

An Excel/Visual Basic for Applications (VBA) Programming Primer

Robert J. Ribando 6/4/2018

Introduction

In the past, even mildly complicated engineering calculations have not mixed well with spreadsheets because of the very strong tendency to wind up with nearly-impossible-to-debug "spaghetti code". While most students seem to enjoy using spreadsheets, instructors recognize that other than by finding the "correct" answer printed somewhere, it is next to impossible to grade assignments or to help the occasional student debug them. However, by using Visual Basic for Applications (VBA) in conjunction with the Excel spreadsheet, the user has the convenience of a spreadsheet for neatly-formatted input/output and for graphical display of results, i.e., to function as a graphical user interface (GUI). Meanwhile well-structured, readable, line-oriented code can be used for the more complicated calculations. This appendix will review a few general aspects of spreadsheets, while providing a brief introduction to the use of functions and subroutines written in VBA. Many books about Excel do not address VBA at all, and if they do, the coverage is limited to a few pages [1-3]. However, several recent books, including several by Walkenbach [4] and others by Orvis [5], Halberg, et al. [6], Chapra [7], Albright [8] and Bullen, et al. [9] do cover VBA extensively. References 7 - 9 are almost entirely devoted to VBA, with the Chapra book aimed at beginning engineering students, including those learning to program for the first time.

In the early days of spreadsheets, a macro was just a recording of a series of keystrokes; that can still be done. (In fact, as will be noted later, recording a series of keystrokes and mouse clicks and then viewing the resulting VBA code generated is a great way to learn the language.) But VBA, which has been available since Excel 5.0 and is a subset of Visual Basic™, is a full-featured, structured programming language and thus far more powerful. Indeed, because VBA allows the programmer to program and thus control Excel's rich collection of drawing and chart objects, it might be considered more potent than VB itself. Consider drawing a rendered cylinder on the screen. From scratch this would be a very formidable job in VB (including hidden surface removal, shading, etc.). With VBA you can just grab a cylinder from the "Autoshapes" collection and manipulate it at will using VBA statements. In addition you don't have to buy another package. Those who have taken a course using any modern structured language like Java, C++, Fortran 90, Pascal, etc. can pick up VBA syntax with very little coaching. Indeed, those students who ordinarily will do anything to avoid writing a traditional computer program find VBA an attractive and enjoyable addition to their skill set.

Several examples given in this document were implemented originally in Excel Version 7.0; however, all are compatible with later versions. Several workbooks discussed here and available on the Heat Transfer Today (<http://www.faculty.virginia.edu/ribando/modules>) website may not be backwardly compatible. Unfortunately recent versions of MS Office for Macs do not support VBA; some institutions have solved this problem by switching to a virtualized computing environment, thus giving Mac and Linux users the capabilities of PC's. Future editions are expected to return VBA capability to Mac users.

General

As with any structured programming language, VBA has certain rules for the naming of variables. Variable names must start with a letter and may consist of letters, numbers and some characters. They are case insensitive and must not include spaces or periods. Certain symbols including #, \$, %, & and ! are forbidden, as are a number of "reserved" words. Many data types,

including Boolean (T or F), integer, long (integer), single (floating point), double (floating point), currency, date, string, object, variant and user - defined, are permitted. Since Excel itself uses double precision arithmetic (~15 significant figures), you may as well declare all VBA floating point variables as “Double”.

The practice of "naming" cells on the spreadsheet is highly recommended. Giving names to cells allows one to reference them in subsequent cell formulae (and also in VBA code) by name rather than cell address, thus making the spreadsheet far more readable than it would be otherwise. For instance, to someone even mildly familiar with fluid mechanics, the cell formula:

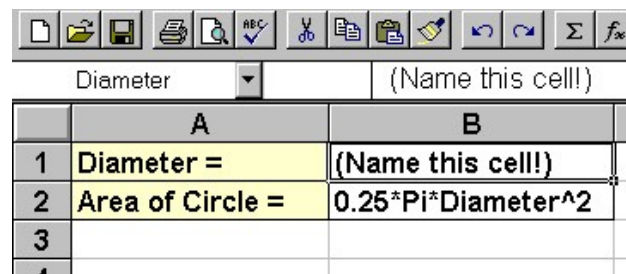
$$= \text{Density_Air}(T_{\text{film}}) * A_{12} * \text{Diameter} / \text{Viscosity_Air}(T_{\text{film}}),$$

and having four of the five variables referenced by name rather than address, looks a lot more like a Reynolds number $\left(\text{Re}_D = \frac{\rho V D}{\mu} \right)$ than does:

$$= \$B\$1 * A12 * \$F\$11 / \$B\$7$$

There are several ways to “name” a cell. The first is to use **Insert – Name - Define** on the 2003 toolbar. This technique allows the programmer to see all cells previously defined, thus avoiding conflicts, and to delete names already assigned should that be needed. If the cell to the left of the one you want to name has what appears to be a valid name in it (e.g., Diameter =), then Excel will suggest that as a possibility. All you have to do is confirm it.¹

The other way to name a cell is to highlight the cell and then simply type the name in the box just above cell A1 and to the left of where the cell formulae appear and hit “enter.” Figure 1 shows a small section of a spreadsheet:



	A	B
1	Diameter =	(Name this cell!)
2	Area of Circle =	0.25*Pi*Diameter^2
3		
4		

Figure 1. Upper Left Corner of the Spreadsheet

Cell B1 is currently the active cell and the name we wish to assign to it has been typed in the Name box above cell A1. Entries shown in cells A1 and A2 are simply text. (Here the equals (=) sign that should precede the formula in Cell B2 has been omitted in order to avoid an error message appearing in the figure.) If, after having named the cell, one supplies a numerical value for **Diameter** in Cell B1 and tries it first without the parenthesis after the **Pi** (and with the equals sign in place) in Cell B2, an error results. A search of the **Help** files for "Pi" reveals that one needs the empty parentheses () in order to invoke this *supplied* (intrinsic) function. Thus the correct cell formula for cell B2 is: **= .25*Pi()*Diameter^2.**

¹ Excel 2007 and up users will find **Define Name** under **Name Manager** under the **Formulas** tab. A list of names previously defined can be found under **Use in Formulas**.

