The built-in functions are volatile, that is they update if their input data changes. In contrast, procedures such as Goal Seek and Solver and the routines in the Analysis ToolPak are static. The results form a 'data dump' not linked to the originating data. Therefore if the input data changes, the procedures have to be repeated.

3 Introduction to VBA

This chapter introduces the use of VBA and macros within Excel, and attempts to do so in the context of examples where VBA enhances spreadsheet functionality. Whilst not intended as a full introduction to programming in VBA, it argues the case for mastering some VBA, touches on the 'object-oriented' focus of the language, and suggests an incremental approach to mastering the code. Most of the examples in this chapter involve simple code, the first 'hands-on' examples being in section 3.3.3. However, in section 3.6, several applications are developed more fully and some fruitful areas for macros identified. These include subroutines to produce charts, to calculate and display normal probability plots, and to generate the efficient frontier. A related workbook, VBSUB.xls, contains the examples discussed in the chapter and can be used for reference. However, the reader is advised to start by entering code in a new workbook (without opening the VBSUB workbook).

Throughout the book, the VBA coding used for routines is reviewed. The functions and macros developed in subsequent chapters build on the basic foundations laid down in this and the next chapter.

3.1 ADVANTAGES OF MASTERING VBA

Macros form an important part of an experienced user's palette for two main reasons: they are an excellent way to control repeated calculations, and they can be written to assist third-party users who might have less familiarity with spreadsheets. From our perspective, the main objective of mastering VBA is to be able to automate calculations using functions and macros. Spreadsheet models are more robust if complicated sequences of calculations are replaced by function entries. Excel provides an excellent range of functions and VBA can be used to extend that range. In addition, VBA macros are a useful way to produce charts and to automate repeated operations, such as simulation. In both instances, the resulting procedures are programs in VBA, but in the first case, they are user-defined functions and in the second, macros (or subroutines). Whilst the code is largely common to both types of procedure, this chapter focuses on writing macros, whereas Chapter 4 deals solely with user-defined functions.

The programming language used in Excel is called Visual Basic for Applications or VBA for short. The word *basic* confirms that the program is derived from the ancient BASIC mainframe language, which means VBA tends to be relatively inefficient for tasks involving very large volumes of computations. As PCs become ever more powerful and spreadsheets accumulate many features, the boundary between spreadsheets and dedicated computational packages has become blurred. The question to pose at this stage is what is the best way to achieve a task, such as finding the 'efficient frontier', or simulating the performance of a stock, or evaluating the 'eigenvectors' of a matrix. The answer will depend on the aims of the user. The spreadsheet will be slower, but the calculation process will be easier to follow. In contrast, the dedicated package will be faster, but the process of calculation more remote and less comprehensible.

It is important to match your usage of VBA to the task in hand, and to be selective in the parts of VBA that you use, bearing in mind that spreadsheets do not provide the answer to all questions. Using VBA subroutines to automate repetitive tasks is efficient programming; using VBA to create your own simple functions is efficient; but trying to program bullet-proof interaction for a novice user accessing a complex spreadsheet model seems like the road to misery. Whilst seasoned applications programmers can probably tackle this task, the pertinent question is should you? VBA is not a panacea for all programming tasks, and should be used selectively since more efficient computing methods exist for some computationally intensive tasks.

Excel's macro recorder translates user keystrokes into VBA code and can be used to suggest appropriate coding. While it is possible to use the recorder and remain ignorant of the underlying VBA code, the aim of this chapter is to provide some explanations and demonstrate practical applications, thus allowing you to become a more efficient user of the whole Excel package. Probably the best approach at first is to use the recorder to generate rudimentary code, then to selectively edit (or rewrite) the results as you become more familiar with VBA.

It is important to be aware of the distinction between VBA subroutines and functions. Whilst both are distinct, named blocks of source code, the raison d'être of functions is to return values. Typically subroutines accept no inputs, but they carry out a sequence of spreadsheet commands (which may use values from cells in the spreadsheet, and change the values in particular cells). In contrast, functions can accept inputs (or 'arguments'), they carry out a series of predominantly numerical calculations away from the spreadsheet, and return a single value (or a single array). However, for both subroutines and functions, the spreadsheet is the first development tool.

3.2 OBJECT-ORIENTED ASPECTS OF VBA

A few concepts that you need to grasp follow from the fact that VBA is an **'object-oriented'** programming language. Each Excel object represents a feature or a piece of functionality in Excel, e.g. workbooks, worksheets, ranges, charts, scenarios, etc. are all Excel objects as is Excel itself (the Application object). You program in VBA to manipulate the properties and apply methods to Excel objects.

Many statements can be made about objects and VBA, but essentially they can be condensed into four rather cryptic statements. Textbooks and VBA help screens abound with explanations, but overall, these tend to be confusing.

The first statement is *Objects come in collections*. For example, the Workbooks collection consists of all open workbooks, similarly the Worksheets (or Sheets) collection (all the sheets in a workbook), the Scenarios collection (all scenarios associated with a particular sheet), the Charts collection (all the charts on a sheet), etc. However, some objects come as singular objects (i.e. collections of one member only), for example, Excel has only one Application object (itself) and for any cell on the spreadsheet there is only one Font object (although this object has several properties, such as Name, Size, etc.). These are singular objects that are referenced directly, e.g. by simply writing Application. or Font. (the object followed by a 'fullstop'). Individual objects in collections are referenced by indexing the collection either by number (1, 2, 3...) or by name, e.g. Workbooks(1). or Sheets("inputs"). The Range object is by definition a singular object, but notice that it is referenced in a way that is similar to that of a collection, either by name or by address, e.g. Range("data"). or Range("A1:B20").

The second statement is *Objects are arranged in a hierarchy*. The following sequence illustrates the hierarchy of objects in Excel. It shows how the cell range named 'data' in sheet 'inputs' of the Model.xls workbook is referenced via its position in the hierarchy:

```
Application.Workbooks("Model.xls").Sheets("inputs").Range("data")
```

It is not necessary to use the full hierarchy, as long as the identification of the cell range (here named 'data') is unique. If the Model.xls workbook is the active book when the VBA code is executing, then Sheets("inputs").Range("data") is adequate; or referencing the active workbook explicitly:

```
ActiveWorkbook.Sheets("inputs").Range("data")
```

Similarly, if only the Model.xls book is open and the 'inputs' sheet is currently active, then ActiveSheet.Range("data") is adequate and simpler to write.

The third statement is *Objects have properties*. Properties are the attributes of an object, the values or settings that describe the object. VBA can be used either to set a property or to get a property setting. Property values are usually numbers, text, True or False and so on. You can control Excel objects by using VBA to change their properties. An example is:

```
Application.ScreenUpdating = False
```

This line of code stops the Excel screen updating during the running of a macro. ScreenUpdating is a property of the Application object, taking values True/False.

Another pair of examples is:

```
Range("B23").Name = "month2"
Range("B23").Value = 4000
```

which gives cell B23 the name 'month2' and value 4000. The syntax follows the 'Object.Property' style. In these examples, 'Application' and 'Range' refer to objects, whereas 'ScreenUpdating', 'Name' and 'Value' are properties of the objects. The following example takes a setting from a spreadsheet cell, and assigns it to variable 'firstval':

```
firstval = Range("B23").Value
```

The fourth statement is *Objects have methods*. Methods are a set of predefined activities that an object can carry out or have applied to it. Here are some examples of methods applied to Range objects:

Range("A1:B3").Select	which selects the cell range A1:B3
Range("A1:B10").Copy	which copies the cell range A1:B10
Range("storerange").PasteSpecial	which pastes the contents of the Clipboard from the previous command to cell 'storerange'

The syntax follows the 'Object.Method' style. In the above examples, the objects are instances of the 'Range' object while 'Select', 'Copy' and 'PasteSpecial' are methods which act on the objects. Workbook objects and Worksheet objects also have methods, for example:

Workbooks("Model.xls").Activate	makes Model.xls the active workbook
Sheets("inputs").Delete	deletes the sheet called 'inputs'

There are some minor exceptions to the above explanation. But if you record your code, you never need to worry about the distinction between properties and methods. You can also refer to the Excel Object Browser in the Visual Basic Editor to check the correct syntax to accompany specific objects.

3.3 STARTING TO WRITE VBA MACROS

Mastering any language is a cumulative process, part formal learning of rules, part informal experimenting and testing out coding ideas. VBA code is developed in the Visual Basic Editor, Excel's code writing environment. Since Excel 97, this has been enlarged and enhanced somewhat from the Excel 5/7 version, providing more support for writing code. We start by examining the code for some simple examples. VBA's MsgBox function is introduced as it provides a simple mechanism for displaying calculated values and feeding back simple diagnostic messages. We show how to generate code using the recorder and contrast the results with written code. The advantages of the dual approach of recording and editing are outlined.

3.3.1 Some Simple Examples of VBA Subroutines

A subroutine is a standalone segment of VBA code; it constitutes the basic building block in programming. The subroutine performs actions and consists of a series of VBA statements enclosed by Sub and End Sub statements. Its name is followed by empty parentheses (unless it takes an argument passed to it from another subroutine).

For example, the LinkOne() subroutine links the contents of one cell to another on the active sheet. The code starts with the Sub keyword, followed by the macro name, LinkOne. A comment statement outlines the purpose of the macro. (Text preceded by an apostrophe is ignored in processing, so this is a useful way to add explanatory comments to the code.) The first statement uses the Value property of the Range object, the second the Formula property:

```
Sub LinkOne()
'enters a value in B3, then links cell B4 to B3
Range("B3").Value = 4000
Range("B4").Formula = "=b3"
End Sub
```

The LinkOne routine works on the active sheet. However, if there is ambiguity about the target sheet for these operations, the cell's range reference should be more precise. The subroutine below sets the value of cell B3 in sheet 'Inputs' to 4000. The value in cell B3 is then copied into cell B4 in the same sheet:

```
Sub LinkTwo()
'enters a value in B3, then links cell B4 to B3 on Inputs sheet
Sheets("Inputs").Range("B3").Value = 4000
Sheets("Inputs").Range("B4").Formula = "=b3"
End Sub
```

Or if the actions are to take place on the currently active sheet, ActiveSheet. can be used instead of Sheets("Inputs").

