A

Programming in VBA

The purpose of this appendix is to provide an introduction to the features of Excel VBA that are used in the book. To learn more about VBA in a finance setting, Jackson and Staunton [40] is a good source.

A.1 VBA Editor and Modules

Subroutines and functions are created in the VBA editor, which is reached through Tools/ Macros/Visual Basic Editor. When the editor opens, click Insert/Module to open an editing screen in which subroutines and functions can be typed. If you have opened a new workbook and inserted a module, you should see on the left a small pane with the heading Project-VBA Project that lists the elements of the workbook, including Module 1, which is the default name for the collection of things you might type in the editing screen (if the pane is not present, click View/Project Window). You should also see on the left a pane with the heading Properties-Module 1 (if it is not there, click View/Properties). You can rename Module 1 to something more useful by highlighting Module 1 in the Properties Window and typing the new name. You can add another module by clicking Insert/Module again. If you save the Excel workbook, all of the modules (and hence all of the subroutines and functions created in them) are saved with the workbook.

If you open the workbook distributed with this book, you will see in the Project Window modules named Chapt2, Chapt3, ..., Chapter 13. Each of these modules contains the VBA code in the corresponding book chapter. To view the code in a particular module, right-click on its name in the Project Window and select View Code. You will see a collection of programs separated by gray lines (which are added by the VBA editor to make things more readable). Each subroutine starts with `Sub` and ends with `End Sub` and each function starts with `Function` and ends with `End Function`. You will also see "Option Explicit" at the top of each module—this will be discussed below.

The organization of subroutines and functions into modules is not important (except for globally defined variables, which are not used in this book). All of the subroutines and all of the functions in all of the modules in any open workbook are available to use in any open workbook. However, you may find it convenient to organize your work into separate modules, for example Homework1, Homework2, etc.

If you open multiple workbooks, and open the VBA editor with one of them, then the Project Window will list each workbook and the modules associated with each. When the workbooks are saved, each set of modules will be saved with the associated workbook.

If VBA catches an error (normally a syntax error or an undeclared variable if "Option Explicit" has been declared) when executing a subroutine or function, a message box will pop up to inform the user. If the "Debug" option is chosen in this box, the offending VBA code will be highlighted in the VBA editor. After correcting the error, you need to click Run/Reset in the editor (or the square button in the editor's toolbar).

VBA ignores everything written on a line following an apostrophe, so comments can be placed on any line by preceding them with an apostrophe. Including comments in your subroutines and functions is very important to make them understandable.

The underscore character indicates that a line is to be continued. For example,

```
y = x + 5
```

is the same as

```
y = x _
+ 5
```

This is useful for breaking long lines.

A.2 Subroutines and Functions

A subroutine is also called a macro. It is a way of automating tasks, including mathematical calculations, cell formatting, and outputting results to cells. The subroutines in this book simulate a random process and output the results to the active worksheet. The other programs in the book are user-defined functions.

To execute a macro, click Tools/Macros. Clicking the name of a macro and then clicking Run will execute it. A macro or function created in one workbook can be used in another. To execute a macro created in another workbook, simply open both workbooks at the same time, click Tools/Macros and choose the option "All Open Workbooks" for the macros to be displayed.[1]

[1] Since you will be running macros and using user-defined functions frequently, it is useful to add buttons to the toolbar to execute the keystrokes of clicking

To create a macro in the VBA editor, type

```
Sub WhateverNameYouWant()
...
list of commands
...
End Sub
```

You will notice that the editor automatically adds the parentheses () at the end of the subroutine name and adds the End Sub statement when you type Sub WhateverNameYouWant.

A user-defined function is executed just like any other Excel function – in a cell of the spreadsheet, type =FunctionName(arguments). The arguments supplied to the functions can be numbers or can be cell references, just as with any other Excel function. To see the user-defined functions that have been created, click Insert/Function and select the category User Defined. You may see a lot of functions created by Excel add-ins in addition to the functions that are in the modules. You can also execute a function by double-clicking on its name here.

