An Introduction to VBA in Excel *

Robert L. McDonald†

First draft: November, 1995

January 22, 2001

Abstract

This is a tutorial showing how to use the macro facility in Microsoft Office—Visual Basic for Applications—to simplify analytical tasks in Excel.

Contents

 Thanks to Jim Dana for asking stimulating questions about VBA.

†Finance Dept, Kellogg School, Northwestern University, 2001 Sheridan Rd., Evanston, IL 60208, tel: 847-491-8344, fax: 847-491-5719, E-mail: r-mcdonald@northwestern.edu.

1 Introduction

Visual Basic for Applications, Excel's powerful built-in programming language, permits you to easily incorporate user-written functions into a spreadsheet.[1] You can easily calculate Black-Scholes and binomial option prices, for example. Lest you think VBA is something esoteric which you will never otherwise need to know, VBA is now the core macro language for all Microsoft's office products, including Word. It has also been incorporated into software from other vendors. You need not write complicated programs using VBA in order for it to be useful to you. At the very least, knowing VBA will make it easier for you to analyze relatively complex problems for yourself.

This document presumes that you have a basic knowledge of Excel, including the use of built-in functions and named ranges. I do not presume that you know anything about writing macros or programming. The examples here are mostly related to option pricing, but the principles apply generally to any situation where you use Excel as a tool for numerical analysis.

All of the examples here are contained in the Excel workbook VBA.XLS.

2 Calculations without VBA

Suppose you wish to compute the Black-Scholes formula in a spreadsheet. Suppose also that you have named cells[2] for the stock price (s), strike price (k), interest rate (r_), time to expiration (t), volatility (v), and dividend yield (d). You could enter the following into a cell:

```
s*exp(-d*t)*normsdist((ln(s/k)+(r-d+v ^ 2/2)* t)/(v*t ^0.5))
  −k * exp(-r_ * t)* normsdist((ln(s/k)+(r_-d-v^2/2)*t)/(v*t^0.5))
```

Typing this formula is cumbersome, though of course you can copy the formula wherever you would like it to appear. It is possible to use Excel's data table feature to create a table of Black-Scholes prices, but this is cumbersome and inflexible. If you want to calculate option Greeks (e.g. delta, gamma, etc...) you must again enter or copy the formulas into each cell

[1] This document uses keystrokes which are correct for Office 97. VBA changed dramatically (for the better, in my opinion) between Office 95 and Office 97. The general idea remained the same, but specific keystrokes changed. So far I have not found changes required for Office 2000.

[2] If you do not know what a named cell is, consult Excel's on-line help.

where you want a calculation to appear. And if you decide to change some aspect of your formula, you have to hunt down all occurences and make the changes. When the same formula is copied throughout a worksheet, that worksheet potentially becomes harder to modify in a safe and reliable fashion. When the worksheet is to be used by others, maintainabiltiy becomes even more of a concern.

Spreadsheet construction becomes even harder if you want to, for example, compute a price for a finite-lived American option. There is no way to do this in one cell, so you must compute the binomial tree in a range of cells, and copy the appropriate formulas for the stock price and the option price. is is not so bad with a 3-step binomial calculation, but for 100 steps you will spend quite a while setting up the spreadsheet. You must do this separately for each time you want a binomial price to appear in the spreadsheet. And if you decide you want to set up a put pricing tree, there is no easy way to edit your call tree to price puts. Of course you can make the formulas quite flexible and general by using lots of "if" statements. But things would become much easier if you could create your own formulas within Excel. You can — with Visual Basic for Applications.

3 How to Learn VBA

Before we look at examples of VBA, it is useful to have appropriate expectations. There are several points:

- *For many tasks, VBA is simple to use.* We will see in a moment that creating simple add-in functions in VBA (for example to compute the Black-Scholes formula) is easy.
- *You can do almost anything using VBA.* If you can dream of something you would like Excel to do, the odds are that VBA will enable you to do it.
- *You will never learn all about VBA by reading a book.* If a macro language is so powerful that it enables you to do everything, it is obviously going to be too complex for you to memorize all the commands. A book or tutorial (like this one) will enable you to use VBA to solve specific problems. However, once you want to do more, you will have to become comfortable figuring out VBA by trial and error. The way to do this is...

- *Learn to use the macro-recorder in Excel.* If you turn on the macro recorder, Excel will record your actions using VBA! Try this: select Tools | Macro | Record New Macro in Excel. Then create a simple graph using the graph wizard. Look at the VBA code that Excel creates. (This example is described in more detail on p. 26 below.) It is daunting when you first look at it, but if you want to use VBA to create graphs, you can now simply modify the code that Excel has recorded for you. You do not need to create the basic code from scratch. Of course this raises the question of how to modify the VBA code you now have. The simple fact is that you *must* be comfortable with trial-and-error.

If you are a serious Excel user, VBA can make your life much simpler and greatly extend the power of Excel. The main emphasis of this tutorial is to help you create your own functions. Most of the examples here are for option pricing, but it should be obvious that there are many other uses of VBA.

4 Calculations with VBA

4.1 Creating a simple function

With VBA, it is a simple matter to create your own function, say *BSCall*, which will compute a Black-Scholes option price. To do this, you must first open a special kind of worksheet, called a macro module. Here are the steps required to open a new macro module, and create a simple formula:

1. *Open a blank workbook using File|New.*
2. *Select Tools|Macro|Visual Basic Editor from the Excel menu.*
3. *Within the VBA editor, select Insert|Module from the menu. You will find yourself in a new window, in which you can type macro commands.*
4. *Within the newly-created macro module, type the following exactly (be sure to include the "_" at the end of the second line):*

```
' Here is a function to add two numbers. _
  Works well, doesn't it?
Function addtwo(x, y)
  addtwo = x + y
End Function
```

5. *Return to Excel. In cell A1 enter*

   =addtwo(3,5)

6. *Hit <Enter>. You will see the result "8".*

Congratulations! You have just created an add-in function! There are a number of things to notice about this example:

- You need to tell the function what to expect as input, hence the list "(x,y)" following the name of the function.
- You specify the value of the function with a line where you set the function name ("addtwo") equal to a calculation ("x+y").
- An apostrophe denotes a comment, i.e. text which is not interpreted by VBA. You do not need to create comments, but they are very useful if you ever return to work you did several months ago and have to figure out what you did.
- VBA is line-oriented. This means that when you start a comment with an apostrophe, if you press <enter> to go to a new line, you must enter another apostrophe or else—since what you type will almost surely not be a valid command—VBA will report an error. You can continue a line to the next line by typing an underscore, i.e. "_" without the quotes. (You can test this out by deleting the "_" in the example above.)
- When you entered the comment and the function, VBA automatically color-coded the text and completed the syntax (the line "End Function" was automatically inserted).. Comments are coded green and the reserved words "Function" and "End" were coded in blue and automatically capitalized.
- The function you typed now appears in the function wizard, just as if it were a built-in Excel function! To see this, open the function wizard using Insert|Function. In the left-hand pane ("function category"), scroll to and highlight "User Defined". Note that "Addtwo" appears in the right-side pane ("function name"). Click on it and Excel pops up a box where you are prompted to enter inputs for the function.

4.2 A Simple Example of a Subroutine

A *function* returns a result. A *subroutine* (called a "sub" by VBA) performs an action when invoked. In the above example we used a function, because we wanted to supply two numbers and have VBA add them for us and tell us the sum. For our purposes, this is by far the most important use of VBA. However, it is often useful to create a subroutine, which is a set of statements which execute when invoked. Here is how to create one (the following is easier to do than it is to explain):

1. *Return to the Visual Basic Editor*
2. *Click on the "Module1" window.*
3. *At the bottom of the module (i.e. below the function we just created) enter the following:*

   Sub displaybox()
   response = **MsgBox**("Greetings!")
   End Sub

4. *Return to Excel*
5. *Run the subroutine by using Tools|Macro, then double-clicking on "displaybox" out of the list.*

We have just created and run a subroutine. It pops up a dialog box which displays a message. The "MsgBox" function can be very useful for giving information to the spreadsheet user.

4.3 Creating a Button to Invoke a Subroutine

In the previous section we ran the subroutine by clicking on Tools|Macro|Macros and then double-clicking on the subroutine name. If you are going to run the subroutine often, creating a button in the spreadsheet is a way to create a shortcut to the subroutine. Here is how to create a button:

1. *Move the mouse to the Excel toolbar and right-click once.*
2. *You will see a list of toolbar names. Move the highlight bar down to "forms" and left-click. A new toolbar will pop up.*

3. *The rectangula icon on this toolbar is the "create button" icon, which looks something like a button (of the software, not the clothing, variety). Click on it.*

4. *The cursor changes to a crosshair. Move the mouse to the spreadsheet, hold down the left mouse button, and drag to create a rectangle. When you lift your finger off the mouse button, a dialog box will pop up and one of the choices will be "Displaybox". Double click on that.*

5. *Now move the mouse away from the button you've created and click once (this de-selects the button). Move the mouse back to the button you created and left-click once on it. Observe the dialog box which pops up, and click either "yes" or "no" to get rid of the dialog.*

Some comments:

- This is obviously a trivial example. However, if you have a calculation which is particularly time-consuming (for example a Monte Carlo calculation) you might want to create a subroutine to activate it, and a button to activate the subroutine would be natural.

- There is a more sophisticated version of the MsgBox function which permits you to customize the appearance of the dialog box. It is documented in the on-line help and an example of its use is contained in DisplayBox2 in the workbook. One nice feature of this more sophisticated version is that within the subroutine, we could have checked the value of the variable "response", and had the subroutine perform different actions depending upon which button the user clicked. For an example of this, see the example Displaybox2.[3]

4.4 Functions can call functions

As a final example, we will demonstrate that a function can call a function (or a subroutine).

1. *Enter this code in the "Module1" window*

[3]If you have examined the code for Displaybox2, you may be puzzled by checking to see if response = "vbYes". VbYes is simply an internal constant which VBA uses to check for a "yes" button response to a dialog box. The possible responses—documented in the help file—are vbOK, vbCancel, vbAbort, vbRetry, vbIgnore, vbYes, and vbNo.

Function addthree(x, y, z)
addthree = addtwo(x, y) + z
End Function

2. *Now in cell A2, enter*

=addthree(3,5,7)

The answer 15 will appear. This is again a trivial example, but it illustrates what you can do with functions.

4.5 Illegal Function Names

You need to be aware of what you can and cannot name a function. In Office 97, the documentation has this useful information: the function name "follows standard variable naming conventions." This is obviously not very helpful.

There are some obvious restrictions. (Please don't take the following as definitive, these are mostly things I've figured out.) You cannot name a function a number. You cannot use the following characters in a function name: space . , + - : ; " ' ‘ # $ % / \. It was easy to produce this list because the Visual Basic compiler lets you know immediately that something is wrong if you try to put any of these characters in a function name. Note that you *can* use an underscore where you would like to have a space for readability of the function name. So "BS_2", for example, is a legal function name.

Here is a more subtle issue. There are function names which are legal but which you should most definitely *not* use. "BS2" is an example. This would be fine as the name of a subroutine, which is not called directly from a cell. But think about what happens if you give this name to a user-defined function. You enter, for example, "BS2(3)", in a cell. How does Excel understand this? The problem is that "BS2" *is also the name of a cell.* So if you try to use it as a function in the spreadsheet, Excel will return the cryptic error "#REF!" and you will likely have no idea what is wrong. This is why, later in this tutorial, you will see functions named "BS_2", "BS_3", and so on.

4.6 Differences Between Functions and Subroutines

Functions and subroutines are *not* interchangable and do not have the same capabilities. Subroutines are meant to be invoked by a button or otherwise

explicitly, while functions are meant to be inserted into cells and return results. Because of their different purpose, some VBA capabilities will work in one but not the other.

In a subroutine, for example, you can write to cells of the workbook. With a subroutine you could perform a calculation and have the answer show up in cell A1. You cannot write to cells from within a function (there are several ways to do this from a subroutine; to my knowledge none of them work within functions). You cannot activate a worksheet or change anything about the display from within a function. Subroutines, on the other hand, can not be called from cells. Ocassionally you may find these restrictions annoying, but they exist because functions and subroutines are intended for different purposes.