The forgoing examples illustrate the type of code which Excel's recorder would generate in response to actual keystrokes. Contrast it with the code for the subroutine Factorial below. This subroutine computes a factorial (denoted fac) for the number in cell B5 (referred to as num) and enters the result in cell C5. It has two new features. Firstly, the code uses variables, *i*, fac and num, which subsequently are assigned values. Secondly, the factorial is computed by repeated multiplication in a For...Next loop with index *i*. (The counter *i* starts at value 1 and increases by 1 each time it passes through the loop until its value goes above the upper limit, num.)

```
Sub Factorial()
'calculates the factorial of a number in B5
num = Range("B5").Value
fac = 1
For i = 1 To num
  fac = i * fac
Next i
Range("C5").Value = fac
End Sub
```

Other than using and returning values to the spreadsheet, this code is straightforward Basic and has little to do with Excel per se. This type of code must be entered manually.

3.3.2 MsgBox for Interaction

In general, a subroutine gets its input either from spreadsheet cells or from the user (via the screen) and its outputs are either pasted into the workbook or displayed to the user. Two useful display screens are generated by the VBA built-in functions, MsgBox() and InputBox(). The simpler of these, MsgBox(), displays a message on screen then pauses for user response, as is illustrated below:

```
Sub MsgBoxDemo1()
MsgBox "Click OK to continue"
End Sub
```

The message (or 'prompt') is within the quotation marks and must be present. Brackets around the prompt are optional in this simple instance of MsgBox. However, when an answer is required from MsgBox, the brackets are required to indicate that MsgBox is being used as a function. The MsgBox display can be made more useful if a variable is 'concatenated' (or 'chained') on to the prompt using ampersand (&). For example, in the Factorial subroutine, the result could be communicated to the user with the statement:

```
MsgBox "Factorial is "& fac
```

as in the code below. In this version, the user supplies the number via an InputBox. This VBA function pops up a dialog box displaying a message and returns the user's input, here called num. The argument in the InputBox syntax below is the message displayed, also known as the 'prompt'. Here InputBox has brackets to indicate it is being used 'as a function', that is, to return an answer, num. The factorial is built up multiplicatively in a For...Next loop:

```
Sub Factorial()
'calculates the factorial of a number
fac = 1
num = InputBox("Enter number ")
For i = 1 To num
fac = i * fac
Next i
MsgBox "Factorial is "& fac
End Sub
```

We use this simple factorial example to explore elements of programming macros in section 3.4.

3.3.3 The Writing Environment

The Visual Basic Editor (referred to as the VBE) is where subroutines are written, tested out and debugged. We assume Excel's workbook environment, its menu, commands and toolbars are familiar territory. When developing VBA procedures, it helps to have Excel's Visual Basic toolbar visible in the Excel window (shown in Figure 3.1). As with other toolbars, this has buttons which are programmed to carry out sequences of relevant menu commands, here Tools Macro commands such as Run Macro, Record Macro and the VBE (first, second and fourth from left), etc.

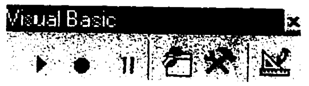


Figure 3.1 Excel's Visual Basic toolbar

VBA code is written (or recorded) not in worksheets but in Module sheets. In Excel 97 (and subsequent versions) these appear in a different window, the Visual Basic (VB) window, and the process of developing macros involves switching to and fro between the Excel and VB windows (using the keystroke combinations Alt+Tab or Alt+F11) or the equivalent buttons.

To try this out for yourself, open a new workbook in Excel, choose Tools then Macro then Visual Basic Editor to go to the VB window. (Or simply click the VBE button on the VB toolbar.) The VB window consists of the Project Explorer and the Properties window as well as the Code window (which is not usually present until the workbook has Modules). The Code window is where programs are written and/or keystrokes recorded. From the VB window you can refer to the Excel Object Browser to check the correct VBA syntax to accompany specific objects. (In the VB window, choose View to access the Object Browser, then choose Excel in the top left-hand References textbox.)

The VBE also contains numerous debugging and editing tools that can be accessed from its command bar or via its own specialist toolbars. More detail on the Visual Basic Environment is at the end of the chapter in Appendix 3A (and could be skimmed now).

3.3.4 Entering Code and Executing Macros

VBA code is entered on a Module sheet. To add one to the workbook, on the VB window menu, choose Insert then Module to get a blank module sheet in the Code window. Begin

typing in the Code window, for example:

```
Sub MsgBoxDemo()
MsgBox "Click OK to continue"
End Sub
```

At the end of the first statement, when you press Enter, Excel adds the End Sub statement. You may also notice the 'quick information' feature, which assists with the entry of recognisable bits of syntax.

There are several ways to run the subroutine:

- From the Code window, with the pointer positioned anywhere in the body of the routine's code, choose Run then Sub/UserForm (alternatively click the Run button on the VB Standard toolbar underneath the VB window menu).
- From the Excel window, choose Tools then Macro then Macros, then select MsgBoxDemo macro and Run it or click on the VB toolbar Run button.
- You can also execute the macro using a chosen keyboard shortcut (for example, the combination of Ctrl+Shift+G) or, if you're really trying to impress, link the macro to a button on the spreadsheet.

When you run your code, execution may stop if a run-time error occurs. The ensuing dialog box suggests the nature of error, although for novice programmers the comments are somewhat opaque. Having noted the error message, on clicking OK the cursor jumps to the statement containing the error and the subroutine declaration statement is shown highlighted (in yellow). Having corrected the error, to continue execution, click Reset on the Run menu (or the Reset button with a black square) to remove the yellow highlight and run the macro again.

For further practice, enter the Factorial subroutine in the updated form given below. In this version, the loop has been replaced by the Excel function FACT, which calculates factorials. Note that to use Excel's functions in VBA code, it is sufficient to preface the function name FACT with Application. (or WorksheetFunction.):

```
Sub Factorial()
'calculates the factorial of a number
num = InputBox("Enter integer number ", "Calculate Factorial ")
fac = Application.Fact(num)
MsgBox "Factorial is "& fac
End Sub
```

To avoid the memory overflowing, run subroutine Factorial with integer numbers less than 25 say.

Both these simple macros have to be entered manually. However, the code for many operations can be generated by Excel's macro recorder, so next we explore this other approach to macro writing.

3.3.5 Recording Keystrokes and Editing Code

The recorder translates keystrokes into VBA code and has two modes when interpreting cell addresses: its default mode (absolute references) and the relative references mode. Initially you will almost always use the recorder with absolute references, however detail on the alternative mode is given at the end of the chapter in Appendix 3B. As an illustration, a simple macro entering data into cells B8:C10 will be recorded in the absolute (default) mode.

The macro recorder is invoked from the Excel window by choosing Tools, then Macro then Record New Macro. Give the macro a name, say Entries, then click OK. (Alternatively, start recording by clicking the Record Macros button on the VB toolbar.) The Stop Recording button appears on a small toolbar (together with the Relative References button). The Relative References button should not be 'pressed in'.

In the sheet, select cell B8 and enter JAN in B8; move down to B9, enter FEB in B9; move down to B10 and enter Total. Then changing to column C, select C8, enter 300; select C9, enter 400; select C10, entering the formula =SUM(C8:C9) in the cell and click Stop Recording. Click the VBE button to see the code recorded from your keystrokes in the Code window. Hopefully, it should be similar to the code below:

```
Sub Entries()
'entering data using absolute addresses
  Range("B8").Select
  ActiveCell.FormulaR1C1 = "JAN"
  Range("B9").Select
  ActiveCell.FormulaR1C1 = "FEB"
  Range("B10").Select
  ActiveCell.FormulaR1C1 = "Total"
  Range("C8").Select
  ActiveCell.FormulaR1C1 = "300"
  Range("C9").Select
  ActiveCell.FormulaR1C1 = "400"
  Range("C10").Select
  ActiveCell.FormulaR1C1 = "=SUM(R[-2]C:R[-1]C)"
  Range("C11").Select
End Sub
```

When a cell is 'selected', it becomes the ActiveCell (a property of the Application object which returns a range). The entries are made using ActiveCell.FormulaR1C1 to put the entries into cells B8 to C10. When you run the macro, by switching to Excel and clicking the Run Macro button (or from the Code window with Run then Sub/UserForm), notice that the macro always produces the same results, i.e. it always fills the range B8 to C10.

For a VBA novice, the best approach is to record keystrokes to suggest code, then edit to improve, make more concise, and generalise the code. The combination of recording, editing and testing out code is a practical approach. Recorded code can produce excellent results when the actions required consist of sequences of menu commands, as for example in producing a chart of some data, doing a data sort or specifying pivot tables, etc. The recorder produces concise code encapsulating quite complex instructions. In contrast, if the actions recorded are mostly a sequence of cursor moves, the recorder generates very verbose code. The best approach in this instance is to write the code, or record bits and edit the results.

As an illustration, look back at the code generated for the Entries macro. Recorded code always includes much 'selecting' of cells, but when writing code it is not necessary to explicitly select cells to enter data into them. The following edited version of the code performs the same operations but is more concise:

```
Sub ConciseEntries()
'entering data using absolute addresses
  Range("B8").Formula = "JAN"
  Range("B9").Formula = "FEB"
  Range("B10").Formula = "Total"
  Range("C8").Formula = "300"
  Range("C9").Formula = "400"
  Range("C10").Formula = "=SUM(C8:C9)"
End Sub
```

In fact for a single cell range object, it turns out to be unnecessary to specify the Formula property, so the code is simply:

```
Sub Entries()
'entering data using absolute addresses
  Range("B8") = "JAN"
  Range("B9") = "FEB"
  Range("B10") = "Total"
  Range("C8") = "300"
  Range("C9") = "400"
  Range("C10") = "=SUM(C8:C9)"
End Sub
```

A practical way to tidy up generated code is to comment out (add apostrophes in front of possibly) redundant statements, then test by running the macro to see the effect.

When developing code, it frequently helps to display messages about the stage of processing reached, particular intermediate values in calculations, etc. The VBA MsgBox function is invaluable for this. At the editing stage it helps to add comments concisely describing the action performed by the macro, together with any unusual features or requirements. More importantly, any useful possibilities for generalising the scope of the code should be explored.

