Notes on Visual Basic for Applications (VBA)

By Konstantinos N. Vonatsos

Table of Contents

INTRODUCTION

Introducing basic terms

Project: The project acts as a container for the modules, class modules, and forms for a particular file.

Module, class module, and form: These three elements act as containers for main programming elements such as procedures and functions. We will deal with modules and forms. Note that each of these elements requires a unique name. In a single project it is possible to have a number of different forms, modules, class modules.

Function and **Sub:** The Function and Sub elements hold individual lines of code. A function returns a value, whereas a Sub does not. You always use a Sub as the entry point for a program. A Sub is used to perform a task and not receive a direct return value. You can also use it to display information. A very important use is to utilize them to break up the code to smaller pieces each performing a specific task. This makes the code easier to read and handle. You use a Function when you want to return a value. Suppose that you need to calculate the Black-Scholes value several times in a program, you can then create a function that returns the price each time you call it.

Important compiler options

You add the following two options at the very beginning of a *module, class module,* or *form* before any other code.

Option base <Number>: Use this option to change how VBA numbers array elements. You can number array elements beginning at 0 or 1. Be careful the default value is 0.

Option Explicit: Always use this option! It tells VBA that you want to define all the variables before using them. It makes the code easier to read and helps you find typos in your code.

Introducing VBA objects

In EXCEL we have *objects.* Each EXCEL object represents a feature or a piece of functionality.

Examples of objects are: workbooks, worksheets, ranges, charts, font etc.

- Objects come in collections

  For example workbooks collection consists of all open workbooks. It is possible for an object to be a singular object (a collection of only one member). For example the Font object for any cell in the spreadsheet.

  Singular objects are referenced directly eg. **Font.**
  Individual objects in collections are referenced either by number (**Worksheets(1).**) or by name (**Worksheets("Sheet 1").**).
  The Range object although a singular object is referenced in a way similar to that of a collection.
  Note that a Cell is not an object on its own it is a Range object.

- Objects are arranged in hierarchy

```
ActiveWorkbook.Sheets("Test1").Range("A1:C10")
Workbooks("Lecture1").Sheets("Example1").Range("A1:C10")
```

- Objects have properties

  Properties are the attributes of an object, the values or settings that describe the object. An object's properties determine how it looks, how it behaves and whether it is visible.

```
Application.ScreenUpdating = False
Range("A2")="Input Value"
Range("A3")= 3000
Volatility=Range("A2").Value
Volatility=Cells(1,2).Value
```

- Objects have methods

  A method is an action you perform with an object. A method can change an object's properties or make the object do something. Notice that because a collection is also an object, collections also have methods.

```
Sheets("Sheet1").Range("A1:B3").Select
ActiveRange.Copy
Sheets("Sheet2").Range("A1").PasteSpecial
Workbooks("Model.xls").Activate
Sheets("Sheets1").Delete
```

- Objects can handle events

To do anything meaningful with an object you must do one of two things:

(a) Read or modify an object's properties

(b) Specify a method of action to be used with an object

INTRODUCING VARIOUS PROGRAMMING ELEMENTS

1. **Open** Excel. Go to Tools **à** Macro **à** Security and **Select** Medium Level.
2. **Start** a new workbook.
3. Press **Alt+F11** or go to Tools **à** Macro **à** Visual Basic Editor to activate VBE.
4. In VBE press Ctrl+R to open Project Explorer.
5. Select the new workbook's name in the Project Explorer window.
6. Choose Insert **à** Module to introduce a VBA module into the project

Type the following lines of code into the module:

Introducing Subroutines

Example

We create a subroutine that to copies a range of cells from ActiveSheet to "sheet2".

```
Option Explicit
 'We must declare all variables
Sub CopyAll()
Range("A1").CurrentRegion.Copy
Sheets("Sheet2").Range("A1").PasteSpecial
End Sub
```

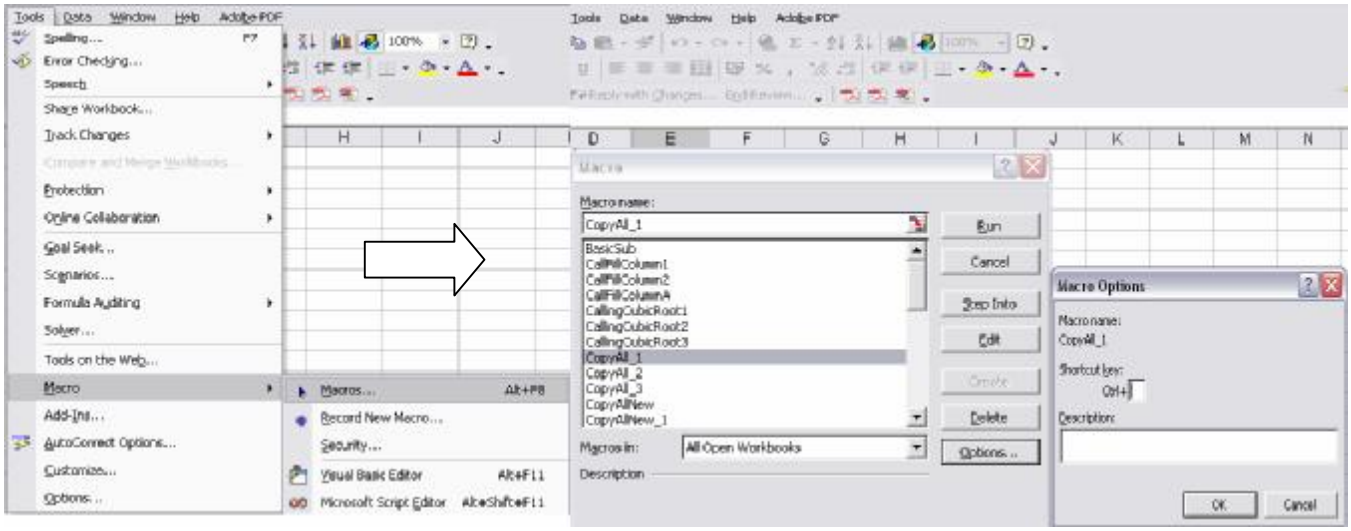
Note the use of objects. We start by using `Range("A1")` object and we use `CurrentRegion` Method. `CurrentRegion` returns a range object representing a set of contiguous data. As long as the data is surrounded by one empty row and one empty column, you can select the table with this method.

At this point to execute the subroutine move the cursor inside the subroutine and press **F5** or select **Run à Run Sub/User Form**.

You could also press **Alt+F11** or **Tools à Macro à Macros** and select CopyAll.

You can also run it by assigning a shortcut key as follows:

1. Tools **à** Macro**à** Macros (see figure below on the left)
2. Select the subroutine in the list box that appears (see figure below on the right)
3. Click the options buttons.
4. Click the Shortcut key option and enter a letter in the box next to Ctrl+. Note that if you insert small *c* then you press Ctrl+c to run the subroutine, whereas if you insert capital C you must press Ctrl+Shift+c.
5. Click ok and then exit from the Macro dialog box.



Assigning a Short-Cut key

Change in `CopyAll()` subroutine

```
Range("A1").CurrentRegion.Copy
```

To

```
Cells(1,1).CurrentRegion.Copy
```

or to

```
ActiveCell.CurrentRegion.Copy
```

The first and the second case produce the same result. The third case copies the `CurrentRegion` around the `ActiveCell` (i.e. the selected cell). Run the subroutine for all three cases and observe what happens.

Introduce the following subroutine:

```
Sub BasicSub()
     Call CopyAll
End Sub
```

Executing `BasicSub()`, calls and executes subroutine `CopyAll()`.

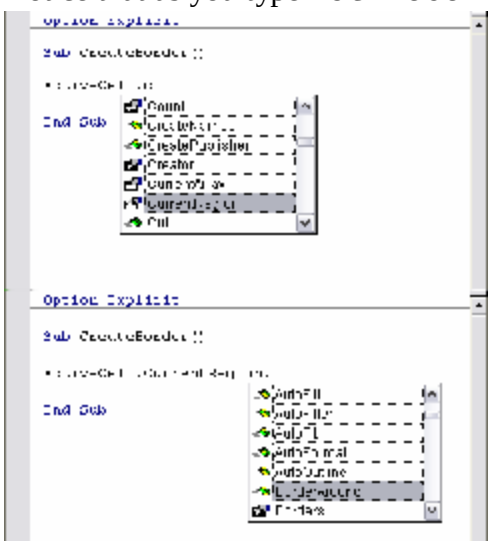
Now let us make the working environment in VB Editor more user-friendly. Choose `Tools` **à** `Options`. Select the `Editor` Tab and check the following options:

- *Auto Syntax Check:* If there is a syntax error while entering your code, VBE alerts you.
- *Auto List Members:* It provides as you type a list with objects/actions that would logically complete the statement you are typing.
- *Auto Quick Info:* VBE displays information regarding the syntax and the arguments of a function as you type it.
- *Default to Full Module View:* It presents all the functions and procedures in a module in a single scrollable list.

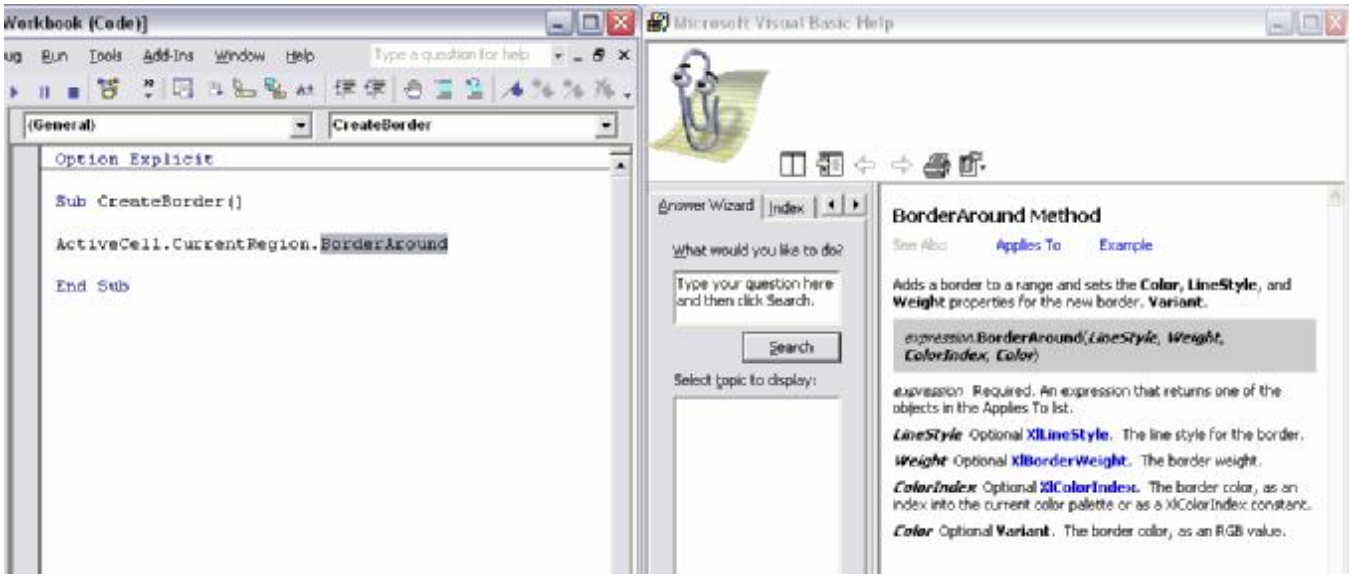
Let us try to create a subroutine that creates a border around a set of contiguous data. As before we will use the range object `CurrentRegion`.

```
Sub CreateBorder()
     ActiveCell.CurrentRegion.BorderAround _
          LineStyle:=xlDot, Weight:=xlThick, Color:=RGB(255, 0, 0)
End Sub
```

