

Example:

```
ampl: option prompt1 “A>” ;
A>
```

The issue of option command with prompt option changes the prompt from ampl to A>.
One can set solver options also by using this command.

```
ampl: option cplex_options;
```

To return all options to their default values use the command ‘reset options’.

Setting up and solving models and data

A model can be run from command prompt window. One can choose the solver for solving the problem by using option command.
To apply a solver to an instance of a model, we use model, data and solve command.

```
ampl: option solver cplexamp;
ampl: model steel4.mod;
ampl: data steel4.dat;
ampl: solve;
```

If the model declares more than one objective function, we can use objective command to select the objective function to pass to the solver. It consist of keyword objective followed by a name from minimize or maximize declaration. AMPL by default chooses first objective function.

```
ampl: objective Total_Number;
```

Modifying Data

To delete the current data for several model components, without changing the current model itself, use reset data command as in:

```
reset data MINREQ, MAXREQ, amt, n_min, n_max;
```

We can then use data command to read in new values for these sets and parameters. To delete all data type 'reset data'.
The update data command works similarly, but does not actually delete any data until new values are assigned. Thus if we type:

```
update data MINREQ, MAXREQ, amt, n_min, n_max;
```

but read in new values for `MINREQ`, `amt` and `n_min`, the previous values for `MAXREQ` and `n_max` will remain. If instead we used reset data, `MAXREQ` and `n_max` would be without values.

The 'reset data' command also acts to resample the randomly computed parameters.

The 'let' command permits us to change particular data value while leaving the model the same, but it is more convenient for small or easy to describe changes than 'reset data' or 'update data'. For example: if a parameter `'T'` in our data for some hypothetical model has a value 4 and we can change it to 3 by let command:

```
ampl: let T:=3;
ampl: solve ;
```

Modifying models

The 'delete' command removes a previously declared model component, provided that no other component use it in their declarations. The format of the command is simply 'delete' followed by a comma-separated list of names of model components:

```
ampl: model dietobj.mod;
ampl: data dietobj.dat;
ampl: delete Total_Number, Diet_Min ;
```

Normally we can-not delete a set, parameter or variable, because it is declared for use later in the model; but we can delete any objective or constraint.
The 'purge' command has the same form, but with keyword 'purge' in place of delete. It removes not only the listed components, but also all components that depend on them either directly or indirectly. If we are not sure which components depend on some given component, we can use 'xref' command to find out.

To change any component's declaration we can use 'redeclare' command.

```
ampl: redeclare param f_min {FOOD} >0 integer;
```

changes the validity conditions on `f_min`. The declarations of all components that depend on `f_min` are left unchanged, as are any values previously read for `f_min`.

Changing the model: fix, unfix; drop, restore

The 'drop' command instructs AMPL to ignore certain constraints or objectives of the current model. As an example, the constraints are:

```
subject to Diet_Max { i in MAXREQ} :
                    sum { j in FOOD} amt[I,j] * Buy [j] <= n_max [i] ;
```

A 'drop' command can specify a particular one of these constraints to ignore:

```
drop Diet_Max["CAL"] ;
```

The entire collection of constraints can be ignored by

```
drop {i in MAXREQ} Diet_Max[i] ;
```

The 'restore' command reverses the effect of drop. It has same syntax, except for the keyword 'restore'.

The 'fix' command fixes specified variables at their current values, as if there were a constraint that the variables must equal these values; the unfix command reverses the effect. These commands have the same syntax as 'drop' and 'restore' except that they name variables rather than constraints.

Relaxing Integrality

Changing option 'relax_integrality' from its default of 0 to any nonzero value:

```
option  relax_integrality 1;
```

tells AMPL to ignore all restrictions of variables to integer values. Variables declared integer gets whatever bounds we specified for them, while variables declared binary are given a lower bound of zero and an upper bound of one. To restore integrality restrictions, set 'relax_integrality' option back to 0.

A variable's name followed by the suffix '.relax' indicates its current integrality relaxation status: 0 if integrality is enforced, nonzero otherwise. We can make use of this suffix to relax integrality on selected variables only. For example,

```
ampl: let Buy['CHK'].relax=1;
```

relaxes integrality only on the variable `Buy['CHK']`.

Some of the solvers that work with AMPL Studio provide their own directives for relaxing integrality but may have different effect as AMPL's 'relax_integrality' option.

DISPLAY Commands

AMPL provides a rich variety of commands and options to help examine and report the results of optimization.

Browsing through results: display command

The easiest way to examine data and result values is to use 'display' command. It is also possible to capture the output of display command in a file, by adding >filename to the end of 'display' command; this redirection mechanism applies as well to other commands that produces the output.

The contents of the sets are shown by typing 'display' and a list of set names. For example a set of week days defined as the set WEEK would give the following result.

```
ampl: display WEEK;
set WEEK : = MON TUE WED THURS FRI SAT SUN;
```

The argument of 'display' need not be a declared set; it can be any of the expression that evaluate to sets. For example, we can see the union of all the sets `AREA[p]` (where `PROD = {prod1, prod2, prod3, prod4}, AREA[prod1] = east, AREA[prod2] = north, AREA[prod3] = west, AREA[prod4] = south`):

```
ampl: display union {p in PROD} AREA[p];
set union {p in PROD} AREA[p] := east north west south;
```

The 'display' command can also be used to see the value of a scalar model component.

```
ampl: display T;
T=4
```

Or the value of individual components from an indexed collection.

```
ampl: display avail[“reheat], avail[“roll”] ;
avail [‘reheat’] = 35
avail [‘roll’] = 40
```

or an arbitrary expression:

```
ampl: display sin(1)^2 + cos(1)^2;
sin(1)^2 + cos(1)^2 = 1
```

The major use of display, however, is to show whole indexed collection of data. For 'one-dimensional' data – parameters or variables indexed over a simple set – AMPL uses a column format. For example, if avail is indexed over some set, the use of display would work as:

```
ampl: display avail;
avail[*] :=
reheat 35
roll 40
;
```

For 'two-dimensional' parameters or variables – indexed over a set of pairs or two simple sets – AMPL forms a list for small amounts of data or a table for larger amounts.

The 'display' command can show the value of any arithmetic expression that is valid in AMPL model. Single valued expression poses no difficulty, as in the case of these three profit components indexed over say set PROD and some set representing time period:

```
ampl: display sum{p in PROD, t in 1..T} revenue[p,t]*sell[p,t];
```

```
sum{p in PROD, t in 1..T} revenue[p,t]*sell[p,t] = 787810
```

Suppose however we want to see all the individual values of revenue[p,t] * sell[p,t]. Since, we can type 'display revenue, sell' to display the separate values of revenue [p,t] and sell [p, t], we might want to ask for the products of these values by typing:

```
ampl: display revenue * sell;
syntax error
context: display revenue >>> * <<< sell ;
```

AMPL does not recognize this kind of array arithmetic. To display an indexed collection of expressions, we must specify the indexing explicitly:

```
ampl: display {p in  PROD, t in 1..T} revenue[p,t]*sell[p,t];
revenue[p,t]*sell[p,t] [*,*] (tr)
:     bands coils
1     15000       9210
2     15600       87500
;
```

To apply the same indexing to two or more expressions, enclose a list of them in parentheses after the indexing expression.

Formatting options for display

The display command uses a few simple rules for choosing a good arrangement of data. By changing several options, we can control overall arrangement, handling of zero values and line width. These options are summarized below with their default values.

Option	**Details**
display_1col	Maximum elements for a table to be displayed in list format(20)
display_transpose	Transpose tables if rows-colums < display_transpose (0)
display_width	Maximum line width (79)
gutter_width	Separation between table columns (3)
omit_zero_cols	If not 0, omit all-zero columns from displays (0)
omit_zero_rows	If not 0, omit all-zero rows from displays (0)

These options can be used with the keyword 'option' like

```
ampl: option display_1col 0;
```

to force the display to a compact form, or can be set to a very large number to force the list format.

List format & Compact format example:

```
ampl: display required;
required [*] :=
Fri1  100
Fri2  78
Fri3  52
Mon1  100
Mon2  78
Mon3  52
Sat1  100
Sat2  78
Thu1  100
Thu2  78
Thu3  52
Tue1  100
Tue2  78
Tue3  52
Wed1  100
Wed2  78
Wed3  52
;
```

In compact format:

```
Required [*] :=
Fri1  100   Mon1  100   Sat1  100   Thu2  78    Tue2  78    Wed2  78
Fri2  78    Mon2  78    Sat2  78    Thu3  52    Tue3  52    Wed3  52
Fri3  52    Mon3  52    Thu1  100   Tue1  100   Wed1  100
;
```

Numeric Options for display

The numbers in a table or list produced by display are the results of a transformation from the computer's internal numeric representation to a string of digits and symbols. AMPL's options for adjusting this transformation are shown in the table below along with their default values. The options falls under two categories: Options that affect only the appearance of numbers and options that affect the underlying solutions values as well.

Option	**Details**
display_eps	Smallest magnitude displayed different from zero (0)
display_precision	Digits of precision to which displayed numbers are rounded; full precision if 0 (6)
display_round	Digits left or (if negative) right of decimal place to which display numbers are rounded, overriding display_precision (" ")
solution_precision	Digits of precision to which solution

	values are rounded; full precision if 0 (0)
solution_round	Digits left or (if negative) right of decimal place to which solution values are rounded, overriding display_precision (" ")

Other output commands: print and printf

The print command

A print command produces a single line of output:

```
ampl: print {t in 1..T, p in PROD} Make [p,t] ;
5990 1407 6000 1400 1400 3500 2000 4200
```

Or, if followed by an indexing expression and a colon, a line of output for each member of the index set:

```
ampl: print {t in 1..T}: {p in PROD} Make[p,t];
5990 1407
6000 1400
1400 3500
2000 4200
```

Print entries are normally separated by a space, but option 'print_separator' can be used to change this.

The keyword 'print' (with optional indexing expression and colon) is followed by a print item or comma-separated list of print items. A print item can be a value, or an indexing expression followed by a value or parenthesized list of values. Thus a print item is much like a 'display' command, except that only individual values may appear.

'print' command has options 'print_precision' and 'print_round' options, which work exactly like the 'display_precision' and 'display_round' options for the display command.

The printf command

The syntax of printf is exactly the same as that of print, except that the first print item is a character string that provides formatting instructions for the remaining items:

```
ampl: printf "Total revenue is $%6.2f.\n", sum {p in PROD, t in 1..T}
revenue[p,t] * Sell[p,t];
          Total revenue is $787810.00
```

The format string contains two types of objects: ordinary characters, which are copied to the output, and conversion specifications, which govern the appearance of successive remaining print items. Each conversion specification begins with the character % and ends with a conversion character. The complete rules are much the same as for the 'printf' function in C programming language.

Related Solution values

AMPL provides ways of examining objectives, bounds, slacks, dual prices and reduced costs associated with the optimal solution. AMPL distinguishes the various values associated with a model component by use of "qualified" names that consist of a variable or constraint identifier, a dot(.), and a predefined "suffix" string.

Objective functions

The name of the objective function (from a *minimize* or *maximize* declaration) refers to the objective's value computed from the current values of the variables. This name can be used to represent the optimal objective value in display, print, or printf.

```
ampl: print 100* Total_Profit;
7000
```

Here `Total_Profit` was an objective function.

Bounds and slacks

The suffixes *.lb* and *.ub* on a variable denote its lower and upper bounds, while slack denotes the difference of a variable's value from its nearer bound.

```
ampl: display Buy.lb, Buy, Buy.ub, Buy.slack;
:     Buy.lb            Buy          Buy.ub            Buy.slack;
BEEF  2                 2            10                0
CHK   2                 10           10                0
FISH  2                 2            10                0
HAM   2                 2            10                0
MTL   2                 6.23596      10                3.76404
SPG   2                 5.25843      10                3.25843
TUR   2                 2            10                0
;
```

The reported bounds are those that were sent to the solver. Thus they include not only the bounds specified in >= and <= phrases of *var* declarations, but also certain bounds that were deduced from the constraints by AMPL's presolve phase.

The suffixes *.lb, .body*, and *.ub* on constraints give the current values of these parts of the constraints, while the suffix *.slack* refers to the difference between the body and the nearer bound.

Dual values and reduced costs

Associated with each constraint in a linear program is a quantity variously known as the dual variable, marginal value or shadow price. In the AMPL command environment, these dual values are denoted by the names of the constraints, without any qualifying suffix. For example, let there be a collection of constraints named 'Demand':

```
subject to Demand {j in DEST, p in PROD}: sum { i in ORIG} Trans[I,j,p]=
demand[j,p];
```

and a table of dual values associated with these constraints can be viewed by

```
ampl: display Demand;
Demand [*,*]
:     bands       coils       plate :=
DET   201         190.714     199
FRA   209         204         211
FRE   266.2       273.714     285
LAF   201.2       198.714     205
STL   206.2       207.714     216
WIN   200         190.714     198
;
```

A nearly identical concept applies to the bounds on a variable. The role of the dual value is played by the variable's so called reduced costs, which can be viewed from the AMPL command environment by use of the suffix .rc
Details about dual values and reduced costs can be found in AMPL book and in standard linear programming textbooks.