To create a function in the VBA editor, type

```
Function AnotherName(argument1, argument2, ..., lastargument)
...
list of commands
...
AnotherName = WhateverTheAnswerMightBe
End Function
```

A.3 Message Box and Input Box

One way for a subroutine or function to deliver information is through the MsgBox function. In Module 1, type

```
Sub WhateverNameYouWant()
MsgBox("Whatever you want to type.")
End Sub
```

When you execute this macro, a message box will pop up, displaying the message. To close the message box, click OK. The message box function is useful primarily for displaying error messages. However, the message box can also return the results of mathematical operations, as the next example shows.

Tools/Macros and Insert/Function, if the buttons are not already there. To add the macro button, click Tools/Customize/Commands/Tools, scroll to Macros, and drag the "Macros ..." button to the toolbar. To add the function button, scroll to Insert and drag the "Insert Function" button to the toolbar.

One way for a subroutine or function to obtain information from the user is via the InputBox function. In Module 1, type

```
Sub AnotherSub()
x = InputBox("What is your favorite number?")
MsgBox("You said your favorite number was  " & x)
End Sub
```

When you execute this macro, a box will pop up displaying the text "What is your favorite number?" and providing a facility for inputting a number. When you hit Enter or click OK, the input box will disappear and the message box will appear, displaying the message and the number you chose.

A.4 Writing to and Reading from Cells

You can write a number, text, or formula to any cell in any worksheet of any open workbook. For example, executing the following macro

```
Sub WritingTest()
Workbooks("Book1.xls").Sheets("Sheet1").Range("B3").Value = 7
End Sub
```

will write the number 7 to cell B3 of Sheet1 of Book1.xls. The statement can be shortened to

```
Sheets("Sheet1").Range("B3").Value = 7
```

if you want to write to the active workbook, and it can be shortened to

```
Range("B3").Value = 7
```

if you want to write to the active sheet in the active workbook. To write text to the cell, enclose it in parentheses; for example, we could replace `Value = 7` with `Value = "some text"`. It is also possible to write a formula to a cell by replacing `Value = 7` with, for example, `Formula = "=A6"`. Running any macro of this sort will over-write anything that may already be in cell B3.

In the macros in this book, rather than writing to a particular cell, we write to the active cell of the active sheet of the active workbook (i.e., the cell in which the cursor is) and to cells surrounding the active cell. This is done as follows:

```
Sub WritingTest()
ActiveCell.Value = 7
ActiveCell.Offset(1,2) = 8
End Sub
```

This macro writes the number 7 to the active cell and the number 8 to the cell that is one row below and two columns to the right of the active cell.

A subroutine or function can also read directly from a cell in a workbook, though we do not use that feature in this book. The syntax is the same as for writing to a cell; for example, `x = ActiveCell.Value` assigns the value in the active cell to the variable `x`.

The formatting of cells (and ranges of cells) can also be changed in Excel macros. Moreover, the active cell/sheet/workbook can also be selected within a macro, and charts can be generated within macros, etc. We use VBA mainly as a computational engine in this book rather than as a means to create and modify worksheets, so we do not use many of the features of Excel VBA.

A.5 Variables and Assignments

Variable names must begin with a letter, be less than 256 characters long, and cannot include various special characters (in particular, they cannot contain blank spaces, hyphens or periods). Variable names are not case sensitive: `a` is the same variable as `A` (in fact, you may find the VBA editor changing the capitalization of names to maintain consistency across a project). It is of course a good idea to use names that mean something, so your programs are easier to read later. You cannot use any name already reserved by VBA or Excel; for example, attempting to create a variable with the name `Sub` will generate an error message.

An expression like `y = x + 3` is an assignment statement (unless it is prefaced by an `If`, `ElseIf` or `Do While`—see below). The computer evaluates the right-hand side, by looking up the value already assigned to `x`, adding 3, and storing this value in the memory space reserved for `y`. A statement like `x = x + 3` is perfectly acceptable. It simply adds 3 to the value of `x`. It doesn't matter whether you add spaces around the = and + signs: the VBA editor will automatically adjust the spacing.