As will be seen later, VBA coding lends itself well to traditional in-line documentation. **Cell Notes** are one effective way to provide documentation on the spreadsheet itself. In Figure 2 a yellow cell note in Cell B2 warns the user that the formula shown is erroneous.

	A	B
1	Diameter =	(Name this cell)
2	Area of Circle =	0.25*Pi*Diameter^2
3		
4		
5		
6		

Figure 2 Use of a Cell Note for Documentation

VBA Functions

Instead of using a formula typed into a cell on the main spreadsheet, one can create a VBA *function* to do it. Creating such an *extrinsic* (user-defined) function is particularly wise if the underlying formula is complicated or includes more than a simple assignment statement. That function will be invoked on the main sheet exactly as would any of the hundreds of intrinsic (supplied) functions (sine, cosine, sum, etc.) provided in Excel by typing the cell formula, e.g.,

= Area_Circle(Diameter)

Now by clicking **Tools, Macro, Visual Basic Editor** and insert a **Module** (the later from the VB **Insert** menu), a blank page will open where the VBA coding will go. One can also click on the VBE icon on the toolbar to get directly into VBA. Users of older versions of Excel may need to use: **Insert, Macro, Module**.²

When you are in the Visual Basic Editor (VBE) you can check the *Project Explorer Window* in the upper left of the screen and you will see a *treeview* picture of your project. An annotated example is seen in Figure 3. (You may only see the project name if someone else has created the workbook and does not want you to see their coding.) You may also find the names of various Excel “Add-ins” that you or some application you previously used has installed on your computer. An Add-in consists of VBA code that the supplier, perhaps a commercial entity, wants you to be able to use, but not to have access to their proprietary source code. An Add-in does not preclude you adding your own VBA coding, whereas simply password protecting modules does.

² Excel 2007 users must click the **Office** button at the top left and under **Excel Options – Popular**, click **Show Developers Tab in the Ribbon**. Once the Developers Tab is showing, then you are able to select Visual Basic and access the development environment.

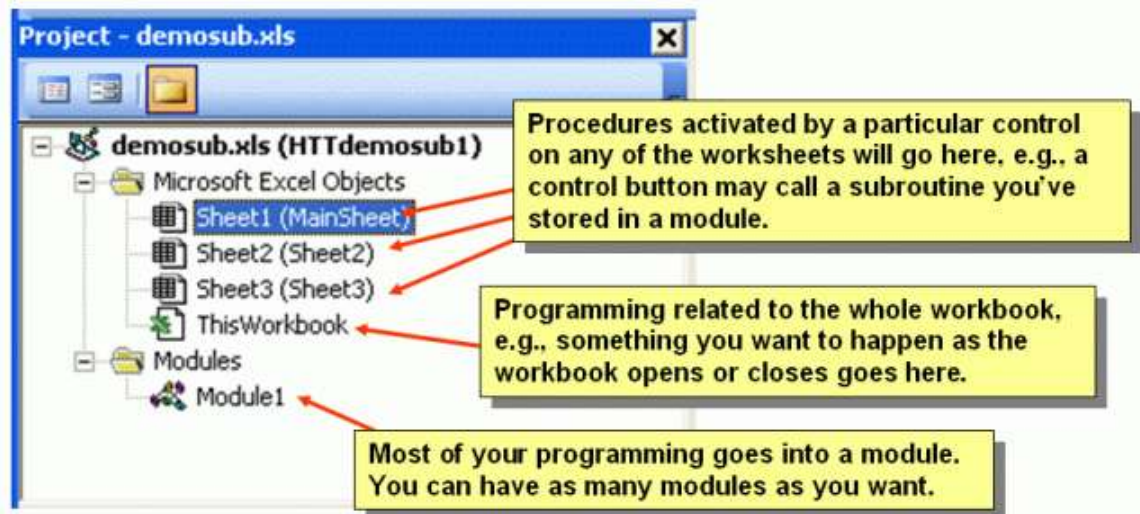


Figure 3. Project Explorer Window. This project (HTTdemsub.xls) has four objects: the Workbook itself, three worksheets plus a single module.

Most of the coding you do should be put in a **Module** (under **Insert**). Coding you put there will be accessible from any of your worksheets and can be exported easily from there to any other projects where it may be reused. If you place a control, e.g., a command button that activates a subroutine on a particular worksheet, then coding for the event you want (calling the subroutine) may appear in the coding for that form. So if you place a command button (from the controls toolbox) on a worksheet and double click it while in design mode (where the eight “handles” you use to resize, reshape and reposition it are visible), you will open the editor and the coding you write to accompany that control will automatically be placed with that sheet instead of in a module. You can put all the code you want executed with the button (or other control) with the worksheet, but putting it in a module as a separate procedure is probably a neater alternative. Recent versions of Excel encourage you to put all your programming in modules by asking you to assign a macro directly to each button you add on a worksheet.

A *module* may consist of a single, user-written function or subroutine or may include multiple subprograms. While, as noted above, one might do some VBA programming under the Excel Objects collection (worksheets and the workbook), it is much more likely that user-written code will go into a module. Every module should start with **Option Explicit**; thus forcing the programmer to declare all variables. This practice helps locate spelling errors, but in addition VBA considers all undeclared variables as type “variant.” That is, for example, if one sets some variable equal to a string, then that is what it will be. If that same variable is set to an integer later, then it will be an integer. This sloppy practice leads to slower execution and requires more memory - for one example in the Walkenbach book [4], the code using type variant was slower by a factor of four. One can have an **Option Explicit** automatically inserted at the beginning of each module by going to **Tools, Options, Editor** and checking **Require Variable Declaration** and clicking “OK” while in the Visual Basic Editor.