Other display features for models and instances

Displaying model components: the show command

show command lists the names of all components of the current model:

```
ampl: model example.mod;
ampl: show;
parameters: demand limit
sets: DEST ORIG PROD
variables: use cost
constraints: Demand supply
```

```
objective: Total_cost
checks: one, called check_1.
```

The display may be restricted to one or more types:

```
ampl: show vars;
variables: use cost
```

The show command can also display the declarations of individual components.

```
ampl: show Total_cost;
minimize Total_cost: sum{ i in ORIG, j in DEST, p in PROD}
demand[p]*use[i,j] ;
```

Since the *check* statements in a model do not have names, AMPL numbers them in the order they appear.

Displaying model dependencies: the xref command

The xref command lists all model components that depend on a specified component, either directly(by refereeing to it) or indirectly (by referring to its dependents). If more than one component is given, the dependents are listed separately for each.

Example:

```
ampl: xref demand, Trans ;
# 2 entities depend on demand:
check 1
Demand
# 5 entities depend on Trans:
Total_Cost
Supply
Demand
Multi
Mini_Ship
```

In general the command is simply the keyword 'xref' followed by a comma-separated list of any combination of set, parameter, variable, objective and constraint names.

Displaying model instances: the expand command

In checking a model and its data for correctness, we may want to look at some of the specific constraints that AMPL is generating. The 'expand' command displays all constraints in a given indexed collection or specific constraints that one identifies.

```
ampl: model example.mod;
ampl: data example.dat;
ampl: expand Supply;
subject to Supply['GRAY']:
            Trans['GRAY','FRA'] +   Trans['GRAY','DEN']     +      Trans
['GRAY', 'LAN'] + Trans
            ['GARY', 'WIN'] = 1400;
subject to Supply['CLEV']:
            Trans['CLEV','FRA'] +   Trans['CLEV','DEN']     +      Trans
['CLEV', 'LAN'] + Trans
            ['CLEV', 'WIN'] = 2600;
```

Similarly objectives can also be expanded. When expand is applied to a variable, it lists all of the nonzero coefficients of that variable in the linear terms of objectives and constraints. When a variable also appears in nonlinear expressions within an objective or constraint, the term *+nonlinear* is appended to represent those expressions.

The command 'expand' alone produces an expansion of all variables, objectives and constraints in a model.

Generic synonyms for variables, constraints and objectives

Synonym	Details
_nvars	Number of variables in the current problem
_ncons	Number of constraints in the current problem
_nobjs	Number of objectives in the current problem
_varname{1.._nvars}	Names of variables in the current problem
_conname{1.._ncons}	Names of constraints in the current problem
_objname{1..n_objs}	Names of objectives in the current problem
_var{1.._nvars}	Synonyms for variables in the current problem
_con{1.._ncons}	Synonyms for constraints in the current problem
_obj{1.._nobjs}	Synonyms for objectives in the current problem

Resource listing

Changing option show_stats from its default of 0 to nonzero value requests summary statistics on the size of the optimization problem that AMPL generates:

```
ampl: model steelT.mod;
ampl: data steelT.dat;
ampl: option show_stats 1;
solve;
Presolve eliminates 2 constraints and 2 variables.
Adjusted problem:
24 variables, all linear
12 constraints, all linear ; 38 nonzeros
1 linear objective; 24 nonzeros.
MINOS 5.5 optimal solution found.
15 iterations, objective 515033
```

Changing option *times* from its default value of 0 to a nonzero value requests a summary of AMPL's translator's time and memory requirements. Similarly, by changing option *gentimes* to a nonzero value, we can get a detailed summary of the resources that AMPL's genmod phase consumes in generating a model instance.

General facilities for manipulating output

Redirection of output

We can direct all output to a file instead of it appearing on display console, by adding a > and the name of the file:

```
ampl: display supply >multi.out;
```

The first command specificying >filename creates a new file by that name(or overwrites any existing file of the same name). Subsequent commands add to the end of the file, until the end of session or a matching close command:

```
ampl: close multi.out;
```

To open a file and append output to whatever is already there(rather than overwriting), use >> instead of >.

Output logs

The log_file option instructs AMPL to save subsequent commands and responses to a file. The option's value is a string that is interpreted as a filename:

```
ampl: option log_file 'multi.tmp';
```

The log file collects all AMPL statements and the output that they produce. Setting log_file to the empty string turns of writing to the file.

Limits on messages

By specifying *option eexit n,* where n is some integer, we determine how AMPL handles error messages. If n in not zero, any AMPL statement is terminated after it has produced abs(n) error messages; a negative value causes only the one statement to be terminated, while a positive value results in termination of the entire AMPL session.
The default value for –eexit is -10. Setting it to 0 causes all error messages to be displayed.

Command Scripts

A *script* is a sequence of commands, captured in a file, to be used and re-used.

Running scripts: include and commands

AMPL provides several commands that cause input to be taken from a file. The command:

```
include filename
```

is replaced by the contents of the named file. An include can even appear in the middle of some other statement, and does not require a terminating semicolon.

The 'model' and 'data' commands are special cases of 'include' that put the command interpreter into model or data mode before reading the specified file. By contrast, 'include' leaves the mode unchanged. For working with a small model, it might be convenient to put the model and data command and all the data statement in a file and then read in by use of 'include' command.

The statement:

```
commands filename;
```

is very similar to include, but is a true statement that needs a terminating semicolon and can only appear in a context where a statement is legal.

For example, 'commands' command may find its use while performing sensitivity analysis on a model by changing a parameter value. In this case we have to solve the model repeatedly by changing the data. So it would be better to put all these statements in a file and then call it by use of 'commands' command.

In many cases 'commands' command can be replaced by 'include' command. In general it is best to use commands within command scripts, however, to avoid unexpected interactions with *repeat, for, if* statements.

Iterating over a set: the 'for' statement

Many times we may have to repeat a few commands a few times. AMPL provides looping commands that can do this work automatically, with various options to determine how long the looping should continue.'for' statement, executes a statement or collection of statements once for each member of some set.

For example:

```
model steelT.mod;
data steelT.dat;
for {1..4} {
solve;
display Total_Profit > steelT.sens;
option display_1col 0;
option omit_zero_rows 0;
display Make > steelT.sens;
display Sell > steelT.sens;
option display_1col 20;
option omit_zero_rows 1;
display Inv > steelT.sens;
let avail[3] := avail[3] + 5;
}
```

The 'for' statement can be iterated over any set also.

Between the opening and closing brace of 'for' statement, we can place other statements like let, print, printf etc.

Iterating subject to a condition: the repeat statement

A second kind of looping construct, the *repeat* statement, continues iterating as long as some logical condition is satisfied.
Generally the 'repeat' statement has the one of the following forms as illustrated:

```
repeat while condition {...};
repeat until condition {…};
repeat {…} while condition;
repeat {…} until condition;
```

The loop body, here indicated by {…}, must be enclosed in braces. Passes through the loop continue as long as the *while* condition is true, or as long as *until* condition is false. A condition that appears before the loop body is tested before every pass; if a *while* condition is false or an *until* condition is true before

the first pass, then the loop body is never executed. A condition that appears after the loop body is tested after every pass, so that the loop is executed at least once in this case. If there is no *while* or *until* condition, the loop repeats indefinitely and must be terminated by other means, like the *break* statement.

Testing a condition: the 'if-then-else' statement

The if-then-else statement conditionally controls the execution of statements or groups of statements.
In the simplest case, the if statement evaluates a condition and takes a specified action if the condition is true:

```
If Make ["coin",2] < 1500 then printf "under 1500\n" ;
```

The action may also be a series of commands grouped by braces as in the *for* and *repeat* commands.

An optional *else* specifies an alternative action that also may be a single command or group of commands:

```
If Make ["coin",2] < 1500 then printf "under 1500\n" else printf  "Over
1500\n" ;
```

AMPL executes these commands by first evaluating the logical expression following *if.* If the expression is true, the command or commands following then are executed. If the expression is false, the command or commands following else, if any, are executed.

Terminating a loop: break and continue

Two other statements work with looping statements to make some scripts easier to write. The *continue* statement terminates the current pass through a *for* or *repeat* loop; all further statements in the current pass are skipped, and execution continues with the test that controls the start of the next pass(if any). The *break* statement completely terminates a *for* or *repeat* loop, sending control immediately to the statement following the end of the loop.

Stepping through a script: step, next, skip

If we suspect that a script might not be doing what we want it to do, we can tell AMPL to step through it one command at a time. This facility can be used to provide an elementary form of "symbolic debugger" for scripts.

To step through a script that does not execute any other scripts, reset the option 'single_step' to 1 from its default value of 0. For example:

```
ampl: option single_step 1;
ampl: commands steelT.sa7;
steelT.sa7:2(18) data…
<2>ampl:
```

The expression `steelT.sa7:2(18)` gives the filename, line number and character number where AMPL has stopped in its processing of the script. It is followed by the beginning of the next command (data) to be executed. On the next line we are returned to the `ampl:` prompt. The `<2>` in front indicates the level of input nesting; "2" means that execution is within the scope of a commands statement that was in turn issued in the original input stream.
At this point we may use **step** command to execute individual commands of the script. If step is followed by a number, that number of commands will be executed.

```
<2> ampl: step;
steelT.sa7:4(36) option …
```

To help through lengthy compound statement (for, repeat or if) AMPL provides alternatives to step. The **next** command steps past a compound command rather than into it. Typing 'next n' step past n commands in this way.
The commands **skip** and 'skip n' works like step and 'step n', except that they skip the next 1 or n commands in the script rather than executing them.

Manipulating character strings

String functions and operators: '&', length, match, substr, sub, gsub

The concatenation operator '&' takes two strings as operands, and returns a string consisting of the left operand followed by the right operand. For example:

```
ampl: model diet.mod;
ampl: data diet2.dat;
ampl: display NUTR, FOOD;
set NUTR :=  A          B1          B2;
set FOOD := BEEF  CHK   FISH;
ampl: set NUTR_FOOD := setoff {i  in NUTR, j in FOOD} i & "_" & j;
ampl: display NUTR_FOOD;
set NUTR_FOOD :=
A_BEEF      B1_BEEF     B2_BEEF
A_CHK       B1_CHK      B2_CHK
A_FISH      B1_FISH     B2_FISH;
```

Numbers as arguments to '&' are automatically converted to strings. Numeric operands are always converted to full precision.

The **'length'** string function takes a string as argument and returns the number of characters in it. The **'match'** function takes two string arguments, and returns the first position where second appears as a substring in the first, or zero if the second never appears as a substring in the first. The **'substr'** function takes a string and one or two integers as arguments. It returns a substring of the first argument that begins at the position given by the second argument; it has the length given by the third argument, or extends to the end of the string if no third argument is given. An empty string is returned if the second argument is greater than the length of the first argument, or if the third argument is less than 1.
AMPL provides two other functions, **'sub'** and '**gsub'**, that look for the second argument in the first, like match, but that then substitute a third argument for either the first occurrence(sub) or all occurrences(gsub) found.

Interactions with Solvers

We briefly discuss the mechanisms used by AMPL to control and adjust the problems sent to solvers, and to extract and interpret information returned by them.

Presolve

AMPL's presolve phase attempts to simplify a problem instance after it has been generated but before it is sent to a solver. It runs automatically when a 'solve' command is given or in response to other commands. Any simplifications that presolve makes are reversed after a solution is returned, so that one can view the solution in terms of the original problem. Thus presolve normally proceeds silently behind the scenes. Its effects are only reported when we change option show_stats from its default value of 0 to 1.
We can determine which variable and constraints presolve eliminated by testing, to see which variables/ constraints have a status of "pre".

```
ampl: print { j in 1.._nvars: _var[j].status = "pre" } : _varname [j];
      Inv['bands', 0]
      Inv['coils', 0]
ampl: print { i in 1.._ncons: _con[i].status = "pre"}:  _conname [i] ;
Init_Inv ['bands']
Init_Inv ['coils']
```

We can then use 'show' and 'display' to examine the eliminated components.

Activities of the presolve phase

- AMPL first assigns each variable whatever bounds are specified in its 'var' declaration or the special bounds '-Infinity' and 'Infinity' when no lower or upper bounds are given.
- The presolve phase tries to use these bounds together with the linear constraints to deduce tighter bounds that are still satisfied by all of the problem's feasible solutions. Concurrently, presolve tries to use the tighter bounds to detect variables that can be fixed and constraints that can be dropped.
- Presolve works on a problem in two parts. In first part it applies some tests to deduct some bounds on variables and deduce linear constraints. In second part, there are a series of passes through the problem, each attempting to deduce still tighter variable bounds from the current bounds and the linear constraints.

Controlling the effects of presolve

To turn off presolve entirely, set option presolve to 0; to turn off the second part only, set it to one (1). A higher value for this option indicates the maximum number of passes made in part two of presolve; the default is 10.

Following presolve, AMPL saves two sets of lower and upper bounds on the variables: ones that reflect the tightening of the bounds implied by constraints that presolve eliminated, and ones that reflect further tightening deduced from constraints that presolve could not eliminate. The problem has the same solution with either set of bounds, but the overall solution time may be lower with one or the other, depending on the optimization method in use and the specifics of the problem.

Some other variables to control presolve effects:
var_bounds : set it to 2 to pass the second set of bounds to the solver.

For integer variables, AMPL rounds any fractional lower bounds up to the next higher integer and any fractional upper bounds down to the next lower integer. To prevent the inaccuracies of finite precision computation, AMPL subtracts the value of option 'presolve_inteps' from each lower bound and adds it to each upper bound. If increasing this value to the value of option 'presolve_intepsmax' would make a difference to the rounded bounds of any of the variables, AMPL issues a warning.

