VBA TUTORIAL

Professor Doug Blackburn

PURPOSE: For students with exposure to programming (which is assumed for incoming MSQF students) to learn the basic syntax of VBA programming. This tutorial is not intended for those with no programming background.

PRELIMINARIES:

VBA (Visual Basic for Application) is a programming language attached to all Microsoft products. We will be using VBA within the context of Excel.

The Developer Tab: To allow for easy movement between VBA and Excel, it is useful to “turn on” the Developer Tab. To do this, in Excel, click on the Microsoft icon at the top left corner of Excel. Select ‘EXCEL OPTIONS’. Under the ‘POPULAR’ tab, check the ‘SHOW DEVELOPER TAB IN THE RIBBON’. After doing this, you will notice the new Developer tab in the ribbon (along with ‘Home’, ‘Insert’, ‘Page Layout’, etc.)

Opening a VBA programming window: you can either click ‘VISUAL BASIC’ from the developer tab or simply press ‘ALT F11’ simultaneously. Now, click on the ‘INSERT’ tab and select ‘MODULE’.

GETTING STARTED

The method I will use to introduce VBA is through many examples.

You can write as many routines within one Module so long as each routine is titled differently.

Here we go …

WRITING TO CELLS IN EXCEL FROM VBA

Our primary goal is to build financial models that allow the user to input preferences in Excel that are then used by VBA with results outputted back into Excel. Therefore, we need to understand how VBA communicates with Excel. The following code demonstrates several ways we can output results from VBA and into a single cell in the Excel spreadsheet.

```
1.      Sub WriteOut()
2.              ' You can write comments by using the apostrophe
3.              ' Notice that comments are written in the color green

4.              Cells(1, 1) = "Welcome MSQF Students!"
5.              Cells(2, 1) = "Good Luck!"
6.              Range("A3") = "and work hard!"

7.              Range("B5").Select
8.              ActiveCell.Value = 1
9.              ActiveCell.Offset(0, 1).Value = 2
10.             ActiveCell.Offset(1, 0) = 3
11.             ActiveCell.Offset(1, 1) = 4
12.      End Sub
```

(4-5) Cells(i,j) refers to the row i and column j of the active worksheet in the active workbook.

(7) Alternatively, we can use the more commonly used letter/number referencing system to indicate the desired cell using the Range function as demonstrated in line (7). I tend to find this a less desirable approach since I often want VBA to change the cell reference. We will see examples of dynamically changing cell references later.

(8-11) We can use the OFFSET(i,j) function to instruct VBA to move from the active cell, B5 in this case, i cells in the downward and j cells to the right. For example, line (10) references the active cell B5 and then moves one cell down and zero cell to the right. As a result, the value ‘3’ is place in cell B6.

Notice that in line (9) I use the Value command but not in lines (10) or (11). These are two different ways to do the same thing. You will find that there are many different ways to perform the same task.

DEFINING VARIABLES

Performing simple mathematical operations and outputting the results into Excel is straightforward.

```
Sub VariableNames()
     ' Adding in VBA and Outputting to Excel
    num1 = 3
    num2 = 5
    Range("b2").Value = num1
    Range("b3") = num2
    Cells(4,2) = num1 + num2
End Sub
```

Num1 and num2 are variable names that are assigned the values of 3 and 5, respectively. Unlike many other programming languages (e.g. Matlab), VBA is not case sensitive. Instead, VBA will automatically change all variable names to have the same appearance in terms of upper and lower case letters.

It is also worth noting that VBA functions all have both lower and uppercase lettering. VBA will automatically adjust the typed function to have the appropriate appearance as long as the function is spelled correctly.

TIP: A simple way to catch syntax errors due to misspelling VBA functions is to always type the functions in lower case letters. VBA will automatically capitalize certain letters only if the function is spelled correctly. Hence, if this does not happen, then you know immediately that you have mistyped the function name.

READING DATA FROM EXCEL

The previous example demonstrates how we can define variable names. Now we show how we can use VBA to take data from Excel, assign the data to a variable name, perform mathematical operations, and then output the results back into Excel.

First, we need to type data into Excel and save the current workbook. Do the following:

- *Input numbers into cells b8, b9, and b10.*
- *Save your workbook as 'VBA TUTORIAL'*

Note than when saving the workbook, you need to select 'Save as type: Excel Macro-Enabled Workbook'.

```
1.      Sub ReadData()

2.              Workbooks("VBA Tutorial.xlsm").Worksheets("sheet1").Select

3.              num1 = Range("b8")
4.              num2 = Cells(9,2)
5.              num3 = Range("b10")

6.              Range("d8") = num1+num3
7.              Range("d9") =2*num2
8.              Range("d10") = sqr(num3)
9.      End Sub
```

Again, this is a straightforward example. We can again use the RANGE and CELLS functions to reference the desired cells.

(2) VBA will always assume that the desired worksheet is the one most recently visited (the active sheet in the active book). I have accidently written over important data because VBA was interacting with the incorrect worksheet. It is a good habit to activate the precise workbook and worksheet in VBA at the top of the code. This line references sheet1 in workbook 'VBA Tutorial.xlms'.

(3-5) These lines instruct VBA to look into specific cells of the active worksheet and assigns the values in the cells to the variable names num1, num2, and num3.

(6-8) Mathematical operations are performed on the values and then the results are outputted into the active worksheet. Notice that in line (8) the SQR function is used. This is takes the square root of the value in parentheses. Recall, however, we use the SQRT function in Excel to perform the same operation. This demonstrates some of the quirkiness with using VBA.

TIP: It is a good habit to specifically select or activate the desired workbook and worksheet.

USING EXCEL WORKSHEET FUNCTIONS

Most Excel worksheet functions may be referenced and used in VBA. The following example requires numeric values to be typed into the cells b8, b9, and b10 of sheet 1 of the ‘VBA Tutorial.xlsm’ workbook.

```
1.     Sub WorksheetFunctions()
2.         Workbooks("VBA Tutorial.xlsm").Worksheets("sheet1").Select

3.         num1 = Range("b8")
4.         num2 = Range("b9")
5.         num3 = Range("b10")

6.         AVG = WorksheetFunction.Average(num1, num2, num3)
7.         Max = Application.WorksheetFunction.Max(num1, num2, num3)
8.         Min = Application.Min(num1, num2, num3)

9.         Cells(12, 1) = "Average = "
10.        Cells(13, 1) = "Max = "
11.        Cells(12, 2) = AVG
12.        Cells(13, 2) = Max
13.     End Sub
```

(6-8)   These three lines demonstrate different ways of referencing Excel worksheet functions.
(9-10) You may output text by simply surrounding the text with quotation marks.