Now here’s what VBA code for finding the area of a circle might look like:

```
Function Area_Circle (Diameter as Double) as Double
    Area_Circle = 0.25 * Pi() * Diameter ^2
End Function
```

Note that both the type of the function itself and the argument passed into it (**Diameter**) have been declared as Double precision floating point numbers. (Numbers in cells on the worksheets are automatically double precision, so declaring them single precision within the function would require a type switch both on entering and leaving the function and would cost you about seven significant figures in precision.) But the function as written above doesn't work at all! The message: **#NAME?** (indicating an error) appears in the cell where the function has been invoked on the main sheet. If one deletes the **Pi()** completely, then this function does work (but gives the wrong answer). The reason is that VBA has some of the intrinsic functions that Excel has, but not all. (And some that do the same thing are even named differently; for instance the square root is **SQRT()** in Excel and **SQR()** in VBA.) To tell it to use the **Pi** function from Excel instead, one uses: **Application.Pi()**. Here "application" tells VBA to get the function from wherever the VBA is being invoked (Excel, PowerPoint, Access, wherever).

With just a single statement assigning a value to the function, this example shows about the simplest VBA programming one can do. Everything needed is passed into the function as an argument and a value is returned to the cell where the function has been invoked. Here with thorough documentation is the finished product:

```
Option Explicit           'This forces programmer to declare all variables.
Function Area_Circle(Diameter As Double) As Double
    ' Programmed 9/6/97 R.J.Ribando.
    ' Takes diameter of circle as input and outputs the area.
    ' There is no supplied function for Pi in VBA, so the Application.pi() instructs it to use the Excel function.

    Area_Circle = 0.25 * Application.Pi() * Diameter ^ 2
End Function
```

Note that the above function and ALL functions (in any programming language) must contain a statement assigning a value to the function name! Here it is the second-to-the-last line. Meanwhile back in Excel, your new function; i.e., **Area_Circle**, ought to appear in the list of "User Defined" functions on the Insert Function menu.

This next example uses a control structure within a function, but here again, all necessary data are passed in as arguments and a single value is returned to the cell where the function is invoked. This example (<http://www.faculty.virginia.edu/ribando/modules/xls/HTTPlnkslaw.xls>) uses Excel's **Chart** function to display values computed over a range of wavelengths and temperatures. The actual equation being evaluated (the Planck distribution for blackbody radiation) is given as:

$$E_{\lambda,b}(\lambda,T) = \frac{C_1}{\lambda^5 \left(e^{\frac{C_2}{\lambda T}} - 1 \right)}$$

The following is a VBA implementation:

```
Option Explicit
Function Planck(Lambda As Double, Temperature As Double) As Double

    'Computes the spectral distribution of blackbody radiation.
```



```
' This is Eqn. 12.26 in Incropera & DeWitt 5th Edition, Wiley, 2002  
' R.J.Ribando, 7/9/97
```

```
Dim C1 As Double, C2 As Double, Eee As Double
```

```
C1 = 374200000# 'S.I. Units are being used.
```

```
C2 = 14390#
```

```
Eee = 2.718281828 'Base of natural logarithms
```

```
If Lambda * Temperature < 200 Then
```

```
Planck = 1E-16 ' To avoid trying to plot 0 value on log scale.
```

```
Else
```

```
Planck = C1 / (Lambda ^ 5 * (Eee ^ (C2 / (Lambda * Temperature)) - 1#))
```

```
End If
```

```
End Function
```

A log-log plot of the results obtained by invoking the Planck function at an array of temperatures and wavelengths is shown in Figure 4. Note that this plot is automatically updated when a new temperature is selected with the scrollbar.

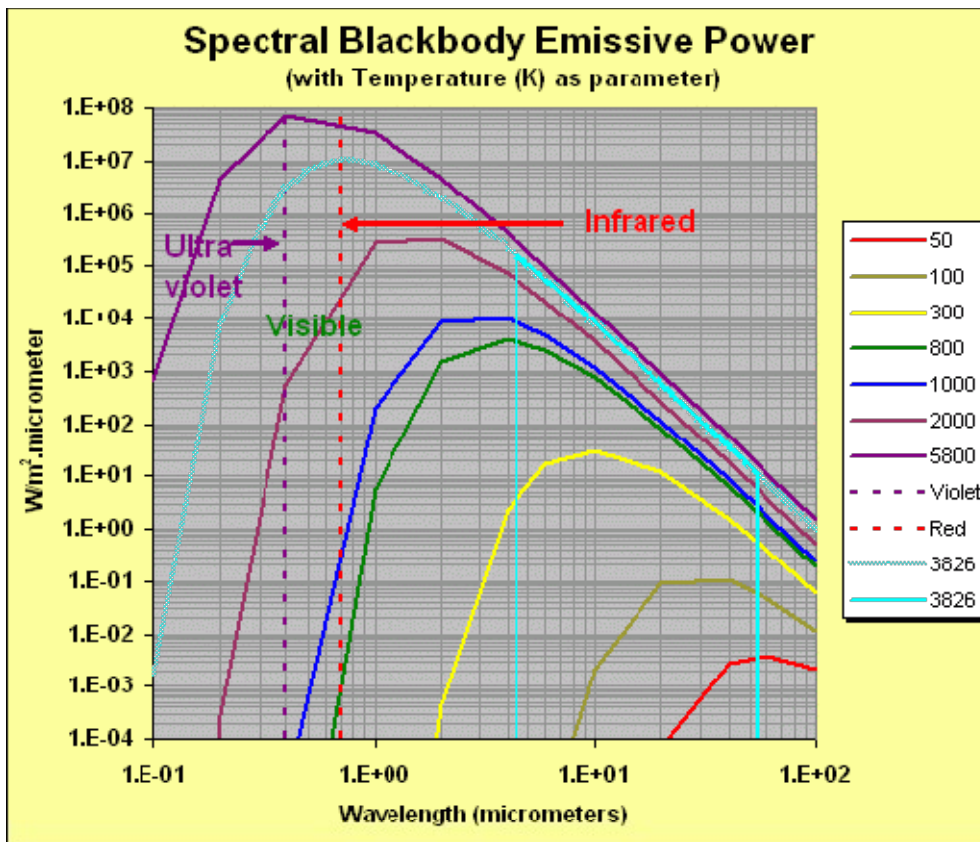


Figure 4. Log-log Plot of Blackbody Emissive Power Generated Using VBA Function Described