Notice that as you type `ActiveCell.` a box appears with all the possible object members that you can



use. Scroll down and select `CurrentRegion`. After you type the "." the pop-up window appears again and as you scroll down you find the `BorderAround` Method. Unfortunately, you do not know how to use the `BorderAround` Method. Select the word `BorderAround` and press *F1*. Excel help appears with a detailed description of the method and an example of how to use it. Compare the example from Excel help, with the Syntax of `BorderAround` Method and the example you have been provided with.



In a module you can select any object, method or procedure and after pressing F1 Excel help appears.

Another option you have in order to browse through the objects available to you is the Object Browser. You can access it by either pressing *F2* in VBE or by selecting `View`**`à`**`Object Browser`. At this stage object browser will not be as helpful as the online Excel help. Usually, for our programming needs we do not need to know a large number of object, methods, and procedures.

In the next examples we present different ways to work with range objects. A range object can consist of many rows and columns.

```
Sub ProvideRowAndColumn()
Dim EndRow As Integer, EndColumn As Integer
Dim i As Integer, j As Integer

     EndRow = ActiveCell.CurrentRegion.Rows.Count
     EndColumn = ActiveCell.CurrentRegion.Columns.Count

For i = 1 To EndRow
    For j = 1 To EndColumn
          Cells(i, j).Value = i + j
    Next
Next
End Sub
```

Now suppose that we have the following table in Excel and we want to fill in the third column with the product of columns A and B

A	**B**	**C**
1	5	
2	4	
3	3	
4	2	
5	1	

```
Sub DoCalculation()
Dim EndRow As Integer, EndColumn As Integer
Dim InitialRow As Integer, InitialColumn As Integer
Dim i As Integer

    InitialRow = ActiveCell.Row
    InitialColumn = ActiveCell.Column

    EndRow = ActiveCell.CurrentRegion.Rows.Count
    EndColumn = ActiveCell.CurrentRegion.Columns.Count

For i = 1 To EndRow
      Cells(InitialRow + i - 1, InitialColumn + 2).Value =
      Cells(InitialRow + i - 1, InitialColumn).Value _
             * Cells(InitialRow + i - 1, InitialColumn + 1).Value
Next
End Sub
```

Select the upper left cell of the table and run the subroutine.

In most cases subroutines need to have arguments. Arguments can be single valued variables, arrays or strings. A subroutine has a fixed number of arguments, which are a combination of optional and required arguments.

In a new module introduce the following subroutine:

```
Sub FillColumnA(NoOfRows As Integer)
Dim i As Integer

For i = 1 To NoOfRows
    Cells(i, 1) = i
Next
End Sub
```

If you try to run this subroutine you will find out that you cannot; the reason being that you have to pass an argument. To run this subroutine we introduce the following subroutine:

```
Sub CallFillColumnA()
Dim NoOfRows As Integer

     NoOfRows = InputBox("Input the number of rows...")

    Call FillColumnA(NoOfRows)
End Sub
```

If you run `CallFillColumnA` it calls the subroutine `FillColumnA` and passes the required argument. Now let us introduce an optional argument. Optional arguments should always be of the variant data type (you could introduce also different data types but it is not recommended). If you introduce an optional argument all subsequent arguments in the argument list must also be optional. To declare an optional argument we use the word optional before the argument. In the subroutine you must check whether an optional argument has been assigned a value and if not you must assign it its default value.

```
Sub FillColumn(NoOfRows As Integer, Optional NoOfColumn As Variant)
Dim i As Integer
'Check whether the optional argument is used

     If IsMissing(NoOfColumn) Then NoOfColumn = 1
```

```
     For i = 1 To NoOfRows
          Cells(i, NoOfColumn) = i
     Next
End Sub
```

To run the `FillColumn` subroutine introduce:

```
Sub CallFillColumn1()
    Call FillColumn(10)
End Sub
Sub CallFillColumn2()
    Call FillColumn(10, 2)
End Sub
```

The `CallFillColumn1` subroutine calls `FillColumn` and passes only the required argument. In `FillColumn` subroutine as the optional argument is missing it is assigned the default value 1. In contrast in `CallFillColumn2` both arguments are passed.

Introducing Functions

The use of arguments in functions is similar to their use in subroutines. We can have optional and required arguments which are defined in exactly the same way as in subroutines. Remember functions always return a value.

Example

Create a function that returns the cubic root of a number.

```
Function CubicRoot(Number As Double) As Double
'This function returns the cubic root
CubicRoot = Number ^ (1 / 3)
End Function
```

You can execute a function by

A. calling it from another function or subroutine
B. using the function in a worksheet formula
    - Select a cell in "sheet1" and type "`+cubicroot(8)`" and press enter. You should have in the cell the return value 2.
    - Another way to call the function is by choosing at the main Excel menu `Insert`**`à`**`functions`. In the box select category you choose `User Defined Functions` and double click on `CubicRoot`.
    - Type in the module

```
Sub CallingCubicRoot()
     MsgBox CubicRoot(8)
End Sub
```

Run `CallingCubitRoot.` Change in `CallingCubitRoot` subroutine `MsgBoX CubicRoot(8)` To `Cells(1,1).Value=CubicRoot(8)` Or to `Cells(2,1).Value=CubicRoot(Cells(1,1).Value).`

Note that in this last statement you read a value from A1 cell and return its cubicroot at A2 cell.

Introducing Variables, Comments, Constants and Data Types

When naming a variable keep in mind that:

- The first character should be a letter
- You cannot use any spaces or periods in a variable name
- VBA does not distinguish between uppercase and lowercase letters. So InterestRate is the same as interestrate and INTERESTRATE.
- You are not allowed to use the following characters: #, $, %, &, !.
- The names can be no longer than 254 characters

When you name a variable write it in a readable and easy to understand format. Use either mixed case format such as TimeToMaturity or use the underscore character Time_To_Maturity.

There are a number of reserved words that cannot be used for variable names or procedure names. For example you cannot use built-in function names such as Ucase, Sqr, Exp, etc or language words such as if, then , sub, with, for, etc.

Data Type	**Bytes Used**	**Range of Values**
Boolean	2	True or False
Integer	2	-32,768 to 32,767
Long	4	-2,147,483,648 to 2,147,483,648
Single	4	-3.402823 E38 to 1.401298 E45
Double (negative)	8	-1,7976931346232 E308 to -4.94065645841247 E-324
Double (positive)	8	4.94065645841247 E-324 to 1,7976931346232 E308
Currency	8	-922,337,203,685,477.5808 to 922,337,203,685,477.5807
Date	8	1/1/100 to 12/31/9999
String	1 per character	Varies
Variant	Varies	Varies

Always choose the data type that uses the smallest number of bytes but can manage with all the data the program assigns to it. We will see in what follows that we must be very careful when we use Excel built-in functions to perform complicated computational task. Note that the more bytes you reserve for a calculation the slower it is performed.

You should always include the following statement as the first statement in a VBA module: **Option Explicit**. This forces you to declare all the variables you use. To ensure that this option is always present go in the Visual Basic Editor, select `Tools`**à**`Options` and tick `Require Variable Declaration` option.

When defining a variable we must also decide the *scope* of the variable. That means that we must decide which modules and procedures can use a variable.

Available in a Single Procedure only: We use a **Dim** or **Static** statement in the procedure that uses the variable. This type of variables can only be used in the procedure in which they are declared.

Example:
```
Dim i as integer, j as integer
    Static NoOfIterations as integer
```

Be careful using: `Dim i, j as integer` declares *j* as integer and *i* as variant.

Note that a `Static` variable retains its value even when the procedure ends, so if you come back to the procedure the value is not reset. This value though is not available to other procedures. You can use it, for example, to keep track of the number of times you execute a subroutine.

Available in a particular Module only: We use a **Dim** statement before the first sub or function statement in the module. This type of variable is available to all procedures in a module. Therefore it can be used in all procedures and it will retain its value from one procedure to another.

Available in all procedures in all modules: We use a **Public** statement before the first sub or function in a module. This type of variable is available in all the VBA module in a workbook.

Example:
```
Public InterestRate as Double
```

To declare a constant we use the following syntax:

```
Const pi as Double= 3.14159265359
```

We can make a constant available to a particular procedure, module or in all procedures in all modules in exactly the same way as we do it for any variable.
Example: `Public Const pi as Double= 3.14159265359`

In many cases you will need to use strings to use and manipulate text. When dealing with strings you can introduce either a *fixed-length string* or a *variable-length string*.

- A fixed-length string is declared as follows:
  `Dim NewString As String *10`
  In this example we specify NewString as a string with maximum 10 characters. The maximum number of characters a fixed-length string can hold is 65,526.
- A variable-length string is declared as follows:
  `Dim NewString As String`
  NewString has an unspecified number of characters and theoretically can hold up to two billion characters. In terms of memory efficient it is recommended to use fixed-length strings.

In computational applications the most important element of a computer language is the array (matrix). We declare an array in the same way we declare any variable. We can declare an array as follows:

```
Dim Underlying(1 to 100) As Double
Dim Underlying(100) As Double
Dim Underlying() As Double
```

Note that the first and second declaration can result in different arrays. Visual Basic assumes by default that the lower index is 0. Therefore the second declaration is equivalent to:

```
Dim Underlying(0 to 100) As Double
```

If we want to force VBA to use 1 as the lower index we must include just below the `Option Explicit` statement, `Option Base 1`. In that case the following two statements are equivalent:

```
Dim Underlying(1 to 100) As Double
Dim Underlying(100) As Double
```

Using the statement `Dim Underlying() As Double`, declares a dynamic array. In order to use this array we must define its size using:

```
ReDim Underlying(100)
```

We can use `ReDim` as many times and as often as we need. Note that every time we use `ReDim` we clear all the values stored in the array. If we need to keep the 100 values we have found, and also need to increase the size of the array to 200 elements we use the following statement:

```
ReDim Preserve Underlying(200)
```

In this case you keep the first 100 elements intact and you have room for 100 more.