SIMPLIFYING USING *WITH*

It may seem cumbersome having to type ‘Worksheetfunction’ in order to use the Excel functions. In cases when you need to reference multiple worksheet functions, we can use a WITH statement to simplify the code. Again, be sure to provide numeric values in cells b8, b9, and b10.

```
1.      Sub UsingWith()
2.              Workbooks("VBA Tutorial.xlsm").Worksheets("sheet1").Select

3.              num1 = Range("b8")
4.              num2 = Range("b9")
5.              num3 = Range("b10")

6.              With WorksheetFunction
7.                      AVG =.Average(num1, num2, num3)
8.                      Max = .Max(num1, num2, num3)
9.              End With

10.             Cells(12,1).Select

11.             With ActiveCell
12.                     .Offset(0, 0) = "Average = "
13.                     .Offset(1, 0) = "Max = "
14.                     .Offset(0, 1) = AVG
15.                     .Offset(1, 1) = Max
16.             End With
17.     End Sub
```

(6-9) Instead of typing ‘WORKSHEETFUNCTION.AVERAGE(…)’ and ‘WORKSHEETFUNCTION.MAX(…)’ we can simplify by using a WITH statement. Now, we can reference the worksheet functions by simply typing ‘.AVERAGE’ and ‘.MAX’. This simplification ends once we write ‘END WITH’.

(11-16) The WITH statement may be applied to any such repeated command. For example, instead of typing ‘ACTIVECELL.OFFSET’ several times, we can instead type ‘WITH ACTIVECELL’ and it is applied to all of the following OFFSET statements until the ‘END WITH’.

REFERENCING A RANGE OF VALUES

With financial modeling, we often need to reference a range of data (e.g. a list of stock returns). Here are several ways to references ranges.

```
1.    Sub Ranges()
2.        Workbooks("VBA Tutorial.xlsm").Worksheets("sheet1").Select

3.        Range("a1") = 100
4.        Range("c5:c10") = 200     ' range from c5 to c10
5.        Range("b3", "b7") = 300   'range from b3 to b7
6.        Range("d1:d10 a1:f1") = "Intersect"  'intersection of two sets
7.        Range("e1,e3,e5,e7,e10") = "union" 'union of sets
8.        Range(Cells(10, 10), Cells(15, 15)) = "square"
9.    End Sub
```

(3) Reference a range consisting of the single cell A1.
(4) This refers to the range starting at C5 and ending at C10 including all cells in between.
(5) This is an alternate syntax for defining the range from B3 to B7 and all cells in between.
(6) We can refer to the intersection of two ranges. This line outputs the word ‘Intersect’ to the single cell D1.
(7) Separating ranges or single cell references with a comma creates a union of the various referenced cells and ranges.
(8) This is the most flexible of the various approaches. When cells and ranges are referenced using numbers, we can then manipulate the referenced cells in VBA. This allows us to create a more dynamic spreadsheet. We will do this later.

COMMUNICATING WITH THE USER (Message Boxes)

There are times when it is beneficial to communicate with the user of the model. For example, if the user inputs incorrect values, then the user needs to be instructed as to what values are required. The two most common methods are Message Boxes and Input Boxes.

```
1.      Sub MessageBox()
2.              MsgBox "Welcome to the MSQF program!  (1) "
3.              X1 = MsgBox("Welcome to the MSQF program! (2) ", vbExclamation , "Fordham University")
4.              X2 = MsgBox("Do you like New York City?", vbYesNo , "Fordham University")
5.              MsgBox X2

6.              MyNumber = Sqr(1790)
7.              MyMessage = "The square root of 1790 is " & MyNumber & "."
8.              Answer = MsgBox(MyMessage, vbInformation, "Square Root Machine")
9.      End Sub
```

(2) Displays a simple message.

(3) Displays same message but adds the heading ‘Fordham University’ and a decoration, a large exclamation point. Other decorations are vbQuestion, vbCritical, and vbInformation.

(4) The default is to have a simple OK button on the message box. We can override the default to have a number of button combinations. The vbYesNo command places Yes and No buttons on the message box. If the YES button is pressed, then X2 will be assigned the value of 6, and if NO is pressed, X2 is assigned the value of 7. Other button combinations are: vbOKCancel, vbAbortRetryIgnore, vbYesNoCancel, and vbRetryCancel.

(6-8) Message boxes can be flexible to accept different messages and provide results of calculations. The ‘&’ sign is used to concatenate two messages. In this case, we are connecting three different pieces to form the entire message. The first part is “The square root of 1790 is “, the second part of the message is the answer represented by the variable name MyNumber, and third part is a period to indicate the end of the sentence.

COMMUNICATING WITH THE USER (Input Boxes)

Input boxes can be used to gather data from the user.

```
1.    Sub MyInputBox()
2.        MyNumber = InputBox("Input any non-negative number.", "Square Root Machine", 0, 10000, 5000)
3.        MyAnswer = Sqr(MyNumber)
4.        MyAnswer2 = Format(MyAnswer, "0.000")

5.        MyMessage = "The square root of " & MyNumber & " is " & MyAnswer2 & "."
6.        Answer = MsgBox(MyMessage, vbInformation, "Square Root Machine")
7.    End Sub
```

(2) The input box allows for a number of inputs. See the VBA help to learn more. This input box asks for a non-negative number. The box has the title ‘Square Root Machine’. The third input is the default value that will appear in the box – ‘0’. The next two inputs indicate the screen position for the top-left corner of the box. The variable name ‘MyNumber’ is assigned the value that the user enters into the input box.

(4) The Format command is used to indicate the number of decimal values to output. Without this statement, VBA will output 14 decimal values.

(5-6) The message to be outputted is created in line (5) and the message is outputted using a message box.

TIP: User input boxes only when there are one or two pieces of information being requested. If a model needs to be run multiple times, it is irritating to the user to have to continuously reenter the same information over again.

LOGICAL STATEMENTS

We are now getting to several topics that clearly show the power and advantages of VBA when used in conjunction with Excel. It is possible to create logical statements in Excel (using the IF function), but it can be clumsy to write complicated logical statements.