To examine first and second set of presolve bounds we can use suffixes, .lb1 and .ub1 and .lb2 and .ub2 respectively. The suffixed .lb and .ub give the bound values currently passed to the solver, based on current values of options 'presolve' and 'var_bounds'.

Detecting infeasibility in presolve

Presolve can determine many conditions that can make the problem infeasible.

a) If any variable's lower bound is greater than its upper bound then there can be no solution satisfying all the bounds and other constraints, and an error message is printed.

b) Presolve's more sophisticated tests can also find infeasibilities that are not due to any one variable.

c) When the implied lower and upper bounds for some variable or constraint body are equal then due to imprecision in the computations, the lower bound may come out slightly greater than the upper bound, causing AMPL's presolve to report an infeasible problem. To circumvent this difficulty, we can reset the option 'presolve_eps' from its default value of 0 to some small positive value. Differences between lower and upper bounds are ignored when they are less than this value. If increasing the current 'presolve_eps' value to a value no greater than 'presolve_epsmax' would change presolve's handling of the problem, then presolve displays a message to this effect.

d) An imprecision in the computations can cause the implied lower bound on some variable or constraint body to come out slightly lower than the implied upper bound. Here no infeasibility is detected, but the presence of bounds that are nearly equal may make the solver's work much harder than necessary. Thus whenever, the upper bound minus the lower bound on a variable or constraint body is positive but less than the value of option 'presolve_fixeps', the variable or constraint body is fixed at the average of two bounds. If increasing the value of 'presolve_fixeps' to at most the value of 'presolve_fixepsmax' would change the results of presolve, a message to this effect is displayed.

e) The number of separate messages displayed by presolve is limited to a value of 'presolve_warnings', which is 5 by default. Increasing option 'show_stats' to 2 may elicit some additional information about the presolve run.

Retrieving results from solvers

AMPL sets two built in parameters after each run of 'solve' command to indicate the solver's status after a run of the optimization problem. These two parameters are:

solve_result_num: Contains a number
solve_result : Contains a character string

This can be interpreted as the following:

Number	String	Interpretation
0-99	solved	Optimal solution found
100-199	solved?	Optimal solution indicated, but error likely
200-299	infeasible	Constraints cannot be satisfied
300-399	unbounded	Objective can be improved without limit
400-499	limit	Stopped by a limit that one sets (such as on iterations)
500-599	failure	Stopped by an error condition in the solver

This status information is used in scripts, where it can be tested to distinguish among cases that must be handled in different ways.
The built in parameter solve_exitcode records the success or failure of the most recent solver invocation. Initially -1, it is reset to 0 whenever there has been a successful invocation and to some system dependent nonzero value otherwise.

Solver status of objectives and problems

Sometimes it is convenient to be able to refer to the solve result obtained when a particular objective was most recently optimized. For this purpose, AMPL associates with each built in solve result parameter a 'status' suffix:

Built in parameter	Suffix
solve_result	.result
solve_result_num	.result_num
solve_message	.message
solve_exitcode	.exitcode

Appended to an objective name, this suffix indicates the value of the corresponding built in parameter at the most recent *solve* in which the objective was current.

Solver statuses of variables

AMPL provides facilities to let solver return an individual status for each variable. The major use of solver status values from an optimal basic solution is to provide a good starting point for the next optimization run. The option 'send_statuses', when left at its default value of 1, instructs AMPL to include statuses with the information about variables sent to solver at each solve.

AMPL refers to a variable's solver status by appending .sstatus to its name. Thus we can print the status of variables with *display* command.

A table of the recognized solver status values is stored in option sstatus_table:

```
ampl: option sstaus_table;
option sstatus_table '\
0     none  no status assigned\
1     bas   basic\
2     sup   superbasic\
3     low   nonbasic<= (normally = ) lower bound \
4     upp   nonbasic>= (normally=) upper_bound\
5     equ   nonbasic  at equal lower and upper bounds\
6     btw   nonbasic between bounds\
';
```

Solver statuses of constraints

Implementation of the simplex method typically adds one variable for each constraint that they receive from AMPL. Each added variable has a coefficient of 1 or -1 in its associated constraint, and coefficients of 0 in all other constraints. If the associated constraint is in inequality, the addition is used as a "slack" or "surplus" variable; its bounds are chosen so that it has effect of turning the inequality into an equivalent equation. If the associated constraint is an equality, the added variable is an "artificial" one whose lower and upper bounds are both zero.

To accommodate statuses of these logical variables, AMPL permits a solver to return status values corresponding to the constraints as well as the variables. The solver status of a constraint, written as the constraint name suffixed by .sstatus, is interpreted as the status of the logical variable associated with that constraint.

AMPL statuses

Only those variables, objectives and constraints that AMPL actually sends to a solver can receive solver statuses on return. So that we can distinguish these from components that are removed prior to a solve, a separate "AMPL status" is also maintained. We can work with AMPL statuses much like solver statuses, by using the suffix .astatus in place of .sstatus and referring to option astatus_table for a summary of the recognized values:

```
ampl: option astatus_table;
option astatus_table '\
0     in          normal state (in problem)\
1     drop        removed by drop command\
2     pre         eliminated by presolve
3     fix         fixed by fix command\
4     sub         defined variable, substituted out\
5     unused      not used  in current problem\
';
```

Exchanging information with solvers via suffixes

AMPL employs various qualifiers or suffixes appended to component names to represent values associated with a model component. AMPL can not anticipate all of the values that a solver might associate with model components, however. The values recognized as input or computed as output depend on the design of each solver and its algorithms. To provide for open ended representation of such values, new suffixes may be defined for the duration of AMPL session, either by the user for sending values to a solver, or by a solver for returning values. For this purpose we have user defined suffixes and solver defined suffixes.
User defined suffixes can be used to pass preferences for variable selection and branch direction to an integer programming solver. Similarly solver suffixes can be used for sensitivity analysis and infeasibility diagnosis. Users are referred to the AMPL book by R. Fourer, D. Gay and B.W. Kernighan (Chapter 14 – Interaction with solvers) for details.

Defining and using suffixes

A new AMPL suffix is defined by a statement consisting of the keyword 'suffix' followed by a suffix name and then one or more optional qualifiers that indicate what values may be associated with the suffix and how it may be used. The suffix statement causes AMPL to recognize suffixed expression of the form component-name.suffix name, where component-name refers to any currently declared variable, constraint or objective. The definition of a suffix remains in effect until the next 'reset' command or the end of the current AMPL session. There are a few optional qualifiers of the suffix statement and they may appear in any order.
The optional 'type' qualifier in a suffix statement indicates what values may be associated with the suffixed expressions, with all numeric values being the default.

Suffix type	**Values allowed**
None specified	Any numeric value
Integer	Integer numeric values
Binary	0 or 1
Symbolic	Character strings listed in option suffix-name_*table*

All numeric-valued suffixed expressions have an initial value of 0. Their permissible values may be further limited by one or two bound qualifiers of the form

>= arith-expr

<= arith-expr

Where arith-expr is any arithmetic expression not involving variables.

For each symbolic suffix, AMPL automatically defines an associated numeric suffix, suffix-name_num. An AMPL option suffix-name_table must then be created to define a relation between the .suffix-name and .suffix-name_num values, as in the following example:

```
suffix iis symbolic OUT;
option iis_table '\
0     non   not in the iis\
1     low   at lower bound\
2     fix   fixed\
3     upp   at upper bound\
';
```

Each line of the table consist of an integer value, a string value, and an optional comment.

The optional *in-out* qualifier determines how suffix values interact with the solver:

In-out	**Handling of suffix values**
IN	written by AMPL before invoking the solver, then read in by solver
OUT	written out by solver, then read by AMPL after the solver is finished
INOUT	both read and written, as for IN and OUT above
LOCAL	neither read nor written

Alternating between Models

We have seen earlier how AMPL commands can be set up to run as programs that perform repetitive actions. In several examples, a script solves a series of related model instances, by including a *solve* statement inside a loop. The result is a simple kind of sensitivity analysis algorithm, programmed in AMPL's command language.

Much more powerful algorithmic procedures can be constructed by using two models. An optimal solution for one model yields new data for the other, and the two are solved in alteration in such a way that some termination condition must eventually be reached. To use two models in this manner, a script must have some way of switching between them. Switching can be done with previously defined AMPL features, or more clearly and efficiently by defining separately-named problems and environments.

Named problems

At any point during an AMPL session, there is a *current* problem consisting of a list of variables, objectives and constraints. The current problem is named *Initial* by default and comprises all variables, objectives and constraints defined so far. We can define other "named" problems consisting of subsets of these components, however, and can make them current. When a named problem is made current, all of the model components in the problem's subset are made active, while all other variables, objectives and constraints are made inactive. More precisely variables in the problem's subset are unfixed and the remainder are fixed at their current values. Objectives and constraints in the problem's subset are restored and the remainder are dropped.

We can define a problem most straightforwardly through a *problem* declaration that gives the problem's name and its list of components. For example:

```
problem Cutting_Opt : Cut, Numer, Fill;
```

A new problem `Cutting_opt` is defined, and is specified to contain all of the `Cut` variables, the objective `Number` and all of the `Fill` constraints. At the same time, `Cutting_opt` becomes the current problem. Any fixed `Cut` variables are unfixed, while all other declared variables are fixed at their current values. The objective `Number` is restored if it had been previously dropped, while all other declared objectives are dropped; and similarly any dropped `Fill` constraints are restored, while all other declared constraints are dropped.

Any *problem* statement that refers to only one problem has the effect of making that problem current.

We can display the current problem by using command 'problem':

```
ampl: model cut.mod;
ampl: data cut.dat;
ampl: problem;
problem Initial;
```

The current problem is always *Initial* until other named problems have been defined. The 'show' command can give a list of the named problems that have been defined.

```
ampl: show problems;
problems: Cutting_opt   Pattern_Gen
```

We can also use 'show' to see the variables, objectives and constraints that make up a particular problem or indexed collection of problems and use 'expand' to see the explicit objectives and constraints of the current problem, after all the data values have been substituted.

Named environments

In the same way that there is a current problem at any point in an AMPL session, there is also a current *environment.* Whereas a problem is a list of non fixed variables and non dropped objectives and constraints, an environment records the value of all AMPL options. By naming different environments, a script can easily switch between different collections of option settings.

At the start of an AMPL session the current environment is named *Initial* and each subsequent problem statement that defines a new named problem also defines a new environment having the same name as the problem. An environment initially inherits all the option settings that existed when it was created, but it retains new settings that are made while it is current. Any 'problem' or 'solve' statement that changes the current problem also switches to the correspondingly named environment, with options set accordingly.

In more complex situations, we can declare named environments independently of named problems, by use of statement that consists of the keyword 'environ' followed by a name:

```
environ Master;
```

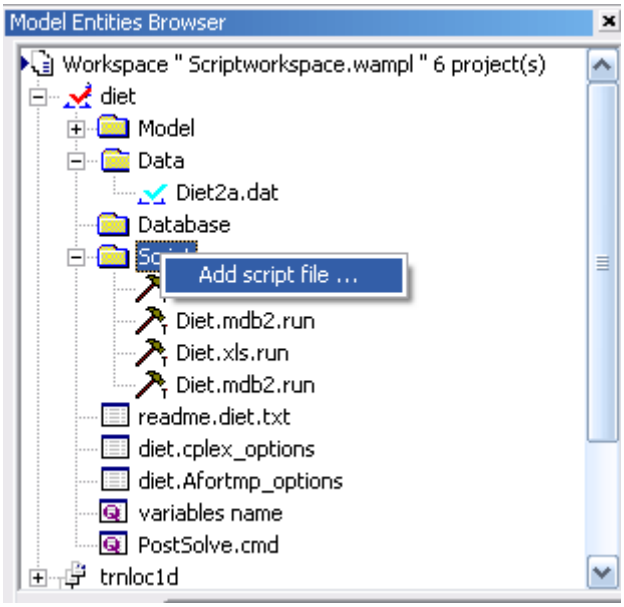
For a more detailed description of the advance features of AMPL language, users are referred to the book on AMPL by R. Fourer, D. Gay and B.W. Kernighan and the AMPL website (www.ampl.com).

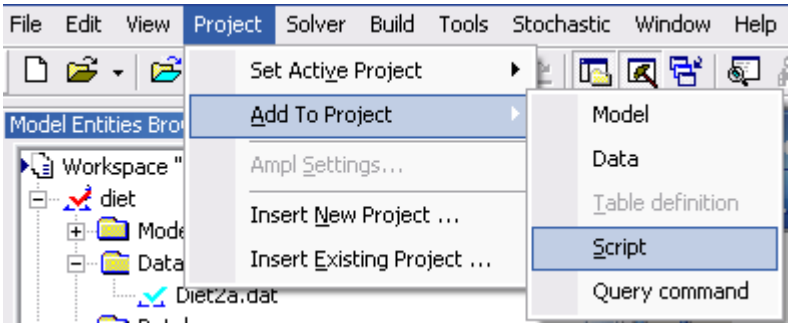
Chapter 9: Scripts, Debugging & Tracing in AMPL Studio

In AMPL studio we can include script file. Prominent among the unique advantages of AMPL studio are the debugging and tracing features.

Scripts

In Chapter 4, we have already seen the ways of running a script file. A new script file can be added to a project by right clicking on the script folder of any project or by choosing to add script file to an active project from the Add to project submenu under project-menu.