For reference, all the macros discussed in this section are stored in ModuleA and can be run from the IntroEx sheet of VBSUB.

Reviewing the Factorial subroutine developed in the previous paragraphs, it is clear that code of this type with its use of variables and a For...Next loop will never result from recording keystrokes. It has to be entered manually. Therefore, some guidelines for conducting this more conventional type of programming are set out in the following section.

3.4 ELEMENTS OF PROGRAMMING

In the financial modelling context, much of the code you will want to write will have as its objective the control of numerical calculations. This involves assigning variables, applying proper control structures on the flow of processing and reporting results. These tasks are common to programming whatever the language. VBA programs differ only in that they often originate from spreadsheet models, where tasks are performed by menu commands and where crucial calculations are in cell formulas, many involving Excel functions. Fortunately, it is easy to incorporate Excel functions in VBA code. Rather than replace the spreadsheet model, VBA programs increment or enhance their functionality. We briefly review a few relevant aspects of programming concerning the use of variables, in particular array variables, structures that control the flow of processing and the use of Excel functions in VBA.

3.4.1 Variables and Data Types

In programming, variables are used to store and manipulate data. Conventionally, data comes in different forms or 'types' and this influences the amount of memory allocated for storage. In most numerical calculation, the main distinctions are between numbers (conventional decimal system numbers otherwise known as scalars), integers, Booleans (True or False), and dates. There are also string variables that hold textual as opposed to numerical information. VBA has a useful data type, variant, which can represent any data type. It is the default data type, which can be useful when a variable's type is uncertain. However, the variant type consumes additional memory, which can affect processing speeds if large amounts of numerical processing are required. There is also an 'object' variable type which can refer to an object such as a range or a worksheet. See Green (1999) for more details of the different data types.

From the viewpoint of novice VBA programmers, it is important to explicitly declare all variables used, as this cuts down errors in coding. This can be forced by ensuring that the Option Explicit statement is included at the top of each Module sheet. (The VBE can be set up to ensure that this happens automatically–via Tools then Options on the VB menu, then checking the 'Require Variable Declaration' code setting.) At run-time, the VBA compiler produces an error message if any undeclared variable is encountered. This is particularly useful in detecting mis-spelt variable names in the code. Declaring the data type (as integer, string, Boolean, etc.) is less important in the financial modelling context, but it can be used as a way of indicating (to the user) a variable's purpose in the program.

Variables are declared with the Dim keyword as shown in the third statement of the Factorial subroutine. Here, the two variables have been declared but not data-typed (as integer say). By default they are variant in type:

```
Sub Factorial()
'calculates the factorial of a number
Dim fac, num
num = InputBox("Enter number ", "Calculate Factorial ")
fac = Application.Fact(num)
MsgBox "Factorial is "& fac
End Sub
```

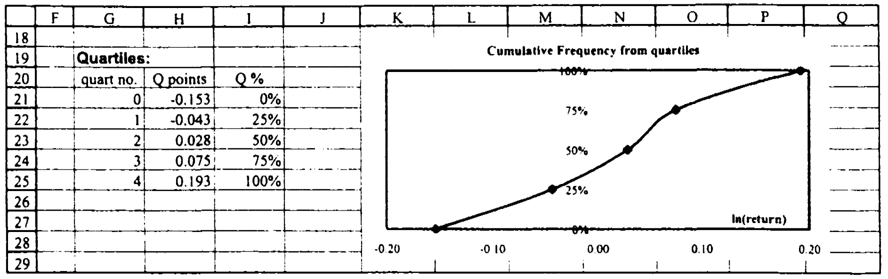
(As an aside, notice that InputBox(), which returns the user's input, has two input parameters or 'arguments' here. The first argument is the prompt 'Enter number ', the second (optional) argument is the title of the dialog box.)

In summary, all variables used in subroutines should be explicitly declared using the Dim keyword. The statement Option Explicit at the top of a Module sheet ensures that all variables must be declared for the subroutine to work correctly.

3.4.2 VBA Array Variables

Where sensible, variables can be grouped by name into arrays (vectors or matrices). For example, the quartiles of the cumulative distribution of a data set can be represented by the array variable, qvec(). Individual elements of the array are referenced as qvec(0), qvec(1), etc. Using the log returns data discussed in the previous chapter, Figure 3.2 shows the quartiles calculated with the Excel QUARTILE function in cells H21:H25, with the first and last Q points being the minimum and maximum of the data set, −0.043 being

the value below which 25% of the data falls, etc. So array qvec() could take its values from the cells in range H21:H25, with qvec(0) taking value −0.153, qvec(1) taking value −0.043, etc.



	F	G	H	I	J
18					
19		**Quartiles:**			
20		quart no.	Q points	Q %	
21		0	-0.153	0%	
22		1	-0.043	25%	
23		2	0.028	50%	
24		3	0.075	75%	
25		4	0.193	100%	
26					
27					
28					
29					

Figure 3.2 Quartile values for log returns data as in Data sheet of VBSUB

Extending the idea, array variables can have several dimensions, e.g. the two-dimensional array variable PQmat() could represent a 5 by 2 array consisting of a column of quartiles and a column of the corresponding 'left-hand side' percentages of the distribution. For example, array variable PQmat() could take its values from the 5 by 2 cell range H21:I25 shown in Figure 3.2. Although there are few rules for the names given to array variables, it is helpful to choose names that distinguish between arrays with one dimension (vectors) and two dimensions (matrices). Hence our use of names such as qvec and PQmat.

As with other variables, array variables should be declared before use in the subroutine. By default, VBA numbers its arrays from 0. So the array qvec(), if declared in the statement:

```
Dim qvec(4)
```

will consist of five elements. By extension, the two-dimensional array for two rows and four columns of quartiles and percentages, PQmat(), if declared in the form:

```
Dim PQmat(1,4)
```

will consist of 10 elements, again assuming the default numbering base for arrays is 0.

If you want VBA to use 1 as the lowest index, you must state Option Base 1 at the start of your Module sheet. If this were done, the array variable qvec(4) would have only four elements, qvec(1), qvec(2), qvec(3) and qvec(4). In fact in most of the VBA procedures developed in this book, the array base is chosen as 1.

The effect of the Option Base setting is illustrated by the following ArrayBase macro. It produces different results according to the Option Base statement at the top of its Module sheet. Since the VBA array function Array() is used to enter the actual array, it is not necessary to specify in advance the dimensions for array variable avec:

```
Sub ArrayBase()
'gives b=20 if Option Base is 1; gives b=30 in absence of Option Base statement
Dim avec, b
avec = Array(10,20,30)
b = avec(2)
```

```
MsgBox "b is "& b
End Sub
```

Often, the number of elements required in an array is not known at the outset, but results from the operations of the macro. This so-called 'dynamic array' does not have a preset number of elements. It is declared with a blank set of brackets, e.g. Dim qvec() or Dim PQmat(). However, before using any non-dimensioned array, a ReDim statement is used to tell VBA how many elements are in the array. In the example below, the named range 'dvec' contains results that are to be stored in array variable cvec. After declaring cvec as an array variable, its dimensions are decided by counting the elements of the named range (dvec), leading to the ReDim statement required before cvec can be used:

```
Sub SelectCount()
're-Dim data array cvec
Dim n, cvec()
Sheets("Data").Range("dvec").Select
n = Selection.Count
ReDim cvec(n)
End Sub
```

Working with arrays in VBA is more complicated than using Excel's array formulas (such as SUM, SUMPRODUCT and the matrix functions). In some cases, array processing can be handled entirely by Excel functions taking array inputs. Note that if individual array elements do not need to be accessed in the code, the array can be referred to by a variable name such as qvec, without the requirement to state its dimensions. However, frequently array processing involves element-by-element operations, which are best handled systematically in a loop. In this case, individual qvec(*i*) elements have to be identified and typically Dim qvec() and ReDim statements need to be included. The coding requires much more care if errors are to be avoided. There is also ambiguity about the layout of VBA arrays constructed by Excel, in particular as to when they are row vectors and when column vectors. To add possible further confusion, if an array is used as an input to a procedure, Excel numbers the array elements from 1.

3.4.3 Control Structures

In common with other programming languages, VBA provides several control structures to sequence the flow of processing. These include conditional statements such as If...Then...Else, which tests a condition and alters the flow of execution depending on the result of the test; and Select Case, which selects a branch from a set of conditions (or cases). A looping structure allows you to run a group of statements repeatedly. Some loops repeat statements a specific number of times, such as the For...Next loop with its counter which we have already used to calculate the factorial. Other constructions, such as the Do While...Loop, repeat statements until a condition is True (or until a condition is False). We look at simple examples of If...Then in this section, and further examples of the For...Next loop and the Do While...Loop in the following section. Textbooks (such as Green, 1999; Walkenbach, 1999; Wells and Harshbarger, 1997) explain these structures much more extensively and can be referenced for other examples.

As noted in the Factorial subroutine, the VBA interaction function, InputBox, returns the user's input. To make our procedure more robust, wrong inputs should be filtered out and amended inputs requested from the user. One approach is to test the input to

check that it is numeric, and to discard it if not. The code below illustrates two uses of the If...Then structure. The first instance is an in-line If...Then with no additional conditions on following code lines. This tests the variable numtype (resulting from a VBA function IsNumeric) and if numtype = True calculates the factorial. The test and ensuing action are contained in a single code line. In the second case, the structure takes the block form of If...Then. If the user input is non-numeric, the user is warned and no calculation takes place. When the conditional statements finish, there is an End If:

```
Sub Factorial()
'calculates the factorial of a number
Dim fac, num, numtype
num = InputBox("Enter number ", "Calculate Factorial ")
numtype = IsNumeric(num)                                   'either True or False
If numtype = True Then fac = Application.Fact(num)         'in-line If Then
If numtype = False Then                                    'block If Then End If
MsgBox ("Not an integer number. Try again") 'don't proceed
End If
End Sub
```

We make much use of If...Then...Else in Chapter 4 on user-defined functions.