The example below demonstrates how If/Then statements can be used to validate data entered by the user. Validation is a common step to ensure that results are reasonable and as a result, prevents the code from crashing (e.g taking the square root of a negative value will cause problems in VBA).

```
1.      Sub IfThen()
2.              MyNumber = InputBox("Input any non-negative number.", "Square Root Machine", 0, 10000, 5000)

3.              If MyNumber < 0 Then
4.                      MsgBox "The number entered is less than zero."
5.                      Exit Sub
6.              End If

7.              MyAnswer = Sqr(MyNumber)
8.              MyAnswer2 = Format(MyAnswer, "0.000")
9.              MyMessage = "The square root of " & MyNumber & " is " & MyAnswer2 & "."
10.             Answer = MsgBox(MyMessage, vbInformation, "Square Root Machine")
11.     End Sub
```

(3-6) The IF statement begins be describing the condition that must be met. If the condition is met, then the lines of code between IF and END IF will execute. Otherwise, these lines of code will be skipped. If the inputted value is less than 0, then the message box appears. Once the user acknowledges the message box, the EXIT SUB statement causes the code to end immediately.

The following code is an extension of the previous one and introduces IF/THEN/ELSE statements.

```
1.      Sub IfThenElse()
2.              MyNumber = InputBox("Input any non-negative number.", "Square Root Machine", 0, 10000, 5000)

3.              If MyNumber < 0 Then
4.                      MsgBox "The number entered is less than zero."
5.                      Exit Sub
6.              End If

7.              MyAnswer = Sqr(MyNumber)
8.              MyAnswer2 = Format(MyAnswer, "0.000")
9.              MyMessage = "The square root of " & MyNumber & " is " & MyAnswer2 & "."
10.             Answer = MsgBox(MyMessage, vbYesNo + vbInformation, "Square Root Machine")

11.             If Answer = vbYes Then
12.                     MsgBox "Good! Have a nice day."
13.             Else
14.                     MsgBox "Sorry. The problem is too difficult."
15.             End If
16.
17.     End Sub
```

(10) Yes no buttons are added to the message box. The response is stored in the variable ‘Answer’.

(11-12) If the variable ‘Answer’ is Yes, then the message box appears wishing you a nice day. Alternatively, we could have written ‘IF ANSWER = 6 THEN’.

(13-14) Alternatively, if the user does not press the YES button, then the ELSE statement is true and the message box indicating the difficulty of the problem is displayed.

The following is an example of using IF/THEN/ELSE that allows for multiple conditions.

```
1.      Sub IfThenElse2()
2.              ActiveWorkbook.Worksheets(1).Cells.ClearContents
3.              Range("A1") = Rnd * 100
4.              Range("A2") = Rnd * 10

5.              MySelection = InputBox("Input 1 to add and 2 to subtract.")

6.              If MySelection = 1 Then
7.                      Cells(4, 1) = Cells(1, 1) + Cells(2, 1)
8.                      Cells(4, 2) = "Add"
9.              ElseIf MySelection = 2 Then
10.                     Cells(4, 1) = Cells(1, 1) - Cells(2, 1)
11.                     Cells(4, 2) = "Subtract”
12.             Else
13.                     Cells(4, 1) = "XXX"
14.                     Cells(4, 2) = "ERROR"
15.             End If
16.     End Sub
```

(2)	This line clears the contents in all the cells in the first worksheet of the active workbook. However, notice that this line does not actually select the first sheet of the active workbook. It is possible that the output of the code will appear in a different sheet than the one that was just cleared.
(3-4)	RND is picks a uniformly random variable over the interval 0 to 1. Multiplying by 100 changes the distribution to the interval 0 to 100.
(6-8)	If the user enters ‘1’ in the input box, the MySelection = 1 is true and the lines 7 and 8 will execute. After line 8 completes, the code exits the IF statement and resumes at line 16.
(9-11)	If MySelection =2 is true then lines 10 and 11 will be executed. The code will exit the IF statement and continue at line 16.
(12-14)	If MySelection does not equal 1 or 2, then the code will default to line 13 and 14.

TIP: Please note that CLEARCONTENTS and DELETE do two different things. CLEARCONTENTS deletes the values typed into the cell while DELETE removes the value plus, formatting, colors, etc. Hence, DELETE is a more powerful command.

LOGICAL STATEMENTS (SELECT CASE)

If the code requires many different possible actions that depend on the value of a variable, then SELECT CASE may be useful.

```
1.      Sub SelectCase()
2.              Age = Inputbox(“Enter your age”)
3.
4.              Select Case Age
5.              Case Is < 0
6.                      Msg = "Age cannot be less than zero."
7.              Case 0 to 16
8.                      Msg = "You are too young to drive."
9.              Case 16 to 100
10.                     Msg = "You are legally an adult and allowed to drive"
11.             Case is >100
12.                     Msg = “Wow! You are old!”
13.             End Select
14.
15.             MsgBox Msg
16.     End Sub
```

(4-13) The SELECT CASE begins by indicating the variable of concern and ends by stating END SELECT. Lines 5, 7, 9, and 11 indicate conditions that the variable Age must satisfy. Line 5 asks whether Age is less than zero. Line 7 checks whether Age is between 0 and 16. A similar condition is set for line 9, and line 11 checks if Age is greater than 100.

CREATING LOOPS (FOR/NEXT)

Loops make it possible to execute the same set of instructions multiple times. This is the feature that really adds significant power to VBA over Excel.

```
1.      Sub Loop1()
2.              ActiveWorkbook.Worksheets(2).Select
3.              ActiveSheet.Cells.ClearContents
4.
5.              For i = 1 To 100
6.                      Cells(i, 1) = i
7.                      Cells(i, 2) = i ^ 2
8.              Next i
9.      End Sub
```

(2) The second sheet of the active workbook is selected.

(3) In the selected sheet, all cells are selected and their contents cleared.

(5-8) The loop is initiated by defining the variable i and setting it to take the values 1 to 100 at intervals of 1. The variable I will first be set to 1. Lines 6 and 7 are executed. Line instructs VBA to go back to line 5 and let the variable i take the value of 2. These steps are repeated until i=100. When i=100, lines 6 and 7 will execute for a final time and then the code exists the loop and continues to line 9.

(6-7) These two lines output data into the active worksheet of the active workbook. Specifically, when i=1, the cell in the first row and first column will take the value of i=1. Next to it in row 1, column 2, the cell will be assigned the square of i. In the next iteration of the loop, i=2. The output will be placed in the first and second columns of the second row. When i=3, output will be placed in row 3, and so on. The use of the Cells command allows us to dynamically change the cells being referenced in each loop.