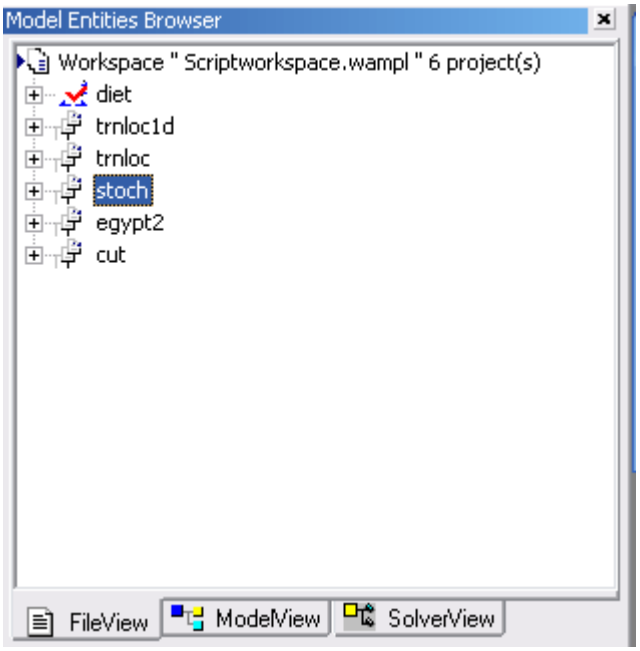


To run a script file for the project, the script file (.sa or .run) file first needs to be opened in AMPL Studio and then by use of  it can be run.

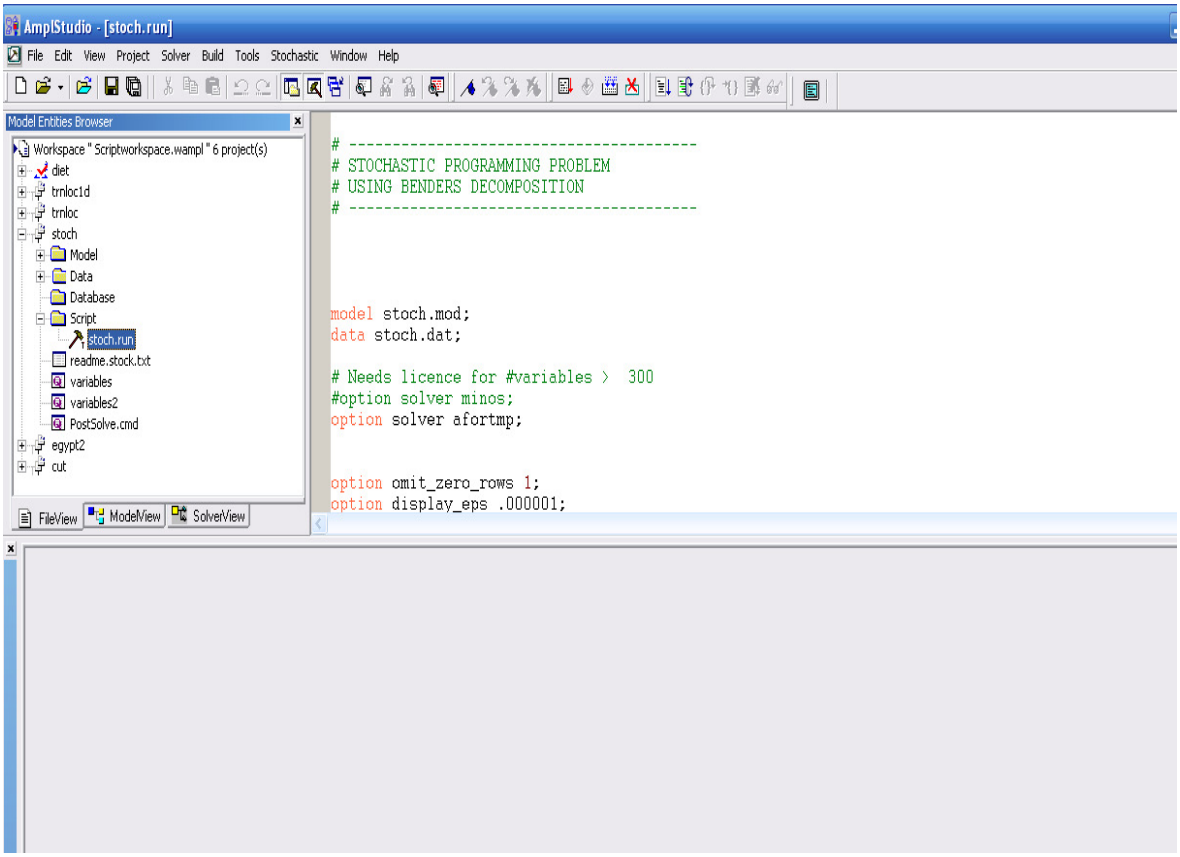
Debugging and Tracing: step by step walk through example

We illustrate the debugging feature by use of an example. We consider the example 'stoch' present under AmplStudio Modeling System 1.6.J\Examples\Script.

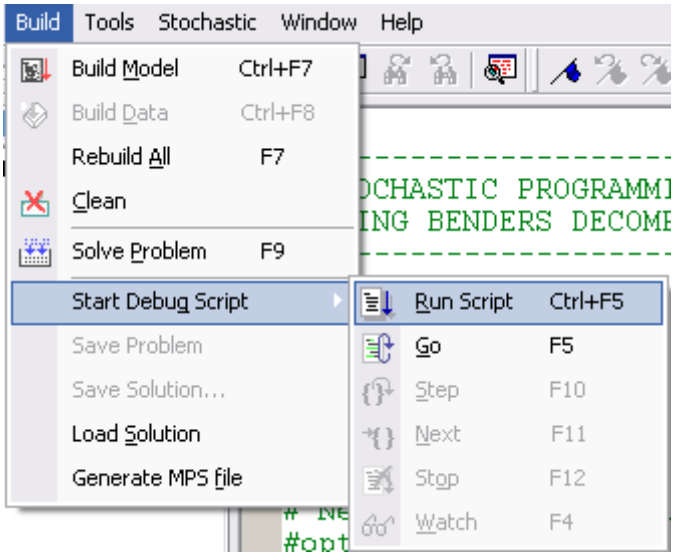
1. We load the workspace Scriptworkspace.wampl and consider the project stoch.



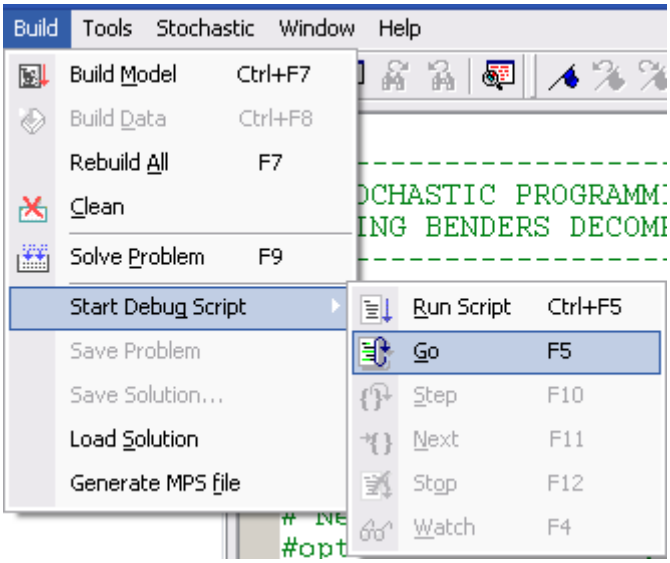
2. We load the stoch.run file by double clicking on it.



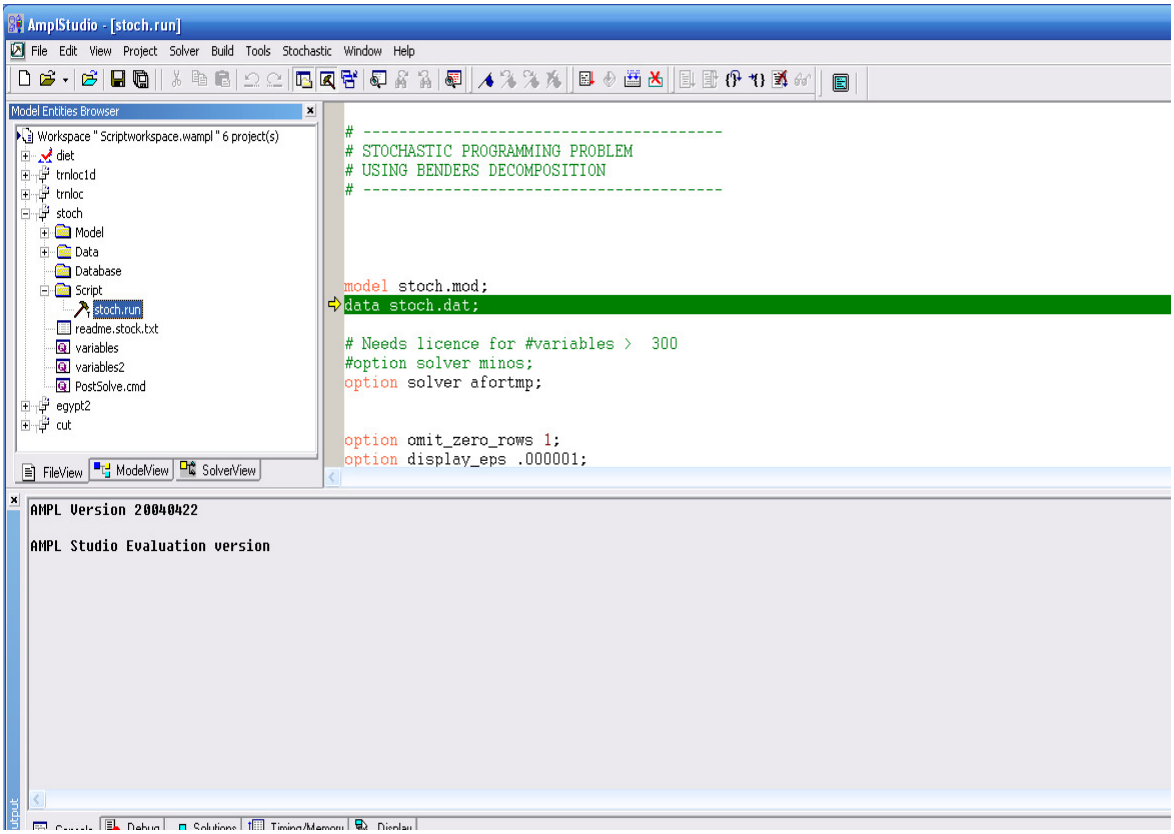
3. The script file can be run in one go by using Start Debug Script -> Run script submenu under the Build menu or the  button on the script bar.



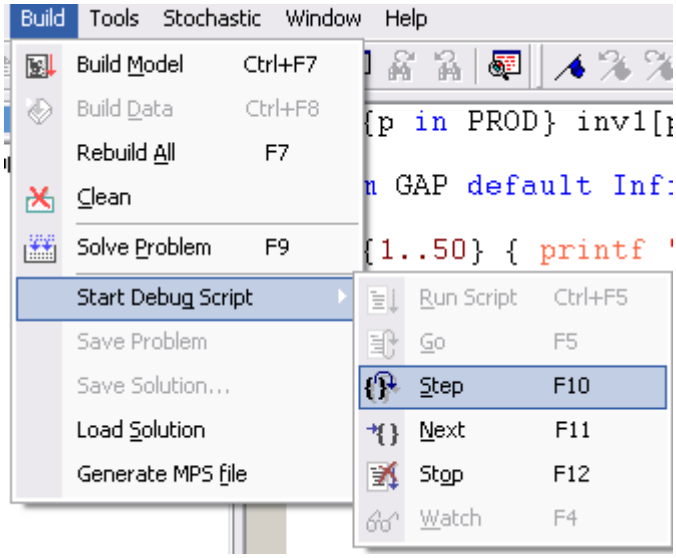
4. To step through the script file use Start Debug Script -> Go submenu from the Build menu or the  button from the script bar.



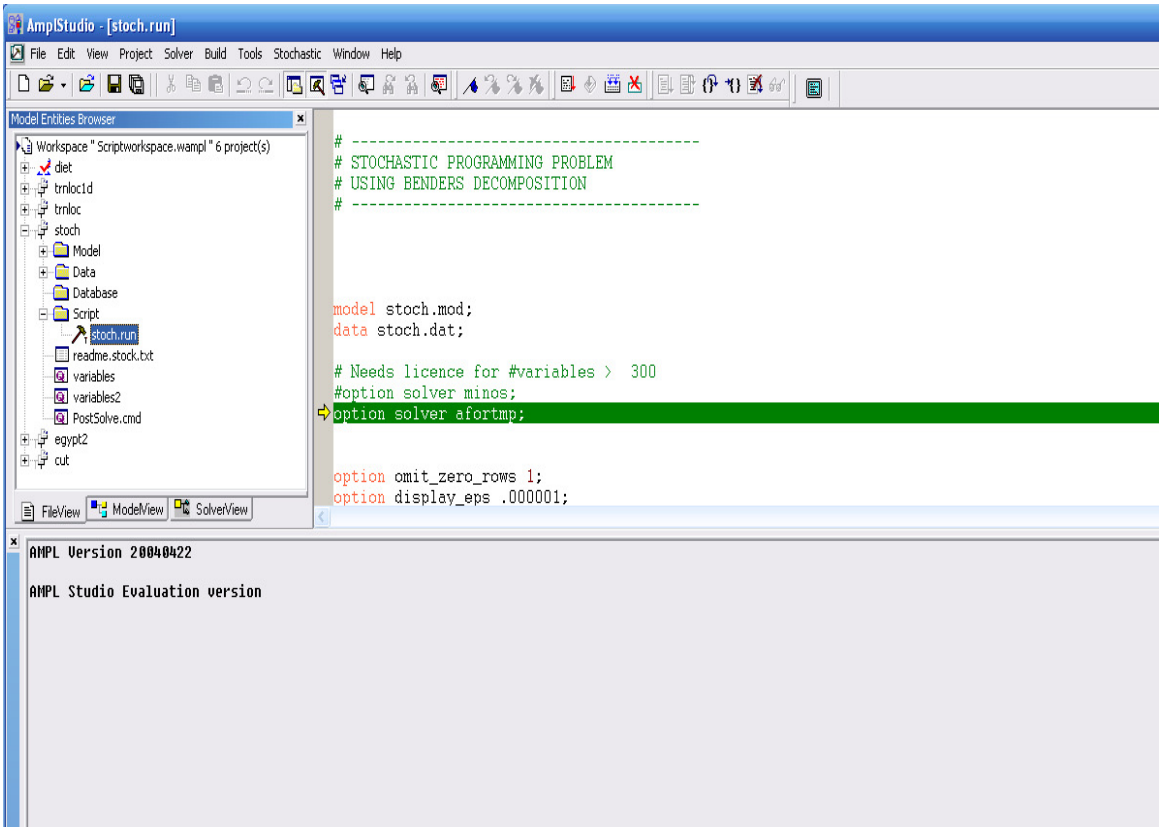
AMPL studio then steps through the script file.