5 Storing and Retrieving Variables in a Worksheet

Suppose that there is a value in the spreadsheet which you want to have affect your function or subroutine. (For instance you might have a variable which determines whether the option to be valued is American or European.) Or suppose you create a subroutine which performs computations. You may want to display the output in the spreadsheet. (For example, you might wish to create a subroutine to draw a binomial tree.) This raises the general question: if you are using VBA, how do you read and write values from the spreadsheet?

Obviously if you are going to read and write numbers to specific locations in the spreadsheet, you must identify those locations. The easiest way to identify a location is to use a named range. The alternatives—which we will examine below—requires that you "activate" a specific location or worksheet within the workbook, and then read and write within this activated region.

There are at least three ways to read and write to cells:

- "Range" lets you address cells by name,
- "Range().Activate" and "ActiveCell" lets you easily get at cells by using traditional cell addresses (e.g. "A1"), and
- "Cell" lets you address cells using a row and column numbering scheme.

Although you are probably thinking that it seems silly to have so many ways to get to cells, each is useful at different times.

5.1 Using a named range to read and write numbers from the spreadsheet

1. *Enter the following function in Module1:*

```
Sub ReadVariable()
  x = Range("test")
  MsgBox (Str(x))
End Sub
```

2. *Select cell A1 in sheet "Sheet2", then Insert|Name|Define, and type "test", then click OK. You have just created a named range*

3. *Enter the value "5" in the cell you just named "test".*

4. *Tools|Macro, then double-click on "ReadVariable".*

At this point you have just read from a cell and displayed the result. Note that "x" is a number in this example. In order to display it using MsgBox, we first need to convert it into a string. We did this using the built-in "Str" function. (You can find locate this function by using—you guessed it—the object browser, looking under VBA, then "conversions".)

As you might guess, you can use the Range function to write as well as read.

1. *Enter the following subroutine in Module1:*

```
Sub WriteVariable()
  Range("Test2") = Range("Test")
  MsgBox ("Number copied!")
End Sub
```

2. *Give the name "test2" to cell sheet2.b2.*

3. *Enter a number in sheet2.b2.*

4. *Go to Tools|Macro, then double click on "WriteVariable".*

5. *The number from Test is copied to Test2.*

5.2 Reading and Writing to Cells Which are not Named.

You can also access a specific cell directly. In order to do this, you first have to "activate" the worksheet containing the cell. Here is VBA code to read a variable:

```
Sub ReadVariable2()
  Worksheets("Sheet2").Activate
  Range("A1").Activate
  x = ActiveCell.Value
  MsgBox (Str(x))
End Sub
```

Notice what is happening: we first activate the worksheet named "sheet2". Next we activate the cell "A1" within Sheet2. You will notice that when you have finished calling this function, the cursor has moved to cell A1 in Sheet2! This is because what Excel means by "Active Cell" is whatever cell the cursor happens to be on; the first two lines instruct the cursor to move to sheet2.a1.

The active cell has properties, such as the font, color of the cell, formatting, etc... All of these properties may be accessed using the ActiveCell function. For fun, insert the line

```
ActiveCell.Font.Bold = True
```

after the MsgBox function. Then switch to sheet2,run the subroutine, and watch what happens to cell A1. (I'll bet you can guess before you run the example!)

We can also assign a value to ActiveCell.Value; this is a way to write to a cell. Here is a macro which does this:

```
Sub WriteVariable2()
  Worksheets("Sheet2").Activate
  Range("A1").Activate
  X = ActiveCell.Value
  Range("B1").Activate
  ActiveCell.Value = X
End Sub
```

This subroutine reads the number from Sheet2.A1 and copies it to Sheet2.B1.

5.3 Using the "Cells" Function to Read and Write to Cells.

There is yet another way to read and write to cells. The "Cells" function lets you address cells using a numerical row and column numbering scheme. Here is an example illustrating how Cell works:

```
Sub CellsExample()
  ' Make "Sheet2" the active sheet
  Worksheets("Sheet2").Activate
  ' The first entry is the row, the second is the column
  ' Write the number 1 into cell A3
  Cells(3, 1) = 1
  ' Write the number 2 into cell A4
  Cells(4, 1) = 2
  ' Copy the number from cell A3 into cell C3
  Cells(3, 3) = Cells(3, 1)
  ' Copy the number from cell A4 into cell C4
  Cells(4, 3) = Cells(4, 1)
End Sub
```

This example reads the numbers 1 and 2 into A3 and A4, and copies them into C3 and C4. Later we will use the Cells function to draw a binomial tree.

6 Using Excel Functions from Within VBA

6.1 Using VBA to compute the Black-Scholes formula

There is only one complicated piece of the Black-Scholes calculation: computing the cumulative normal distribution, the "N()" function in the formula. Based on the very first example above, we would like to do something like the following:

```
function BS(s,k,v,r,t,d)
  BS=s*exp(-d*t)*normsdist((ln(s/k)+(r-d+v^2/2)*t)/(v*t^0.5)) _
   -k*exp(-r*t)*normsdist((ln(s/k)+(r-d-v^2/2)*t)/(v*t^0.5))
end function
```

Unfortunately, this doesn't work. The reason it doesn't work is that VBA does not understand either "ln" or "NormSDist". Though these are functions in Excel they are not functions in VBA, even though VBA is part of Excel! (Who says software is getting easier to use?)

Instead of using "ln", we can use "log", which is the VBA version of the same function. However, there is no VBA versions of NormSDist.

Fortunately, there is a way for you to tell VBA that NormSDist is located inside of Excel. This will then make the function available for use in VBA. The following example will first show you the error you get if you fail to call NormSDist correctly:

1. *Click on the "Module1" tab*

2. *Enter the following:*

   Function BS(s, k, v, r, t , d)
   d1 = (**Log**(s / k) + (r − d + v ^ 2 / 2) ∗ t) / (v ∗ **sqr**(t))
   d2 = d1 − v ∗ sqrt(t)
   bs = s∗**exp**(−d∗t)∗normsdist(d1)−k∗**exp**(−r∗t)∗normsdist(d2)
   End Function

   Comment: To save a little typing and to make the function more readable, we are defining the Black-Scholes "d1" and "d2" separately. You will also notice that instead of entering "ln", we entered "log", which—as we noted above—is built into VBA.