3.4.4 Control of Repeating Procedures

The early version of the factorial subroutine (in section 3.3.1) illustrates the simplest type of repeating procedure, the For...Next loop. As a different example, consider the code for the Quartiles subroutine below. This calculates the quartiles of a data set (taken from a named range, dvec, in the Data sheet of the VBSUB workbook), and displays the results to the user one-by-one in a sequence of dialog boxes. (You may wish to access this sheet of VBSUB to follow the argument in the next paragraphs.)

```
Option Base 0

Sub Quartiles()
'displays quartiles of range named 'dvec'
  Dim i As Integer
  Dim quart                               'to hold the current quartile
  Dim dvec As Variant                'col vec of data
  'fill array variable dvec from spreadsheet range named dvec
  dvec = Worksheets("Data").Range("dvec")

  'calculate quartiles one at a time & display
  For i = 0 To 4
    quart = Application.Quartile(dvec, i)
  MsgBox "Quartile no. "& i & "has value "& quart
  Next i
End Sub
```

Inspection of the current range names in the VBSUB workbook, via the commands Insert then Name then Define shows that the name dvec is attached to the LnReturns data range in column E of the Data sheet. In the code for the Quartiles subroutine, we also use the name dvec for the array variable containing the data set values. To indicate (to us) that dvec is a vector not simply a scalar variable, we declare it as a Variant. Examining the code, we note that after filling array dvec from the spreadsheet cells, the individual quartiles are calculated and displayed element-by-element in the For...Next loop. Note that to use the Excel function QUARTILE in the VBA code, it needs to be prefaced by Application. or WorksheetFunction., as explained in the next section.

The MsgBox prompt has the values of the For...Next counter and the current quartile concatenated, which works, but is not the most satisfactory solution for handling output. We give an improved version of the Quartiles subroutine in section 3.5.

Another useful structure for repeated operations is the Do While...Loop. Repeated operations continue whilst a condition is satisfied and cease when this is not the case. The DelDuplicates subroutine, which searches for and deletes duplicate rows in a database, illustrates the use of the Do While...Loop. Having sorted the database on a specific 'code' field, the entries in the code column are processed row-by-row. If the values of the currentCell and the nextCell are the same, the currentCell row is deleted. Here currentCell and nextCell are used as object variables, since they reference objects (here Range objects). Because they are object variables, their values have to be assigned with the keyword Set.

```
Sub DelDuplicates()
Dim currentCell, nextCell
Range("database").Sort key1:= Range("code")
Set currentCell = Range("code")
Do While Not IsEmpty(currentCell)
      Set nextCell = currentCell.Offset(1, 0)
      If nextCell.Value = currentCell.Value Then
         currentCell.EntireRow.Delete
      End If
      Set currentCell = nextCell
  Loop
End Sub
```

This subroutine illustrates some of the advantages of VBA's online help. This can be accessed from the VBE, via the Help command and the Help Index. Here the index entry on Delete Method elicited the above subroutine, which provided exactly the operation required. Unfortunately, not all queries posed to VBA's online help are so fruitful, nevertheless, it contains much useful information on syntax. These sections have only touched on a few of the most widely used control structures. Other useful sources for VBA programming techniques are textbooks such as Green (1999), Walkenbach (1999) or Wells and Harshbarger (1997).

3.4.5 Using Excel Functions and VBA Functions in Code

As seen in the Quartiles subroutine, to use an Excel function in a VBA procedure, you need to prefix the function name with Application. In Excel 2000, the function name can be prefixed by WorksheetFunction or Application. For compatibility with earlier versions of Excel, we continue to use Application. In contrast, VBA functions require no prefix, as we have seen with MsgBox, IsNumeric, InputBox, etc. In fact, there are relatively few specific numeric functions built into VBA, currently just Abs, Cos, Exp, Int, Log, Rnd, Sgn, Sin, Sqr and Tan, where Log is the natural log (usually denoted Ln). So some VBA functions are spelt differently from the parallel Excel function (e.g. Sqr in VBA for square root, whereas the Excel function is SQRT). To resolve the conflict, if both a VBA and an Excel function exist for the same calculation, the VBA form must be

used rather than the Excel function. (For example, write Log rather than Application.Ln when you require the natural logarithm of some quantity; similarly write Sqr rather than Application.Sqrt.)

3.4.6 General Points on Programming

To conclude this section, some more general advice about the process of tackling a programming project is given. In structured programming, the objective is to write programs that progress in an orderly manner, that are easy to follow and, most important, easy to modify.

It helps to work out on paper the overall application's main stages, breaking the overall task down into distinct and separable subtasks. Then the program can be built in steps, coding and testing a series of self-contained and separate subprograms to accomplish the subtasks. Where possible, it helps to keep the subprogram segments reasonably small. One of the tenets of structured programming is that a code segment should have one entry and one exit point. Program control should not jump into or exit from the middle of code segments. If this practice is adhered to, the process of linking the separate segments of code together is greatly simplified.

Useful programs frequently require updating or modifying in different ways. Program logic is made easier to follow if comments are added when the program is first written. Wherever possible, important constants in the numerical calculations should be parameterised (explicitly declared) so that they are easy to change if the program is to be used for different circumstances. If code can be written to handle slightly more general situations, it is usually worth the extra effort.

3.5 COMMUNICATING BETWEEN MACROS AND THE SPREADSHEET

Having introduced the use of variables and control structures in programming, this section discusses how VBA macros can obtain inputs directly from the spreadsheet and how results can be returned. It concentrates on the communication between macros and the spreadsheet.

In most cases, a subroutine consists of three parts: input of data, calculations (or manipulation of inputs), then output of results. Writing the VBA code for the calculations usually involves conventional programming techniques. The novel aspect of VBA programming is the interaction with the spreadsheet. Taking the three parts separately:

- Input can be from spreadsheet cells or directly from the user via dialog boxes, with the input stored in variables.
- Output can be written to cells or displayed in dialog boxes.
- Calculation can be done in code ('offline'), or by using formulas already in cells or by writing to cells via statements in the VBA subroutine.

The following three versions of the familiar factorial example illustrate combinations of different types of input, calculation and output.

Subroutine Factorial1 gets input from the user, employs the Excel FACT function for the calculation and returns the output via MsgBox. The subroutine does not interact with the contents of the worksheet at all:

```
Sub Factorial1()
Dim fac, num
num = InputBox("Enter number ", "Calculate Factorial ")
fac = Application.Fact(num)
MsgBox "Factorial is "& fac
End Sub
```

In contrast, Factorial2 takes its input from a spreadsheet cell (B5) and returns the factorial answer to another cell (C5):

```
Sub Factorial2()
'gets number from spreadsheet, uses Excel Fact function, returns answer to spreadsheet
Dim fac, num
num = Range("B5").Value
fac = Application.Fact(num)
Range("C5").Value = fac
End Sub
```

As a further variation, in Factorial3 the input number is written to a cell (B6), the factorial formula is written to an adjacent cell with the code:

```
Range("C6").Formula = "=FACT(b6)"
```

and the result displayed back to the user.

```
Sub Factorial3()
'gets number from InputBox, calculates factorial in spreadsheet
'returns answer via MsgBox
  Dim fac, num
  num = InputBox("Enter number ", "Calculate Factorial ")
  Range("B6").Value = num
  Range("C6").Formula = "=FACT(b6)"
  fac = Range("c6").Value
  MsgBox "Factorial is "& fac
End Sub
```

To consolidate, try developing these input–output subroutines with your own formulas and requirements. For reference, the factorial subroutines are all in ModuleF of the VBSUB workbook, and are best run from the IntroEx sheet. When you adapt or write your own subroutines, you may wish to take note of the material in Appendix 3A on the Visual Basic Editor, especially the paragraphs on stepping through and debugging macros.

The process of reading the values of spreadsheet ranges, and writing results to other spreadsheet cells and ranges, is simple if cell-by-cell computation is not required. Suppose the input data is in the spreadsheet range named 'avec', the values of whose cells have to be pasted to another range, with *anchor* cell (the top, left-hand cell) named 'aoutput'. The recorder produces the code in subroutine ReadWrite1, which can be improved by editing out the 'selection' operations, as shown in ReadWrite2:

```
Sub ReadWrite1()                                    'recorded macro
  Application.Goto Reference:="avec"
  Selection.Copy
  Application.Goto Reference:="aoutput"
  Selection.PasteSpecial Paste:=xlValues, Operation:=xlNone, SkipBlanks:=_
```

```
    False, Transpose:=False
  Application.CutCopyMode = False
End Sub

Sub ReadWrite2()                                    'edited version
'reads data, writes values to another range
  Range("avec").Copy
  Range("aoutput").PasteSpecial Paste:=xlValues
  Application.CutCopyMode = False                   'cancels Copy mode
End Sub
```

The coding becomes a little more complicated if the calculation routine has to be performed on the individual cells of avec. These calculations are likely to be carried out in a repeating loop, and the results pasted one-by-one into an output range. Counting the range avec provides the number of cells in avec to be read and values to be pasted to the output range. Suppose the calculation routine produces the running product of the elements of avec. For the calculation, two variables x and y are used, x for the current element of avec, and y for the current product. Suppose the top cell of the input range, avec, is named 'atop' and the top cell of the output range is named 'aoutput'. The following subroutine reads data cell-by-cell, calculates the product to date and outputs the current product at each step:

```
Sub ReadWrite3()
'reads data cell-by-cell, calculates, writes results
Dim i As Integer, niter As Integer                  'niter is counter for iterations
Dim x, y
niter = Range("avec").Count                         'no. of cells in avec
y=1                                                 'initial value for product
For i = 1 To niter
  x = Range("atop").Offset(i - 1, 0)
  y = x *y                                          'calculation routine
  Range("aoutput").Offset(i - 1, 0) = y
Next i
Application.CutCopyMode = False
End Sub
```

Offset(i, j) is a most useful method which returns the cell i rows below and j columns to the right of the referenced cell, here Range("atop"). For example, the cell given by Range("atop").Offset(1, 0) is one row below the cell named atop, etc. Initially x takes the value Range("atop").Offset(0, 0), that is the value of Range("atop").