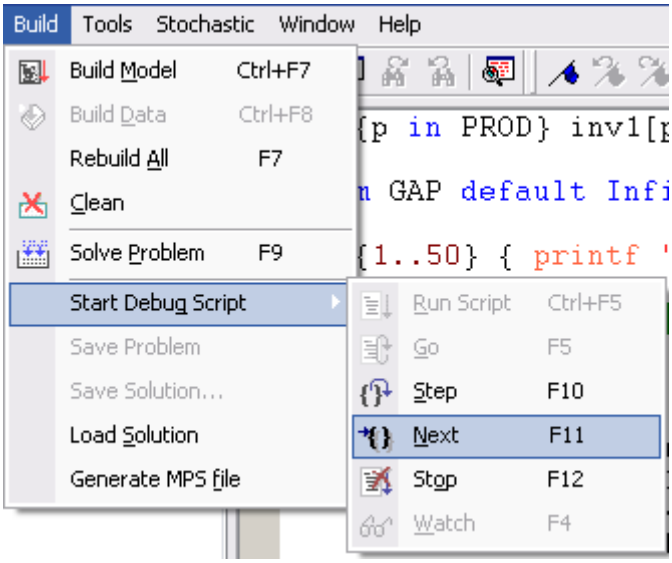
5. When using step by step debugging feature (Step 4) to proceed a single step (analogous to AMPL's step command) use Start Debug Script -> Step under Build menu or  button on script bar



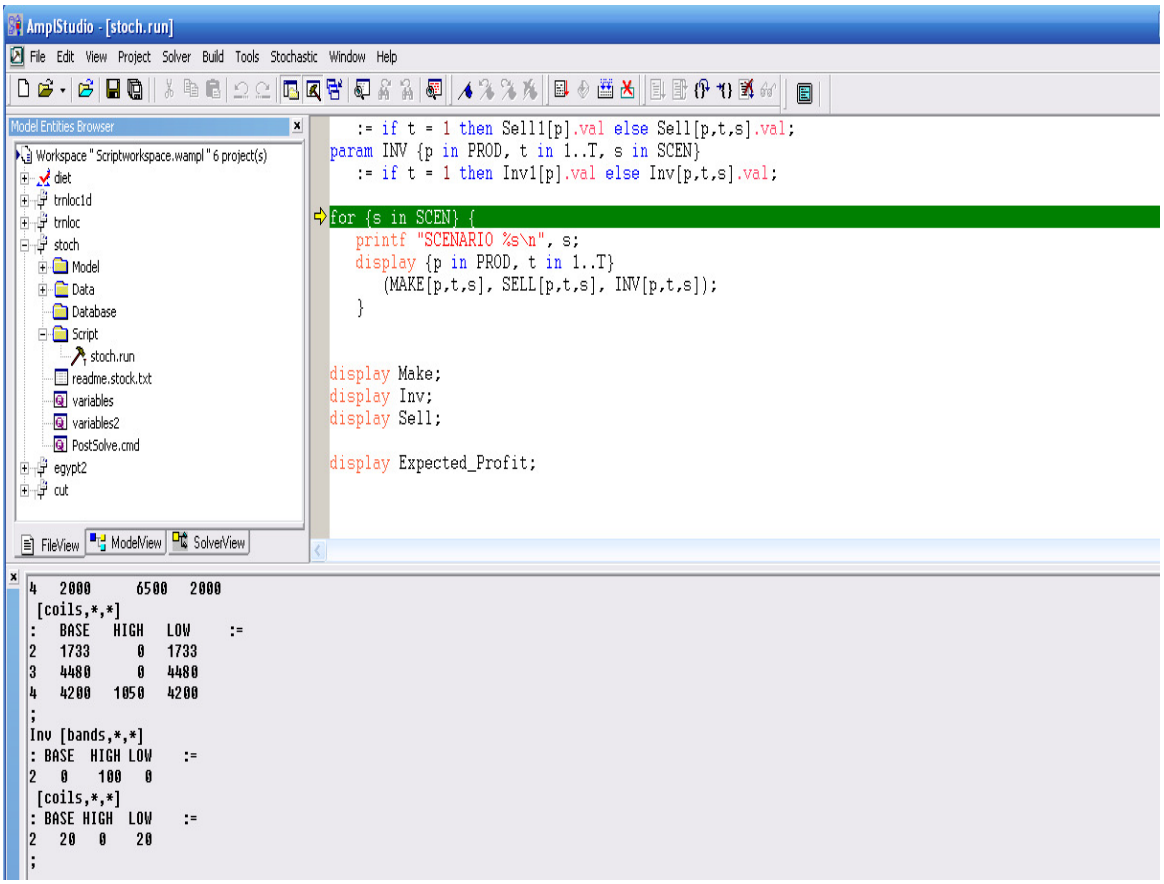
AMPL studio then processes one step at a time.



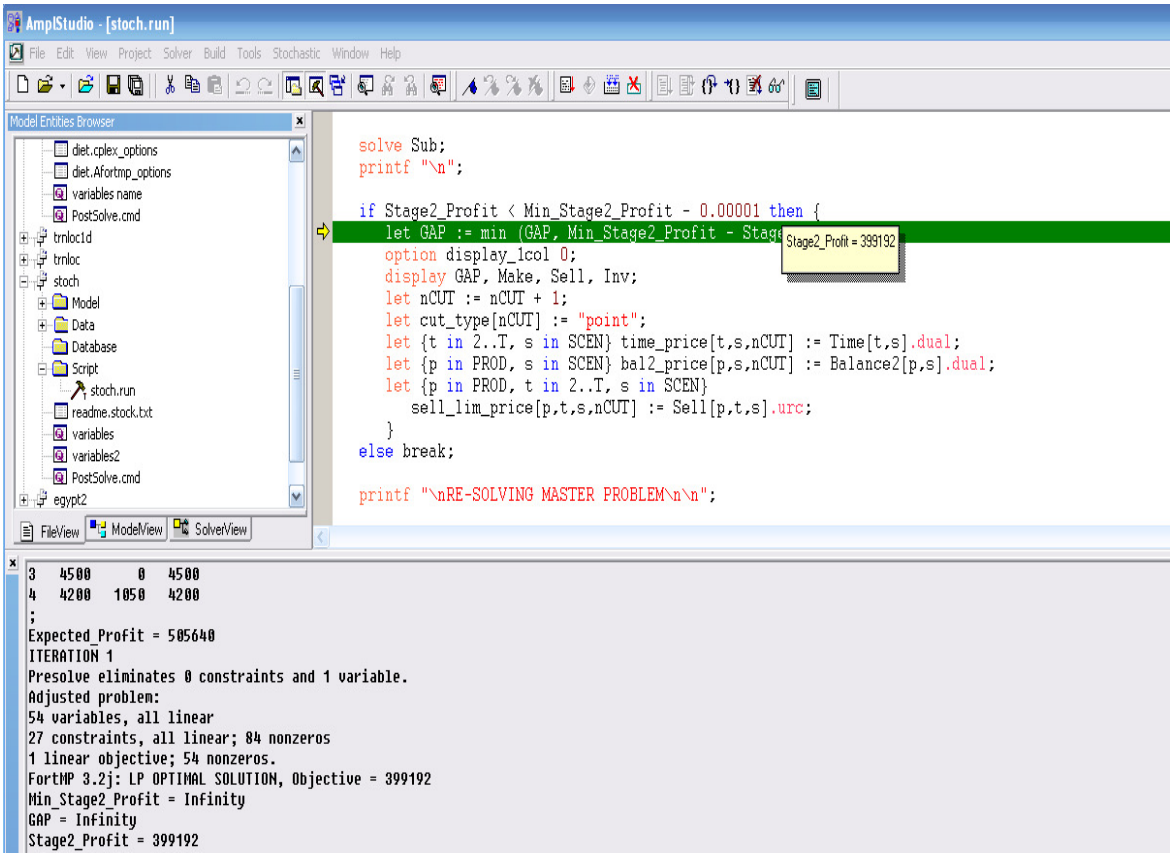
6. To step past a compound statement (analogous to AMPL's next command) rather than into it, use Start Debug Script ->Next under Build menu or use  button on the script bar.



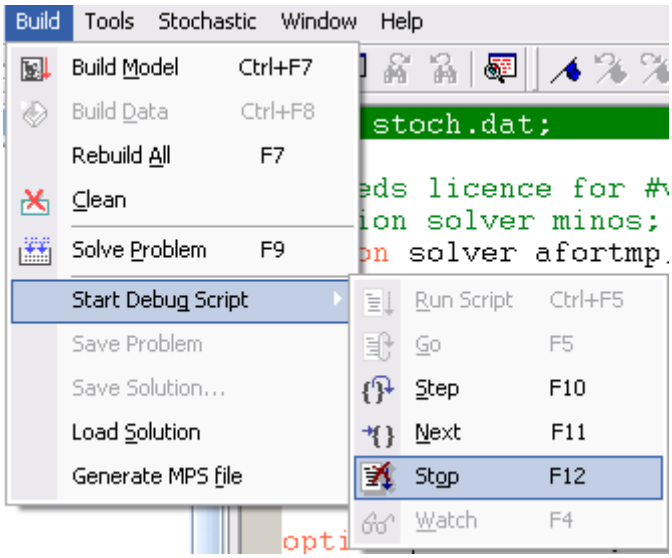
AMPL studio then just steps past the compound statement (For loop in the screen shot)



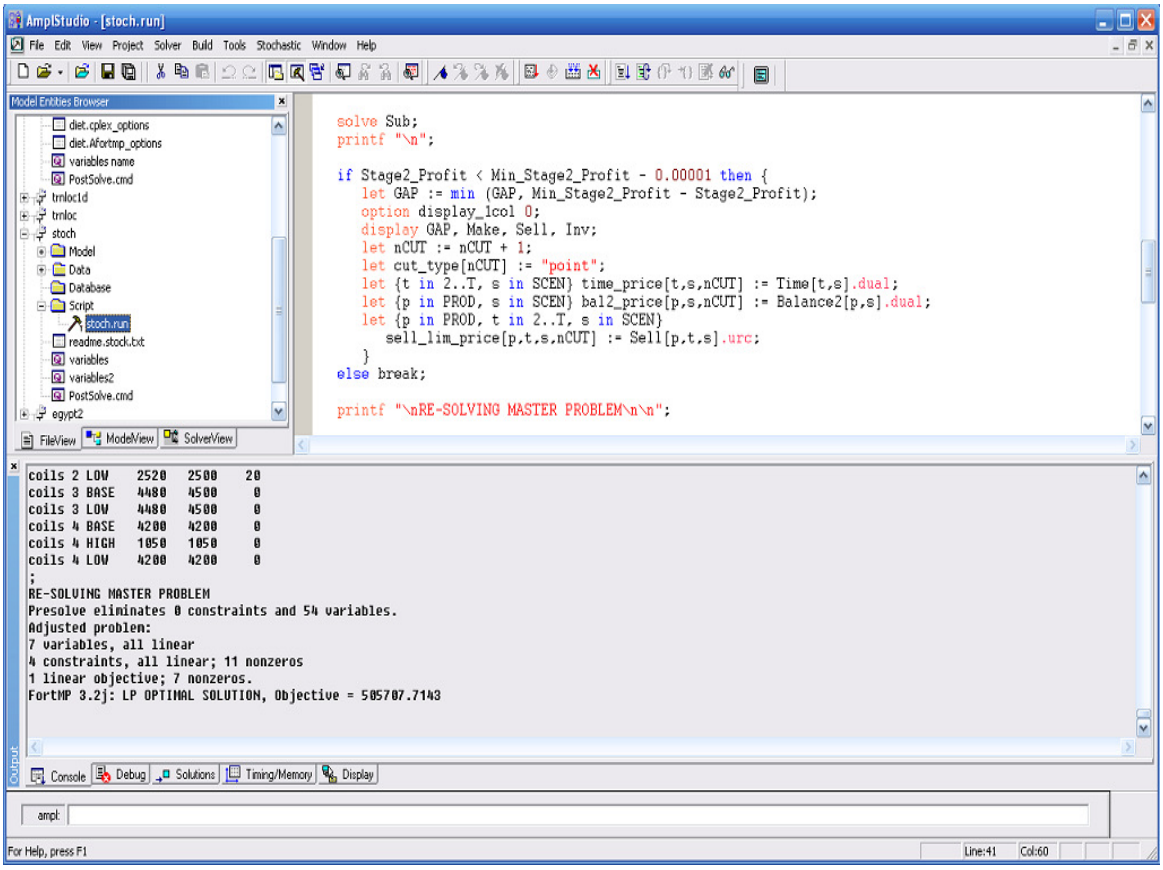
7. To see the value of any variables or parameter double click on the variable or parameter name in the script file. AMPL studio displays its value in Output console as well as in the script file.