3. *Enter into the spreadsheet*

   =bs(40,40,.3,.08,.25,0)

   Hit <Enter>. You will get the error message "sub or function not defined".

This error occurs because there is no version, however spelled, of "NormSDist" which is built-in to VBA. Instead, we have to tell VBA where to look for "NormSDist". We do this by typing instead "WorksheetFunction.NormSDist".[4]

Correctly typed, the function becomes:

[4] If you are curious about this, do the following: Select View|Object Browser or press F2. Click on the drop-down arrow under "libraries/workbooks", then select "Excel". Under "Objects/Modules" click on "Application", then under "Methods/Properties" scroll down to "NormSDist". You have now just located "NormSDist" as a method available from the application. If you scroll around a bit you will see that there is an enormous and overwhelming number of functions available to be called from VBA.

By the way, you should not make the mistake of thinking that you can call any Excel function simply by prefacing it with "WorksheetFunction" Try it with "sqrt" and it won't work. The only way to know for sure which functions you can and can't call is by using the object browser.

```
Function BS(s, k, v, r, t, d)
  d1 = (Log(s / k) + (r - d + v ^ 2 / 2) * t) / (v * t ^ 0.5)
  d2 = d1 - v * t ^ 0.5
  bs = s*exp(-d*t)*WorkSheetFunction.NormSDist(d1) _
    -k*exp(- \_ r*t)*WorkSheetFunction.NormSDist(d2)
End Function
```

The Black-Scholes function will now evaluate correctly to 2.78.

6.2 The Object Browser

The foregoing example illustrates an extremely powerful feature of VBA: it can access the functions built into Excel if you can tell it where to find them. The way you locate other functions is to use the *Object Browser*, which is part of VBA. Here is how to use it:

1. *From within a macro module, press the F2 key. This will pop up a dialog box with the title "Object Browser".*
2. *In the top left you will see a drop-down box which says "All Libraries". Click on the down arrow at the right of this line. You will see a drop-down list with, at a minimum, "VBA" and "Excel" as two entries. (There may be other entries, depending upon how you have set up Excel.)*
3. *Click on VBA.*
4. *In the "Classes" list, click on "Math".*
5. *To the right, in the "Members of Math" box, you now have a list of all the math functions which are available in VBA. Note that "log" is included in this list, but not "ln" or "NormSDist". If you right-click on "log" and then click on "help", you will see that "log" returns the natural logarithm.*
6. *Move back to the top left box which now says "VBA". Click on the down arrow at the right of this line.*
7. *Click on Excel.*
8. *In the "Objects/Modules" list, click on "WorksheetFunction".*

9. *To the right, in the "Members of WorksheetFunction" box, you now have a list of Excel built-in functions which may be called from a macro module by specifying* "WorksheetFunctionfunctionname". Note that both *"ln" and "NormSDist" are included in this list. Note also that "log" is included in this list, but be aware that Excel's "log" function is base 10 by default (you can specify a different base), while VBA's is base e.*[5] *Note also that "Sqrt" is not included. For some reason Excel's Sqrt function is not available in VBA, which means that you cannot count on an Excel function automatically being available (though most are).*

If you create any VBA functions which are even moderately ambitious, you are going to use the object browser. It is the heart and soul of VBA.

7 Checking for Conditions

Frequently you want to perform a calculation only if certain conditions are met. For example, you would not want to calculate an option price with a negative volatility. It makes sense to check to see if your inputs make sense before proceeding with the calculation, and aborting—possibly with an error message—if they do not.

The easy way to check is to use the construct is If ... Then ... Else.[7] Here is an example of its use in checking for a negative volatility in the Black-Scholes formula:[8]

```
Function BS_2(s, k, v, r, t, d)
  If v > 0 Then
    BS_2 = BS(s, k, v, r, t, d)
  Else
    MsgBox ("Negative_volatility!")
    BS_2 = CVErr(xlErrValue)
  End If
End Function
```

[5] Scroll down to the ":Log" entry and then click on the "?" button at the bottom left. If you use "log" in a spreadsheet, or if you use "worksheetfunction.log" in a function, you will get the base 10 logarithm. However, if you use "log" in a function, you will get the base e logarithm! Be very very sure you know what you are doing when you use a function!

[7] There is also a Case ... Select construct which we will not use.

[8] You need to be aware that VBA will expect the "If then", "Else", and "End if" pieces to be on separate lines. If you write "else" on the same line as "If then", for example, the code above will fail.

This function checks to see if volatility is greater than 0; if it is, the function computes the Black-Scholes formula using the BS function we created earlier. If volatility is not greater than zero, then two things happen: (i) a MsgBox pops up which informs you of the mistake and (ii) the function returns a value indicating that there is an error.

In general you should be cautious about putting message boxes into a function (as opposed to a procedure), since every time the spreadsheet is recalculated the message box will pop up.

Since error-checking is often critically important (you would not want to quote a client a price on a deal for which you had accidentally entered a negative volatility!), it is worth expanding a bit on the use of the CVErr function.

If the user enters a negative volatility, you *could* just have Excel return a nonsense value for the option, such as -99. This would be a bad idea, however. What if you have a complicated worksheet with many option calculations? If you failed to notice the error, the -99 would be treated as a true option value and propagated throughout your calculations.

Alternatively, you could have the function return a string such as "You dweeb, you entered a negative volatility". Apart from the fact that a colleague might use your spreadsheet and take the message personally, this is also not a good idea. Entering a string in a cell when you should have a number could have unpredictable effects on calculations which depend on the cell. While it is obvious that an addition between a string and a number will fail, suppose you are performing a regression or a frequency count. Are you sure what will happen to the calculation if you introduce a string among the numbers in your data?

By using CVErr along with one of the built-in error codes (all of which are called "xlErr*something*") you are guaranteeing that your function will return a result which Excel recognizes as a numerical error. The Excel programmers have already thought through the issues of how subsequent calculations should respond to a recognized error, and Excel usually does something reasonable in those circumstances. The Excel error codes are documented in the on-line VBA help.