We can use loops to help with data validation.

```
Sub Loop2()
    ActiveWorkbook.Worksheets("sheet1").Select
    ActiveSheet.Cells.ClearContents

    For i = 1 To 10
        MyNumber = InputBox("Input number between 0 and 10", , 0)

        If MyNumber >= 1 And MyNumber <= 10 Then
            Exit For
        ElseIf MyNumber = "" Then
            Exit Sub
        Else
            MsgBox "Input is invalid"
            If i=10 then Exit Sub
        End If
    Next i

    Select Case i
        Case Is = 1
            msg = "You did it on the first try! Good Job!"
        Case 2 To 5
            msg = "Not bad"
        Case 6 To 9
            msg = "You are a slow learner."
        Case Is = 10
            msg = "You are not very smart!"
    End Select

    A = MsgBox("It took you " & i & " attempts." & vbCrLf & msg)
End Sub
```

(4-14) This is the data validation step. The FOR/NEXT statements are used to allow the user ten attempts to enter a number between 0 and 10.

(6-13) This is a set of conditional statements checking the validity of the data.

- (6) The IF statement checks whether the submitted value is between 0 and 10. If yes, then EXIT FOR is executed causing the loop to be terminated immediately. The code jumps from line 6 to line 11.
- (8) This condition is true if either of two possibilities arise. First, the cancel button assigns the variable MyNumber a value of “” … nothing. Hence, pressing the cancel button will cause this condition to be true. Second, notice that the input box has a default value of 0. If the user deletes the 0 and presses ok, the condition will also be true. In either case, the EXIT SUB command will cause the entire code to be terminated immediately.
- (10) In all other cases, the ELSE statement will be executed causing a message box to appear informing that the input was invalid. If i=10, then the user inputted values incorrectly 10 times. In this case, the Exit Sub command will execute causing the code to terminate immediately.
- (13) The code loops back to line 4 unless either (6) is true or (8) is true.

(15-24)This section provides instructions for various values of the variable i using SELECT CASE.

(25) This is a message outputted using a message box. The command VBCRLF is a carriage return. It forces the second part of the message to continue on the next line. It is the same as pressing the enter key in Microsoft Word, for example.

CREATING LOOPS (DO LOOP)

DO LOOPs are an alternate way to repeat lines of code. One caution, the careless use of DO loops can result in the code being stuck in an infinite loop. Be sure to include appropriate EXIT DO commands.

The following example is another example of data validation. The user is stuck in an infinite loop until an acceptable value is entered.

```
1.      Sub DoLoop1()
2.
3.              Do
4.                      X = InputBox("Enter a number between 0 and 10." & vbCrLf & "Press Cancel to end.")
5.                      If X = "" Then Exit Sub
6.                      If X >= 0 And X <= 10 Then Exit Do
7.              Loop
8.
9.              A = MsgBox("You entered the number " & X & ".")
10.     End Sub
```

(3-7) The loop begins with a DO command and ends with a LOOP statement. All of the code in between will be repeated an infinite number of times unless a line of code provides instructions to exit the loop using an EXIT DO statement. In this case, line 6 instructs VBA to exit the loop when the user enters a value between 0 and 10.

TIP: Data validation is an important element in financial modeling. It is generally a good practice to simply provide an error message and end the program than to force the user to provide correct information through the use of an infinite loop.

CREATING LOOPS (DO UNTIL / LOOP UNTIL)

DO UNTIL and LOOP UNTIL are variations of the DO LOOP. The loop will continue until a condition is satisfied. The following example places the UNTIL command with the LOOP statement. We can also place the UNTIL command with the DO statement.

```
1.      Sub DoLoop2()
2.              MyCounter = 0

3.              Do
4.                      X = InputBox("Enter a number between 0 and 10." & vbCrLf & "Press Cancel to end.")

5.                      If X = "" Then Exit Sub
6.                      If X >= 0 And X <= 10 Then
7.                              A = MsgBox("You entered the number " & X & ".")
8.                              Exit Do
9.                      End If

10.                     MyCounter = MyCounter + 1
11.             Loop Until MyCounter = 5

12.     End Sub
```

(2) The variable MyCounter is assigned the value of 0.

(4-11) The loop begins in line (4) with the DO statement and ends in line (14) with the LOOP UNTIL statement. The loop will continue until the variable MyCounter takes the value of 5.

(5) The code will immediately terminate when the user either leaves the input box empty or presses the Cancel button.

(6-9) When the use inputs a correct value, then the condition is true and lines 9 and 10 will execute. The message box will appear and the code will exit the loop and resume at line 16.

(10) The variable MyCounter starts is initially assigned the value of zero in line (2). Here, MyCounter increases in unit increments with each iteration of the loop.

(11) This line can be read as “continue to loop until the variable MyCounter takes the value of 5.” The code will either loop back up to line 3 or the loop will terminate allowing the code to continue to line (12). The latter occurs when MyCounter = 5.

USING ARRAYS

An array is a collection of objects with the same size and type (e.g. integers, strings, etc.). Elements of an array are indexed using integer values.

The following is a simple example of an array. The array, called MyArray, is one dimensional and stores five objects. Using a FOR/NEXT loop, we assign values to the array: MyArray(1) = 101, MyArray(2) = 102, …, MyArray(5) = 105.

```
1.    Sub Array1()
2.        Dim MyArray(1 To 5) As Integer

3.        For i = 1 To 5
4.            MyArray(i) = i + 100
5.        Next i

6.        MsgBox MyArray(3)
7.    End Sub
```

(2) Arrays must be declared at the beginning of the code and we must indicate the dimension of the array. The DIM statement is used to declare the array. The size of the array is established. There are a number of ways to do this in VBA. Here, we define the array to have five elements and the indexing begins with one and ends with five. Additionally, the type of data stored in the array is also declared - the array will hold integers.

(3-5) A loop is used to assign integer values to the array. The variable i ranges from 1 to 5. In line 4, when i=1, then MyArray(1)=101. When i=2, then MyArray(2)=102, and so on.

(6) A message box is used to output the value of the third element in the array, MyArray(3). This value should be 103.

TIP: The default base for arrays in VBA is zero. This means that the statement DIM MYARRAY(5) creates an array with 5 elements where indexing begins with 0 and ends with 4. We can change the base by specifically stating DIM MYARRAY(1 TO 5) thus instructing VBA to index the elements using integers 1 through 5. Or, we can type OPTION BASE 1 prior to any array declaration. This will change the base for all arrays in the module and therefore only needs to be used once in the module.

The following example creates a two-dimensional array. You can think of this as begin a grid with multiple rows and columns. In this case, the array has five rows and five columns … 25 elements in total. Values are assigned to the array using nested FOR/NEXT loops.

```
1.    Sub Array2()
2.        Dim NewArray(1 To 5, 1 To 5)

3.        For i = 1 To 5
4.            For j = 1 To 5
5.                NewArray(i, j) = i+j
6.            Next j
7.        Next i

8.        MsgBox NewArray(5, 2)
9.    End Sub
```

(2) The array NewArray is defined at the beginning of the code and the dimensions are established.