In all codes it is a good practice to introduce comments. Use comments whenever and wherever you think that you need them. A good comment should be easy to understand and should help clarify how the code works. VBA treats as a comment anything on a line that follows the apostrophe ‘. A good practice for bits of code that you do not need is to comment them out instead of completely deleting them. Furthermore, try to make your code more readable. If you type all the statement for your program one after the other you have a cramped result that is difficult to read. The code still works, but makes it difficult to handle. Leave white space between particular statements or steps in a function or subroutine. Furthermore, use indention to make the main part of a subroutine or function clearer.

Introducing Labels and the On Error statement

In some cases in VBA you will need to use labels. Labels are introduced if you plan to use the GoTo statement. A label begins with a character and ends with a colon.

Example

```
Sub CopyAllNew()
On Error GoTo ErrorHandle

    Sheets("Sheet1").Range("A1").CurrentRegion.Copy
    Sheets("Sheet4").Range("A1").PasteSpecial
    MsgBox "Data Successfully Pasted"

    Exit Sub
ErrorHandle:
    MsgBox "Cannot find Sheet4"
End Sub
```

In this subroutine we introduced a number of new elements. First of all we introduced the `On Error` statement, which bypasses Excel default error handling and uses our own error handling code.

Run the previous subroutine with the `On Error GoTo ErrorHandle` statement and without and observe the difference.

Suppose now that we want instead of simply having a `MsgBox` informing us that "Sheet4" does not exist to be able to add "sheet4" and redo the process. We have no idea of what sort of commands to use, but there is help at hand. We can record a macro a see the code produced to get an idea of the commands we need. Note that the macro recorder creates only subroutines (not functions). The macro recorder is suitable for straightforward tasks such as formatting, copying, creating a graph etc. You cannot use it to introduce conditional actions, iterations etc.

To use the macro recorder go to `Tools` **à** `Macro` **à** `Record New Macro.`

While recording select in Excel Insert **à** WorkSheet then right-click on the new WorkSheet and select rename. Type in a new name and then stop recording.

Go to VB Editor and check under module 1 the code inserted. It should look something like the following code:

```
Sub Macro1()
    Sheets.Add
    Sheets("Sheet4").Select
    Sheets("Sheet4").Name = "New"
End Sub
```

Based on the recorded macro, we introduce the following subroutine:

```
Sub CopyAllNew_1()
        On Error GoTo ErrorHandle

        Sheets("Sheet1").Range("A1").CurrentRegion.Copy
        Sheets("NewSheet").Range("A1").PasteSpecial
        MsgBox "Data Successfully Pasted"

        Exit Sub
ErrorHandle:
        Sheets.Add
        ActiveSheet.Name = "NewSheet"
        Resume
End Sub
```

First of all we must be extra careful so that the `ErrorHandle` label is reached only if an error occurs. Use `Exit Sub` or `Exit Function` just before the error handling part of your code. After we have taken care of the error we ask the subroutine to `Resume`.

The `resume statement` resumes execution after an error-handling routine is finished:

`Resume:`	Execution resumes with the statement that caused the error.
`Resume Next:`	Execution resumes with the statement immediately following the statement that caused the error.
`Resume label:`	Execution resumes at the specified label.

Remove worksheet “NewSheet” and introduce the following subroutine:

```
Sub CopyAllNew_2()
    On Error GoTo ErrorHandle

        Sheets("Sheet1").Range("A1").CurrentRegion.Copy
        Sheets("NewSheet").Range("A1").PasteSpecial
        MsgBox "Data Successfully Pasted"

        Exit Sub
ErrorHandle:
        Sheets.Add
        ActiveSheet.Name = "NewSheet"
        Resume Next
End Sub
```

Notice that when you run this last subroutine while the new worksheet is added the data are not pasted in the “NewSheet”. The error is caused by the statement:

```
Sheets("NewSheet").Range("A1").PasteSpecial
```

After the error-handling we resume at the statement immediately following this statement. That is why we get the message regarding the successful completion. In the error handling part of the code we could have also used `Resume Label.` Consider the following subroutine:

```
Sub SquareRoot_1()
Dim Number As Double, Answer As VbMsgBoxResult
Start:
    On Error GoTo ErrorHandle
        Number = InputBox("Provide a number")
        MsgBox "The square root is " & Str(Sqr(Number))

        Exit Sub
ErrorHandle:
        Answer = MsgBox("An error occured. Do you want to try
                 again?", vbYesNo)
        If Answer = vbYes Then Resume Start
End Sub
```

This implementation is more cumbersome and should be used only if after the error handling process you have to move to a different point in the procedure. The specific subroutine represents a poor example of handling errors. The use of a large number of labels can make a code difficult to understand and more importantly difficult to alter and correct if problems arise. Consider instead the following subroutine:

```
Sub SquareRoot_2()
Dim Number As Double, Answer As VbMsgBoxResult

    On Error GoTo ErrorHandle
        Number = InputBox("Provide a number")
        MsgBox "The square root is " & Str(Sqr(Number))

        Exit Sub
ErrorHandle:
       Answer = MsgBox("An error occured. Do you want to try again?",
                    vbYesNo)
       If Answer = vbYes Then
            Number = InputBox("Provide a number")
            Resume
        End If
End Sub
```

In this subroutine we resolve the error and resume from the statement that the error occurred. Let us try to understand what resume does in the error handling process. Run the `SquareRoot_1` subroutine as follows: Type -2, select Yes, Type -2, Select Yes, Type 2.

Now, change in `SquareRoot_1` the statement

```
If Answer = vbYes Then Resume Start
```

To

```
If Answer = vbYes Then GoTo Start
```

And run the subroutine again as follows: Type -2, select Yes, Type -2, Select Yes, Type 2.

You will see that your error-handling process cannot understand that an error occurs as we have not cleared the original error condition. When we use Resume the error condition clears!

On Error statements:

On Error GoTo Label :	If an error occurs you go to the specified lined by the label.
On Error Resume Next:	If an error occurs it is ignored and you move to the next line of code, clearing the error conditon.

A good way to avoid problems with the use of your code is to identify the possible sources of run-time errors. If you write codes that you only use, you might think that error handling is unnecessary. It is true that if you have written a code of which you are the sole user and an error occurs you can try and correct it. Error handling though is very important if you expect other people to also use your code. In that case you should make certain that proper messages appear that would help them to overcome the problem. In the following subroutine we do exactly that.

```
Sub SquareRoot()
Dim Number As Variant, Answer As VbMsgBoxResult

Start:
    Number = InputBox("Please, provide a number:")

        If IsNumeric(Number) = False Then
              Answer = MsgBox("No numeric input! The input should be
                       a positive number. Do you want to try again?",
                       vbYesNo)
                If Answer = vbYes Then
                    GoTo Start
                Else
                    Exit Sub
                End If
        ElseIf Number < 0 Then
               Answer = MsgBox("Negative number! The input should be
                        a positive number. Do you want to try
                        again?", vbYesNo)
                If Answer = vbYes Then
```

```
                GoTo Start
            Else
                Exit Sub
            End If
        End If

    MsgBox "The square root is " & Str(Sqr(Number))

End Sub
```

Introducing VBA and Excel functions

Excel and VBA provide you with a number of different functions that you can use in you VBA programs. Extra care is needed though when we use such functions to make certain that we fully understand how these functions should be used.

Commonly Used VBA Built-in functions

`Abs`:	Returns the absolute value of a number
`Atan`:	Returns the arctangent of a number
`CInt:`	Converts a numeric or string expression into an integer
`CDbl:`	Converts a numeric or string expression into a double
`CLng:`	Converts a numeric or string expression into a long
`Cos`:	Returns the cosine of a number
`Exp`:	Returns the base of the natural logarithm (e) raised to a power
`InputBox`:	Displays a box to prompt a user for an input
`Int`:	Returns the integer part of a number
`IsArray`:	Returns True if a variable is an array
`IsEmpty`:	Returns True if a variable has been initialized
`IsNull`:	Returns True if an expression contains no valid data
`IsNumeric`:	Returns True if an expression can be evaluated as a number
`LCase`:	Returns a string converted to lowercase
`Log:`	Returns the natural logarithm (Ln) of a number
`LTrim:`	Strips the leading spaces from a string
`MsgBox:`	Displays a modal message box
`Rnd:`	Returns a random number between 0 and 1
`RTrim:`	Strips the trailing spaces from a string
`Sin:`	Returns the sine of a number
`Str:`	Returns the string representation of a number
`Sqr:`	Returns the square root of a number
`Tan:`	Returns the tangent of a number
`Trim:`	Returns a string without leading or trailing spaces
`UCase:`	Returns a string converted to uppercase
`Val:`	Returns the numbers contained in a string

In VBA you are able to use all the worksheet function available in Excel. In order to do so you must type `Application.WorksheetFunction.TheFunctionYouWantToUse`

Let us create in a new module a subroutine that finds the maximum value in a column array.

```
Option Explicit

Sub HowToUseExcelFunctions()
Dim NoOfPoints As Integer, MyArray() As Double
Dim i As Integer, LargestNumber As Double

     NoOfPoints = Val(InputBox("Define the size of the matrix"))

ReDim MyArray(NoOfPoints)

     For i = 1 To NoOfPoints
          MyArray(i) = i
          ActiveSheet.Cells(i,1)=i
     Next

     LargestNumber = Application.WorksheetFunction.Max(MyArray)
     MsgBox ("The largest number is " & Str(LargestNumber))
End Sub
```

Run the `HowToUseExcelFunctions()` subroutine for:

(a) `NoOfPoints` =20

(b) `NoOfPoints` =32,766

(c) `NoOfPoints` =32,767

Notice that you get an overflow error message in the third case because you have declared `i` and `NoOfPoints as Integers`. Declare them as `Long`.

Run the `HowToUseExcelFunctions()` subroutine for:

(a) `NoOfPoints` =20

(b) `NoOfPoints` =65,536

(c) `NoOfPoints` =65,537

If you scroll down in excel up to the last row you will see that it ends at 65,536 (=2^{16}). Select the last row and try to insert a new row. Notice that no new row is inserted. All functions that deal with ranges can accept a maximum of 65,536 elements for a specific column.