8 Arrays

Often you will wish to use a single variable to store many numbers. For example, in a binomial option calculation, you have a stock price tree. After n periods, you have n possible stock prices. It can be useful to write the

lowest stock price as P(1), the next as P(2), and the highest as P(n). The variable P is then called an array—it is a single variable which stores more than one number. You access the different elements of the array by selecting the appropriate argument.

8.1 Defining Arrays

When you create an array, it is necessary to tell VBA how big the array is going to be. You do this by using the Dim statement. Here are some examples of how to use Dim:

Dim P(2) as double

This creates an array of three double precision real numbers, with the array index running from 0 to 2. (By default, the first subscript in an array is 0.) If you had written

Dim P(3 to 5) as double

you would have created a 3-element array with the index running from 3 to 5.

In this example we told Excel that the variable is type "double". This was not necessary—we could have left the type unspecified and permitted Excel to determine the type automatically. It is faster and easier to detect mistakes, however, if we specify the type.

Here is a routine which defines a 3-element array, reads numbers into the array, and then writes the array out into dialog boxes:

```
Sub UseArray()
  Dim X(2) As Double
  X(0) = 0
  X(1) = 1
  X(2) = 2
  y = X(0)
  MsgBox (Str(y))
  y = X(1)
  MsgBox (Str(y))
  y = X(2)
  MsgBox (Str(y))
End Sub
```

You should enter this code and execute it to see what happens. The subroutine UseArray can also be written as follows

Sub UseArray2()

```
  X = Array(0, 1, 2)
  y = X(0)
  MsgBox (Str(y))
  y = X(1)
  MsgBox (Str(y))
  y = X(2)
  MsgBox (Str(y))
End Sub
```

There is an interesting difference between UseArray and UseArray2 in the way arrays are declared. In UseArray, there is a dimension statement, and then array elements are created one by one. In UseArray2, there is *no* dimension statement, and the array function (built into VBA) is used to initialize the array. UseArray will fail without the "Dim" statement, and UseArray2 will fail *with* the "Dim" statement. I don't know why.

You might wonder why in this example I write

```
y = X(2)
MsgBox (Str(y))
```

instead of just

```
MsgBox (Str(X(2)))
```

The reason is that "MsgBox(Str(X(2)))" does not work. It seems that it should, but it doesn't.[1]

Finally, notice the repetition in these examples! The statements which put numbers into the array are essentially repeated three times (albeit more compactly in UseArray2), and the statements which read numbers out of the array are repeated three times. Suppose the array had 100 elements! Surely there must be a better way. There is. It is time to learn about how to perform repetitive calculations by iterating.

9 Iterating

Many kinds of option calculations are repetitive. For example, when we compute a binomial option price, we generate a stock price tree and then traverse the tree, calculating the option price at each node. Similarly, when

[1] This is just one example of why *real* programmers might kick silicon in your face if you tell them you're using VBA instead of, say, C++. But you can just smile, get your work done, and close the deal, while they're still yapping about how VBA is nothing but a toy.

we compute an implied volatility we need to perform a calculation repeatedly until we arrive at the correct volatility. VBA provides us with the ability to write one or more lines of code which can be repeated as many times as we like.

9.1 A simple *for* loop

Here is an example of a *for loop*, which does exactly the same thing as the UseArray subroutine above:

```
Sub UseArrayLoop()
  Dim X(2) As Double
  For i = 0 To 2
  X(i) = i
  Next i
  For i = 0 To 2
    y = X(i)
    MsgBox (Str(y))
  Next i
End Sub
```

This table translates the syntax in the first loop above:

For i = 0 to 2	*Repeat the following statements three times, the first time setting i = 0, the next time i = 1, and finally i = 2.*
y = X(i)	*Set the variable y equal to the ith value of X.*
Next i	*Go back and repeat the statement for the next value of i.*

9.2 Creating a binomial tree

In order to create a binomial tree, we need the following information:

- The initial stock price
- The number of time periods
- The magnitudes up and down by which the stock moves.

Suppose we wish to draw a tree where the initial price is 100, we have 10 binomial periods, and the moves up and down are u = 1.25 and d = 1/u

= .8. Here is a subroutine, complete with comments explaining the code, which will create this tree. You first need to name a worksheet "output", and then we will write the tree to this worksheet. The number of binomial steps and the magnitude of the moves are read from named cells, which can be in any worksheet. I have placed those named cells in Sheet1. We will put these named cells in Sheet2.[6]

```
Sub DrawBinomialTree()

  ReDim stock(2) ' provide default of 2 steps if no steps specified
  Dim i as integer
  Dim t as integer
  n = Range("n") ' number of binomial steps
  u = Range("u") ' move up
  P0 = Range("p0") ' initial stock price
  d = 1 / u ' move down
  ReDim stock(n + 1) ' array of stock prices
  Worksheets("Output").Activate
  ' Erase any previous calculations
  Worksheets("Output").Cells.ClearContents
  Cells(1, 1) = P0
  ' We will adopt the convention that the column holds the
  ' stock prices for a given point in time. The row holds
  ' stock prices over time. For example, the first row
  ' holds stock prices resulting from all up moves, the
  ' second row holds stock prices resulting from a single
  ' down move, etc...

  ' The first loop is over time
  For t = 2 To n
    Cells(1, t) = Cells(1, t - 1) * u
    ' The second loop is across stock prices at a given time
    For i = 2 To t
      Cells(i, t) = Cells(i - 1, t - 1) * d
    Next i
  Next t
End Sub
```

[6] If it is not obvious to you that using the "Cells" function is the best way to write to cells here, you should think about how you would do it using the other methods we discussed earlier.

Several comments:

- There is a simple command to clear an entire worksheet, namely: Worksheets("*worksheetname*").Cells.ClearContents.
- The use of the Cells function means that you can perform the calculation exactly as you would if you were writing it down, using subscripts to denote which price you are dealing with. Think about how much more complicated it would be to use traditional row and column notation (e.g. "A1") to perform the same function.

Note that this function uses the "ReDim" command to specify a flexible array size. Sometimes you do not know in advance how big your array is going to be. In this example you are unsure how many binomial periods the user will specify. If you are going to use an array to store the full set of prices at each point in time, this presents a problem—how large do you make the array? You could specify the array to have a very large size, one larger than any user is ever likely to use, but this kind of practice could get you into trouble someday. Fortunately, VBA permits you to specify the size of an array using a variable. You do it by using the Redim ("dimension") statement.