It is optional whether you must specifically allocate memory space for a variable. If you type "Option Explicit" in a VBA module, then all variables must be declared. This is done with the keyword `Dim` at the beginning of the program (more on this below). If you do not type "Option Explicit," then you can create a new variable in the middle of a program simply by assigning it a value. For example, you can type `y = x+3`. If `y` has not been previously defined, then it will be created and assigned the value `x+3`. If `x` has not been defined, it will be created and given the value 0.

The main virtue of selecting "Option Explicit" is that it helps to avoid typographical errors. Suppose for example that you intend to assign a new value to a variable named `HardToSpell`. If you misspell the name in the assignment statement and have not declared "Option Explicit," then VBA will create a new variable with the misspelled name. The program will still execute, but it will not calculate what you intended it to calculate. Likewise, if you intend to perform some operation with `HardToSpell` and assign the result to another variable and you misspell `HardToSpell`, then a new variable will be created with the misspelled name, given a value of zero, and the operation will be performed with the value zero rather than with the value of `HardToSpell`. In both cases, with "Option Explicit" declared, VBA will generate an error message alerting you to the misspelling.

A.6 Mathematical Operations

The basic mathematical operations are performed in VBA in the same way as in Excel: addition, subtraction, multiplication (the asterisk symbol), division (/), and exponentiation (the caret symbol – 3^2 is 3 squared). The natural exponential is also the same in VBA as in Excel: Exp(6) is e^6. The square root and natural logarithm functions are also available in VBA but with different names than in Excel. The name of the square root function is Sqr in VBA (rather than Sqrt as in Excel) and the name of the natural logarithm function is Log in VBA (rather than Ln as in Excel). It does not matter whether or not you capitalize the names: the VBA editor will automatically capitalize, converting for example exp to Exp.

Other mathematical functions are used in VBA by preceding their Excel names with Application. For example Application.Max(3,4) returns the larger of 3 and 4. Of course, VBA means "Visual Basic for Applications" and the application being used here is Excel, so the name Application.Max indicates that the Excel Max function is to be used. A function that we use frequently is Application.NormSDist(d), which returns the probability that a standard normal random variable is less than or equal to d.

A.7 Random Numbers

Computers do not behave in a random way (though of course it may seem like it when one crashes) but they can generate sequences of numbers that pass statistical tests for randomness. The basic construction is the generation of a random integer in some range $[0, N]$ with each integer in the range being "equally likely." Dividing by N gives a number between 0 and 1 that has the appearance of being uniformly distributed. This number can then be transformed to give the appearance of a normal distribution or other standard distributions. Random integers are generated sequentially by an algorithm of the type $I_j = aI_{j-1} + c \bmod N$, for constants a and c. "mod N" means the remainder after dividing by N (7 mod 5 is 2, 10 mod 3 is 1, etc.). In this construction, I_{j-1} is called the "seed," and each integer becomes the seed for the next. This is certainly not a random construction, but if the constants and N are suitable chosen (N must be very large) then the integers will have the appearance of unpredictability, both to the human observer and according to formal statistical tests.

VBA has a built-in function for generating random variables that are uniformly distributed between 0 and 1. This function is called Rnd(). The same function is in Excel but called Rand(). Applying the inverse of the standard normal cumulative distribution function to a random variable that is uniformly distributed between 0 and 1 will generate a random variable with the standard normal distribution (i.e, the normal distribution with mean 0 and variance 1). The inverse of the standard normal cumulative distribution function is provided in Excel as the NormSInv function, and hence it can be called in VBA as Application.NormsInv. Given the existence of the NormSInv function, this is the simplest, though not the fastest, way to transform a uniformly distributed random variable into a normally distributed one. To reduce typing, the following function is used throughout this book.

```
Function RandN()
   RandN = Application.NormSInv(Rnd())
End Function
```

A.8 For Loops

A loop is a command or set of commands that executes repeatedly either for a fixed number of times or until some condition is violated. To execute the commands for a fixed number of times, use a "for loop."