As a final example of communication between macro and spreadsheet, we reconsider the Quartiles subroutine, improving the coding so that the five quartiles are returned as an array to the spreadsheet. The new subroutine, Quartiles1, retains the five quartile values in the array variable qvec() and outputs this to the previously named range qvec1. (The range name qvec1 is attached to range K20:K24 in the Data sheet, as can be confirmed by choosing Insert then Name then Define and inspecting the dialog box.)

In the code below, the array variable qvec(4) is dimensioned to hold the five values. The input and output array variables, dvec (the data set) and qvec1 (the results array), have been declared as Variants to indicate (to us) that they are not simply scalar variables. Since dvec and qvec1 are not accessed element-by-element, their variable declaration Dim statements do not include the brackets (). After filling array dvec from the spreadsheet range dvec, the quartiles array is assembled element-by-element in the For. . .Next loop. By default, VBA sets up arrays such as qvec() in a worksheet row, so Excel's Transpose function is applied to output the quartiles as a column vector:

```
Option Base 0
Sub Quartiles1()
'pastes 4 by 1 col vector of quartiles into range named 'qvec1'
'requires 2 named ranges in spreadsheet, dvec with data and qvec1
Dim i As Integer
Dim qvec(4)                                  'quartiles array with 5 elements
Dim dvec As Variant                          'col vec of data
Dim qvec1 As Variant                         'results array
  'fill array variable from spreadsheet range
  dvec = Worksheets("Data").Range("dvec")
  'calculate quartiles and assemble as an array
  For i = 0 To 4
    qvec(i) = Application.Quartile(dvec, i)
  Next i
  qvec1 = Application.Transpose(qvec)        'to make a column vector
  'transfer results into spreadsheet range qvec1
  Worksheets("Data").Range("qvec1") = qvec1
End Sub
```

Notice that only one statement is executed in the loop. This illustrates the practice of never putting unnecessary statements in loops. Any statements unaffected by the processing in the loop should always be put outside the loop.

Once again, you should consolidate your understanding by trying out some of these subroutines with your own data and formulas. For reference, the ReadWrite subroutines are in ModuleI and the Quartiles subroutines in ModuleQ of the VBSUB workbook.

This concludes our brief introduction to writing VBA subroutines. For further amplification at this stage, Chapter 2 of Green's (1999) text provides an excellent primer in Excel VBA. The next section contains three further applications, developed and explained at greater length. They represent some realistically useful applications for macros and taken together illustrate further ideas about writing VBA code.

3.6 SUBROUTINE EXAMPLES

In this section, a number of subroutines are developed more fully to illustrate firstly, the process of incremental improvement and secondly, some useful application areas for macros. These examples combine use of the recorder and subsequent code editing. They include subroutines for generating particular types of charts, cumulative frequency and normal probability plots, and repeated optimisation with Solver.

3.6.1 Charts

To start, we develop a subroutine for charting some data such as the cumulative frequency data shown tabulated and plotted out in Figure 3.3.

Suppose the frequency data is in the 'Data' sheet of a workbook in a range named 'chartdata' and the objective is to produce an XY chart with cumulative percentage frequencies on the Y axis. The chart is to be placed as an object on the Data sheet.

Frequency Distribution:	
interval	%cum freq
-0.16	0%
-0.12	10%
-0.08	16%
-0.04	27%
0.00	44%
0.04	52%
0.08	77%
0.12	92%
0.16	98%
0.20	100%

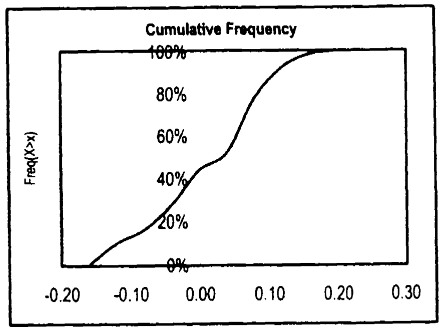


Figure 3.3 Frequency distribution and chart in Data sheet of workbook VBSUB

For a novice VBA user, the code required to manipulate charts is likely to be unknown, so the best approach is to use the recorder to generate the code. The VBA recorder is switched on and the keystrokes required to plot the data in range 'chartdata' are recorded. Wherever possible, use range names (selecting the names from the namebox or clicking the Paste Name button) to specify cell ranges. The code generated for macro here called ChartNew() is likely to be similar to the following:

```
Sub ChartNew()
Charts.Add
  ActiveChart.ChartType = xlXYScatterSmoothNoMarkers                       'WizStep1
  ActiveChart.SetSourceData Source:=Sheets("Data").Range("chartdata"), PlotBy:=_
    xlcolumns                                                              'WizStep2
ActiveChart.Location Where:=xlLocationAsObject, Name:="Data"               'WizStep4
  With ActiveChart                                                         'WizStep3
    .HasTitle = True                                                       'Titles
    .ChartTitle.Characters.Text = "Cumulative Frequency"
    .Axes(xlCategory, xlPrimary).HasTitle = False
    .Axes(xlValue, xlPrimary).HasTitle = True
    .Axes(xlValue, xlPrimary).AxisTitle.Characters.Text = "Freq(X> x)"
  End With
  With ActiveChart.Axes(xlCategory)                                        'Gridlines
    .HasMajorGridlines = False
    .HasMinorGridlines = False
  End With
  With ActiveChart.Axes(xlValue)
    .HasMajorGridlines = False
    .HasMinorGridlines = False
  End With
  ActiveChart.HasLegend = False                                            'Legend
End Sub
```

(The annotations on the right-hand side above have been added to help explain this somewhat lengthy set of statements. WizStep1 is short for Chart Wizard Step 1, etc.)

The Charts collection in the workbook is augmented by a new chart [Charts.Add] of type XYScatterSmoothNoMarkers [Active.ChartType] taking as source data the cell range named 'chartdata' [ActiveChart.SetSourceData]. The code generated relates closely to the answers given in the first two Chart Wizard steps. In the first step, the chart type is chosen, and in the second, the data range containing the source data is defined and its structure (rows or columns) specified. [The syntax for the method applied to the object ActiveChart is: SetSourceData(Source, PlotBy).] In passing, note that part of the ActiveChart.SetSourceData statement overflows onto the following line. This is indicated in the code by a space followed by underscore (_).

The code line: ActiveChart.Location Where:=xlLocationAsObject, Name:="Data" corresponds to choices made on the fourth wizard screen and ensures that the chart will be located as an embedded object on the Data sheet itself. (If the chart is to be in a separate chart sheet, the code simplifies to ActiveChart.Location Where:=xlLocationAsNewSheet.)

The majority of the subsequent code relates to the choices made in the Wizard's third step, where Chart Options are specified. Notice the useful bit of syntax:

'With... End With'

which Excel's recorder frequently uses when several changes are made to an object, here the ActiveChart object. Hence the code:

```
With ActiveChart
      .HasTitle = True
      .ChartTitle.Characters.Text = "Cumulative Distribution"
      .Axes(xlCategory, xlPrimary).HasTitle = False
      .Axes(xlValue, xlPrimary).HasTitle = True
      .Axes(xlValue, xlPrimary).AxisTitle.Characters.Text = "P(X> x)"
End With
```

defines the various titles, and subsequent code segments define the Gridlines, Axes and Legend in turn.

Much of the code generated by the recorder leaves default settings unchanged and can be edited out to make the macro more concise. A first step is to 'comment out' possibly redundant statements (by adding an apostrophe at the start of the statement), and then to check that the code still works as wished. For example, the following subroutine, ChartNew1 works perfectly well without the comment lines:

```
Sub ChartNew1()
  Charts.Add
  ActiveChart.ChartType = xlXYScatterSmoothNoMarkers
  ActiveChart.SetSourceData Source:=Sheets("Data").Range("chartdata"), PlotBy:=_
    xlColumns
  ActiveChart.Location Where:=xlLocationAsObject, Name:="Data"
  With ActiveChart
    .HasTitle = True
    .ChartTitle.Characters.Text = "Cumulative Frequency"
    .Axes(xlCategory, xlPrimary).HasTitle = False
    .Axes(xlValue, xlPrimary).HasTitle = True
    .Axes(xlValue, xlPrimary).AxisTitle.Characters.Text = "Freq(X> x)"
  End With
  'With ActiveChart.Axes(xlCategory)
  '   .HasMajorGridlines = False
  '   .HasMinorGridlines = False
  'End With
  With ActiveChart.Axes(xlValue)
```

```
      .HasMajorGridlines = False
  '   .HasMinorGridlines = False
  End With
  ActiveChart.HasLegend = False
End Sub
```

The code can therefore be reduced to the following:

```
Sub ChartNew2()
'same as ChartNew Macro after removing redundant lines
  Charts.Add
  ActiveChart.ChartType = xlXYScatterSmoothNoMarkers
  ActiveChart.SetSourceData Source:=Sheets("Data").Range("chartdata"), PlotBy:=_
    xlColumns
  ActiveChart.Location Where:=xlLocationAsObject, Name:="Data"
  With ActiveChart
    .HasTitle = True
    .ChartTitle.Characters.Text = "Cumulative Frequency"
    .Axes(xlValue, xlPrimary).HasTitle = True
    .Axes(xlValue, xlPrimary).AxisTitle.Characters.Text = "Freq(X> x)"
  End With
  With ActiveChart.Axes(xlValue)
    .HasMajorGridlines = True
  End With
  ActiveChart.HasLegend = False
  ActiveChart.PlotArea.Select                        'new code added
  Selection.Interior.ColorIndex = xlAutomatic        'remove plot area colour
  ActiveChart.Axes(xlValue).Select
  With ActiveChart.Axes(xlValue)
    .MaximumScale = 1                                'reset max. value on Y axis
  End With
End Sub
```

The code from ActiveChart.PlotArea.Select onwards is additional. It ensures that the Y axis scale stops at 1 (or 100%) and removes the default grey colour in the Plot area of the chart.

The various chart macros which can be run with the Data sheet equity returns are in ModuleC of the workbook.

3.6.2 Normal Probability Plot

One useful data visualisation is the so-called normal probability plot. This plot of a set of readings shows whether the variation in the readings can be assumed to be statistically normal. Suppose there are 50 readings. Then the order statistics of a set of 50 normally distributed readings (called norm-scores) are plotted against the standardised values of the readings (called *z*-scores). If the readings are effectively normal, the points resulting from the two scores will lie more or less on a straight line. An illustrative normal probability plot shown in Figure 3.4 indicates some departures from normality, especially on the tails of the distribution.