The syntax used in the preceding function for **selection** has just two options. One can also allow for multiple selection options:

```
If (...) Then
    (...)
Elseif (...) Then
    (...)
Else
    (...)
End if
```

The next example (<http://www.faculty.virginia.edu/ribando/modules/xls/HTTtwoDss.xls>) is a little more complicated, but still is a function with parameters passed in as arguments. In addition to an **If - Then, Else, End If** for **Selection**, it uses a **For - Next** construct for **Repetition**. This example uses both flat and raised contour plots (see below) to show the computed results for a range of inputs. The function evaluates the following analytical solution (obtained using a technique called separation of variables) for steady-state conduction in a rectangular region having one boundary at unit temperature ($\theta(x, y = W) = 1.0$) and the other three at zero ($\theta(x, y = 0) = \theta(x = 0, y) = \theta(x = L, y) = 0.0$):

$$\theta(x, y) = \frac{2}{\pi} \sum_{n=1}^{\infty} \frac{(-1)^{n+1} + 1}{n} \sin \frac{n\pi x}{L} \frac{\sinh(n\pi y/L)}{\sinh(n\pi W/L)}$$

Because the above is a fairly complicated calculation, there is plenty of documentation included within this function:

```
Option Explicit
Function AnalyticSoln(Nterms As Integer, X As Double, Y As Double) As Double

' This Visual Basic for Applications function evaluates the solution
' to the two-dimensional, steady-conduction problem for a square
' region, fixed value of 1.0 on one boundary, 0.0 on other three.
' The analytical solution from separation of variables is displayed
' on the main sheet and evaluated here. Here W = L = 1.0
' Inputs to this function are the number of terms to be included in
' the series and the x and y coordinates of the point where the
' evaluation is to be made.

' This is a brute-force implementation. If a subroutine instead of a function
' had been used, I could have computed and saved the x-dependent part
' to use at all y values, etc., but I wanted to have an independent function
' that could be invoked anywhere in the solution domain. Main sheet has been
' set up to evaluate this function on a 21 x 21 grid and then makes contour
' plots. Two different contour map styles are demonstrated. User
' is encouraged to vary the number of terms to include in the series, in fact,
' that cell is the only unlocked one on the main sheet.

' VBA doesn't have Pi or hyperbolic sine, so the "application...." tells it to use
' the Excel function instead. The hyperbolic sine gets very large for large
' arguments, so the reduced form of the solution in terms of exponentials is
```

```

' used then.

' R.J. Ribando 6/7/97

Dim K As Integer
Dim Coef As Double, Arg1 As Double, Arg2 As Double, Eee As Double
Dim Pie As Double, Xterm As Double, Yterm As Double

AnalyticSoln = 0#
Eee = 2.7182818
Pie = Application.Pi 'Use Excel's function for Pi

For K = 1 To Nterms Step 2 ' Note coefficient = 0 for K even.
    Coef = 2# * ((-1) ^ (K + 1) + 1) / (K * Pie)
    Xterm = Sin(K * Pie * X)
    If (K * Pie < 5#) Then
        Arg1 = K * Pie * Y
        Arg2 = K * Pie
        Yterm = Application.Sinh(Arg1) / Application.Sinh(Arg2)
    Else
        Yterm = Eee ^ (K * Pie * (Y - 1))
    End If
    AnalyticSoln = AnalyticSoln + Coef * Xterm * Yterm
Next K

End Function

```

A raised contour plot created from a field of data generated by evaluating this function on a 21x21grid (i.e., at 441 points) using nine terms in the series and created using the Excel Chart function is seen in Figure 5. (This is a 3-D Surface Plot.) On the spreadsheet itself, values of the X coordinate were contained in one row; Y values were computed and stored in one column and both are passed as arguments into the function. A second worksheet in the HTTtwodss.xls workbook implements the same solution using a VBA subroutine that takes advantage of the fact that one factor in the above equation is only a function of the horizontal coordinate (x) and another is only a function of the vertical coordinate (y). This strategy makes the evaluation fast enough to give the illusion of animation while building up the solution term by term.

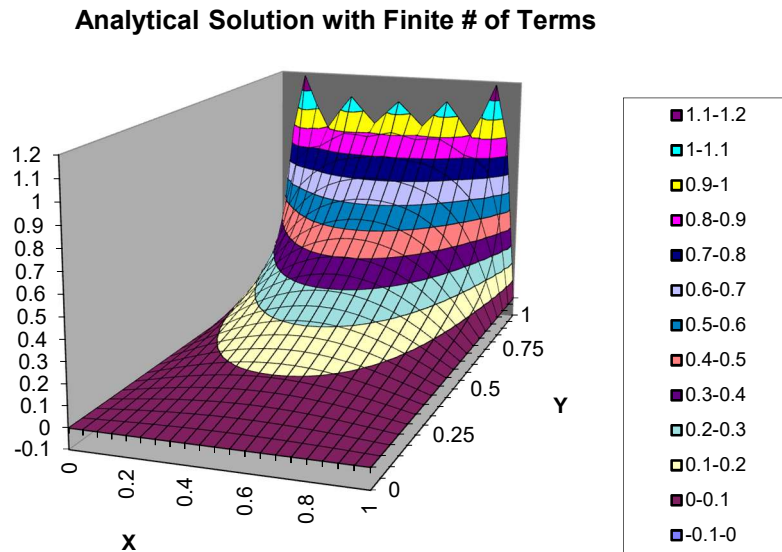


Figure 5. Raised Contour Plot of Analytical Solution (9 terms) Evaluated on a 21x21 Grid

Typing Shift F3 with a particular cell selected can access another feature that helps in debugging functions and subroutines. Figure 6 below shows the display for one particular cell where the above function (AnalyticSoln) has been invoked. It shows the variables passed into the function (the number of terms selected and the X and Y coordinates for that cell) as well as their numerical values for that cell. In the upper right is the value returned for this cell. Providing “Help” for such user defined functions is beyond the scope of this document (See, e.g., Walkenbach’s book).