(3-7) Nested loops are used to assign values to the array. Notice that lines 3 and 7, the outer loop, describe the variable i while the inner loop, lines 4 and 6, describe the variable j.

(8) A message box is used to output the (5,2) element of the array.

ARRAYS WITH UNKNOWN DIMENSION

It is commonly the case that the dimension of the array is not known when the code is being written. Consider, for example, the case where we want to fill an array with the past N days of stock prices where we allow the user to determine N. In this case, we still need to declare the array at the beginning of the code, but the parentheses are left empty. Later in the code when N is known, the REDIM command is used to redimension the array.

Before running this code, type numeric values in column 1 of the spreadsheet starting with cell A1. (Note: usually the programmer will know where the data begins but not necessarily where the data ends.)

```
1.      Sub Array3()
2.              Dim MyArray()

3.              Worksheets("sheet1").Select
4.              Cells(1, 1).Select
5.              N = ActiveCell.CurrentRegion.Rows.Count

6.              ReDim MyArray(1 To N)

7.              For i = 1 To N
8.                      MyArray(i) = Cells(i, 1)
9.              Next i

10.             MySum = WorksheetFunction.Sum(MyArray)
11.             MsgBox MySum
12.     End Sub
```

(2) The array MyArray is declared. Since the dimension is unknown, the parentheses are empty.

(3-4) The worksheet holding the data is selected. It is known that the data is inserted beginning in cell A1. Therefore, we select cell A1.

(5) The number of data points must be determined. There are a number of ways to do this. Here, the CURRENTREGION function is used. This function is similar to highlighting a range in Excel by simultaneously holding the shift+control keys then pressing the arrow down and the arrow right keys. A range has both rows and columns. We instruct VBA to COUNT the number of ROWS in the CURRENTREGION that includes cell A1.

(6) Now that the number of rows in the dataset are known, we REDIM the array. This time we are able to define the array's dimensions.

(7-9) Using a FOR/NEXT loop that allows i to range from 1 to N, the data from the Excel spreadsheet in column one is stored into the array one element at a time. MyArray(1) is equal to the data point in cell A1.

(10) We can perform mathematical operations on the elements in the array. Here, the worksheet function SUM is used to sum all the elements in the array.

(11) The sum of the elements in the array are presented to the user in a message box.

TIP: CURRENTREGION refers to a range bounded by any combination of blank rows and blank columns. To use, any cell within the range must first be selected. Be careful, the current region will also include column headers and row labels.

MATRIX OPERATIONS ON RANGE OBJECTS

Matrix operations may be performed on range objects using the Excel's worksheet functions (MMULT, MINVERSE, TRANSPOSE, etc.). In this example, an NxN matrix is read into an array. The matrix is multiplied by its inverse and the result (the identity matrix) is outputted to Excel. Two checks are made. First, the code checks to ensure that the range is square. Second, the determinant is checked to ensure that the matrix is invertible.

What is the difference between a range and an array? The two are related. However, a range is a single object consisting of collection of cells. An array is a collection of objects. Therefore, in an array, it is possible to reference the individual elements. In a range, since there is only one object, individual cells may not be referenced.

Before running the following example, input data in an NxN range in Excel where the top left corner of the data is in cell b2.

```
1.      Sub Matrix()
2.              Dim myrange As Range     ‘input range
3.              Dim newrange As Range   ‘output range

4.              Set myrange = Range("b2").CurrentRegion
5.              col_num = myrange.Columns.Count
6.              row_num = myrange.Rows.Count

7.              If col_num <> row_num Then
8.                      MsgBox "The matrix is not square."
9.                      Exit Sub
10.             End If

11.             MyDeterm = WorksheetFunction.MDeterm(myrange)
12.             If Abs(MyDeterm) < 0.0001 Then
13.                     MsgBox "The matrix is not invertible."
14.                     Exit Sub
15.             End If

16.             Set newrange = Range(Cells(row_num + 5, 2), Cells(row_num + 5 + row_num - 1, 2 + col_num - 1))

17.             With WorksheetFunction
18.                     newrange.Value =.MMult(myrange,.MInverse(myrange))
19.             End With
20.     End Sub
```

(2-3) Two variables are dimensioned as being range objects. The range myrange will be the range where the data is stored. The range newrange will be the range where the result will be outputted.

(4) CURRENTRANGE is used to establish the boundaries of the data.

(5-6) These two lines determine the number of rows and number of columns of the range.

(7-10) Since we intend to take the inverse of the matrix, the matrix must be square. If the number of rows does not equal the number of columns, then a message box will appear describing the error and the code will terminate.

(11) The determinant of the matrix is calculated.

(12-15) If the determinant of a matrix is equal to zero, then the inverse does not exist. The IF/THEN statement checks whether the absolute value of the determinant is near zero. If so, an error message appears and the code terminates.

(16) The output range is determined and given the variable name newrange. The output range is defined to be 5 cells below the input range.

(17-20) The worksheet functions MMULT and MINVERSE are used to perform the matrix multiplication. To save on some typing, a WITH statement is used. Otherwise, it would be necessary to write out WORKSHEETFUNCTION.MMULT and WORKSHEETFUNCTION.MINVERSE. The result of the matrix operation is outputted to the range defined by newrange.

TIP: It should also be noted that matrix operations may be performed on arrays as well as range objects.

WRITING ARRAY INTO A RANGE

There are several ways to output values in arrays into the spreadsheet. The most obvious way is to use loops to output each element of the array into the spreadsheet one by one. Alternatively, we can simply output the entire array into a range. The following is a classic example of a horserace between the two approaches that demonstrates the difference in speed between the two approaches.

```
1.      Sub OutputArray()
2.              Dim MyArray() As Single
3.              Dim myrange As Range

4.              Worksheets(1).Select
5.              Cells.Clear

6.              num1 =Val( InputBox("Number of rows"))
7.              num2 = Val(InputBox("Number of Columns"))
```

```
        ReDim MyArray(1 To num1, 1 To num2)

        'POPULATE THE ARRAY
        For i = 1 To num1
            For j = 1 To num2
                MyArray(i, j) = Rnd * 100
            Next j
        Next i

        'OUTPUT ARRAY ELEMENT BY ELEMENT
        BeginTime1 = Timer
        Application.ScreenUpdating = False
        For i = 1 To num1
            For j = 1 To num2
                Cells(i, j) = MyArray(i, j)
            Next j
        Next i
        msg1 = "Method 1: " & Format(Timer - BeginTime1, "00.00") & " seconds"
        ' END OF ELEMENT BY ELEMENT APPROACH

        Worksheets(2).Select
        Cells.Clear

        'OUTPUT ARRAY USING RANGE
        BeginTime2 = Timer
        Set myrange = Range(Cells(1, 1), Cells(num1, num2))
        myrange.Value = MyArray
        msg2 = "Method 2: " & Format(Timer - BeginTime2, "00.00") & " seconds"
        'END OF RANGE APPROACH

        Application.ScreenUpdating = True
        MsgBox msg1 & vbCrLf & msg2
    End Sub
```

(2-3) MyRange and MyArray are declared. The dimensions of the array and the actual coordinates of the range are not yet known and will be defined later.