Whenever you use a built-in function be certain that you fully understand how it works. Let us create a subroutine that finds the average value in a column array.

```
Option Explicit

Sub FindAverage()
Dim NoOfPoints As Integer, MyArray() As Double
Dim i As Integer, Average As Double

     NoOfPoints = Val(InputBox("Define the size of the matrix"))

ReDim MyArray(1, NoOfPoints)

     For i = 1 To NoOfPoints
          MyArray(1, i) = 1
     Next

     Average = Application.WorksheetFunction.Average(MyArray)
     MsgBox ("The average is " & Str(Average))
End Sub
```

Run the `HowToUseExcelFunctions ()` subroutine for:

(a) `NoOfPoints` =20

(b) `NoOfPoints` =50,000

You would expect to find average 1 but you do not. The problem is that you introduce an array `MyArray(1, NoOfPoints)` but the base is 0 in Excel by default. That means that you have

introduced not a column matrix but a (2 x `NoOfPoints`) matrix and the first columns when it is initialized is set equal to 0. Introduce `Option Base 1` and rerun the code. Note that if you only change `MyArray(1, NoOfPoints)` to `MyArray(NoOfPoints)` you still don't get average equal to 1 as `MyArray(0)=0`. Run the code with `Option Base 1` for `NoOfPoints = 65,536` and `65,537`. Notice that once more you have a problem. You should keep in mind these limitations of built-in Excel functions, especially when you deal with Monte-Carlo simulation and you need to calculate the mean.

Introducing control of the program flow

`GoTo:` jump to a particular statement (you have to introduce a label). Never use `GoTo` statement to do loops. Use a standard loop statement like "`for-next`"; such practice helps others to understand your intentions and keeps bugs like endless iterations to a minimum. Avoid using `GoTo` to exit a subroutine or a function, instead use `Exit Sub` or `Exit Function.` We discuss the `Exit` statement in more detail later in this section.

`If-Then statement`

The structure of the command is:

```
If [condition] Then [statement]
```

Example

```
Sub SelectOptionType()
Dim OptionType As String
     OptionType = InputBox("Call or Put option? ")

     If LCase(OptionType) = "call" Then MsgBox "You selected a call
     option"

     If LCase(OptionType) = "put" Then
          MsgBox "You selected a put option"
     End If
End Sub
```

Type in the inputbox:

(a) Call

(b) CALL

(c) Put

(d) American

In the code we convert the `OptionType` string to lowercase using the `LCase` function. That means that even if someone types `"CaLl"` we convert it to `"call"` and the comparison we make in the code is meaningful. As we will discuss later we have to introduce one more action if we want to be 100% certain before we compare the introduced string with the expected strings in our code. Note in case (d) that nothing happens and the code exits without message.

If-Then-Else statement

The structure of the command is:

```
If [condition] Then
    [statement 1]
Else
    [statement 2]
End if
```

Change the subroutine to be:

```
Sub SelectOptionType()
Dim OptionType As String
     OptionType = InputBox("Call or Put option? ")

     If LCase(OptionType) = "call" Then
          MsgBox "You selected a call option"
     Else
          MsgBox "You selected a put option"
     End If
End Sub
```

Type in the inputbox:

(a) Call

(b) Put

(c) American

Note in this case the wrong use of the `else` statement.

If-Then-ElseIf statement

The structure of the command is:

```
If [condition] Then
    [statement 1]
ElseIf
    [statement 2]
Else
    [statement 3]
End if
```

Now, change the subroutine in the following form:

```
Sub SelectOptionType()
Dim OptionType As String
     OptionType = InputBox("Call or Put option? ")

     If LCase(OptionType) = "call" Then
          MsgBox "You selected a call option"
     ElseIf LCase(OptionType) = "put" Then
          MsgBox "You selected a put option"
     Else
          MsgBox "Not a valid selection"
     End If
End Sub
```

Type in the inputbox:

(a) Call

(b) Put

(c) American

What do you observe?

Using Logical Operators

VBA has a number of logical operators that can be used. The most commonly used logical operators are the following:

`Not`: performs a logical negation on an expression

`And`: performs a logical conjunction on two expressions

`Or`: performs a logical disjunction on two expressions

```
Sub SelectOptionType ()
Dim OptionType As String, ExerciseRights As String

     OptionType = InputBox("Call or Put option? ")
     If LCase(OptionType) <> "call" And LCase(OptionType) <> "put"
     Then
          MsgBox "Not a valid selection"
          Exit Sub
     End If

     ExerciseRights = InputBox("American or European option? ")
     If LCase(ExerciseRights) <> "american" And LCase(ExerciseRights)
     <> "european" Then
          MsgBox "Not a valid selection"
          Exit Sub
     End If

     If LCase(OptionType) = "call" And LCase(ExerciseRights) =
     "american" Then
          MsgBox "You selected an American call option"
     ElseIf LCase(OptionType) = "call" And LCase(ExerciseRights) =
     "european" Then
          MsgBox "You selected a European Call option"
     ElseIf LCase(OptionType) = "put" And LCase(ExerciseRights) =
     "american" Then
          MsgBox "You selected a American Put option"
```

```
     ElseIf LCase(OptionType) = "put" And LCase(ExerciseRights) =
     "european" Then
          MsgBox "You selected a European Put option"
     End If
End Sub
```

In many cases users introduce by error strings with trailing or leading spaces. For example it is possible that a user mistakenly types "`Call `". Go back and check to see what happens. The program cannot recognize that the input was call. To avoid this type of errors you use the trim function.

Change:

`OptionType = InputBox("Call or Put option? ")` to

`OptionType = Trim(InputBox("Call or Put option? "))`

And

`ExerciseRights = InputBox("American or European option? ")` to

`ExerciseRights = Trim(InputBox("American or European option? "))`

Type `Call` or `Put`, `American` or `European` with leading and trailing spaces and see what happens.

Select Case statement

The structure of the command is:

```
          Select Case [expression]
               Case [condition 1]
                         [statements 1]
               Case [condition 2]
                         [statements 2]
               Case Else
                         [else statements]
          End Select
```

```
Sub SelectOptionTypeCase()
Dim OptionType As String, ExerciseRights As String

OptionType = Trim(InputBox("Call or Put option? "))

Select Case LCase(OptionType)
    Case "call"
     ExerciseRights = Trim(InputBox("American or European option? "))
         Select Case LCase(ExerciseRights)
             Case "american"
                    MsgBox "You selected an American call option"
             Case "european"
                    MsgBox "You selected an European call option"
             Case Else
                    MsgBox "Not a valid selection"
         End Select
    Case "put"
     ExerciseRights = Trim(InputBox("American or European option? "))
         Select Case ExerciseRights
             Case "american"
                    MsgBox "You selected an American put option"
             Case "european"
                    MsgBox "You selected an European put option"
             Case Else
                    MsgBox "Not a valid selection"
         End Select
    Case Else
                    MsgBox "Not a valid selection"
End Select

End Sub
```

In scientific computing in the various calculations that you have to perform there is a need to do iterations, or loop through a process until a certain conditions has been met. When designing a loop make certain that it will always end. In the following pages we deal with the following looping structures:

(A) `For-Next-Step`

(B) `Do-While`

(C) `Do-Until`

To understand how we can use these looping structures we will create a subroutine that solves a non-linear algebraic equation using the bisection method, which is based on the following theorem:

Suppose that f:[a,b] **à** *R is continuous and f(a)f(b)<0. Then for some c in (a,b), f(c)=0.*

`For-Next Statement`

```
Sub BisectionMethod_ForNext(UpperBound As Double, LowerBound As Double, TOLERANCE As Double, MaximumNumberOfIterations As Integer)

Dim i As Integer, MidPoint As Double

     'Check that the required conditions for the bisection method to
     be applied are satisfied
    If F(UpperBound) * F(LowerBound) > 0 Then
        MsgBox "Change the interval so that f(a)*f(b)<0"
        Exit Sub
    End If

    'Check whether the endpoints are roots
    If Abs(F(UpperBound)) < TOLERANCE Then
        MsgBox "F(" & Str(UpperBound) & ")<TOL. This is a root"
        Exit Sub
    End If
    If Abs(F(LowerBound)) < TOLERANCE Then
        MsgBox "F(" & Str(UpperBound) & ")<TOL. This is a root"
        Exit Sub
```

```
    End If

    'Begin iterations
    For i = 1 To MaximumNumberOfIterations

        MidPoint = (UpperBound + LowerBound) / 2

        'Check whether the midpoint is a root
        If Abs(F(MidPoint)) < TOLERANCE Then
                MsgBox "The root is " & Str(MidPoint)
                Exit Sub
        End If

        'Update the interval
        If F(UpperBound) * F(MidPoint) > 0 Then
            UpperBound = MidPoint
        Else
            LowerBound = MidPoint
        End If
    Next
    MsgBox "Maximum number of iterations was exceeded"

End Sub

Function F(x As Double) As Double
F = x ^ 3 - 2 * x - 1
End Function

Sub RunBisectionMethod_WithForNext()
    Call BisectionMethod_ForNext(2, 0, 10 ^ (-10), 1000)
End Sub
```

Note in the use of `For-Next` statement that the code ends with certainty if we reach the maximum number of iterations. If we want to iterate from i=100 to t=1 we should use the `Step` statement:

`For i=100 To 1 Step -1 … Next`

Do-While Statement

The bisection method can be implemented using the Do-While Loop as follows:

```
Sub BisectionMethod_DoWhile(UpperBound As Double, LowerBound As Double, TOLERANCE As Double, MaximumNumberOfIterations As Integer)

Dim i As Integer, NumberOfIterations As Integer, MidPoint As Double

     'Check that the required conditions for the bisection method to
     'be applied are satisfied
    If F(UpperBound) * F(LowerBound) > 0 Then
        MsgBox "Change the interval so that f(a)*f(b)<0"
        Exit Sub
    End If

    'Check whether the endpoints are roots
    If Abs(F(UpperBound)) < TOLERANCE Then
        MsgBox "F(" & Str(UpperBound) & ")<TOL. This is a root"
        Exit Sub
    End If

    If Abs(F(LowerBound)) < TOLERANCE Then
        MsgBox "F(" & Str(UpperBound) & ")<TOL. This is a root"
        Exit Sub
    End If

    NumberOfIterations = 0

    MidPoint = (UpperBound + LowerBound) / 2

    'Begin iterations
     Do While Abs(F(MidPoint)) > TOLERANCE And NumberOfIterations <=
     MaximumNumberOfIterations

        MidPoint = (UpperBound + LowerBound) / 2
```

```
        If F(UpperBound) * F(MidPoint) > 0 Then
            UpperBound = MidPoint
        Else
            LowerBound = MidPoint
        End If

        NumberOfIterations = NumberOfIterations + 1

    Loop

    If NumberOfIterations > MaximumNumberOfIterations Then
        MsgBox "Maximum number of iterations was exceeded"
        Exit Sub
    End If

        MsgBox "The root is " & Str(MidPoint)

End Sub

Sub RunBisectionMethod_DoUntil()
    Call BisectionMethod_DoWhile(2, 0, 10 ^ (-10), 1000)
End Sub
```