At present, Excel does not include this useful plot in its statistical armoury. (The plot referred to as a normal probability plot in the Data Analysis ToolPak Regression option is not a conventional normal probability plot at all, but simply the cumulative distribution of the dependent variable.)

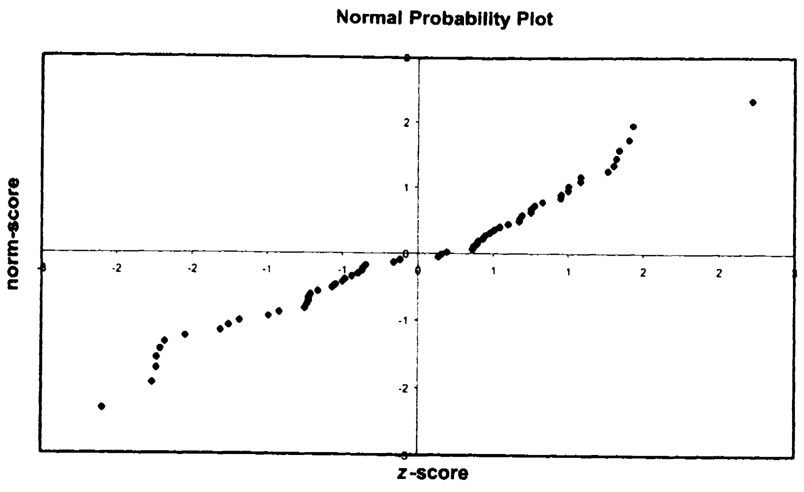


Figure 3.4 A normal probability plot of log returns data

Suppose the readings are in a range named dvec in the Data sheet of the workbook. In the VBA code, an array variable dvec is set up to take the values from the range named dvec. Because we need to process individual array elements, dvec(i), dvec is declared a Variant (and is actually an object variable, specifically a Range object). A second array variable, Znvec, is set up to take vectors of z-scores and norm-scores, one pair of scores for each data value in dvec. These array variables are declared at the beginning of the subroutine NPPlotData.

Since a range variable is referenced by dvec, its values are assigned with the Set keyword, hence:

```
Set dvec = Sheets("Data").Range("dvec")
```

Once dvec has been specified, the number of readings in dvec (say n) can be evaluated and previously declared arrays dimensioned correctly. Hence the statement:

```
ReDim Znvec(n, 2)
```

For each reading in the data set, a z-score and a norm-score are calculated within a For...Next loop. The calculations for the z-score use the Excel AVERAGE and STDEV functions, whereas for the norm-score the calculations involve the Excel RANK function and the inverse standard normal probability function (NORMSINV). The scores are stored in the n by 2 matrix called Znvec, and subsequently output to the named range Znvec in the Data sheet. Once the vectors of scores have been calculated, they can be plotted against each other in an XY chart.

The full code for obtaining the scores for the plot is given below. The penultimate step calls the subroutine ChartNormPlot, which is similar to the chart subroutine developed in section 3.6.1. You can try out the NPPlot macro on the log returns data in the Data sheet of the VBSUB workbook.

```
Option Explicit
Option Base 1

Sub NPPlotData()
  'returns normal probability plot of data from named range dvec

  'declaring variables
  Dim m1, sd1, rank1, c1
  Dim i As Integer, n As Integer
  Dim Znvec() As Variant
  Dim dvec As Variant

  'data input from worksheet
  Set dvec = Sheets("Data").Range("dvec")           'use Set because dvec(i) required
  n = Application.Count(dvec)
  ReDim Znvec(n, 2)                                 'number of readings
                                                    'Znvec dimensioned as n by 2 matrix

  'calculating elements of Znvec array
  m1 = Application.Average(dvec)                    'mean of readings
  sd1 = Application.StDev(dvec)                     'standard deviation of readings
  For i = 1 To n
    Znvec(i, 1) = (dvec(i) -m1) / sd1               'z-score for ith reading
    rank1 = Application.Rank(dvec(i), dvec, 1)
    c1 = (rank1 -3 / 8) / (n + 1 / 4)           'using a continuity correction
    Znvec(i, 2) = Application.NormSInv(c1)          'n-score for ith reading
  Next i

  'output results to range Znvec
  Sheets("Data").Range("Znvec") = Znvec             'matrix of scores output to 'range Znvec
  With Sheets("Data").Range("Znvec")
    .NumberFormat = "0.00"                          'output data formatted
    .Name = "npdata"                                'output range given name npdata
  End With
  ChartNormPlot                                     'subroutine ChartNormPlot called
  End Sub
```

The code for the NPPlot subroutine and ChartNormPlot (called from the NPPlotData macro) are stored in ModuleN of the workbook.

3.6.3 Generating the Efficient Frontier with Solver

This application requires a degree of familiarity with portfolio theory and optimisation with Excel's Solver add-in. The reader may prefer to delay inspecting this material until Chapter 6 has been studied.

In looking at portfolios with several risky assets the problem is to establish the asset weights for efficient portfolios, i.e. those that have minimum risk for a specified expected return. It is straightforward to get the optimum weights for a target return with Solver. If the optimisation is repeated with different targets, the efficient portfolios generate the risk–return profile known as the efficient frontier.

For example, Figure 3.5 shows the weights (40%, 50%, 10%) for one portfolio made up of three assets with expected return 2.2%. The target return is 7% and the efficient portfolio is determined by the set of weights that produce this return with minimum standard deviation.

	A	B	C	D	E	F	G	H	I	J
4	**Asset Data**		Exp Ret	Std Dev						
5		TBills	**0.6%**	**4.3%**			Target exp return		7.0%	target1
6		Bonds	**2.1%**	**10.1%**						
7		Shares	**9.0%**	**20.8%**						
8										
9	**Correlation Matrix**		TBills	Bonds	Shares		**Portfolio weights:**			
10		TBills	1.00	**0.63**	**0.09**			TBills	40.0%	
11		Bonds	0.63	1.00	**0.23**			Bonds	**50.0%**	change1
12		Shares	0.09	0.23	1.00			Shares	**10.0%**	
13										
14	**VCV matrix**		TBills	Bonds	Shares					
15		TBills	0.0018	0.0027	0.0008			Exp Ret	**2.2%**	portret1
16		Bonds	0.0027	0.0102	0.0048			Std Dev	**7.0%**	portsd1
17		Shares	0.0008	0.0048	0.0433					

Figure 3.5 Portfolio weights, expected return and standard deviation of return from Eff1 sheet

To perform this optimisation, Solver requires 'changing cells', a 'target cell' for minimisation (or maximisation) and the specification of 'constraints', which usually act as restrictions on feasible values for the changing cells. In Figure 3.5, we require the weights in cells I10:I12 to be proportions, so the formula in cell I10 of =1−SUM(I11:I12) takes care of this requirement. For the optimisation, the 'changing cells' are cells I11:I12, named 'change1' in the sheet. The target cell to be *minimised* is the standard deviation of return (I16) named 'portsd1'. There is one explicit constraint, namely that the expected return (cell I15) named 'portret1' equals the target level (in cell I5 named 'target1'). The range names are displayed on the spreadsheet extract to clarify the code subsequently developed for the macro.

Applying Solver once, the optimum weights suggest buying Bonds and Shares and short selling TBills in the proportions shown in Figure 3.6. Cell I15 confirms that this efficient portfolio with minimum standard deviation of 15.7% achieves the required return of 7%.

Changing the target expected return in cell I5 and running Solver again with the same specifications as before would produce another efficient portfolio. Clearly, if the entire efficient frontier is required, the best approach is via a macro. If the optimisation using Solver is recorded, the following type of code is obtained:

```
Sub TrialMacro()
        SolverReset
        SolverAdd CellRef:="$I$15", Relation:=2, FormulaText:="target1"
        SolverOk SetCell:="$I$16", MaxMinVal:=2, ValueOf:="0", ByChange:="$I$11:$I$12"
        SolverSolve
End Sub
```

	A	B	C	D	E	F	G	H	I	J
2	**Efficient Frontier points using Solver (no constraints on weights) Eff1**									
3										
4	**Asset Data**		Exp Ret	Std Dev						
5		TBills	**0.6%**	**4.3%**			Target exp return		7.0%	target1
6		Bonds	**2.1%**	**10.1%**						
7		Shares	**9.0%**	**20.8%**						
8										
9	**Correlation Matrix**		TBills	Bonds	Shares		**Efficient frontier portfolio**			
10		TBills	1.00	**0.63**	**0.09**			TBills	-5.6%	
11		Bonds	0.63	1.00	**0.23**			Bonds	**35.8%**	change1
12		Shares	0.09	0.23	1.00			Shares	**69.8%**	
13										
14	**VCV matrix**		TBills	Bonds	Shares					
15		TBills	0.0018	0.0027	0.0008			Exp Ret	**7.0%**	portret1
16		Bonds	0.0027	0.0102	0.0048			Std Dev	**15.7%**	portsd1
17		Shares	0.0008	0.0048	0.0433					

Figure 3.6 Optimum portfolio weights for expected return of 7% produced by Solver

The recorded code hints at some of the special VBA functions in the Solver add-in. In particular, the SolverAdd function (for specifying a constraint) has three arguments: the first is a cell reference for the constraint 'LH side', the second an integer code for the relation (2 for =) and the third either a cell reference or a single number. Similarly, the arguments in the SolverOk function specify the particular optimisation problem to be solved.

The recorded code can be improved firstly by writing the functions and their arguments with brackets together with comma delimited arguments, and secondly by employing range names in preference to cell addresses. The Call keyword used with each Solver function highlights the fact that control has been passed to a Function procedure. (Also, it suppresses the value returned by the function.) After editing, the code for the first optimisation is:

```
Sub Eff0()
'to return an efficient portfolio for given target return
        SolverReset
        Call SolverAdd(Range("portret1"), 2, Range("target1"))
        Call SolverOk(Range("portsd1"), 2, 0, Range("change1"))
        Call SolverSolve(True)
        SolverFinish
End Sub
```

The SolverAdd function ensures that the expected return (in the cell named 'portret1') meets the specified target, in cell 'target1'. The SolverOk function controls the optimisation task, ensuring that the entries in cell range 'change1' are such that the portfolio standard deviation (in cell 'portsd1') is minimised. The SolverFinish function is equivalent to selecting options and clicking OK in the Solver Results dialog shown after solution. Effectively, with this statement, the Solver Results dialog box is not displayed under macro control.