9.3 Other kinds of loops

We will not go into them here, but you should be aware that there are other looping constructs available in VBA. The following kinds of loops are also available:

- Do Until .... Loop *and* Do ... Loop Until
- Do While ... Loop *and* Do ... Loop While
- While ... Wend
- For Each ... In ... Next

If you ever think you need them, you can look these up in the on-line help.

10 Reading and Writing Arrays

A powerful feature of VBA is the ability to read arrays as inputs to a function and also to write functions which return arrays as output.

10.1 Arrays as Output

Suppose you would like to create a single function which returns two numbers: the Black-Scholes price of a call option, and the option delta. Let's call this function **BS_3** and create it by modifying the function **BS** from Section 6.1.

```
Function bs_3(s, k, v, r, t, d)
  d1 = (Log(s / k) + (r − d + 0.5 ∗ v ^ 2) ∗ t) / (v ∗ t ^ 0.5)
  d2 = d_1 − v ∗ t ^ 0.5
  nd1 = WorksheetFunction.NormSDist(d1)
  nd2 = WorksheetFunction.NormSDist(d2)
  delta = Exp(−d ∗ t) ∗ nd1
  price = s ∗ delta − k ∗ Exp(−r ∗ t) ∗ nd2
  bs_3 = Array(price, delta)
End Function
```

The key section is the line

```
bs_3 = Array(price, delta)
```

We assign an array as the function output, using the array function introduced in Section 8.

If you just enter the function bs_3 in your worksheet in the normal way, in a single cell, it will return a single number. In this case, that single number will be the option price, which is the first element of the array. If you want to see both numbers, you have to enter bs_3 as an array function: select a range of two cells, enter the formula in the first, and then hit Ctrl-Shift-Enter to create the array function.

There is a 50% probability you just discovered a catch. The way we have written bs_3, the array output is *horizontal*. If you enter the array function in cells a1:a2, for example, you will see the option price repeated twice. If you enter the function in a1:b1, you will see the price and the delta. What happens if we want vertical output? The answer is that we transpose the array using the Excel function of that name, modifying the last line to read

```
bs_3 = WorksheetFunction.Transpose(Array(price, delta))
```

This will make the output be vertical.

There is also a way to make the output *both* horizontal and vertical. We just have to return a 2×2 array. Here is an illustration of how to do that:

```
Function bs_4(s, k, v, r, t, d)
  Dim temp(1, 1) As Double
```

```
  d1 = (Log(s / k) + (r − d + 0.5 ∗ v ^ 2) ∗ t) / (v ∗ t ^ 0.5)
  d2 = d_1 − v ∗ t ^ 0.5
  nd1 = WorksheetFunction.NormSDist(d1)
  nd2 = WorksheetFunction.NormSDist(d2)
  delta = Exp(−d ∗ t) ∗ nd1
  price = s ∗ delta − k ∗ Exp(−r ∗ t) ∗ nd2
  temp(0, 0) = price
  temp(0, 1) = delta
  temp(1, 0) = delta
  temp(1, 1) = 0
  bs_4 = temp
End Function
```

Now it does not matter whether you select cells a1:a2 or a1:b1; either way, you will see both the price and the delta.[2]

10.2 Arrays as Inputs

We may wish to write a function which processes many inputs, where we do not know in advance how many inputs there will be. Excel's built-in functions "sum" and "average" are two familiar examples of this. They both can take a *range* of cells as input. It turns out it is easy to write functions which accept ranges as input. Once in the function, the array of numbers from the range can be manipulated in at least two ways: as a *collection*, or as an actual array with the same dimensions as the range.

10.2.1 The Array as a Collection

First, here are two examples of how to use a collection.

Sumsq takes a set of numbers, squares each one, and adds them up:

```
Function sumsq(x)
  Sum = 0
  For Each y In x
    Sum = Sum + y ^ 2
  Next
  sumsq = Sum
End Function
```

The function **sumsq** can take a range (e.g. "a1:a10") as its argument. The "for each" construct in VBA loops through each element of a collection,

[2] What do you see if you select cells a1:b2? What about a1:d4?

without our having to know in advance how many elements the collection has.

There is another way to loop through the elements of a collection. The function **SumProd** takes two equally-sized arrays, multiplies them element by element, and returns the sum of the multiplied elements. In this example, because we are working with two collections, we need to use a more standard looping construct. To do this, we need to first count the number of elements in each array. This is done using the "count" property of a collection. If there is a different number of elements in each of the two arrays, we exit and return an error code.

```
Function SumProd(x1, x2)
  n1 = x1.Count
  n2 = x2.Count
  If n1 <> n2 Then
     'exit if arrays not equally-sized
     sumprod = CVErr(xlErrNum)
     End
  End If
  Sum = 0
  For i = 1 To n1
    Sum = Sum + x1(i) * x2(i)
  Next i
  i = 1
  SumProd = Sum
End Function
```

10.2.2 The Array as an Array

We can also treat the numbers in the range as a literal array. The only trick to doing that is that we need to know the dimensions of the array, i.e. how many rows and columns it has. The function **RangeTest** illustrates how to do this.

```
Function RangeTest(x)
prod = 1
r = x.Rows.Count
c = x.Columns.Count
For i = 1 To r
    For j = 1 To c
       prod = prod * x(i, j)
```

```
    Next j
Next i
RangeTest = WorksheetFunction.Transpose(Array(prod, r, c))
End Function
```

This function again multiplies together the cells in the range. It returns not only the product, but also the number of rows and columns.

When x is read into the function, it is considered by VBA to be an array.[3] "Rows" and "Columns" are properties of an array. The construct

```
x.Rows.Count
```

tells us the number of rows in the array. With this capability, we could multiply arrays, check to see whether two ranges have the same dimensions, and so on.