Note the difference in the way the bisection method is implemented in this case compared to the `For-Next` case. In both cases the iteration ends if one of two conditions is met. We stop if the maximum number of iterations is reached or if a root to a predetermined level of tolerance has been found. With the `For-Next` statement we iterate up to a maximum acceptable number of iteration. In each iteration we check whether a root has been found and end the iterations if the answer is affirmative. With the `Do-While` statement we iterate for as long as neither of the two conditions has been met and we exit the moment one of the two is satisfied. Outside the loop though we must find which condition was responsible for exiting the loop. The `Do-While` statement can be implemented in a way that is closer to the `For-Next` implementation.

```
Sub BisectionMethod_DoWhile_2(UpperBound As Double, LowerBound As Double, Tolerance As Double, MaximumNumberOfIterations As Integer)

Dim i As Integer, NumberOfIterations As Integer, MidPoint As Double

    'Check that the required conditions for the bisection method to
    'be applied are satisfied
    If F(UpperBound) * F(LowerBound) > 0 Then
       MsgBox "Change the interval so that f(a)*f(b)<0"
       Exit Sub
    End If

    'Check whether the endpoints are roots
    If Abs(F(UpperBound)) < Tolerance Then
       MsgBox "F(" & Str(UpperBound) & ")<TOL. This is a root"
       Exit Sub
    End If

    If Abs(F(LowerBound)) < Tolerance Then
       MsgBox "F(" & Str(UpperBound) & ")<TOL. This is a root"
       Exit Sub
    End If

    NumberOfIterations = 0

    MidPoint = (UpperBound + LowerBound) / 2

    'Begin iterations
    Do While Abs(F(MidPoint)) > Tolerance


       If NumberOfIterations > MaximumNumberOfIterations Then
           MsgBox "Maximum number of iterations was exceeded"
           Exit Sub
       End If
```

```
        MidPoint = (UpperBound + LowerBound) / 2

        If F(UpperBound) * F(MidPoint) > 0 Then
            UpperBound = MidPoint
        Else
            LowerBound = MidPoint
        End If

        NumberOfIterations = NumberOfIterations + 1

    Loop

        MsgBox "The root is " & Str(MidPoint)
End Sub
```

Do-Until Statement

The bisection method can be implemented using the `Do-Until Loop` as follows:

```
Sub BisectionMethod_DoUntil(UpperBound As Double, LowerBound As
Double, Tolerance As Double, MaximumNumberOfIterations As Integer)

Dim i As Integer, NumberOfIterations As Integer, MidPoint As Double

    'Check that the required conditions for the bisection method to
    'be applied are satisfied
    If F(UpperBound) * F(LowerBound) > 0 Then
        MsgBox "Change the interval so that f(a)*f(b)<0"
        Exit Sub
    End If

    'Check whether the endpoints are roots
    If Abs(F(UpperBound)) < Tolerance Then
        MsgBox "F(" & Str(UpperBound) & ")<TOL. This is a root"
```

```
        Exit Sub
    End If

    If Abs(F(LowerBound)) < Tolerance Then
        MsgBox "F(" & Str(UpperBound) & ")<TOL. This is a root"
        Exit Sub
    End If

    NumberOfIterations = 0

    MidPoint = (UpperBound + LowerBound) / 2

    'Begin iterations
    Do Until Abs(F(MidPoint)) < Tolerance Or NumberOfIterations >
MaximumNumberOfIterations

        MidPoint = (UpperBound + LowerBound) / 2

        If F(UpperBound) * F(MidPoint) > 0 Then
            UpperBound = MidPoint
        Else
            LowerBound = MidPoint
        End If

        NumberOfIterations = NumberOfIterations + 1

    Loop

    If NumberOfIterations > MaximumNumberOfIterations Then
      MsgBox "Maximum number of iterations was exceeded"
      Exit Sub
    End If

    MsgBox "The root is " & Str(MidPoint)
End Sub
```

Note how in the use of `Do-Until` statement we iterate up until either of the two conditions is met. As is the case for the `Do-While` Loop we can set only one condition in the `Do-Until` statement and check whether the other condition has been met during each iteration.

```
Sub BisectionMethod_DoUntil_2(UpperBound As Double, LowerBound As Double, Tolerance As Double,  MaximumNumberOfIterations As Integer)

Dim i As Integer, NumberOfIterations As Integer, MidPoint As Double

     'Check that the required conditions for the bisection method to
     'be applied are satisfied
     If F(UpperBound) * F(LowerBound) > 0 Then
        MsgBox "Change the interval so that f(a)*f(b)<0"
        Exit Sub
     End If

     'Check whether the endpoints are roots
     If Abs(F(UpperBound)) < Tolerance Then
        MsgBox "F(" & Str(UpperBound) & ")<TOL. This is a root"
        Exit Sub
     End If

     If Abs(F(LowerBound)) < Tolerance Then
        MsgBox "F(" & Str(UpperBound) & ")<TOL. This is a root"
        Exit Sub
     End If

     NumberOfIterations = 0

     MidPoint = (UpperBound + LowerBound) / 2

     'Begin iterations
     Do Until Abs(F(MidPoint)) < Tolerance

        If NumberOfIterations > MaximumNumberOfIterations Then
```

```
            MsgBox "Maximum number of iterations was exceeded"
            Exit Sub
        End If
            MidPoint = (UpperBound + LowerBound) / 2

        If F(UpperBound) * F(MidPoint) > 0 Then
            UpperBound = MidPoint
        Else
            LowerBound = MidPoint
        End If

        NumberOfIterations = NumberOfIterations + 1
    Loop

    MsgBox "The root is " & Str(MidPoint)

End Sub
```

A very useful statement is the `Exit` statement, which is used to exit a `Do-Loop`, a `For-Next`, a `Function` or a `Subroutine`:

`Exit Sub:` The code exits the specific subroutine the statement is contained and continues with the statement that follows the line of code that called the exited subroutine.

`Exit Function:` The code exits the specific function the statement is contained and continues with the statement that follows the line of code that called the exited function.

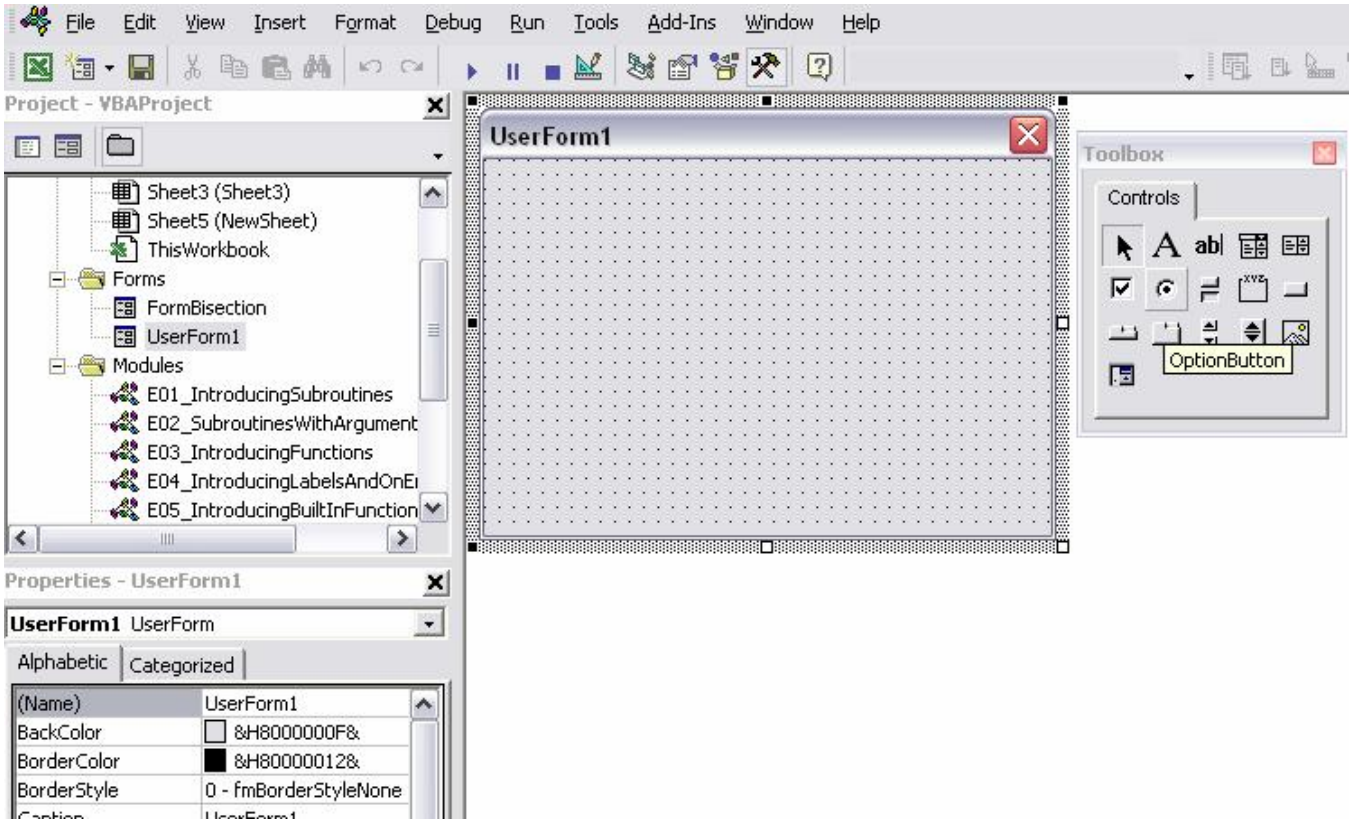
`Exit For:` The code exits a `For-Next` loop and control is transferred to the statement following the `Next` statement. In nested `For-Next` loops the control is transferred to the loop nested one level above the loop where the `Exit For` occurs.

`Exit Do:` The code exits a `Do-Loop` statement and control is transferred to the statement following the `Loop` statement. In nested `Do-Loops` the control is transferred to the loop nested one level above the loop where the `Exit Do` occurs.