8. At any time to stop stepping through the script file use Start Debug Script -> Stop under build menu or the  button on the script bar.

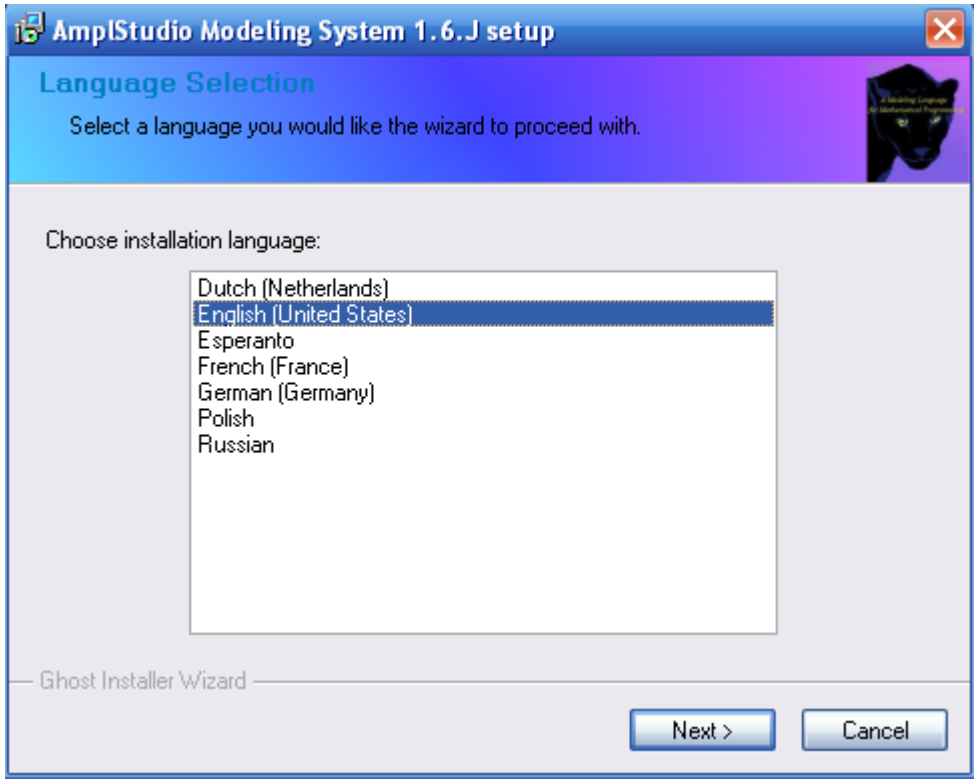


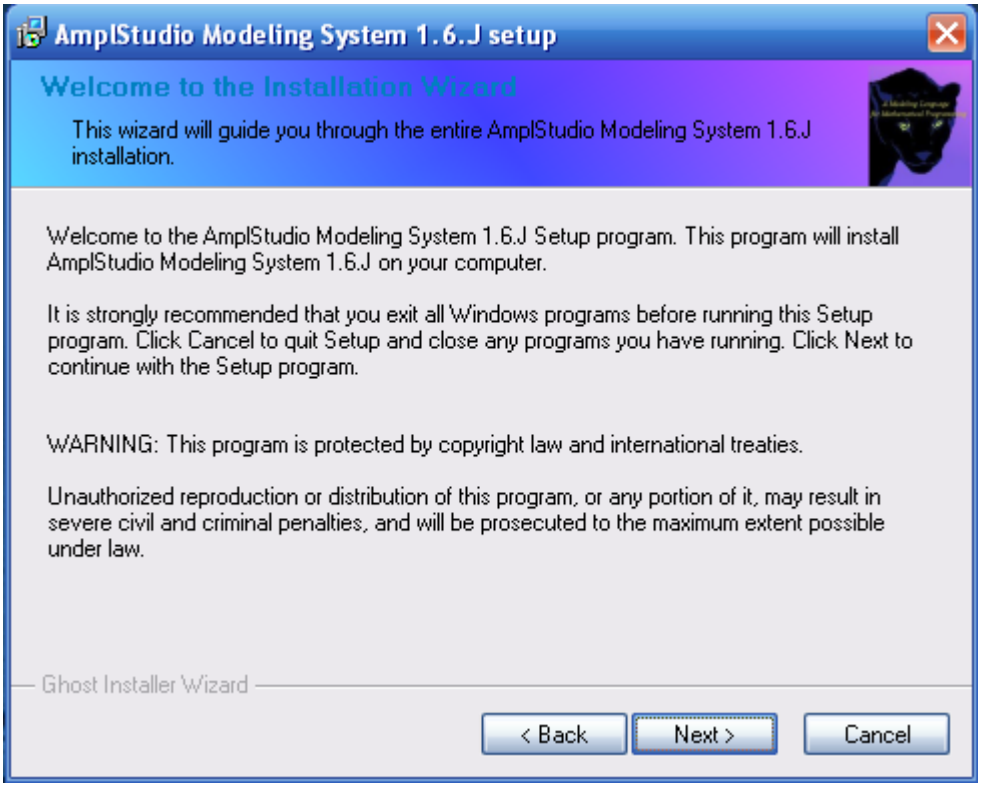
AMPL studio processes the complete script file and displays the final results.

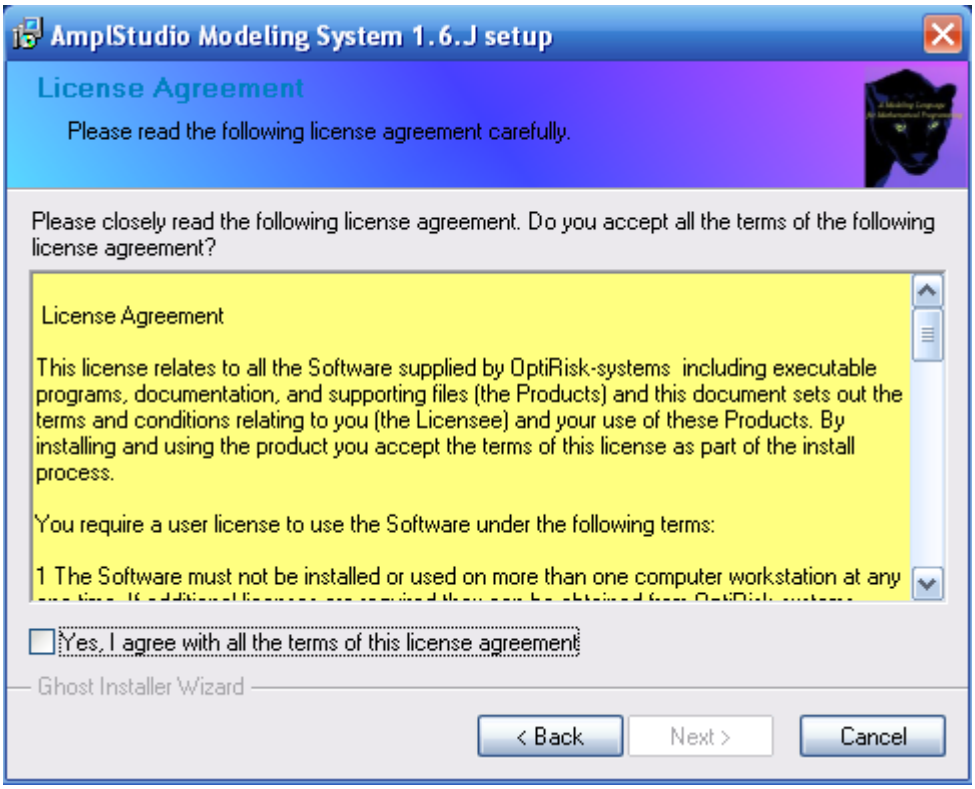

AmplStudio - [stoch.run]
File Edit View Project Solver Build Tools Stochastic Window Help
Model Entities Browser
diet.cplex_options
diet.Afortmp_options
variables name
PostSolve.cmd
trnloc1d
trnloc
stoch
Model
Data
Database
Script
stoch.run
readme.stoch.txt
variables
variables2
PostSolve.cmd
egypt2
FileView ModelView SolverView
solve Sub;
printf "\n";
if Stage2_Profit < Min_Stage2_Profit - 0.00001 then {
let GAP := min (GAP, Min_Stage2_Profit - Stage2_Profit);
option display_1col 0;
display GAP, Make, Sell, Inv;
let nCUT := nCUT + 1;
let cut_type[nCUT] := "point";
let {t in 2..T, s in SCEN} time_price[t,s,nCUT] := Time[t,s].dual;
let {p in PROD, s in SCEN} bal2_price[p,s,nCUT] := Balance2[p,s].dual;
let {p in PROD, t in 2..T, s in SCEN}
sell_lim_price[p,t,s,nCUT] := Sell[p,t,s].urc;
}
else break;
printf "\nRE-SOLVING MASTER PROBLEM\n\n";
coils 2 LOW 2520 2500 20
coils 3 BASE 4480 4500 0
coils 3 LOW 4480 4500 0
coils 4 BASE 4200 4200 0
coils 4 HIGH 1050 1050 0
coils 4 LOW 4200 4200 0
;
RE-SOLVING MASTER PROBLEM
Presolve eliminates 0 constraints and 54 variables.
Adjusted problem:
7 variables, all linear
4 constraints, all linear; 11 nonzeros
1 linear objective; 7 nonzeros.
FortMP 3.2j: LP OPTIMAL SOLUTION, Objective = 505707.7143
Output
Console Debug Solutions Timing/Memory Display
ampl:
For Help, press F1
Line:41 Col:60

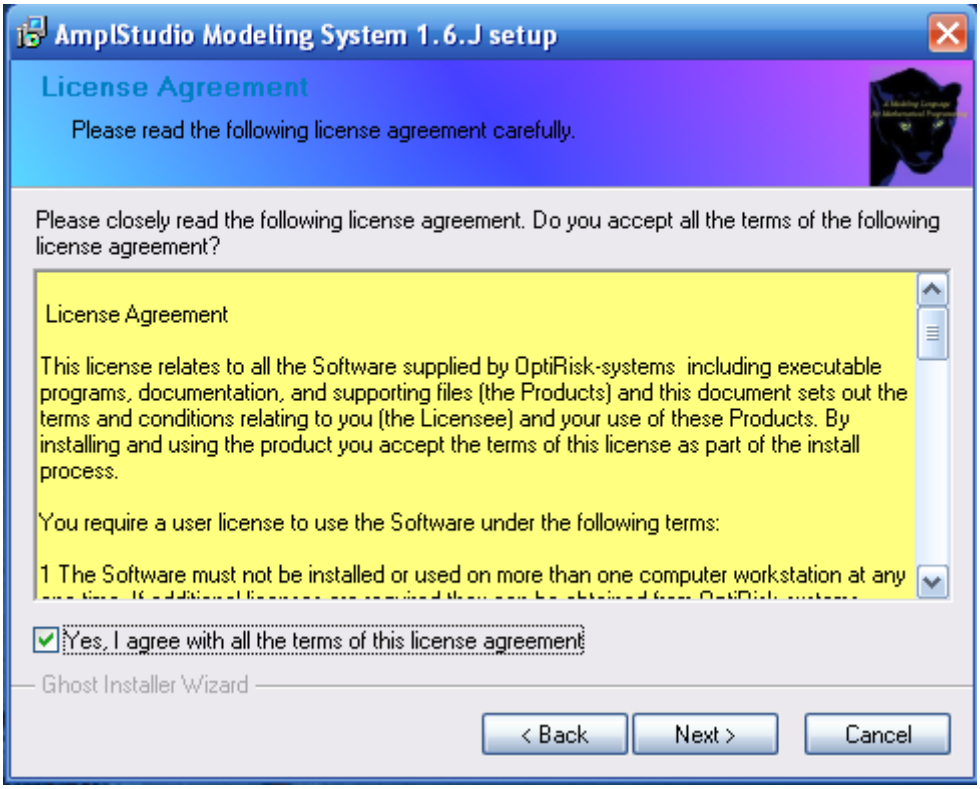
Appendix A: Installation and Licensing

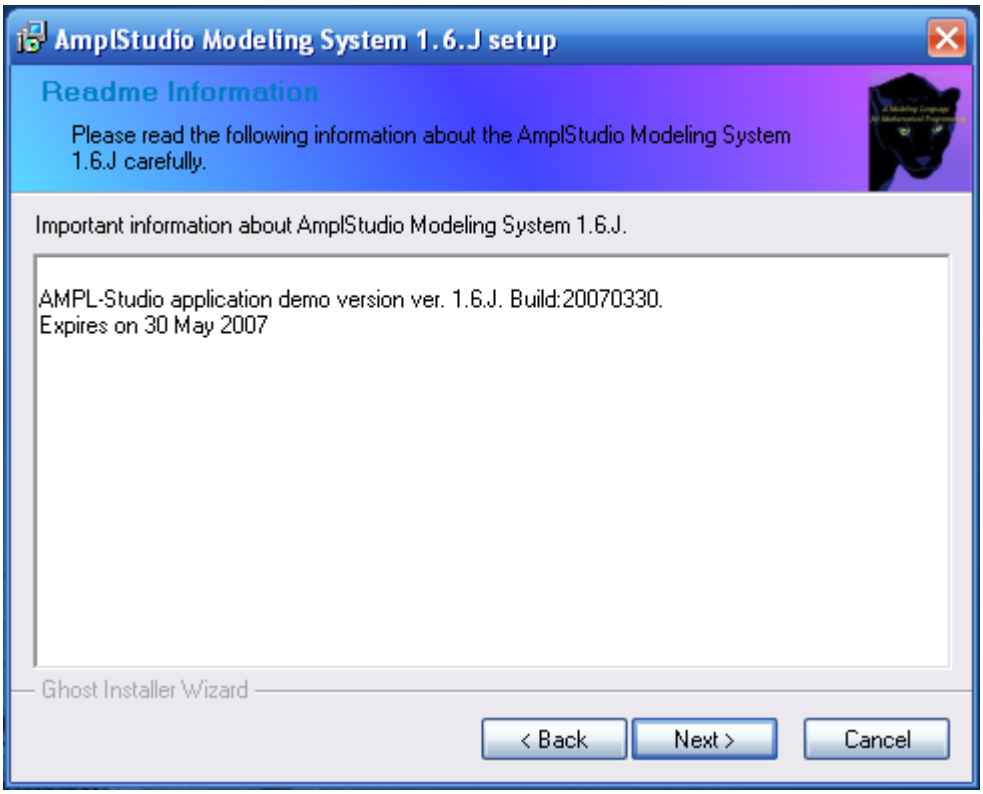
Users should follow the step by step procedure as illustrated by screenshots below:

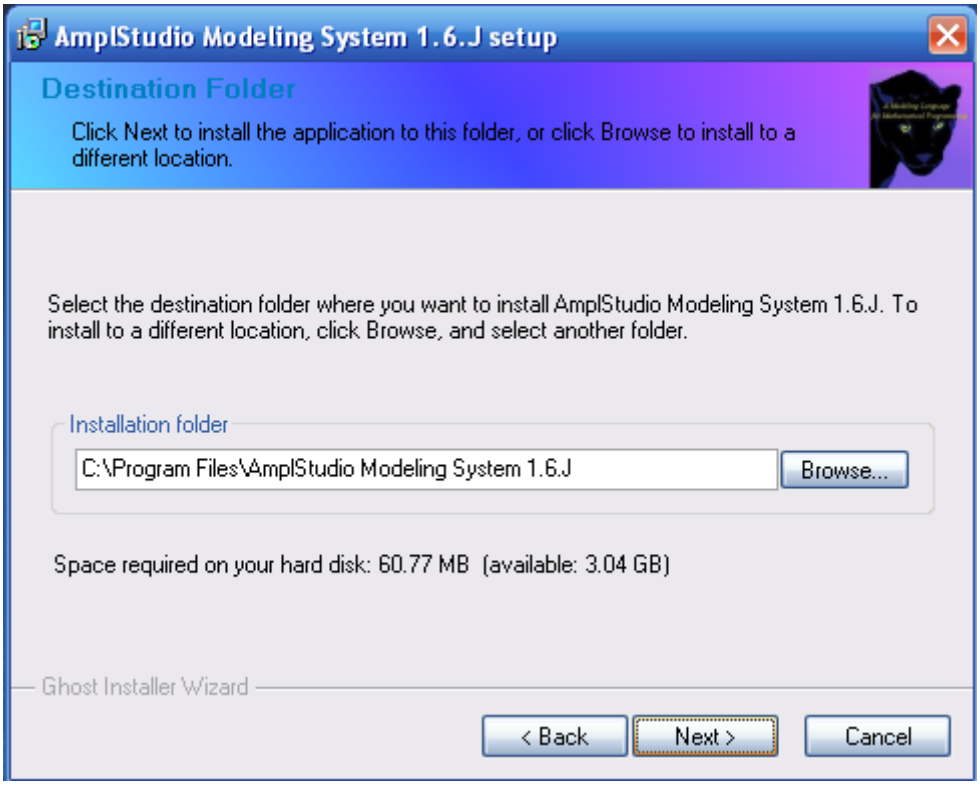


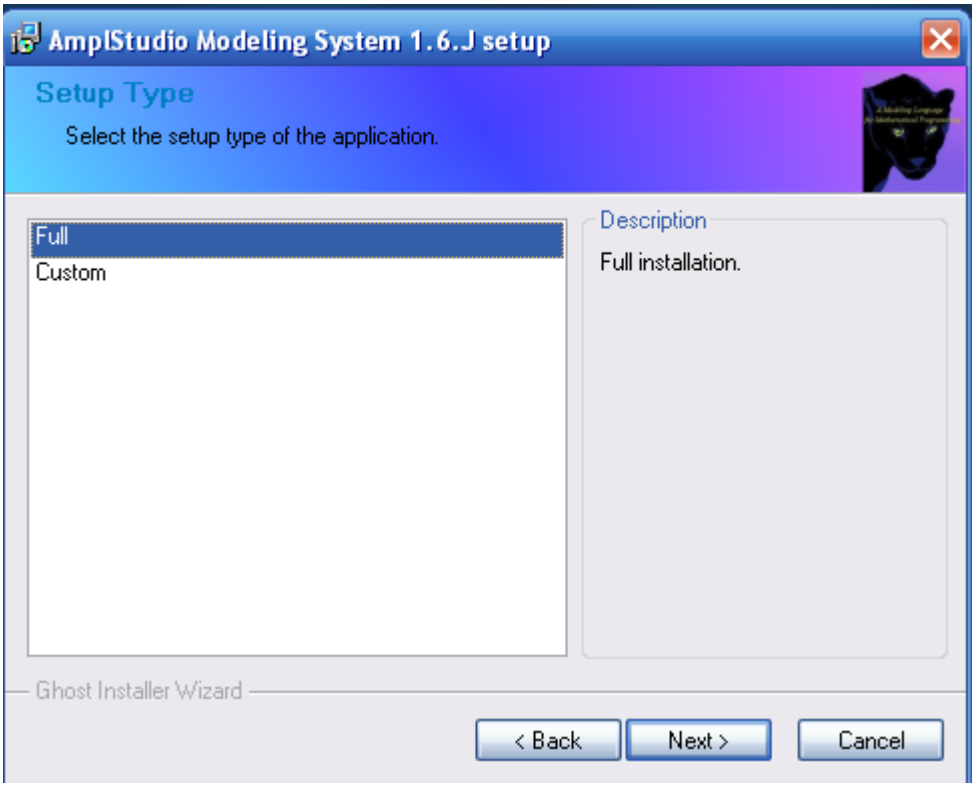

AmplStudio Modeling System 1.6.J setup
Welcome to the Installation Wizard
This wizard will guide you through the entire AmplStudio Modeling System 1.6.J installation.
Welcome to the AmplStudio Modeling System 1.6.J Setup program. This program will install AmplStudio Modeling System 1.6.J on your computer.
It is strongly recommended that you exit all Windows programs before running this Setup program. Click Cancel to quit Setup and close any programs you have running. Click Next to continue with the Setup program.
WARNING: This program is protected by copyright law and international treaties.
Unauthorized reproduction or distribution of this program, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under law.
Ghost Installer Wizard
< Back
Next >
Cancel

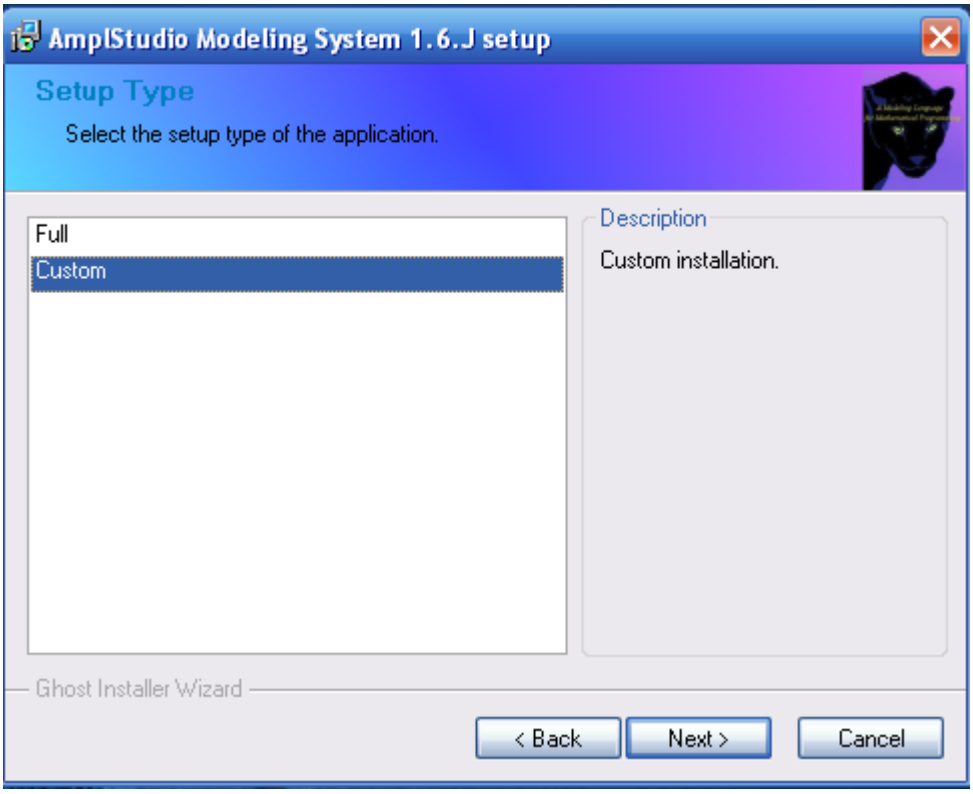

AmplStudio Modeling System 1.6.J setup
License Agreement
Please read the following license agreement carefully.
Please closely read the following license agreement. Do you accept all the terms of the following license agreement?
License Agreement
This license relates to all the Software supplied by OptiRisk-systems including executable programs, documentation, and supporting files (the Products) and this document sets out the terms and conditions relating to you (the Licensee) and your use of these Products. By installing and using the product you accept the terms of this license as part of the install process.
You require a user license to use the Software under the following terms:
1 The Software must not be installed or used on more than one computer workstation at any
Yes, I agree with all the terms of this license agreement
Ghost Installer Wizard
< Back
Next >
Cancel

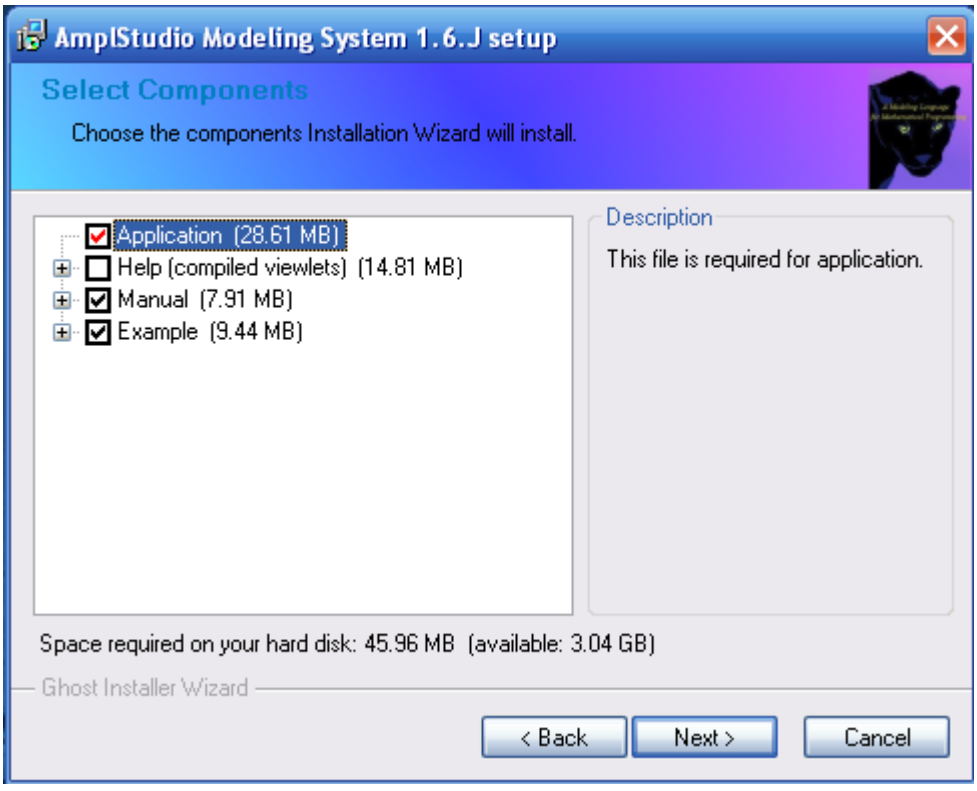

AmplStudio Modeling System 1.6.J setup
License Agreement
Please read the following license agreement carefully.
Please closely read the following license agreement. Do you accept all the terms of the following license agreement?
License Agreement
This license relates to all the Software supplied by OptiRisk-systems including executable programs, documentation, and supporting files (the Products) and this document sets out the terms and conditions relating to you (the Licensee) and your use of these Products. By installing and using the product you accept the terms of this license as part of the install process.
You require a user license to use the Software under the following terms:
1 The Software must not be installed or used on more than one computer workstation at any
Yes, I agree with all the terms of this license agreement
Ghost Installer Wizard
< Back
Next >
Cancel