(6-7) Input boxes are used to ask for the number of rows and columns of the two dimensional array. Notice that the VAL function is used with the input boxes. This function changes a string to a numeric. By default, values entered into input boxes are defined as being a string. In some cases, this causes problems when using the value entered later in the code. To see this, delete the VAL function and rerun the code.

(8) The REDIM function is used to define the dimensions of the array.

(9-14) Nested loops are used to populate the array with random values.

(15-24)The values of the array are outputted into sheet 1 of the spreadsheet one element at a time using nested loops.

(16) The Timer command is used to record the start time of the element by element approach.

(23) The Timer command is used again to record the stop time. The difference between the start and stop times records the speed of the method.

(25-26)Worksheet 2 is selected and cleared to prepare for the range method for outputting a range.

(27-32)The values of the array are outputted all at one time into a range. The range is defined in line (29). The time is recorded using lines (28) and (31).

(34) A message box displays the time required to output the array for each method.

FREEZING THE SCREEN

Whenever VBA interacts with Excel by placing or deleting values in a sheet, selecting ranges, changing sheets, etc, all of the actions are visible to the user. This can be distracting to the user, but more importantly, the speed of the code is greatly hindered. We can prevent the screen from updating during the running of the code by using the SCREENUPDATING function.

The example demonstrates the difference in execution time between the cases of allowing the screen to updating and then preventing the screen from updating. Nested loops are used to populate a row in the first, second and third sheet and then looping back to populate the next row of the first three sheets, and so on. With screen updating, the user will observe the rows being populated and the sheets being activated and deactivated. Run this code from the spreadsheet and not from VBA.

```
Sub ScreenUpdate()
    BeginTime = Timer
    For i = 1 To 100
        For j = 1 To 3
            For k = 1 To 100
                Worksheets(j).Activate
                Cells(i, k) = Rnd * 100
            Next k
        Next j
    Next i
    msg1 = "With updating: " & Format(Timer - BeginTime, "00.00") & " seconds"
    MsgBox "Method 1 is complete. Press ok to reset for method 2."

    For i = 1 To 3
        Worksheets(i).Activate
        Cells.Clear
    Next i

    MsgBox "Method 2 begins"
    Application.ScreenUpdating = False
    BeginTime = Timer
    For i = 1 To 100
        For j = 1 To 3
            For k = 1 To 100
                Worksheets(j).Activate
                Cells(i, k) = Rnd * 100
            Next k
        Next j
    Next i
    msg2 = "Without updating: " & Format(Timer - BeginTime, "00.00") & " seconds"
    Application.ScreenUpdating = True

    MsgBox msg1 & vbCrLf & msg2
```

```
31.             For i = 1 To 3
32.                     Worksheets(i).Activate
33.                     Cells.Clear
34.             Next i
35.     End Sub
```

(2-12) Nested loops are used to toggle through the cells and worksheets. The time required to run these lines is record.
(13-16) The worksheets are cleared.
(17-29) Nested loop are used again to toggle through the various cells and worksheets. This time, however, the SCREENUPDATING function is used to stop the screen from updating in line (18) and then to allow the screen to update in line (29).
(30) The result of the horserace is provided in a message box.
(31-33) The cells in worksheets 1, 2, and 3 are cleared.

DECLARING VARIABLES

Variable declarations are generally optional in VBA (with some exceptions such as with arrays). Declaring a variable essentially specifies its data type … e.g. what type of data can be assigned to the variable. For example, a variable may be declared to take only 2 values (such as True and False), or a variable may be declared to only take integer values.

There are many reasons why declaring variables is important. First, when speed is a concern, declaring variables helps VBA know how to allocate memory. Boolean variables take very little memory while long variables take a lot of memory. In VBA, the default data type is Variant which means that VBA will automatically adjust the data type to match the data. This takes up a lot of memory and can slow the code. Second, for complicated codes, declaring variables at the top of the code helps to keep track of all your variables.

To help create the habit of declaring, we can set VBA to require variable declarations. One way to do this is to type OPTION EXPLICIT at the very top of the module. Alternatively, you can select TOOLS, OPTIONS and check ‘Require Variable Declarations.’

Variable types are: Byte (integers 0-255), Boolean(True(-1) or False(0)), Integer(-32,768 to 32,767), Long(integers ranging -2,147,483,648 to 2,147,483,647), Currency, Single (single precision – 32 bits), Double (double precision – 64 bits), Date, String (characters and not numeric data), Object name (array, range, worksheet, workbook, chart, etc), and others.

Examples:
```
DIM MyVariable As String
DIM TF as Boolean
DIM int1 As Integer, int2 As Integer, int3 As Long
DIM int1, int2, int3            ‘all variables here are given the data type Variant by default
DIM MyWrkbk AS Workbook
```

It is possible to create your own user-defined types. This is related to more serious object oriented programming and will not be covered here.

CALLING SUBROUTINES FROM SUBROUTINES

There are generally two ways to writing a code to perform a task. The first is to simplify write a very long code that runs from top to bottom. Alternatively, and this is the approach used with object-oriented programming, is to write a series of subroutines with each subroutine having a specific purpose. In the latter approach, the main subroutine will call other subroutines as needed.