11 Miscellany

11.1 Getting Excel to generate your macros for you

Suppose you want to perform a task and you don't have a clue how to program it in VBA. For example, suppose you want to create a subroutine to set up a graph. You can set up a graph manually, and tell Excel to record the VBA commands which accomplish the same thing. You then examine the result and see how it works. To do this, select Tools|Record Macro|Record New Macro. Excel will record all your actions in a new module located at the *end* of your workbook, i.e. following Sheet16. You stop the recording by clicking the -?- button which should have appeared on your spreadsheet when you started recording. Macro recording is an *extremely* useful tool for understanding how Excel and VBA work and interact; this is in fact how the Excel experts learn how to write macros which control Excel's actions.

For example, here is the macro code Excel generates if you use the chart wizard to set up a chart using data in the range A2:C4. You can see among other things that the selected graph style was the fourth line graph in the graph gallery, and that the chart was titled "Here is the title". Also, each data series is in a column and that the first column was used as the x-axis ("CategoryLabels:=1").

[3] You can verify this by using the VBA function IsArray. For example, you could write

```
y = IsArray(x)
```

and y will have the value "true" if x is a range input to the function.

```
' Macro1 Macro
' Macro recorded 2/17/99 by Robert McDonald
'
'
Sub Macro1()
Range("A2:C4").Select
ActiveSheet.ChartObjects.Add(196.5, 39, 252.75, 162).Select
Application.CutCopyMode = False
ActiveChart.ChartWizard Source:=Range("A2:C4"), Gallery:=xlLine,_
  Format:=4, PlotBy:=xlColumns, CategoryLabels:=1,SeriesLabels _
  :=0, HasLegend:=1, Title:="Here is the Title", CategoryTitle _
  :="X-Axis", ValueTitle:="Y-Axis", ExtraTitle:=""
End Sub
```

11.2 Using multiple modules

You can split up your functions and subroutines among as many modules as you like—functions from one module can call another, for example. Using multiple modules is often convenient for clarity. If you put everything in one module you will spend a lot of time scrolling around.

11.3 Recalculation speed

One unfortunate drawback of VBA—and of most macro code in most applications—is that it is slow. When you are using built-in functions, Excel performs clever internal checking to know whether something requires recalculation (you should be aware that on occasion it appears that this clever checking goes awry and something which should be recalculated, isn't). When you write a custom function, however, Excel is not able to perform its checking on your functions, and it therefore tends to recalculate everything. This means that if you have a complicated spreadsheet, you may find *very* slow recalculation times. This is a problem with custom functions and not one you can do anything about.

There are tricks for speeding things up. Here are two:

- If you are looping a great deal, be sure to declare your looping index variables as integers. This will speed up Excel's handling of these variables. For example, if you use i as the index in a *for* loop, use the statement

```
Dim i as integer
```

- While a lengthy subroutine is executing, you may wish to turn off Excel's screen updating. You do this by

  `Application.ScreenUpdating = False`

  This will only work in subroutines, not in functions. If you want to check the progress of your calculations, you can turn ScreenUpdating off at the beginning of your subroutine. Whenever you would like to see your calculation's progress (for example every 100^{th} iteration) you can turn it on and then immediately turn it off again. This will update the display.

- Finally, here is a good thing to know:

  Ctrl-Break will (Usually) stop a recalculation!

  Remember this. Someday you will thank me for it. Ctrl-Break is more reliable if your macro writes output to the screen or spreadsheet.

11.4 Debugging

We will not go into details here, but VBA has very sophisticated debugging capabilities. For example, you can set breakpoints (i.e. lines in your routine where Excel will stop calculating to give you a chance to see what is happening) and watches (which means that you can look at the values of variables at different points in the routine). Look up "debugging" in the on-line help.

11.5 Creating an Add-in

Suppose you have written a useful set of option functions and wish to make them broadly available in your spreadsheets. You can make the functions automatically available in *any* spreadsheet your write by creating an add-in. To do this, you simply save the file as an add-in, by selecting File|Save As, and then selecting the type of file to be "Microsoft Excel Add-in (*.xla)". Excel will create a file with the XLA extension which contains your functions. You can then make these functions automatically available through Tools|Add-ins, and browse to locate your own add-in module if it does not appear on the list.

Any functions available through an add-in will automatically show up in the function list under the set of "user-defined" functions.

12 A Simulation Example

Suppose you have a large amount of money to invest. Suppose that at the end of the next five years you wish to have it fully invested in stocks. It is often asserted in the popular press that it is preferable to invest it in the market gradually, rather than all at once. In particular, consider the strategy of each quarter taking a pro rata share of what is left and investing it in stocks. So the first quarter invest 1/20th in stocks, the second invest 1/19th of money remaining in stocks, etc... It is obvious that the strategy in which we invest in stocks over time should have a smaller average return and a lower standard deviation than a strategy in which we plunge into stocks, but how much lower and smaller? Monte Carlo simulation is a natural tool to address a question like this. We will first see how to structure the problem and then analyze it in Excel.[9,10] You may not understand the details of how the random stock price is generated. That does not matter for purposes of this example; rather, the important thing is to understand how the problem is structured and how that structure is translated into VBA.

12.1 What is the algorithm?

To begin, we describe the investment strategy and the evolution over time of the portfolio. Suppose we initially have $100 which is invested in bonds and nothing invested in stock. Let the variables BONDS and STOCK denote the amount invested in each. Let h be the fraction of a year between investments in stock (so for example if h = .25, there are 4 transfers per year from bonds to stock), and let r, μ, and σ denote the risk-free rate, the expected return on the stock, and the volatility of the stock.

Suppose we switch from bonds to stock 20 times, once a quarter for 5 years. Let n = the number of times we will switch. We need to know the stock price each time we switch. Denote these prices by PRICE(0), PRICE(1), PRICE(2), ... , PRICE(20). Now each period, at the beginning

[9]The following example is considerably more complicated than those that precede it. It is designed to illustrate many of the basic concepts in a non-trivial fashion. You may wish to skip it initally, and return to it once you have had some experience with VBA.