In addition to the **For ... Next** construct, there are several other constructs for **Repetition**: the **Do ... While** and **Do ... Until**, and two ways to invoke each of them:

<pre>Do While () . . Loop</pre>	<pre>Do . . Loop While ()</pre>
----------------------------------	----------------------------------

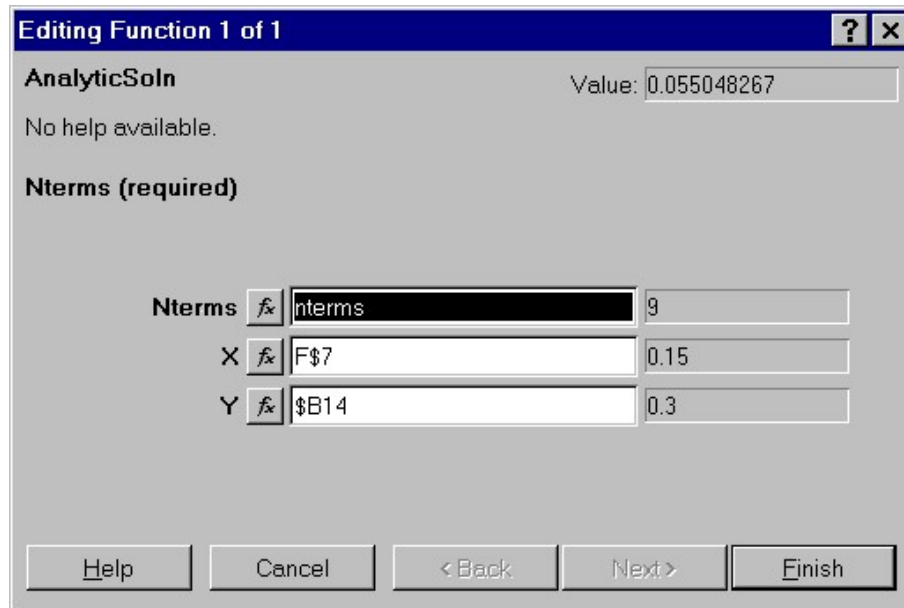


Figure 6. Debugging Window for a Particular Cell in Previous Example

The next example uses the **Case** construct for **Selection** within a function. In addition, it shows how to create a **Message Box** (which will appear on the active worksheet) and also how to put two statements on one line (which is generally not a good practice, but not that detrimental here). In creating the message it also *concatenates* a string and a number.

Option Explicit

Function Discount(Quantity as Integer) as Double

- ' Demonstrates use of Case construct as well as a Message Box.
- ' Note use of colon (:) as statement separator to allow multiple statements on a single line.
- ' (Based on example on pg.183 of Walkenbach's book)

Select Case Quantity

```
Case 0 To 24: Discount = 0.1      ' Multiple statements on a line
Case 25 To 40: Discount = 0.15   ' separate by a colon (:)
Case 50 To 74: Discount = 0.2
Case Is >= 75: Discount = 0.25
```

End Select

```
MsgBox "Discount= " & Discount    ' Concatenate a label and the corresponding
' value. Note that the value doesn't get posted into the cell until user clears the
' message box.
```

End Function

The next and last example function, this one for finding the area of a circle, has nothing passed in explicitly as an argument, but the needed input value (radius) is gained by reference to a particular cell named that on the main sheet. But one must read the note at the top of it very carefully! Note also that the worksheet was named “Mainsheet” in the [HTTdemosub.xls](http://www.faculty.virginia.edu/ribando/modules/xls/HTTdemosub.xls) (http://www.faculty.virginia.edu/ribando/modules/xls/HTTdemosub.xls) workbook.

Option Explicit

```
Function Area_Circle() As Double
' Demonstrates how to get information into a subprogram by using references
' to the worksheet rather than passing values as arguments.
' NOTE! In the cell where this function is invoked on the main sheet,
' there is no mention of radius. So if user changes the cell containing
' the radius, there will NOT be an automatic recalculation as there would
' be if the radius appeared explicitly as an argument. You need to use
' a cntr - alt - F9 to update the whole workbook.
' R.J.Ribando, 9/10/97

Dim ShMain As Worksheet
Dim Radius As Double
Set ShMain = Worksheets("MainSheet")
Area_Circle = Application.Pi() * ShMain.Range("Radius").Value ^ 2

End Function
```

Input – Output: Getting Variables To and From VBA

With the exception of the last one, all the example functions presented so far have gotten data from the spreadsheet into VBA as arguments, and the output has come back to the spreadsheet as the function value returned. Since, unlike functions, subroutines can change more than one value (this distinction between functions and subroutines is true in any programming language), it is appropriate here to point out several other ways to get data into and out of VBA routines. (See Walkenbach's books for much more thorough discussions.)

The *Cells* property allows one to get data to and from worksheets using the row and column indices. So to get data from Cell B7 on the main sheet (See how to declare a worksheet in the previous example), one could use the statement:

```
Diameter = Shmain.Cells(7, 2)
```

where “7” refers to the row, “2” to the column. Similarly, to return data to cell C9 on the main sheet, one could write the VBA statement:

```
Shmain.Cells(9,3) = Area
```

Obviously the *Cells* property works well if one wants to get a whole array or vector to or from a particular area on the spreadsheet.