To illustrate the idea, we revisit the subroutine called Screenupdating(). In this code, we were running a horse race showing the time benefits of using the SCREENUPDATING function. In the process, we typed and ran the same lines of code twice. Additionally, we cleared the workbook at two different times in the same code. We can simplify the coding by creating a separate subroutine that will handle the task of running the nested loops that output values into the spreadsheet and another separate subroutine that handles the task of clearing the workbook. Three separate subroutines will be written

```
Sub ScreenUpdate()

        Call MyLoops(TheTime1)

        msg1 = "With updating: " & TheTime1 & " seconds"
        MsgBox "Method 1 is complete. Press ok to reset for method 2."

        Call MyClear

        MsgBox "Method 2 begins"
        Application.ScreenUpdating = False

        Call MyLoops(TheTime2)

        msg2 = "Without updating: " & TheTime2 & " seconds"
        Application.ScreenUpdating = True

        Call MyClear

        MsgBox msg1 & vbCrLf & msg2
End Sub
```

```
14.     Private Sub MyLoops(MyTime)
15.             BeginTime = Timer
16.             For i = 1 To 10
17.                     For j = 1 To 3
18.                             For k = 1 To 10
19.                                     Worksheets(j).Activate
20.                                     Cells(i, k) = Rnd * 100
21.                             Next k
22.                     Next j
23.             Next i
24.             MyTime = Format(Timer - BeginTime, "00.00")
25.     End Sub

26.     Private Sub MyClear()
27.             For i = 1 To 3
28.                     Worksheets(i).Activate
29.                     Cells.Clear
30.             Next i
31.     End Sub
```

(2) The CALL function is used to run the sub routine called MyLoops. The subroutine MyLoops allows for one variable to be passed between the two routines. The value of the variable is the time it takes for the MyLoops subroutine to run. That is, the value of MyTime is determined in the MyLoops subroutine but is then used in the ScreenUpdate subroutine in line 3.

(5) The CALL function is used to run the MyClear subroutine. This subroutine simply clears all the data from the first three worksheets of the active workbook.

(8) The MyLoops subroutine is called a second time and the execution time is passed between the two subroutines.

(11) The MyClear subroutine is called a second time.

PUBLIC VERSUS PRIVATE

You will inevitably run into sample codes that begin their subroutines with PUBLIC SUB or PRIVATE SUB. What does this mean? First, SUB and PUBLIC SUB are the same. Defining a macro as public means that it is accessible by all modules and to the Excel user. A private sub is one that is only accessible to other macros within the same module and is generally unavailable to the Excel user. The private subroutine will not appear in the list of macros in Excel. Generally speaking, the private sub usually works in conjunction with another subroutine and performs a task that only makes sense in the context of another subroutine.

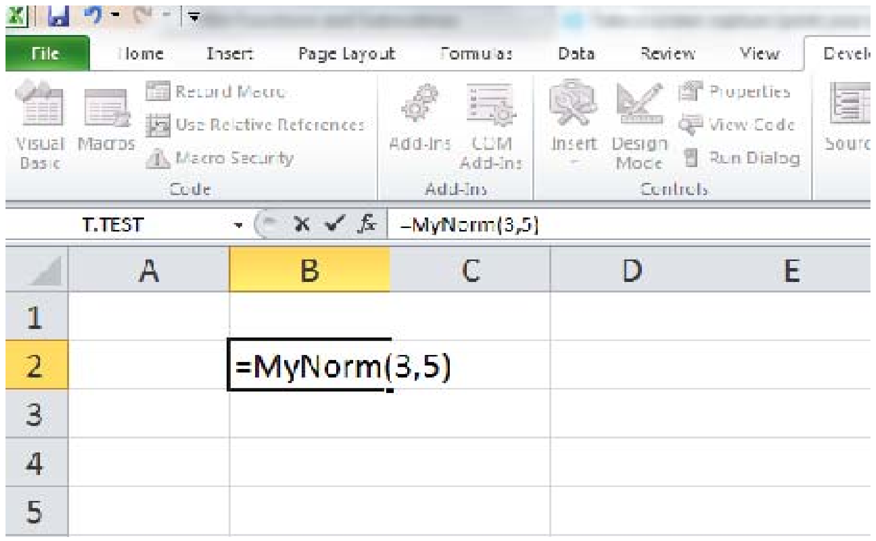
USER-DEFINED FUNCTIONS

Functions and subroutines are different in several ways. Subroutines are run by selecting the sub from a list of macros. The subroutine will perform a set of tasks that may or may not return a result. Functions, however, are used by typing the function name into a particular cell and providing the appropriate arguments. Functions always return a result. The result can be either a single value or string, or it can be an array.

To write a function, we begin by declaring that a function (and not a subroutine) is being created. The name of the function with the list of required arguments follows.

The following are a list of simple examples. Remember, functions are used by typing them into a particular cell. Do not use the green arrow key on the VBA toolbar.

```
Function MyNorm(num1, num2)
        MyNorm = Sqr(num1 ^ 2 + num2 ^ 2)
End Function
```



=MyNorm(3,5)

As seen in the figure to the right, the function MyNorm is entered directly into a cell exactly like all other worksheet functions. The functions has two arguments. When the enter key is pressed, the function returns the result. In this case, the value 5.83095 will appear in cell B2.

The above example requires the user to list two arguments. However, some functions allow the user to input a range of values. The previous function is modified to allow for a range.

```
1.      Function SumOfSq(MyRange)
2.              SS = 0

3.              For Each xxx In MyRange
4.                      SS = SS + xxx ^ 2
5.              Next xxx

6.              SumOfSq = Sqr(summing)
7.      End Function
```

(3-5) FOR EACH/NEXT is used to loop through all objects in a collection. Recall that a range is a collection of objects called cells. Then line (3) essentially tells VBA to perform the following set of instructions on each cell (xxx is the variable name that indicates the cell) in the range that is selected by the user. Here, xxx refers to both the cell (line 3) and the value in the cell (line 4). So, in line (4), the value in the cell is first squared and then added to SS (the sum of squares up to the particular cell in the loop).

(6) The name of the function is assigned the value of the result to be outputted. All functions must end with this type of statement: FunctionName = Result.

ARRAY FUNCTIONS

Excel provides a set of functions that output arrays (for example, MMULT, MINVERSE, etc.). To output the array properly, the user must select the range equal to the size of the output, type the function in the cell, and then press SHFT+CTRL+ENTER.

This function tests whether the mean of the data (MyRange) is statistically different from a given value (Center) using the standard t-test. The function outputs four values in a column: the number of data points, the average, standard deviation, and t-statistic. See figure below for an illustration.