[10]If you are thinking about option-pricing, you might expect this example to be computed using the risk-neutral distribution. Instead, we will compare the actual payoff distributions of the two strategies in order to compare the means and standard deviations. If we wished to value the two strategies, we would substitute the risk-neutral distribution by replacing the 15% expected rate of return with the 10% risk-free rate. After making this substitution, both strategies would have the same expected payoff of $161 (100 $\times$1.15). Since both strategies entail buying assets at a fair price, there is no need to perform a valuation! Both will be worth the initial investment.

of the period we first switch some funds from bonds to stock. At the end of the period, we figure out how much we earned over the period. If we wish to switch a roughly constant proportion each period, we could switch 1/20 the first period, 1/19 with 19 periods to go, and so forth. This suggests that at the beginning of period j,

bonds(j)=bonds(j−1) ∗ (1−1/(n+1−j))
stock(j)=stock(j−1)+bonds(j−1)/(n+1−j)

At the end of the period we have

stock(j)=stock(j) ∗ price(j)/price(j−1)
bonds(j) = bonds(j) ∗ **exp**(r ∗ h)

In words, during period j, we earn interest on the bonds and capital gains on the stock. We can think of the STOCK(j) and BONDS(j) on the right-hand side as denoting beginning of period values after we have allocated some dollars from bonds to stock, and the values on the right-hand side as the end-of-period values after we have earned interest and capital gains.

We compute capital gains on the stock by

```
price(j) = price(j−1) ∗ exp((mu − 0.5 ∗ v ^ 2) ∗ h _
   + v ∗ h ^ (0.5) ∗ WorksheetFunction.NormSInv(rnd()))
```

As mentioned above, it is not important if you do not understand this expression. It is the standard way to create a random lognormally-distributed stock price, where the expected return on the stock is mu, the volatility is v, and the length of a period is h. At the end, j = n, and we will invest all remaining bonds in stock, and earn returns for one final period.

This describes the stock and bond calculations for a single set of randomly-drawn lognormal prices. Now we want to repeat this process many times. Each time, we will save the results of the trial and use them to compute the distribution.

12.2 VBA code for this example.

We will set this up as a subroutine. The first several lines in the routine simply activate the worksheet where we will write the data, and then clear the area. We need two columns: one to store the terminal portfolio value if we invest fully in stock at the outset, the other to store the terminal value if invest slowly. Note that we have set it up to run 2000 trials, and we also clear 2000 rows. We tell VBA that the variables “bonds”, “stock”, and “price” are going to be arrays of type double, but we do not yet know what size to make the array. The “Worksheets(“Invest Output”).Activate makes

the "invest output" worksheet the default worksheet, so that all reading and writing will be done to it unless another worksheet is specified.

```
1  Sub Monte_invest()
2  Dim bonds() As Double
3  Dim stock() As Double
4  Dim price() As Double
5  Worksheets("Invest Output").Activate
6  Range("a1..b2000"). Select
7  Selection.Clear
8  ' number of monte-carlo trials
9  iter = 2000
```

Now we set the parameters. The risk-free rate, mean return on the stock, and volatility are all annual numbers. We invest each quarter, so h = 0.25. There are 20 periods to keep track of since we invest each quarter for 5 years. Note that once we specify 20 periods, we can dimension the bonds, stock, and price variables to run from 0 to 20. We do this using the "Redim" command.

```
10 ' number of reinvestment periods
11 n = 20
12 ' Reset the dimension of the bonds and stock variable
13 ReDim bonds(0 To n), stock(0 To n), price(0 To n)
14 ' length of each period
15 h = 0.25
16 ' expected return on stock
17 mu = 0.15
18 ' risk-free interest rate
19 r = 0.1
20 ' volatility
21 v = 0.3
```

Now we have an outer loop. Each time through this outer loop, we have one trial, i.e. we draw a series of 20 random stock prices and we see what the terminal payoff is from our 2 strategies. Note that before we run through a single trial we have to initialize our variables: the initial stock price is 100, we have $100 of bonds and no stock, and "Price(0)", which is the intial stock price, is set to 100.

```
22 ' each time through this loop is one complete iteration
23 For i = 1 To iter price(0) = 100
24   bonds(0) = 100
```

```
25  stock(0) = 0
```

This is the heart of the program. Each period for 20 periods we perform our allocation as above. Note that we draw a new random stock price using our standard lognormal expression.

```
26  For j = 1 To n
27      ' allocate 1/n of bonds to stock
28      stock(j) = stock(j-1) + bonds(j-1) / (n + 1 - j)
29      bonds(j) = bonds(j-1) * (1 - 1 / (n + 1 - j))
30
31      ' draw a new lognormal stock price
32      price(j) = price(j-1) * Exp((mu - 0.5 * v ^ 2) * h + _
33          v * h ^ (0.5) * WorksheetFunctionNormSInv(Rnd()))
34
35      ' earn returns on bonds and stock
36      bonds(j) = bonds(j) * Exp(r * h)
37      stock(j) = stock(j) * ( price(j) / price(j-1))
38
39  Next j
```

Once through this loop, all that remains is to write the results to "sheet1". The following two statements do that, by writing the terminal price to column 1, row i, and the value of the terminal stock position to column 2, row i.

```
40  ActiveSheet.Cells(i, 1) = price(n)
41  ActiveSheet.Cells(i, 2) = stock(n)
42
43  Next i
44
45  End Sub
```

Note that you could also write the data across in columns: you would do this by writing

```
ActiveSheet.Cells(1, i) = p1
```

This would write the terminal price across the first row.

12.3 A trick to speed up the calculations

Modify the inner loop by adding the two lines referring to "screenupdating":

```
' each time through this loop is one complete iteration
For i = 1 To iter
  Application.ScreenUpdating = false
  ...
  If (i mod 100 = 0) then application.screenupdating = true
  ActiveSheet.Cells(i, 1) = price(i)
  ActiveSheet.Cells(i, 2) = stock(i)
Next i
```

The first line prevents Excel from updating the display as the subroutine is run. It turns out that it takes Excel quite a lot of time to redraw the spreadsheet and graphs when numbers are added.

The second line at the end redraws the spreadsheet every 100 iterations. The "mod" function returns the remainder from dividing the first number by the second. Thus,

```
i mod 100
```

will equal 0 whenever i is evenly divisible by 100. So on iteration numbers 100, 200, etc..., the spreadsheet will be redrawn. This cuts the calculation time approximately in half.

Note that "Application.ScreenUpdating" is an example of a command which only works within a subroutine. It will not work within a function.