The *Range* property allows the programmer to address cells by the address, e.g., B7. Using the *Range* property the equivalent of the previous two statements is:

```
Diameter = Shmain.Range("B7")
Shmain.Range("C9") = Area
```

A true range of cells may also be addressed, so that if one wanted to fill several columns using VBA coding, the following statement would fill 18 cells with the number 5.0:

```
Shmain.Range("A5:C10") = 5.0
```

Finally the Range property may be used to address a cell by its name. So assuming the programmer has named a particular cell on the main sheet as “Circumference” and another as “Diameter,” the following statement will retrieve the latter and return the former.

```
Shmain.Range("Circumference") = Application.Pi()* Shmain("Diameter")
```

You can also use Range and Cells addressing to control cell properties like fonts, background colors, etc.

VBA Subroutines

In high level programming languages functions return only a single value. In a similar fashion a VBA function can only affect the one cell where it is invoked. Just as subroutines in Fortran or any other high level language can return more than one value, a subroutine in VBA can change more than one cell. The following module from the HTTdemsub.xls workbook is a subroutine that returns values to four different cells. It also illustrates both the range and cells methods to pass values to and from cells: by name, by address, and by row and column index.

```
Option Explicit
Sub CompAreaVol()

' Computes area and volume of cylinder and sphere and returns the values
' to appropriate place on main sheet, i.e., it is changing values in
' more than one cell with just one invocation. Note there are no arguments
' passed. Three different ways of obtaining data from the mainsheet (which has been
' named "MainSheet") are demonstrated: (1) By name of the cell "Radius"), (2) by cell
' address ("B2"), and (3) using the cells method (Row index, 'Column index).
' Obviously the assignment statements below could be made a lot shorter by
' introducing local variables for the radius, length, pi, etc.

' This subroutine has to be invoked in some way, hitting F5 key from this
' window is one; you can also create a RUN button for the user on the main sheet.

' R.J.Ribando, 9/10/97

Dim ShMain As Worksheet
Dim Radius As Double
Set ShMain = Worksheets("MainSheet") 'This saves a little typing later.

' This one accesses the radius by the cell name and returns values by cell name also.
ShMain.Range("Vol_Cyl").Value = Application.Pi() * _
    ShMain.Range("Radius").Value ^ 2 * ShMain.Range("Length").Value

' This one accesses the radius by its cell address.
ShMain.Range("Vol_Sph").Value = (4# / 3#) * Application.Pi() * _
    ShMain.Range("B2").Value ^ 3

' Next calculation includes area of two ends, plus the sides. It accesses the
' radius using the cells method (rowIndex, columnIndex)

ShMain.Range("Area_Cyl").Value = 2.0*Application.Pi() * _
    ( ShMain.Cells(2, 2).Value ^ 2 + _
```

```

ShMain.Cells(2, 2).Value = ShMain.Range("Length").Value

ShMain.Range("Area_Sph").Value = 4# * Application.Pi() * _
ShMain.Range("Radius").Value ^ 2

End Sub

```

Notice that before running this subroutine (as well as the previous `Area_Circle` function) the first worksheet (Sheet1) had been named "MainSheet" – by right-clicking on its tab and typing in the name. The underscore (_) at the end of several lines indicates that that statement is continued on the next line. One of the cells (B2) on that sheet had been named "radius" and another "length". Figure 7 shows the upper left corner of the MainSheet after the subroutine has been run.

	Radius		6
	A	B	C
1			
2	Radius	=	6
3	Length	=	6
4			
5			
6			
7	Volume of Cylinder	=	1884.956
8	Volume of Sphere	=	4188.78
9	Area of Cylinder	=	1005.31
10	Area of Sphere	=	1256.637
11			
12			

Figure 7. Detail of MainSheet for Subroutine Demonstration

Two of the workbooks on the website, the one for air and water properties [7] (**Airwater.xls**) and the projectile motion project (**Projmotn.xls**), especially the latter, make use of these concepts. In addition, the projectile workbook includes a command button control to activate the subroutine.

Recording and Editing Macros

Especially if you want your VBA coding to interact with Excel's drawing and charting tools, you must learn to record and edit a macro. By running the macro recorder while you draw objects or spruce up graphs and then viewing the VBA code that Excel generates itself, you can incorporate similar coding into your own VBA. Then you can change the size of an object, rescale axes, change line weights or nearly anything else you might want to accomplish by writing your own code. No book can possibly cover all the coding tricks you can pick up this way! The following section shows an excellent student-written example.³

³ As an "advanced" feature, the options for recording and playing back macros are under the Developer tab in Office 2007.

There was a gap beginning with Office 2007 during which any coding involving application of Excel's graph and chart tools was not recorded. Fortunately that feature has been reintroduced in the latest versions.

Use of Drawing Tools and Controls

Any of the drawing and charting capabilities of Excel may be implemented through VBA coding. ActiveX controls may be incorporated similarly. (Use **Tools – Customize – Toolbars – Control Toolbox** to bring up the controls toolbox. The toolbox includes an icon that will switch you between **Design** and **Run** mode.)⁴ In the example seen in Figure 8 below the user enters the radius of the two coaxial parallel disks and their separation distance. A VBA function computes the viewfactor (a geometric quantity representing the fraction of radiation emitted by one disk that is intercepted by the other assuming the first surface is a diffuse emitter and used in radiative heat transfer analysis and in computer graphics) based on the analytical solution, and then invokes a subroutine that draws the two disks to scale within the gray window. As an additional feature, the user can change the viewing angle using the vertical slider bar control.