To add the first 10 integers together we can create the following macro:

```
Sub AddIntegers()
x = 1
For i = 2 To 10
    x = x + i
Next i
ActiveCell.Value = x
End Sub
```

In the above, we first initialized the value of x to be 1. The statement(s) between the For statement and the Next statement are executed repeatedly. In the first passage through the loop, the variable i has the value 2 and the statement x = x + i translates as x = 1 + 2, so x is given the value 3. In the next passage, i has the value 3 and x has the value 3, so the statement x = x + i translates as x = 3 + 3, and x is given the value 6, etc.

Any variable name (not just i) can be used as a counter. The indentation of the line x = x + i is optional and serves only to make the program easier to read.

The number of iterations need not be fixed when the program is written. We can use variables in the For statement like For i = y To z. The number of iterations will then be determined by the values of y and z when the for loop is encountered.

In the statement For i = 2 To 10, MATLAB increases i by one each time it reaches the statement Next i. This is the default, but it can be changed. If you want i to increase by two each time, you can write

```
For i = 2 To 10 Step 2.
```

Negative step sizes and non-integer step sizes are also acceptable. For example, the statement For i = 10 To 1 Step -1 produces a loop that executes "backwards," starting from i = 10 and counting down until i = 2.

A.9 While Loops and Logical Expressions

A "while loop" executes a block of statements repeatedly until some condition is violated. For example a crude way to add the first 10 integers would be with the following macro:

```
Sub AddIntegers2()
x = 0
i = 1
Do While i <= 10
    x = x + i
    i = i + 1
Loop
ActiveCell.Value = x
End Sub
```

When the program first encounters the Do While statement, it checks whether the condition $i \leq 10$ is true. If it is, then the statements preceding the Loop statement are executed. The condition $i \leq 10$ is then checked again, and the statements are executed repeatedly in this way until the condition $i \leq 10$ is false. Be careful that the statements being executed will eventually cause the condition to be false.

The comparison operators that can be used in the Do While statement (and If and ElseIf described below) are less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=), and equal to (=).

The expression Not(i > 10) is equivalent to (i <= 10). Multiple conditions can be combined: the expression i <= 10 And y > 6 is true if (and only if) both $i \leq 10$ and $y > 6$ are true, and the expression i <= 10 Or y > 6 is true if either or both of its component statements is true.

A.10 If, Else, and ElseIf Statements

You can cause a statement to execute only when a certain condition is satisfied by prefacing it with an If statement. The format is

```
If y <= 10 Then
    x = 2 * x
End If
```

which doubles x if $y \leq 10$ and does nothing otherwise. Rather than doing nothing otherwise, you can cause a different statement or block of statements to execute when the condition is violated by including an Else. For example,

```
If y <= 10 Then
    x = 2 * x
Else
    x = 3 * x
End If
```

In this case, if $y > 10$, the statements following the Else statement execute, tripling x. Finally, you can check multiple conditions sequentially with ElseIf. Consider the following:

```
If y <= 10 Then
    x = 2 * x
ElseIf y <= 20 Then
    x = 3 * x
ElseIf y <= 30 Then
    x = 4 * x
Else
    x = 5 * x
End If
```

The conditions are checked sequentially as follows. If $y \leq 10$, then x is doubled and execution of of the If block ends. If $y > 10$, the condition $y \leq 20$ is checked. If this is true, x is tripled. If it is not true, the next condition is checked, etc. The result is that x is doubled when $y \leq 10$; it is tripled when $10 < y \leq 20$; it is quadrupled when $20 < y \leq 30$; and it is quintupled when $y > 30$.

A.11 Variable Declarations

As mentioned before, if "Option Explicit" is declared, each variable must be declared at the beginning of a subroutine or function. A variable can be declared to be of a specific type or the type can be left unspecified and VBA will choose what seems to be the appropriate type. For numerical calculations, the important types are Integer, Long, Double, and Variant. The Integer data type is for storage of integers between -32,768 and 32,769. The Long data type can store integers between plus or minus 2 billion (actually a bit more than 2 billion). The Double data type stores arbitrary (floating point) numbers, to sixteen digits of accuracy. The Variant data type is the default type for variables whose type is not specified, and it adjusts itself automatically to the data stored within it. To declare a variable to be of a particular type, there are two equally acceptable syntaxes. For example, the Double type can be declared either as Dim x As Double or Dim x#. The Integer type can be declared either as Dim x As Integer or Dim x%. Note that the syntax Dim i, j, k As Integer is acceptable but it declares only k as being of type Integer, with i and j still being of type Variant. On the other hand, Dim i%, j%, k% declares i, j and k as being of type Integer.