The efficient frontier requires the choice of optimal portfolio weights for a range of target returns. Suppose a range of different targets is set up starting with an initial target of 1% (*min_tgt* in general) increasing in steps of 2% (*incr*) as many times as specified (*niter*). The different target returns are produced in a Do While...Loop which repeats *niter* times. For each target return, Solver has to produce optimal weights. The only part of the problem specification that changes on each iteration is the target return. Thus, in the code, most of the Solver specification is set up before the repeated optimisation. Only the SolverChange function that changes the target return sits inside the loop.

```
Sub Eff1()
' repeated optimisation with given min_target & increment
' initialisation
  Dim target1: Dim incr
  Dim iter As Integer, niter As Integer
  target1 = Range("min_tgt").Value          'value taken from spreadsheet cell
  incr = Range("incr").Value                'value taken from spreadsheet cell
  niter = Range("niter").Value              'value taken from spreadsheet cell
  iter = 1                                  'initial value for loop counter
' code to clearout previous results
  ClearPreviousResults                      'a subroutine

' set up Solver
  SolverReset
  Call SolverAdd(Range("portret1"), 2, Range("target1"))
  Call SolverOk(Range("portsd1"), 2, 0, Range("change1"))
' repeated part
  Application.ScreenUpdating = False        'turns off screen recalculation
  Do While iter <= niter
    Range("target1").Value = target1        'put current value of target1 in cell
    Call SolverChange(Range("portret1"), 2, Range("target1"))
    Call SolverSolve(True)
    SolverFinish
'   code to copy & paste results in sheet
    ReadWrite                               'a ReadWrite subroutine
    target1 = target1 + incr                'update value of variable target1
    iter = iter +1                          'increment counter
  Loop
  Range("target1").Select
  Application.CutCopyMode = False
  Application.ScreenUpdating = True         'turn screen updating back on
End Sub
```

After writing the code, you can add a description for the macro and assign a keyboard shortcut, just as occurs when you record a macro. To do this, start in the Excel window, choose Tools then Macro then Macros. In the Macro dialog box, select the macro name (Eff1) then click the Options button.

For reference, the macros to generate the efficient frontier are in ModuleS of VBSUB and can be run from the Eff1 sheet. Before you use any macro containing Solver, you must establish a reference to the Solver add-in. With a Visual Basic module active, click References on the Tools menu, then Browse and find Solver.xla (usually in the \Office\Library subfolder).

SUMMARY

VBA is an object-oriented version of the Basic programming language. As well as familiar programming ideas concerning variables and coding, there are also methods and properties for use with Excel objects.

You automate operations in Excel first by writing the code for subroutines and functions in a VBA Module within the workbook, then by running your macros. The Visual Basic Environment provides some tools for debugging macros and access to VBA libraries and online help.

The purpose of VBA subroutines is to carry out actions: in contrast, VBA user-defined functions return values. In the context of financial modelling, we have found that functions are more useful than subroutines. The exceptions are operations that require the production of charts, and automating optimisation tasks using Solver. Subroutines are frequently useful for one-off administrative tasks, which require lots of repetition.

The macro recorder can be used to translate your actions into VBA code. Recording keystrokes in macros is sometimes useful as a starting point, for getting coding clues and insights. Whereas the recorder tends to generate verbose code for simple operations, it can produce highly sophisticated code for many menu-based procedures.

Most Excel functions can be included in VBA macros and there are some special VBA functions for use in macros. This is a useful way of leveraging the knowledge of the experienced Excel model builder.

There are advantages to mastering VBA. The replacement of calculations by functions can lead to more powerful cell formulas and makes calculation sequences more robust and faster. Skilful use of macros to control repeated operations can remove other sources of error from spreadsheet operations.

REFERENCES

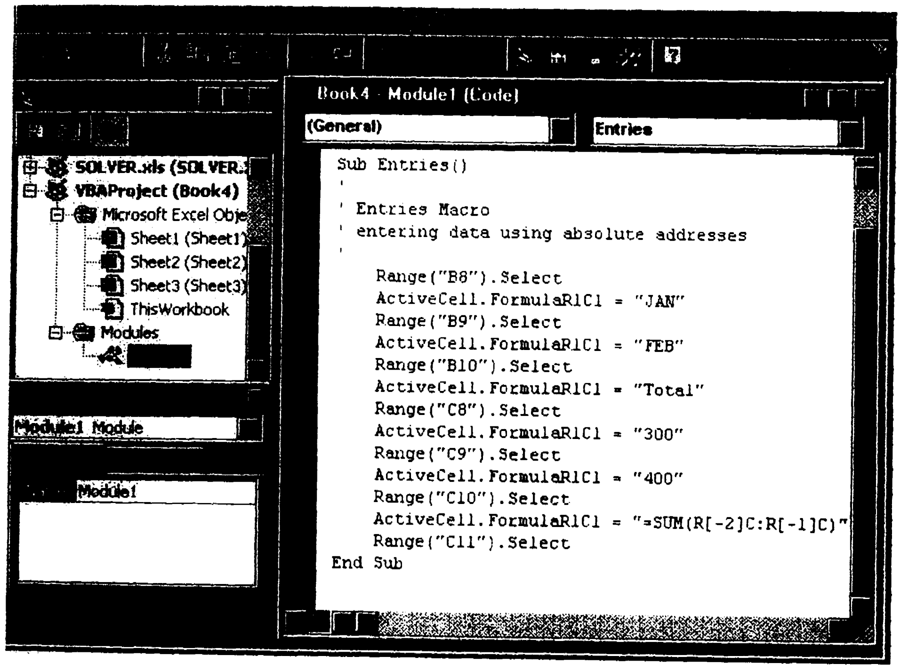
Green, J., 1999, *Excel 2000 VBA Programmer's Reference*, Wrox Press Ltd., Birmingham.
Leonhard, W., L. Hudspeth and T. J. Lee, 1997, *Excel 97 Annoyances*, O'Reilly & Associates, Inc., Sebastopol, CA.
Walkenbach, J., 1999, *Excel 2000 Power Programming with VBA*, IDG Books, Foster City, CA.
Wells, E. and S. Harshbarger, 1997, *Microsoft Excel 97 Developer's Handbook*, Microsoft Press.

APPENDIX 3A THE VISUAL BASIC EDITOR

The Visual Basic Editor was substantially changed for Excel 97 (and later versions) and this appendix details some of its features. It can be activated by pressing the VBE button (on the VB toolbar) or by pressing Alt+F11 in the Excel window. Once activated, you can toggle between the Excel and the Visual Basic windows with Alt+Tab in the usual manner. The Visual Basic Editor can also be opened via the Excel menubar by choosing Tools then Macro then Visual Basic Editor.

The Visual Basic window should look something like that shown in Figure 3.7, which includes a menubar and a toolbar across the top; the Project Explorer and the Properties windows on the left-hand side and the Code window on the right. In the illustration, the

Code window contains the code in Module1, in fact recorded code for the data entry operations described in section 3.3.5.



```
Sub Entries()
'
' Entries Macro
' entering data using absolute addresses
'
    Range("B8").Select
    ActiveCell.FormulaR1C1 = "JAN"
    Range("B9").Select
    ActiveCell.FormulaR1C1 = "FEB"
    Range("B10").Select
    ActiveCell.FormulaR1C1 = "Total"
    Range("C8").Select
    ActiveCell.FormulaR1C1 = "300"
    Range("C9").Select
    ActiveCell.FormulaR1C1 = "400"
    Range("C10").Select
    ActiveCell.FormulaR1C1 = "=SUM(R[-2]C:R[-1]C)"
    Range("C11").Select
End Sub
```

Figure 3.7 Visual Basic window with Project Explorer, Properties and Code windows open

The toolbar and windows described may not be visible when you first activate the Visual Basic Editor. If the toolbar is not visible, use View Toolbars and click once on the Standard option. Similarly, choose View then Project Explorer and View then Properties window to display the two left-hand windows. To display the Code module on the right, double click on the relevant Module (here Module1) in the Project Explorer window.

As the name suggests, the code modules contain the VBA code for procedures. These modules are inserted by choosing Insert then Module from the menubar. (These are Modules, not Class Modules, which are more advanced and will not be covered in this text.) You can insert as many modules as you like in the workbook. One practice is to put related macros into the same module.

The Project Explorer shows the component parts of all the open 'projects'. Its main use is to act as a navigation tool, from which the user can open new modules in which to store code or remove (i.e. delete) a module or to activate the Code windows for existing modules. Selecting a module sheet, the File menu allow you to Remove the module sheet (i.e. delete it) and also to Print out its contents. If you want to copy VBA code from one Module to another, avoid the Export File option. Copying code from one module sheet

to another one in the same workbook or any other is best achieved via the Code window with the usual copy and paste operation.

Each Excel object shown in the Project Explorer for a particular 'project' has its own set of 'properties', for example, each worksheet in an active workbook has the 'name' property. These properties can be changed via the Properties window. However, since modifying properties is more relevant for worksheets than for modules, we will not expand on the features of this window further in the context of macros, except to note that the name of each code module can be modified here. (For example, we frequently change the module names to M, 0 and 1, collecting together any macros in ModuleM, functions with base 0 in Module0, and the remaining functions with base 1 in Module1.) In practice, when developing macros, it is more convenient to close the Properties window and possibly the Project Explorer to maximise the size of the Code window.

The Visual Basic window has its own Standard toolbar, as shown in Figure 3.8, with the View Excel button at the left-hand which returns the user to the Excel window. The middle set of buttons concern editing tasks, running and testing out macros, and the right-hand group open other windows such as the Immediate window (for testing individual statements) and the Object Browser (for checking the objects, methods and properties available in VBA). Note in particular the Run Macro button (a right pointing triangle), the Reset button (a square) and the Object Browser (third from the end).