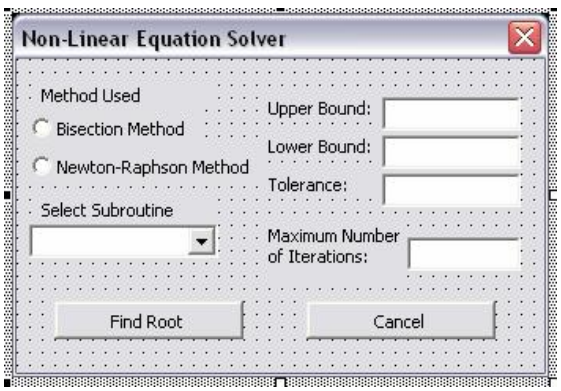
Note that the previous subroutines that use the bisection method to find a root of a non-linear equation could have been implemented as functions that would return the root.

Creating a Non-Linear Equation Solver

In this section we introduce a visual interface for the easier use of the non-linear equation solvers we have developed. The first step is to introduce a `UserForm` object. Choose in the VB Editor `Insert à UserForm.` Double clicking on the `UserForm` in the VBA project window produces the following screenshot. If the toolbox is not displayed choose `View à Toolbox`. Make certain that the `properties window` is visible (if it is not press `F4` or choose `View à Properties Window`). Using the properties window we can change the properties of the various objects we use in designing a `UserForm`.



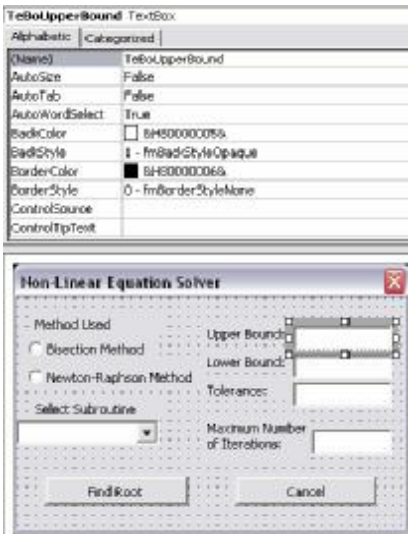
Snapshot of VBE with a new UserForm. The properties and toolbox windows are present.



Using the toolbox we create the `UserForm` shown on the left. If you click on the form in the properties window you will see that the Name of the form is `FormBisection` and the Caption has been changed to "Non-Linear Equation Solver". We have introduced four `TextBox` objects for the user to provide the upper bound, the lower bound, the tolerance and the maximum number of iterations. Each `TextBox` has on its left a `Label` that provides information on the entry required in the specific

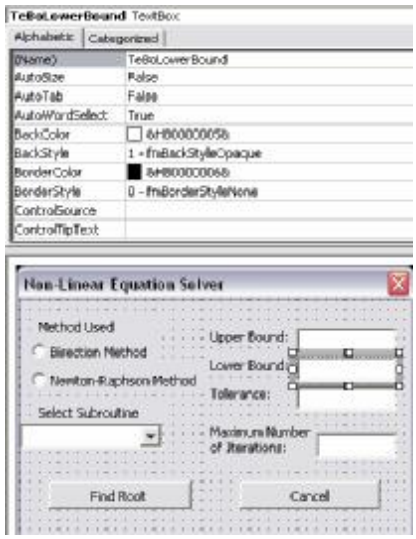
TextBox. If you click on the TextBox introduced for the Upper Bound you will see that its name has been set to TeBoUpperBound. We have also introduced two Option Buttons, one Combo Box and two Command Buttons. We use the option buttons to select the method that we want to use to solve the non-linear equation and the combo box to select based on the method we use which implementation to use i.e. For-Next, Do-While. The only reason for introducing this second choice is for showing you how the Combo Box object is used.

Below you can see for all objects on the form the respective properties.



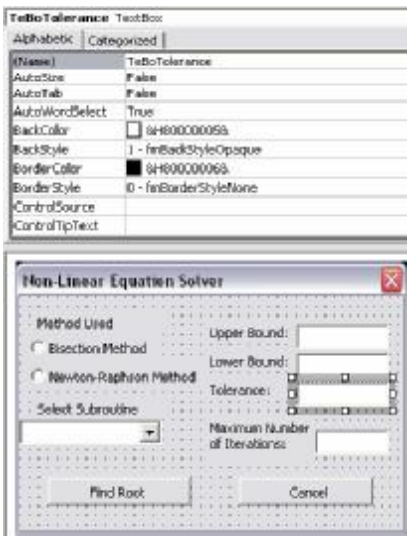
We have set:

Name: TeBoUpperBound



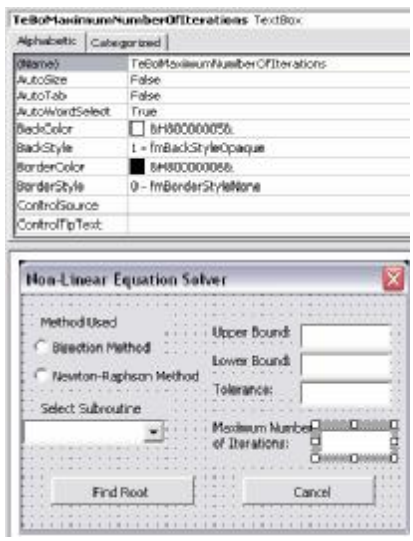
We have set:

Name: TeBoLowerBound



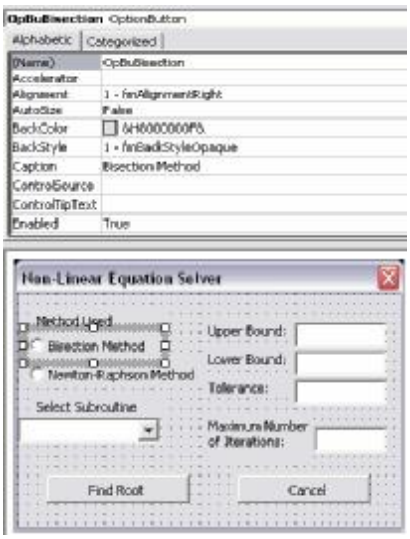
We have set:

Name: TeBoTolerance



We have set:

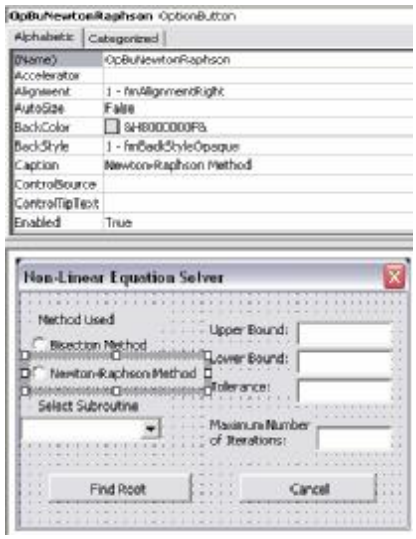
Name: TeBoMaximumNumberOfIterations



We have set:

Name: `OpBuBisection`

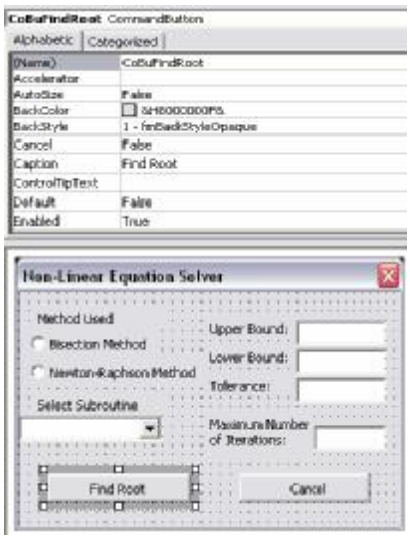
Caption: `Bisection Method`



We have set:

Name: `OpBuNewtonRaphson`

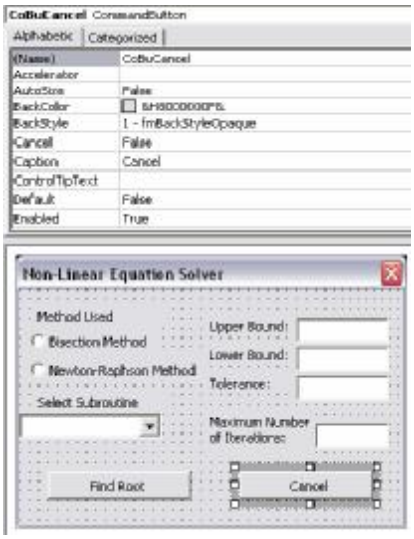
Caption: `Newton-Raphson Method`



We have set:

Name: `CoBuFindRoot`

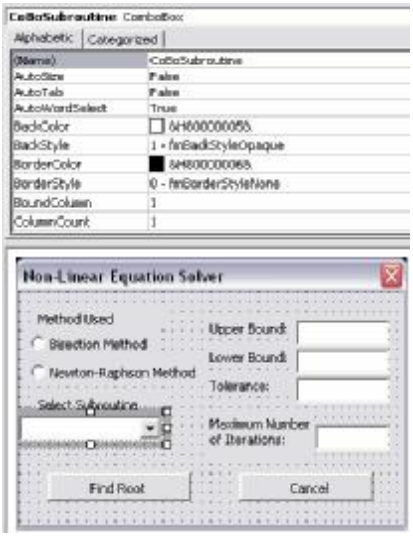
Caption: `Find Root`



We have set:

Name: `CoBuCancel`

Caption: `Cancel`



When we want to introduce a visual interface we must plan for an action that will make this interface available to the user. We insert a new module named `FormHandling` and in this module we enter the following subroutine:

```
Sub ShowFormBisection()
    Call InitializeFormBisection
    FormBisection.Show
End Sub
```

We have set:

Name: CoBoSubroutine

The `FormBisection.Show` command displays the form named `FormBisection`. Before we show the form we call a subroutine that initializes the form by introducing numerical values and the various available options in the objects we have inserted on the form.

```
Sub InitializeFormBisection()

 With FormBisection
       'Select the option button for the Bisection method
        .OpBuBisection.Value = True

        'Setup the Combo Box
        With .CoBoSubroutine
            .AddItem "Next-For Loop"
            .AddItem "Do-While Loop"
            .AddItem "Do-Until Loop"
        End With

        'Set the default value for the Combo Box
        .CoBoSubroutine.Value = "Next-For Loop"

        'Initialize the remaining inputs
```

```
        .TeBoLowerBound = -10
        .TeBoUpperBound = 10
        .TeBoTolerance = 10 ^ (-10)
        .TeBoMaximumNumberOfIterations = 1000
    End With
End Sub
```

In this subroutine we introduced the `With` statement. For more information on this statement select the `With` statement in VB editor and press `F1`.