In this book, the data type is left unspecified (hence as Variant), with the exception that the type of large arrays is declared. The Variant data type requires more memory for storage, so this is a bit inefficient.

Variables declared within a function or subroutine are "local variables." They can only be accessed within the function or subroutine within which they are defined. To understand this, consider the following simple example of a function (TestFunction) calling another function (AddTwo).

```
Function TestFunction(x)
TestFunction = x * AddTwo(x)
End Function

Function AddTwo(x)
Dim y
y = x + 2
AddTwo = y
End Function
```

The result of TestFunction(3) is $3 \times 5 = 15$. Consider now the following (strange) change to TestFunction.

```
Function TestFunction(x)
Dim z
z = x * AddTwo(x)
TestFunction = y
End Function
```

The new feature is that TestFunction(x) attempts to return y, which is defined only in AddTwo. If TestFunction(3) is executed, then one of two things will happen: (i) if "Option Explicit" has been declared, an error message will appear with the information that the variable y has not been declared within TestFunction, or (ii) if "Option Explicit" has not been declared, the function will return a value of zero. The reason in both cases is that the variable y defined within AddTwo is not available to TestFunction—it is local to AddTwo. In case (ii), a new variable y is created within TestFunction and, like all new variables, is given a default value of zero. The error message is probably preferable in this circumstance, which points again to the value of the "Option Explicit" declaration.

It is possible to declare a variable so that it is available to (and can be modified by) all of the functions in a module, or all of the functions in a workbook, or even all of the functions in all open workbooks. Such variables are called "global variables." That facility is useful in some situations, but it is not used in this book.

A.12 Variable Passing

As we have seen, functions and macros can call other functions or macros to perform part of their work. For example, macros shown previously in this appendix call the MsgBox function. The default arrangement in VBA is that variables are passed to functions (or to macros—though variables are not passed to macros in this book) "by reference" rather than "by value." This means that the actual memory location of the variable is given to the function, and any changes made to a variable by a function will affect the use of the variable in a calling function. Consider, for example the following simple change to the function AddTwo:

```
Function AddTwo(x)
x = x + 2
AddTwo = x
End Function
```

This function still adds the number 2 to its input. Now when we execute

```
TestFunction(3)
```

and it reaches the line

```
TestFunction = x * AddTwo(x)
```

it will be multiplying x by 5 as before. However, now x has been changed in AddTwo from 3 to 5, so the result of TestFunction(3) is $5 \times 5 = 25$.

This may sometimes be what one wants, but it is more likely that it will produce mistakes. There are two possible solutions. One is to change the function AddTwo as follows:

```
Function AddTwo(ByVal x)
x = x + 2
AddTwo = x
End Function
```

This forces VBA to pass only the value of x and not the memory location. So when 2 is added to x and returned to TestFunction(3), the value of x in TestFunction is still 3.

The second solution is more straightforward: simply do not change input variables within a function. That is, we can use our first version of AddTwo, which created a new variable to store the sum of x and 2, rather than changing the value of x (or we could use the simpler one-line function AddTwo = x + 2). The functions in this book follow this second approach—**we avoid changing the values of input variables**.

A.13 Arrays

It is very useful to be able to use a single variable name to store multiple values. For example, we can write loops such as

```
For i = 1 To 10
    x(i) = whatever
Next i
```

An array variable must be declared, regardless of whether "Option Explicit" is declared. Normally, the declaration takes the form Dim x(10) if the largest index number of x is known (to equal 10) when the function or macro is written. The default in VBA is that the first element is indexed by 0.[2] Therefore,

[2] This can be changed so that the default is for the first element to be indexed by 1 with the statement "Option Base 1."