In order to see how various geometric shapes are incorporated, one may draw that shape manually on the worksheet, while recording the macro. So for instance, if one wanted to learn the VBA coding to draw a red rectangle on the screen, one turns on “Record macro,” draws the rectangle manually (using AutoShapes) and then selects the fill color, shading (if any), etc. Stop recording, and then if you edit the macro you have just recorded, you will find something like:

```
Sub Drawbox()  
,  
' Drawbox Macro  
' Macro recorded 7/19/99 by Robert J. Ribando  
,  
    ActiveSheet.Shapes.AddShape(msoShapeRectangle, 96#, 66#, 144.6, 118.8). _  
        Select  
    Selection.ShapeRange.Fill.ForeColor.SchemeColor = 10  
    Selection.ShapeRange.Fill.Visible = msoTrue  
    Selection.ShapeRange.Fill.Solid  
End Sub
```

Now one can include similar coding into a custom VBA application program, changing for instance, the size, colors and location to suit your needs. In Figure 8, several “special effects” have been used, including horizontal shading of both disks to give the illusion of depth.

⁴ Excel 2007 users will find these controls and the Design/Run switch under “Insert” on the Developer tab.

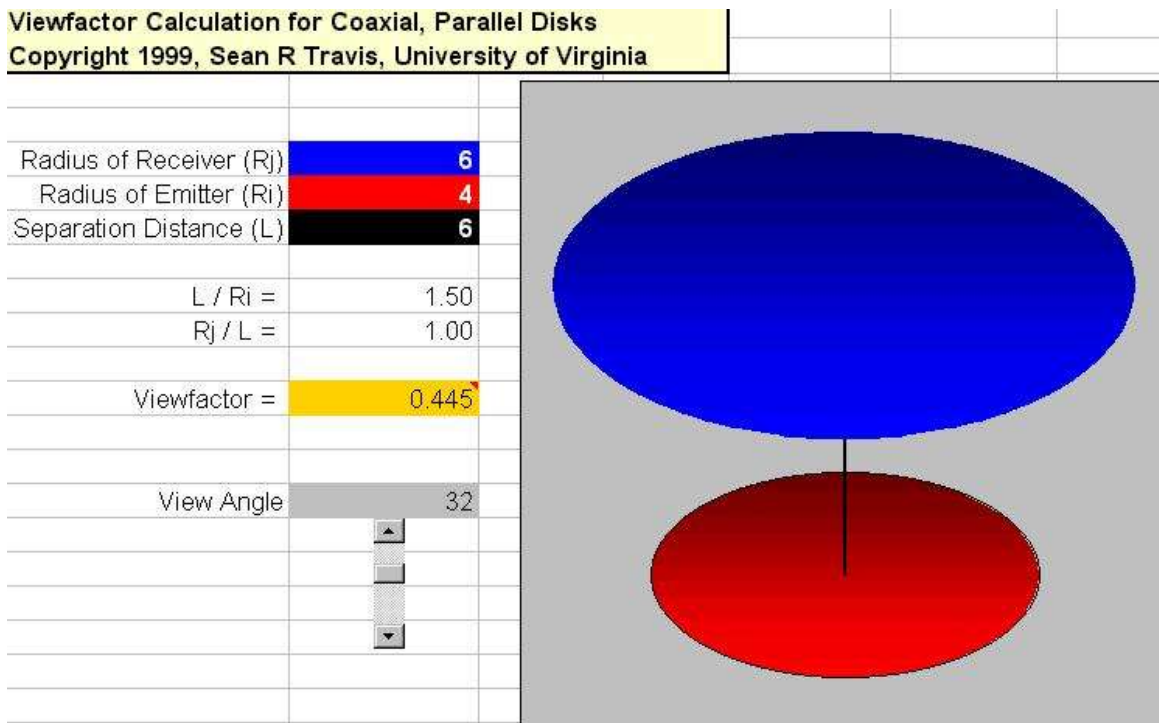


Figure 8. Excel worksheet for viewfactor for coaxial, parallel disks

Two especially useful controls are the command button and the scrollbar. While a function value is updated automatically at the cell where that function is invoked whenever one of its input parameters is changed, one must provide a means to call a subroutine. The command button provides an easy route. One simply draws the button, double clicks on it and writes the coding to call the subroutine. Thus to call the subroutine used in the section above, one writes:

Call CompAreaVol()

under the button, i.e.:

```
Private Sub CommandButton1_Click()
    Call CompAreaVol
End Sub
```

This single line accompanying the button calls the subroutine that you have already saved in a module. You could write all the code you want executed in the Button_click subroutine, but putting it separately as part of a module makes for more portable, usable code. (And to be sure it is often the case of “do as I say, not as I do”!)

Excel 2007 and newer versions enforce the concept of putting all VBA code in modules rather than within worksheet objects. When you add a command button on an Excel 2007 worksheet, you are asked immediately to assign the macro that it is supposed to run. This macro should be a subroutine that you have already written and stored in a module. Since your actual macro may not be ready to go when you insert the button, you may want to create a temporary one first.

Certain controls, e.g., the scrollbar, have the property of a “linked cell.” In Figure 8, the cell that in this static picture has a value of 32 is linked to the scrollbar and holds the current value set by the scrollbar. VBA code can be made to access this particular cell, thus allowing the scrollbar to be used for data input. The “value” property of the scrollbar is an integer that may be set to run from 0 to 32767 ($= 2^{15}-1$). Thus if you wanted the actual property being controlled to run from -1.0 to 1.0 in increments of .01, you could take the limits for the scrollbar to run from 0 to 200 and put that number in the assigned linked cell (which you may want to hide somewhere). In another cell or in VBA code, set the actual value $= -1.0 + \text{scrollbar value}/100$. When the scrollbar value is at its minimum value of 0, then your property has the value of -1.0; at the maximum value of the scrollbar (200), your actual variable has the desired value of +1.0.

Scrollbars can be a very handy way to limit user input to only the range that you, the developer, intended. This prevents the consumer from trying input values that may be completely bogus in your application or for which you have neither verified your calculations nor validated your model.

User Forms

A user form can be a useful way to provide additional documentation or to allow user input to your spreadsheet. Controlled by a command button, a user form can be kept out of the way when not needed so that your spreadsheet doesn’t look cluttered. You can include labels, textboxes, images, command buttons, etc., on your user form. While in the development environment just insert a user form.