Run the `ShowFormBisection` subroutine and see what happens. On the form that appears select the Newton-Raphson method. Notice that you are allowed to select only one of the option buttons. In the properties box of an `OptionButton` we find the `GroupName` property. `OptionButtons` with the same `GroupName` are associated and only one of them can be selected each time out of the particular group. Next, select the combo box. A drop-down menu appears from which you can select any of the three subroutines. For the time being though we have not introduced any event-handler subroutines, so our form cannot actually do something. Close the form and go back to the VB editor, select the form and double click on the `Find Root` button. At this point you should see:

```
Private Sub CoBuFindRoot_Click()

End Sub
```

The subroutine `CoBuFindRoot_Click()` is executed whenever the Find Root button is clicked. Type the following statements inside the subroutine:

```
Private Sub CoBuFindRoot_Click()
Dim UpperBound As Double, LowerBound As Double, Tolerance As Double
Dim MaximumNumberOfIterations As Integer

On Error GoTo ErrorHandling

    'Check that the entries are in the correct format
    If Trim(TeBoUpperBound.Value) = "" Then
```

```
        MsgBox "Problem with the upper bound value"
        Exit Sub
    End If

    If Trim(TeBoLowerBound.Value) = "" Then
        MsgBox "Problem with the lower bound value"
        Exit Sub
    End If


    If Trim(TeBoTolerance.Value) = "" Then
        MsgBox "Problem with the tolerance value"
        Exit Sub
    End If

    If Trim(TeBoMaximumNumberOfIterations.Value) = "" Then
        MsgBox "Problem with the maximum number of iterations value"
        Exit Sub
    End If

    'Read the required data
    UpperBound = Val(TeBoUpperBound.Value)
    LowerBound = Val(TeBoLowerBound.Value)
    Tolerance =  Val(TeBoTolerance.Value)
MaximumNumberOfIterations =CInt(TeBoMaximumNumberOfIterations.Value)


    If OpBuBisection.Value = True Then

        Select Case CoBoSubroutine.Value
            Case "Next-For Loop"
               Call BisectionMethod_ForNext(UpperBound, LowerBound, _
               Tolerance, MaximumNumberOfIterations)

            Case "Do-While Loop"
```

```
                Call BisectionMethod_DoWhile(UpperBound, LowerBound, _
                Tolerance, MaximumNumberOfIterations)

            Case "Do-Until Loop"
                Call BisectionMethod_DoUntil(UpperBound, LowerBound, _
                Tolerance, MaximumNumberOfIterations)
        End Select

    End If



    If OpBuNewtonRaphson.Value = True Then

        MsgBox "Under construction"
        Exit Sub

    End If

    Exit Sub
ErrorHandling:
    MsgBox "Check that all the entries are correct. There was an error."
End Sub
```

Finally, go back to the form and double click the `Cancel` button. Enter the following subroutine:

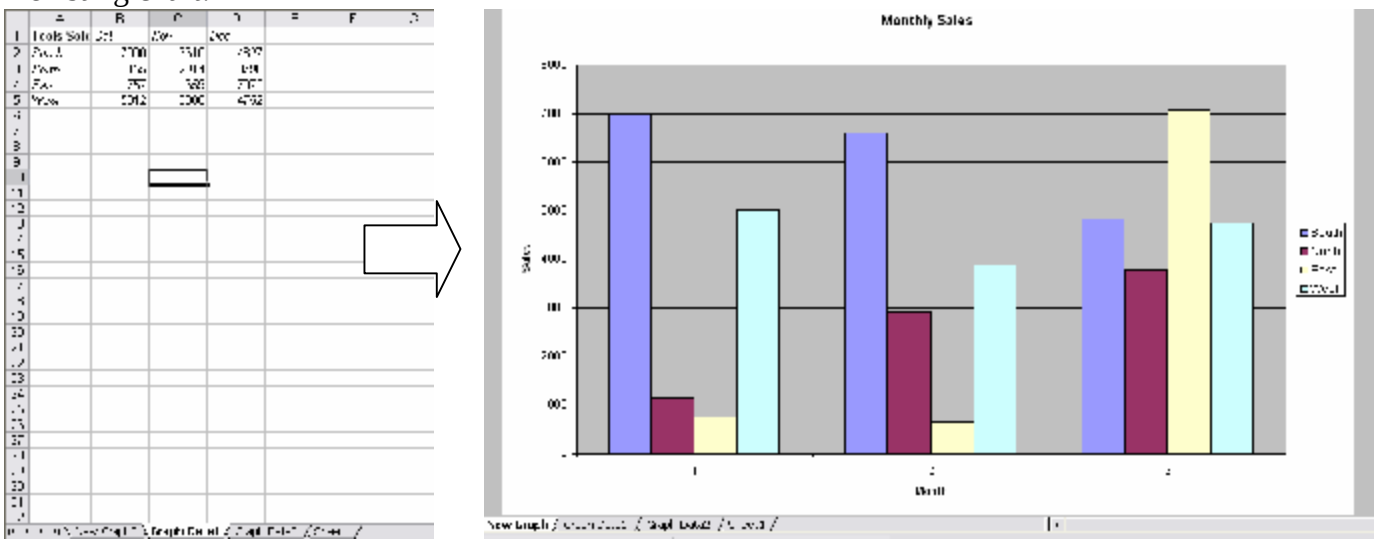
```
Private Sub CoBuCancel_Click()
    Unload FormBisection
End Sub
```

For a user to use our calculator he needs to call `ShowBisectionForm()`. An easy and neat way to call it is by assigning a short-cut key. Activate the Excel window and select `Tools` **à** `Macro` **à** `Macros`. In the menu that appears select `ShowBisectionForm` and click the

options button. In the dialog box that appears enter an uppercase S and click ok. Exit from the Macro dialog box. In the Excel window press `Ctrl+Shift+S` and your solver appears on the screen.

Introducing Charts

Using charts in VBA can be very confusing and we will not deal with them in detail. You will be provided with two subroutines that are quite helpful which you can alter to meet your needs. When dealing with charts it is a good idea to use the macro recorder to get an idea of the commands you need to use. Both subroutines read data from an Excel worksheet and either create a new chart or update an existing chart.



In worksheet `Graph Data1` we have the data seen in the figure above on the left. Suppose that we need to plot the sales in each region per month (figure on the right). Introduce the following subroutine:

```
Sub AddChartSheetXlColumnClustered(chtName As String, chtSheet As
String, chtRange As String, chtTitle As String, _
chtAxisTitleCategory As String, chtAxisTitleValue As String)
'Subroutine that updates a chart if it already exists or it creates a
new one

On Error GoTo ErrorHandling

    With Charts(chtName)
        .ChartType = xlColumnClustered
```

```
        'Link to source data range
        .SetSourceData Source:=Sheets(chtSheet).Range(chtRange), _
                              PlotBy:=xlRows
        .HasTitle = True
        .ChartTitle.Text = chtTitle
        .Axes(xlCategory, xlPrimary).HasTitle = True
        .Axes(xlCategory, xlPrimary).AxisTitle.Characters.Text = _
                    chtAxisTitleCategory
        .Axes(xlValue, xlPrimary).HasTitle = True
        .Axes(xlValue, xlPrimary).AxisTitle.Characters.Text = _
                    chtAxisTitleValue
        '.HasLegend = False
     End With

    Exit Sub

ErrorHandling:
    MsgBox "The referred chart does not exist. A new graph will be _
                introduced"
    Charts.Add
    ActiveChart.Name = chtName
    Resume
End Sub
```

To use this subroutine for plotting introduce the following subroutine:

```
Sub Main()

Call AddChartSheetXlColumnClustered("New Graph", "Graph Data1", _
      "A1:D5", "Monthly Sales", "Month", "Sales")

End Sub
```

Run the `Main()` subroutine and see what happens. Add on `Sheets("Graph Data1")` data for January and change in `Main()` subroutine "A1:D5" to "A1:E5" so that the new data are also plotted.

Note that if you go to `Sheets("Graph Data1")` and change the data the graph is automatically updated. Let us now automate the selection process.

Make in `AddChartSheetXlColumnClustered` subroutine the following changes:

Change

```
Sub AddChartSheetXlColumnClustered(chtName As String, chtSheet As String, chtRange As String, chtTitle As String, _
chtAxisTitleCategory As String, chtAxisTitleValue As String)
```

To

```
Sub AddChartSheetXlColumnClustered(chtName As String, chtSheet As String, chtRange As Range, chtTitle As String, _
chtAxisTitleCategory As String, chtAxisTitleValue As String)
```

Change

```
.SetSourceData Source:=Sheets(chtSheet).Range(chtRange), _
                                                    PlotBy:=xlRows
```

To

```
.SetSourceData Source:=chtRange, PlotBy:=xlRows
```

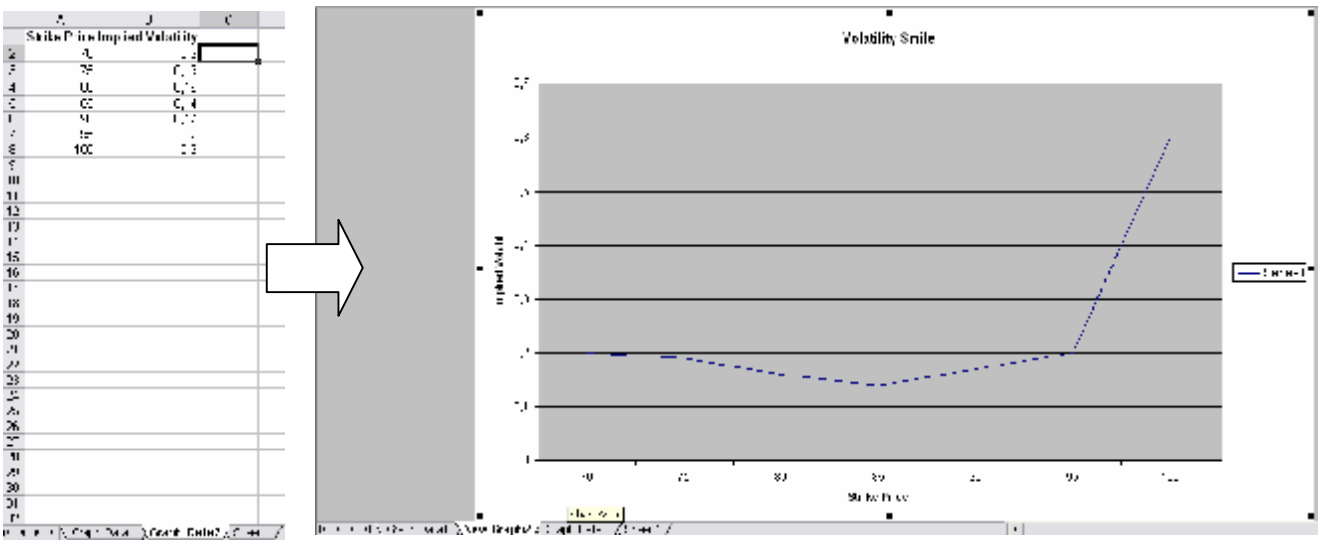
Change the `Main()` subroutine as follows:

```
Sub Main()
Dim chtRange As Range

Set chtRange = Sheets("Graph Data1").Range("A1").CurrentRegion

Call AddChartSheetXlColumnClustered ("New Graph", "Graph Data1", _
               chtRange, "Monthly Sales", "Month", "Sales")
End Sub
```

Add on `sheet("Graph Data1")` data for January, February and March and run this new `Main()` subroutine. The graph is automatically created.