`Dim x(10)` creates a vector with 11 elements, which are accessed as `x(0)`, ..., `x(10)`. The type of each element is Variant, unless it is declared otherwise—for example, `Dim x(10) As Integer` reserves memory locations for 11 integers. Multidimensional arrays can also be used. For example, `x(10, 6, 12)` creates a 3-dimensional array, with 11 × 7 × 13 elements. The first index does not have to be zero. The declaration `Dim x(1 To 10)` creates a vector with 10 elements, which are accessed as `x(1)`, ..., `x(10)`. Likewise, one can use, for example `Dim x(-6 To 3)` to start the indexing at -6 and end at 3.

If the dimension of the array is not fixed, which is often the case, then normally it must be declared with empty parentheses—for example, `Dim x()`. The dimension will depend on the input arguments, or on calculations based on the input arguments. Before the array is used, the program must include a statement specifying the dimension, of the form `ReDim x(N)`, or `ReDim x(1 To N)`, where the variable `N` is either an input argument to the function or has been calculated prior to the statement `ReDim x(N)`.

The exception to the above statements about declaring array variables, whether the number of elements is known in advance or not, is when an array is assigned to a variable by a call to a function. The `Array` function is one example of a function that creates an array. For example

```
Dim x
x = Array(3, 6, 7)
```

will create an array with elements `x(0)=3`, `x(1)=6`, and `x(2)=7`. Replacing `Dim x` with `Dim x(2)` in this context will not work.

Functions can take arrays as inputs and return arrays as outputs. Arrays can be input by (i) typing the array as an argument of the function, (ii) inputting the worksheet cells in which the array resides, or (iii) passing the array as an output from another function. An array created in one function is passed to another function in the same way that any other variable is passed. To type an array as an input, enclose it curly braces, separate items in each row with a comma, and separate rows with a semicolon—for example $\{3,1,2;4,6,2\}$ is an array with two rows and three columns, the first row being $\{3,1,2\}$ and the second row being $\{4,6,2\}$. The same array might be input via cell references as `B3:C5`.

Arrays can also be output to Excel worksheets. Consider the following:

```
Function MyArray(x)
Dim y(3)
For i = 1 To 3
  y(i) = i * x
Next i
MyArray = y
End Function
```

Note that the array y has four elements. The program does not define element 0, so it is zero by default. If we execute MyArray(2), the other elements will be `y(1) = 2`, `y(2) = 4`, and `y(3) = 6`. If we execute the function by typing `=MyArray(2)` in a cell of a worksheet, the number 0 will appear. (To avoid this and have the output show up in three cells instead of four, we could have declared `Dim y(1 To 3)`.) To see the rest of the output, highlight the active cell and the three cells immediately to the right on the same row. Click the function key F2 and then hold down the key combination CTRL-SHIFT-ENTER. This is the standard Excel procedure for displaying arrays returned by functions. For example, the output of Excel's matrix algebra functions, such as `MMULT`, is revealed in the same way.[3] Two-dimensional arrays can be output to worksheets in the same way.

A.14 Debugging

Errors (bugs) are inevitable. VBA will catch some types (for example, syntax errors) and inform you. The more troublesome errors are those that do not prevent the program from running but lead to incorrect results. It is essential therefore to debug each program carefully.

To debug a subroutine, put the cursor on the subroutine name in the Visual Basic editor. Click on Debug/Step Into (or the function key F8) to step through the subroutine one line at a time. Putting the cursor over any variable will show the value of the variable at that stage of the program. To observe the values of variables more systematically, you can include statements of the form `Debug.Print x` or `Debug.Print "The value of x is " & x` in the subroutine. The subroutine will then print to the Immediate Window. To view the Immediate Window, click View/Immediate Window. Click on Run/Reset (or the square button on the toolbar) to discontinue debugging.

To debug a function, one can rewrite it as a subroutine, defining values for the input variables in the beginning of the subroutine. The VBA debugger has many other features. Debug/Step Over is particularly useful for stepping over a line that does not need to be checked and will be time consuming to check, for example, a call to another function.

[3] Once this is done, the individual cells in which the array was output cannot be changed. Attempting to do so will generate an error message, and it may be necessary to hit the Escape key once or twice to allow any use of the worksheet after the error message appears.