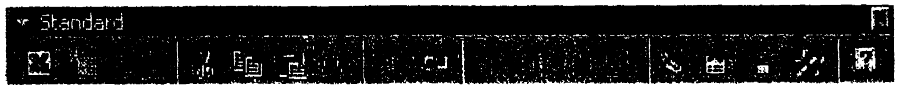


Figure 3.8 Standard toolbar in VB window

There are two types of procedure which we code in VBA: subroutines (or macros) and functions. Only subroutines (or macros) can be recorded. Usually they will require some editing in the Code window and possibly testing using Excel's debugging tools. Since functions cannot be recorded, they must be written in a code module. Therefore the recording tools are of little use in developing functions.

As outlined in the main text, subroutines can be run from the Excel window (via menu commands or by clicking the Run button on the VB toolbar or by using a chosen keyboard shortcut). They can also be run from the Code window (with the pointer positioned anywhere in the body of the routine's code, by clicking the Run button on the VB Standard toolbar underneath the VB window menu). Possibly more user-friendly than either of these ways, subroutines can be attached to buttons. However, since our workbooks rely on functions rather than macros, it is best to refer the reader to other texts for details of attaching macros to buttons. (For example, see Leonhard et al., 1997 for further details on the VBE and developing macros.)

Note that when you try to run subroutines and the VBE intervenes to tell you of an error preventing compilation, macro operation is suspended in so-called 'debug mode', the subroutine name being illuminated in yellow. With the cursor pointing to the offending statement, you can edit simple mistakes, then easily jump out of debug mode via Run

Reset on the Run menu in the VBE command bar or by clicking the Reset button (with the black square). The yellow illumination disappears and the macro can be re-run.

Stepping Through a Macro and Using Other Debug Tools

If your macros don't work as you expect, it helps to step slowly through them operation-by-operation. If you run the macro from the menu, choosing Tools then Macros then selecting the 'macro name', you need to click the **Step Into** button on the Macro dialog box instead of choosing Run. This allows you to step through the macro. Alternatively, provided the cursor is not in a Code window, you can click the **Run** button on the VBA toolbar, choose the macro to be tested then click the **Step Into** button. The VBA Code window opens with the first statement of the chosen macro highlighted (in yellow).

Click the F8 function key (or if visible, the small Step Into button) to move to the next line of code and so on. If you want to see the actions in the spreadsheet window, reduce the size of the VBE window so you can see the Excel sheet underneath or use the Alt+Tab combination to move between windows and watch the macro working line-by-line in the Code window. Click F8 (or the Step Into button) repeatedly to step through the macro. There is a **Step Out** button to escape from step mode. Alternatively, the VBE Editor Window command bar Run menu has a **Reset** choice which gets you out of step or debug mode.

The Debug toolbar in Figure 3.9 contains the Step Into, Step Over and Step Out buttons for proceeding through the macro code. A macro being 'stepped through' line-by-line is said to be in 'break mode'. If the macro calls other secondary routines, the Step Over button executes these secondary routines without stepping through them line-by-line.

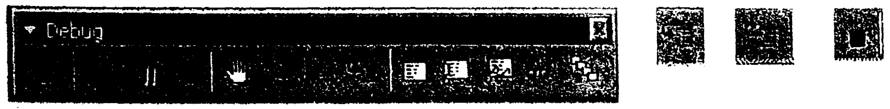


Figure 3.9 The Debug toolbar, with Step In, Step Out and Reset buttons shown separately

To illustrate, display the Debug toolbar in the VB window (using View then Toolbars) and try stepping through the Factorial macro (whose code is in ModuleA of the VBSUB workbook). Starting from the IntroEx sheet, choose Tools then Macro then Macros, select the Factorial macro as shown in Figure 3.10, and click the Step Into button. The VB window opens with the first code line of the Factorial macro highlighted (in yellow). Continue stepping through with the Step Into button. When the Input Box asking for the number appears, enter 3 say and click OK. Go on stepping through (checking that the loop is executed three times) until the MsgBox dialog appears with the answer 6.

At any stage, you can jump out of step mode by clicking Reset (or via Run then Reset from the VBE standard command bar).

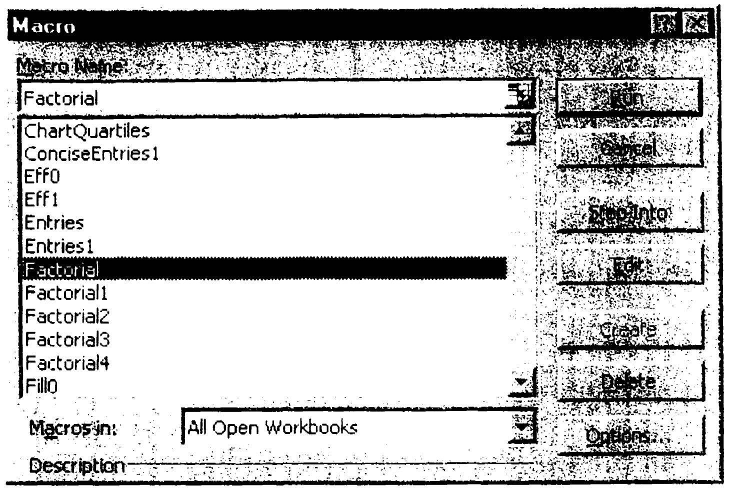


Figure 3.10 Entering step mode for the Factorial macro

As a further illustration, step through the Factorial macro once again after clicking the Locals window button (see the Debug toolbar, fifth button from the right in Figure 3.9). In the new window, the values of variables num, fac and *i* are shown as successive lines of the macro code are executed.

Another useful facility in debugging macros is to insert breakpoints, especially in longer subroutines. When the macro runs, operations cease when a breakpoint is reached. Execution jumps to break mode, and the macro can be explored in step mode. To insert a breakpoint, select the position in the code and click in the adjacent grey margin or simply click the breakpoint button (shown in Figure 3.9 with a small hand). To remove the breakpoint, simply click in the grey margin once again or toggle the breakpoint button. With the Factorial macro, try making the MsgBox statement a breakpoint, then run the macro to check that it runs normally until the point at which the results should be displayed. Then execution changes to 'step mode' for the remaining code lines and macro operations proceed by pressing the F8 key. (Breakpoints can also be used in debugging functions, since they provide a means of changing into step mode to examine the operation of the function code.)

APPENDIX 3B RECORDING KEYSTROKES IN 'RELATIVE REFERENCES' MODE

The recorder translates keystrokes into VBA code and has two modes when interpreting cell addresses: its default mode (absolute references) and the relative references mode. In section 3.3.5, the macro entering data into worksheet cells was recorded using absolute references. Here, we look at the code generated when recording in relative references mode.

Start with the pointer on cell B8. The macro recorder is invoked from the Excel window by choosing Tools, then Macro then Record New Macro. Give the macro a name, say RelEntries. This time when the Stop Recording button appears on screen, click the Relative References button. It should appear as if 'pressed in'. Then just as before, key in the names of the months. The code when you've finished should look different from the previous macro:

```
Sub RelEntries()
'recorded with relative references
  ActiveCell.FormulaR1C1 = "JAN"
  ActiveCell.Offset(1,0).Range("A1").Select
  ActiveCell.FormulaR1C1 = "FEB"
  ActiveCell.Offset(1,0).Range("A1").Select
  ActiveCell.FormulaR1C1 = "Total"
  ActiveCell.Offset(-2,1).Range("A1").Select
  ActiveCell.FormulaR1C1 = "300"
  ActiveCell.Offset(1,0).Range("A1").Select
  ActiveCell.FormulaR1C1 = "400"
  ActiveCell.Offset(1,0).Range("A1").Select
  ActiveCell.FormulaR1C1 = "=SUM(R[-2]C:R[-1]C)"
  ActiveCell.Offset(1,0).Range("A1").Select
End Sub
```

Try running the macro from different cells in the spreadsheet. The entries are made relative to the pointer position (the ActiveCell referred to as Range("A1")), which itself moves down one row at a time, then to the adjacent column. The Offset method, say Offset(1, 0), returns a range object, the cell in the same column, but one row below relative to the current position. The cell selected is always referenced as "A1" (the ActiveCell), despite the fact that the keystrokes did not involve cell A1 in any way.

To get most assistance from the recorder, it is worth thinking out in advance which mode is likely to be more helpful: absolute or relative references. Usually, it is absolute references, the default mode of recording. However, if you are trying to automate operations that repeat themselves row-by-row down the worksheet, the relative references mode is likely to throw up more clues as to coding. For example, if you want your macro to select a specific cell, perform an action, then select another cell relative to the active cell, record with relative references. However, you need to remember to switch off the Relative References button to return to absolute referencing.

The RelEntries macro can be edited to make it less verbose. Recorded code always includes much 'selecting' of cells, however when writing code, it is not necessary to explicitly select cells to enter data into them. The following edited version of the code performs the same operations but is more concise:

```
Sub ConciseEntries()
'entering data using relative positions
  ActiveCell.FormulaR1C1 = "JAN"
  ActiveCell.Offset(1, 0).FormulaR1C1 = "FEB"
  ActiveCell.Offset(1, 0).FormulaR1C1 = "Total"
  ActiveCell.Offset(-2, 1).FormulaR1C1 = "300"
  ActiveCell.Offset(1, 0).FormulaR1C1 = "400"
  ActiveCell.Offset(1, 0).FormulaR1C1 = "=SUM(R[-2]C:R[-1]C)"
End Sub
```

In fact for a single cell range object, it turns out to be unnecessary to specify the Formula property, so the code is simply:

```
Sub ConciseEntries()
'entering data using relative positions
  ActiveCell.Offset(0, 0) = "JAN"
  ActiveCell.Offset(1, 0) = "FEB"
  ActiveCell.Offset(2, 0) = "Total"
  ActiveCell.Offset(0, 1) = "300"
  ActiveCell.Offset(1, 1) = "400"
  ActiveCell.Offset(2, 1) = "=SUM(R[-2]C:R[-1]C)"
End Sub
```

A practical way to tidy up generated code is to comment out (add apostrophes in front of possibly) redundant statements, then test by running to see the effect.