In worksheet `Graph Data2` we have the data seen in the figure above on the left. Suppose that we need to plot the implied volatility against the strike (figure on the right). Introduce the following subroutine:

```
Sub AddChartSheetXlLine(chtName As String, chtSheet As String, _
 chtRangeX As String, chtRangeY As String, chtTitle As String, _
 chtXAxisTitle As String, chtYAxisTitle As String)

On Error GoTo ErrorHandling

    With Charts(chtName)
        .ChartType = xlLine

        'Link to source data range
        .SetSourceData Source:=Sheets(chtSheet).Range(chtRangeY), _
                    PlotBy:=xlColumns
        .SeriesCollection(1).XValues= _
                                Sheets(chtSheet).Range(chtRangeX)
        .HasTitle = True
        .ChartTitle.Text = chtTitle
        .Axes(xlCategory, xlPrimary).HasTitle = True
        .Axes(xlCategory, xlPrimary).AxisTitle.Characters.Text = _
                                                    chtXAxisTitle
        .Axes(xlValue, xlPrimary).HasTitle = True
```

```
        .Axes(xlValue, xlPrimary).AxisTitle.Characters.Text = _
                                                    chtYAxisTitle

    End With

    Exit Sub
ErrorHandling:
    MsgBox "The referred chart does not exist. A new graph will be _
 introduced"
    Charts.Add
    ActiveChart.Name = chtName
    Resume
End Sub

Sub Main ()
     Call AddChartSheetXlLine("New Graph2", "Graph Data2", "A1:A8", _
 "B1:B8", "Volatility Smile", "Strike Price", "Implied Volatility")
End Sub
```

Running the `Main()` subroutine produces the required graph. Suppose that we want to automate the process as well. We need to be able to find the last row in A and B columns. We will use two different ways with the second being the preferable one. First of all change in `AddChartSheetXlLine` subroutine:

Change:

```
Sub AddChartSheetXlLine(chtName As String, chtSheet As String, _
 chtRangeX As String, chtRangeY As String, chtTitle As String, _
 chtXAxisTitle As String, chtYAxisTitle As String)
```

To

```
Sub AddChartSheetXlLine_New(chtName As String, chtSheet As String, _
     chtRangeX As Range, chtRangeY As Range, chtTitle As String,_
     chtXAxisTitle As String, chtYAxisTitle As String)
```

Change

```
.SetSourceData Source:=Sheets(chtSheet).Range(chtRangeY), PlotBy:= _
                                                              xlColumns
.SeriesCollection(1).XValues = Sheets(chtSheet).Range(chtRangeX)
```

To

```
.SetSourceData Source:=chtRangeY, PlotBy:=xlColumns
.SeriesCollection(1).XValues = chtRangeX
```

In this particular case the number of elements in both the implied volatility and the strike price are the same. That means that the rows used in both column A and column B are the same. To find the number of columns we can use the `CurrentRegion` method as follows:

```
NumberOfRows = Sheets("Graph Data2").Range("A1"). CurrentRegion._
                                                          Rows.Count
```

Having found the number of rows we can define the two ranges using `Range(Cells1, Cells2).` More specifically to define `Sheets("Graph Data2").Range(“A1:ANoOfRows”)` we use

```
Set chtRangeX = Range(Sheets("Graph Data2").Cells(1, 1), Sheets("Graph Data2"). _
                                                          Cells(NumberOfRows, 1))
```

Based on these the `Main()` subroutine becomes:

```
Sub Main()
Dim NumberOfRows As Integer, chtRangeX As Range, chtRangeY As Range

'Find the number of rows using the CurrentRegion method. You can use
'it as you need the same number of rows for both x-variable and y-
'variable
NumberOfRows = Sheets("Graph Data2").Range("A1"). _
                                        CurrentRegion.Rows.Count

'Based on this define the ranges
Set chtRangeX = Range(Sheets("Graph Data2").Cells(1, 1), _
```

```
                    Sheets("Graph Data2").Cells(NumberOfRows, 1))

Set chtRangeY = Range(Sheets("Graph Data2").Cells(1, 2), _
                    Sheets("Graph Data2").Cells(NumberOfRows, 2))

Call AddChartSheetXlLine_New("New Graph2", "Graph Data2", chtRangeX,_
 chtRangeY, "Volatility Smile", "Strike Price", "Implied Volatility")

End Sub
```

For the second way we will use the `End` method, which returns a range object that represents the cell at the end of the region that contains the source range. It is equivalent to pressing END+UP ARROW (xlUp), END+DOWN ARROW (xlDown), END+LEFT ARROW (xlToLeft), or END+RIGHT ARROW (xlToRight).

For example to select from A4 to the cell just before the first blank cell in a row on the right we use
`Range("A4", Range("A4").`**`End`**`(xlToRight)).Select`
To select from A1 to the cell above the first blank cell in the column we use
`Range("A1").`**`End`**`(xlDown).Select`

Therefore, a different implementation for `Main()` would be:

```
Sub Main2()
Dim chtRangeX As Range, chtRangeY As Range

'Use the End method to select the range from the cell you define till
‘the cell just above the first blank cells

'The End method returns a range object
Set chtRangeX = Range(Sheets("Graph Data2").Cells(1, 1), _
               Sheets("Graph Data2").Cells(1, 1).End(xlDown))


Set chtRangeY = Range(Sheets("Graph Data2").Cells(1, 2), _
```

```
                Sheets("Graph Data2").Cells(1, 2).End(xlDown))

Call AddChartSheetXlLine_New("New Graph2", "Graph Data2", chtRangeX,_
 chtRangeY, "Volatility Smile", "Strike Price", "Implied Volatility")

End Sub
```

In many cases you may need to export a graph as a `GIF` image file. The following subroutine saves a chart to the hard disk

```
Sub ExportingCharts()
Dim chtName As String
Dim chartFile As String

On Error GoTo ErrorHandling

    chtName = "New Graph2"
    chartFile = "D:\NewChart.gif"

    Charts(chtName).Export Filename:=chartFile, filtername:="GIF"

    Exit Sub
ErrorHandling:

    MsgBox "The graph does not exist"
End Sub
```

If you want you can take the saved chart and load it in a worksheet. Add in the `ExportingCharts()` Subroutine:

`Dim WorkSheetName as string` (below `Dim chartFile As String`)

`WorkSheetName = "Graph Data2"` (below `chartFile = "D:\NewChart.gif"`)

`WorkSheet(WorkSheetName).Pictures.Insert(chartFile)` (before `Exit Sub`)

Note that the GIF file will not be automatically updated if the data change, whereas the graph will be automatically updated, if any data in the defined source ranges that the graph is using, change.

Finally, if you want you can remove the chart sheet using the following command after the export command:

```
Charts(chtName).Delete
```

When you run the subroutine there is an alert message that the graph will be permanently deleted. To avoid these alert messages insert before the delete action:

```
Application.DisplayAlerts = False
```

When the subroutine ends Excel automatically sets the `Application.DisplayAlerts` back to true. The application object is very important as it represents Excel itself (we have already used it when

we introduced Excel built-in functions). There are a number of very useful properties that belong to this object. For example the `CalculationInterruptKey` property, which sets or returns the key that interrupts Excel while performing calculations. It can have one of three values `xlNoKey or value 0`, `xlEscKey value 1`, `xlAnyKey value 2`. Note that using this key does not abort a VBA subroutine that goes into an endless loop. Excel normally calculates each cell as you enter it and minimizes the number of calculation by recalculating only the cells that were linked to the changed cell. You can request a recalculation (in the manual mode) of all changed cells by pressing `F9`, and Excel will recalculate all the changed cells and all the cells linked to them. This can be a very resource intensive process that you can interrupt using the `xlInterruptKey`. The following subroutine clarifies how the `CalculationInterruptKey` property can be used.

```
Sub ApplicationObject()
     Application.CalculationInterruptKey = xlNoKey
     MsgBox Application.CalculationInterruptKey

     Application.CalculationInterruptKey = xlEscKey
     MsgBox Application.CalculationInterruptKey

     Application.CalculationInterruptKey = xlAnyKey
     MsgBox Application.CalculationInterruptKey
End Sub
```

In general, if in your worksheet you have a large number of cells with complex formulas and calculations, you can speed up your calculations if you change the calculation mode to manual.

```
Application.Calculation = xlManual
Application.Calculation = xlAutomatic
```

Furthermore, when you run a macro the screen is automatically updated. This can significantly slow down your calculations. You can turn on/off the screen updating using:

`Application.ScreenUpdating = False` (turns screen updating off)
`Application.ScreenUpdating = True` (turns screen updating on)

Finally, in a number of subroutines dealing with graphs we introduced a new command, the `Set` command. Using the `Set` command we create an object variable. For example the following statement:

```
Set MyRange = Worksheets("Sheet1").Range("A1:A8")
MsgBox Str(Application.WorksheetFunction.Average(MyRange))
```

is the same as:

```
MsgBox Str(Application.WorksheetFunction.Average _
                        (Worksheets("Sheet1").Range("A1:A8")))
```

Using the `Set` command created a Range object and we can use the `MyRange` variable instead of the `Worksheets("Sheet1").Range("A1:A8")` statement anywhere in the code. This results in simpler and easier to understand codes which also run faster.

Note

These notes are not meant to be a substitute for a book on Visual Basic for Applications. They are provided as a quick introduction to VBA, but you should also use books that cover the specific subject matter. I would suggest the following three books:

Excel 2000 Programming for Dummies, by John Walkenbach

This is the best introduction to Excel programming that I am aware of. Very easy to read and covers all the basics and much more than what you will probably ever need.

VBA and Macros for Microsoft Excel, by Bill Jelen and Tracy Syrstad

This is a cookbook for Excel users. It contains a large number of VBA procedures that deal with virtually every possible need you might have.

Advanced modeling in finance using Excel and VBA, by Mary Jackson and Mike Staunton

This is not the book to use to learn VBA, but it is a very good book in teaching what the title promises. The book assumes that you know the underlying theory and focuses on the implementation issues.