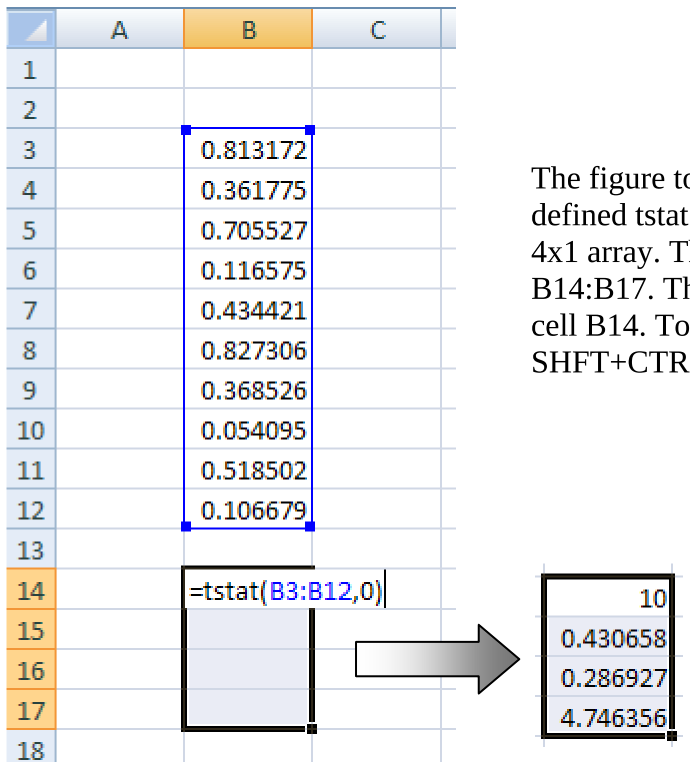
```
1.      Function Tstat(MyRange,Center)

2.              N=MyRange.Count

3.              With WorksheetFunction
4.                      Avg =.Average(myrange)
5.                      STD =.stdev(myrange)
6.                      T=(Avg-Center)/STD*Sqr(N)
7.                      Tstat =.Transpose(Array(N,Avg, STD,T))
8.              End With

9.      End Function
```

(2) The COUNT function is used to determine the number of populated cells in the range.
(3) The WITH/END WITH statement us used to simplify the code. As can be seen, three worksheet functions are used.
(4-5) The average and standard deviation of the data is calculated using the worksheet functions.
(6) The t-statistic is calculated.
(7) The function name is assigned the array of results consisting of the number of observations, the average, standard deviation, and the t-statistic. By default, the array created using the ARRAY function is a row and not a column. Therefore, the TRANSPOSE matrix operation is used to change the row array into a column array.



	A	B	C
1			
2			
3		0.813172	
4		0.361775	
5		0.705527	
6		0.116575	
7		0.434421	
8		0.827306	
9		0.368526	
10		0.054095	
11		0.518502	
12		0.106679	
13			
14		=tstat(B3:B12,0)	
15			
16			
17			
18			

10
0.430658
0.286927
4.746356

The figure to the left illustrates the use of the user-defined tstat function. The array function outputs a 4x1 array. Therefore, we start by highlighting cells B14:B17. The formula '=tstat(B3:B12,0)' is typed into cell B14. To output the array, press simultaneously SHFT+CTRL+ENTER.

DEBUGGING CODE

Finding mistakes in your code will probably be the most challenging part of building financial models. Here are a few tips:

1. The number one most common mistake is mistyping variable names. Check for typos. It is often helpful to dimension all variable names at the top of the code and to define variable names using a combination of upper and lower case letters. VBA will automatically capitalize letters of variable names to match the name in the DIM statement.
2. Be sure that arrays are indexed properly. By default, indexing begins with 0 (e.g. MyArray(5) holds five elements but the indexing ranges from 0 to 4 and not 1 to 5).
3. VBA debugger attempts to identify the line with the error. Sometimes it is correct, but many times, the error is in a completely different part of the code. Do not place all your trust in the VBA debugger.
4. You can use the immediate window to help debug your code. Be sure the immediate window is open (it is the long window below the module). If you do not see it, go to VIEW and select IMMEDIATE WINDOW or simply press CTRL+G. In strategic locations throughout your code, insert the line DEBUG.PRINT *variable_name*. The value of the variable will be outputted to the immediate window.
5. If you have no idea where the problem is located, as an alternative to the above tip, place at strategic locations throughout your code the lines DEBUG. PRINT 1, DEBUG.PRINT 2, DEBUG.PRINT 3, and so on. You will see the values 1, 2, 3, … in the immediate window. If you see the value 2 but not 3 in the immediate window, then you know there is a problem somewhere between the points in the code where you inserted the DEBUG statements for 2 and 3.
6. If you are still at a loss, you can use the ‘step into’ functionality in VBA by pressing F8. This functionality allows you to run one line at a time by continuously pressing F8. The line being executed will be highlighted in yellow.

PROBLEMS:

1. Write a VBA code that (1) computes the first N numbers of the Fibonacci sequence and (2) outputs the numbers in a column starting with the currently active cell (the cell that is highlighted when the macro is run). Use an input box to ask the user for the value N. The Fibonacci sequence is: X(1) = 1, X(2) = 1, X(n) = X(n-1)+X(n-2) … hence, X(3) = X(2)+X(1) = 1+1 = 2.

2. The user inputs N values into the A column of Excel beginning in cell A2. Write a VBA code that will multiply the column of data by its transpose to form a square NxN matrix. Output the matrix into Excel so that the top left corner is positioned in cell C2.

3. The user types a single value into some cell in some worksheet in the workbook. Write a VBA code that will find the single entry, cut (delete) the entry, and then copy it to cell A1 of sheet 1.

4. In mathematics, a norm is a function that calculates the length of a vector. One possible norm is the p-norm defined as $\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p}$ for some $p \geq 1$. Write a user-defined function that calculates the p-norm. The function should have two arguments. The first is a user selected range of values representing the vector, and the second is a value for p. The code should check to insure that $p \geq 1$.

5. Write a VBA code that places randomly generated numbers in a 20x20 grid in sheet 1. Without using the MAX worksheet function, the code should change the color of the cell containing the maximum to green and the cell containing the minimum value to the color red. (You will need to learn how to change cell colors.)

VBA FUNCTIONS/COMMANDS/METHODS USED IN THIS TUTORIAL

ACTIVATE	CURRENTREGION	FORMAT	RANGE	VALUE
ACTIVECELL	DEBUG	IF / THEN / ELSE	REDIM	VBCRLF
ACTIVESHEET	DELETE	INPUTBOX	ROWS	VBEXCLAMATION
ACTIVEWORKBOOK	DIM	LOOP UNTIL	SCREENUPDATING	VBINFORMATION
APPLICATION	DO LOOP	MAX	SELECT	VBYES
AVERAGE	DO UNTIL	MDETERM	SELECT CASE	VBYESNO
CALL	ELSEIF	MIN	SET	WITH / END WITH
CELLS	EXIT DO	MINVERSE	SQR	WORKBOOKS
CLEAR	EXIT FOR	MMULT	STDEV	WORKSHEETFUNCTION
CLEARCONTENTS	EXIT SUB	MSGBOX	SUM	WORKSHEETS
COLUMNS	FOR / NEXT	OFFSET	TIMER	
COUNT	FOR EACH / NEXT	PRINT	TRANSPOSE	