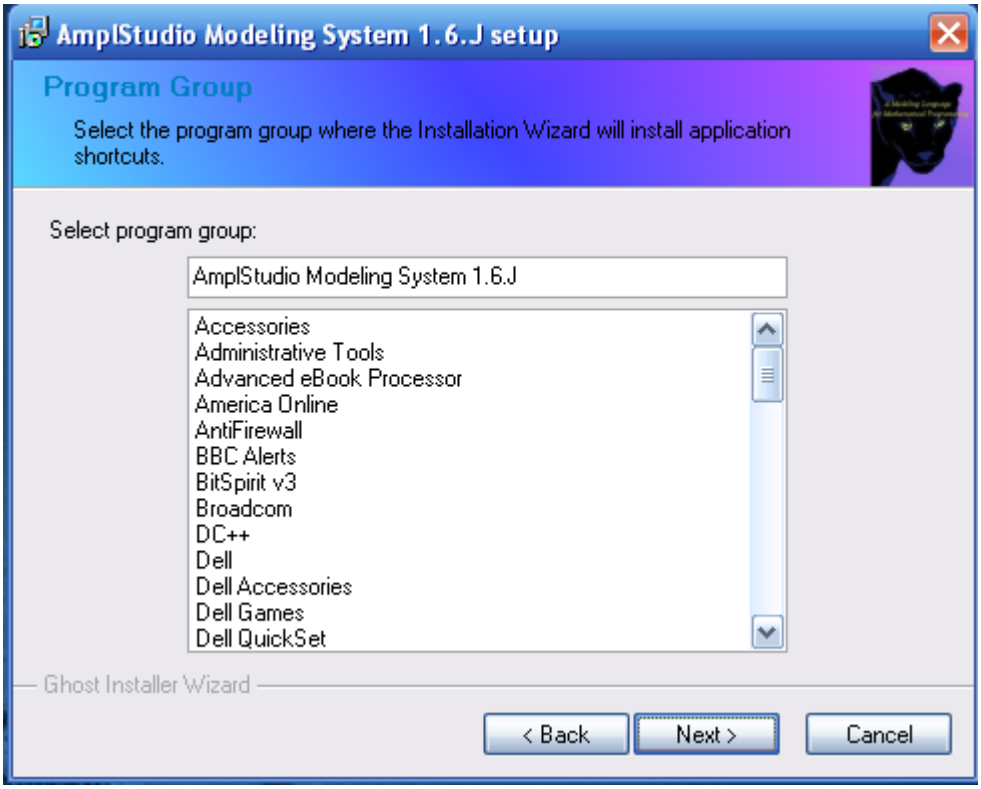

AmplStudio Modeling System 1.6.J setup
Readme Information
Please read the following information about the AmplStudio Modeling System 1.6.J carefully.
Important information about AmplStudio Modeling System 1.6.J.
AMPL-Studio application demo version ver. 1.6.J. Build:20070330.
Expires on 30 May 2007
Ghost Installer Wizard
< Back
Next >
Cancel

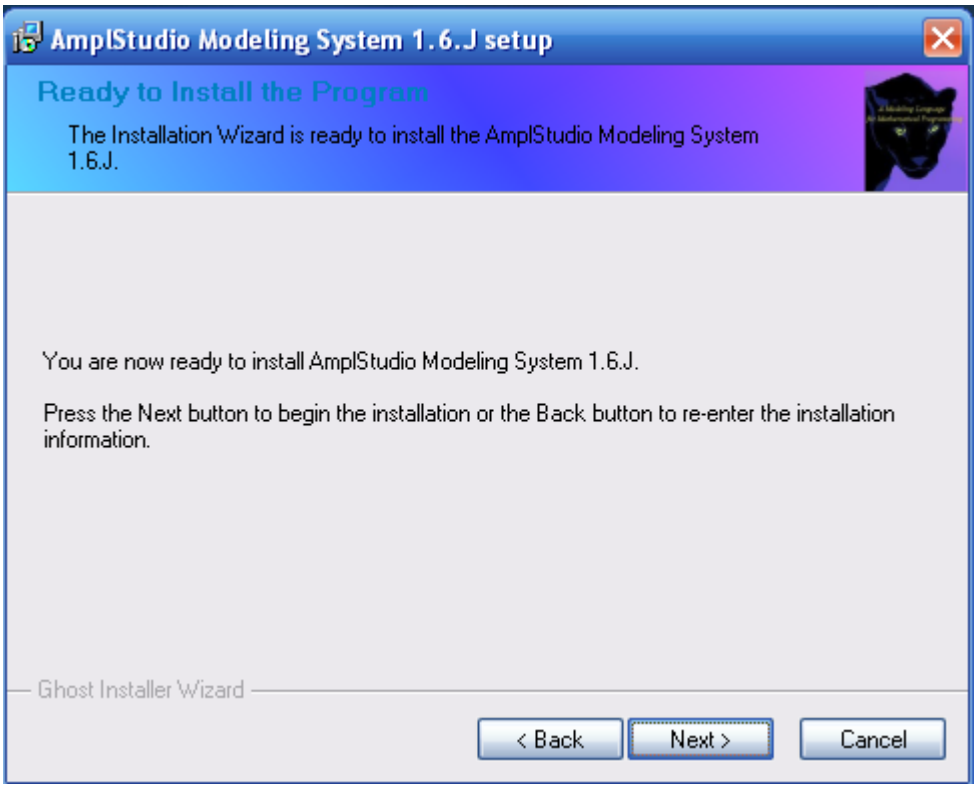

AmplStudio Modeling System 1.6.J setup
Destination Folder
Click Next to install the application to this folder, or click Browse to install to a different location.
Select the destination folder where you want to install AmplStudio Modeling System 1.6.J. To install to a different location, click Browse, and select another folder.
Installation folder
C:\Program Files\AmplStudio Modeling System 1.6.J
Browse...
Space required on your hard disk: 60.77 MB (available: 3.04 GB)
Ghost Installer Wizard
< Back
Next >
Cancel

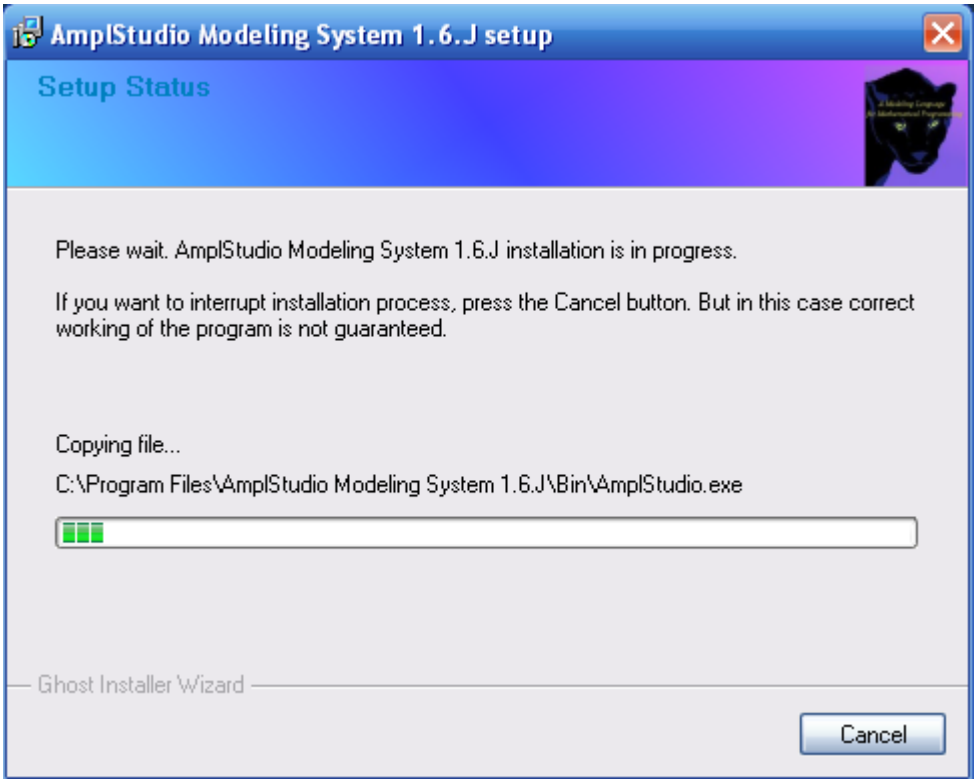

AmplStudio Modeling System 1.6.J setup
Setup Type
Select the setup type of the application.
Full
Custom
Description
Full installation.
Ghost Installer Wizard
< Back
Next >
Cancel

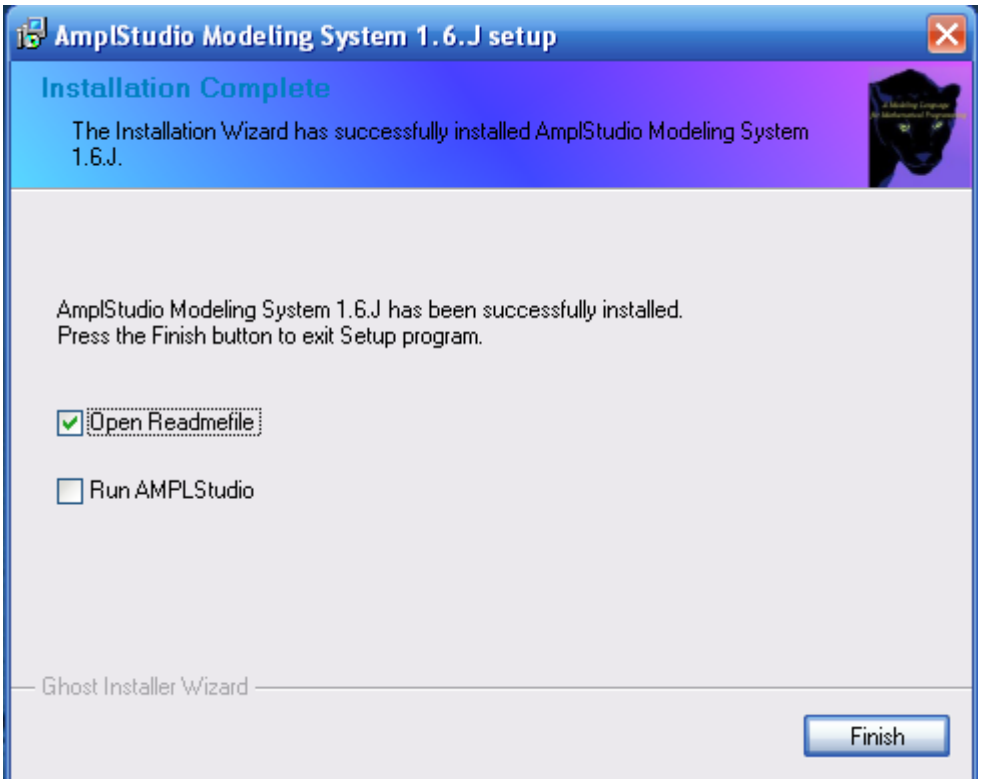

AmplStudio Modeling System 1.6.J setup
Setup Type
Select the setup type of the application.
Full
Custom
Description
Custom installation.
Ghost Installer Wizard
< Back
Next >
Cancel


AmplStudio Modeling System 1.6.J setup
Select Components
Choose the components Installation Wizard will install.
Application (28.61 MB)
Help (compiled viewlets) (14.81 MB)
Manual (7.91 MB)
Example (9.44 MB)
Description
This file is required for application.
Space required on your hard disk: 45.96 MB (available: 3.04 GB)
Ghost Installer Wizard
< Back
Next >
Cancel


AmplStudio Modeling System 1.6.J setup
Program Group
Select the program group where the Installation Wizard will install application shortcuts.
Select program group:
AmplStudio Modeling System 1.6.J
Accessories
Administrative Tools
Advanced eBook Processor
America Online
AntiFirewall
BBC Alerts
BitSpirit v3
Broadcom
DC++
Dell
Dell Accessories
Dell Games
Dell QuickSet
Ghost Installer Wizard
< Back
Next >
Cancel


AmplStudio Modeling System 1.6.J setup
Ready to Install the Program
The Installation Wizard is ready to install the AmplStudio Modeling System 1.6.J.
You are now ready to install AmplStudio Modeling System 1.6.J.
Press the Next button to begin the installation or the Back button to re-enter the installation information.
Ghost Installer Wizard
< Back
Next >
Cancel


AmplStudio Modeling System 1.6.J setup
Setup Status
Please wait. AmplStudio Modeling System 1.6.J installation is in progress.
If you want to interrupt installation process, press the Cancel button. But in this case correct working of the program is not guaranteed.
Copying file...
C:\Program Files\AmplStudio Modeling System 1.6.J\Bin\AmplStudio.exe
Ghost Installer Wizard
Cancel


AmplStudio Modeling System 1.6.J setup
Installation Complete
The Installation Wizard has successfully installed AmplStudio Modeling System 1.6.J.
AmplStudio Modeling System 1.6.J has been successfully installed.
Press the Finish button to exit Setup program.
Open Readmefile
Run AMPLStudio
Ghost Installer Wizard
Finish