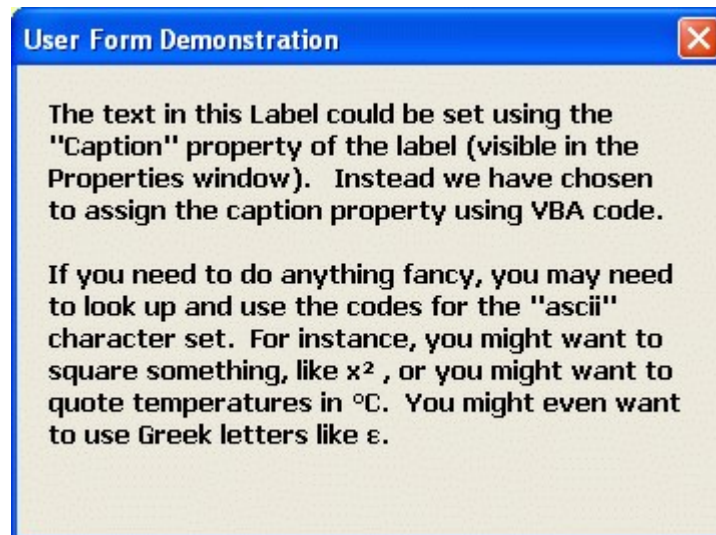


Figure 9. Sample User Form including a Label

User forms, like any object, have their own set of properties. In Figure 9, for instance, the “caption” property of the form was changed from its default value to what is seen above. A label was placed on this form and the text font property of the label was changed from the default value to make the text larger and bold. The caption property of the label could be set in the properties window of the label, but instead we chose to set that property using VBA some code. That code gets executed when the control button is clicked and the form undergoes the “show” procedure. VBA code is used to assign a value to the Caption property of a label on the3 form. The ascii character set is used for the special symbols and the Greek letter.

```
Sub UserFormDemo()  
    UserForm1.Label1.Caption = "The text in this Label could be set using " & _  
        "the "Caption" property of the label (visible in the Properties window). " & _  
        "Instead we have chosen to assign the caption property using VBA code. " & _  
        vbCrLf & vbCrLf & "If you need to do anything fancy, you may need to " & _  
        "look up and use the codes for the "ascii" character set. " & _  
        "For instance, you might want to square something, like x" & Chr(178) & _  
        " , or you might want to quote temperatures in " & Chr(176) & "C." & _  
        " You might even want to use Greek letters like " & ChrW("949") & "."  
    UserForm1.Show (0)  
End Sub
```

VBA for “Heavy Duty” Calculations

VBA can be used for some fairly extensive calculations including the “number-crunching” applications one encounters, for instance, in computational fluid dynamics (CFD) instruction. Because formatting numbers for output to Excel cells is computationally intensive, one is strongly encouraged to pass all the inputs from the Excel sheet into VBA and store them there as local variables, typically declared as “Double”. Only when the calculation is complete should the final numbers be returned to the Excel sheet, where the user will probably want to plot them. If an occasional piece of data is desired during a transient calculation (as, for example, to monitor convergence), one can pass just that data back to the spreadsheet and use the VBA “DoEvents” function or “Application.Calculate” to force the calculation to stop and allow the plotting to be done.

One may also want to provide coding so that any large data sets generated by the program are automatically cleared when the workbook is saved. Using ClearContents as part of the BeforeSave event of your workbook can dramatically reduce the file size of the resulting saved workbook. For example, Shmain.Range(“C1:D1000”).ClearContents immediately clears anything occupying any of the 2000 cells in that range.

If you are going to use Excel and VBA to any extent, then you ought to learn how to use both Excel’s “Goal Seek” routine and its “Solver” add-in. Being an add-in, you will not be able to see the coding behind Solver, but its capability can be extraordinarily useful in serious calculations. You can invoke Solver itself in a workbook or you can use it within your own VBA code.

Enabling Macros

Once you have created a working function or subroutine, you’ll want to save the workbook containing your code and then open and use it at some time in the future. Office 2007 and more recent editions warn you immediately on the command line that the workbook you are opening contains “active content” and asks if you want to enable it. If this workbook is one you wrote yourself or comes from a reliable source and you want the macros to be operable, then you MUST enable them.

The procedure in older versions of Excel is a little more cumbersome. Your security must be set to “medium” even before opening the workbook. Once you have set

your Excel security to medium on your own computer, it will stay that way. In a public environment it is likely reset to the default “high” setting at least once a day. Then when you open the workbook, you will be given the option of enabling the content.

Conclusion

Student response to use of VBA within other courses, both undergraduate and graduate, has been excellent. Since all have had a structured programming course earlier (some in Fortran 90, some in C++, others in Java), the migration to VBA is very straightforward. Most welcome the opportunity to add another package to their inventory of proficiencies. Several graduate students in a recent programming-intensive computational fluid dynamics (CFD) elected to do all assignments using Excel/VBA.

The topics covered in this note are the subjects of only a couple chapters in the Walkenbach book, but are certainly sufficient to address nearly all the sorts of calculations that an undergraduate in engineering (and probably most graduate students) might find him or herself wanting to do. There are a myriad of other operations and features that one can implement in VBA; the Walkenbach books are clearly excellent sources of help. About a score of samples of Excel/VBA coding, mostly applied to heat transfer, thermodynamics and fluid mechanics problems, may be downloaded at: <http://www.faculty.virginia.edu/ribando/modules/xls>.

Disclaimer

I am neither a computer scientist nor a programmer, but rather an engineer. That being the case, I have found VBA to be a quick and easy way to do most of the things I want to do in my teaching and research. Code that you will find described here or available on the website is not guaranteed to be error free or even to follow good programming practice.

References

1. Etter, D.M., *Microsoft Excel for Engineers*, Addison-Wesley Publishing Company, Menlo Park, CA (1995).
2. Gottfried, B.S., *Spreadsheet Tools for Engineers - Excel 2000 Version*, McGraw-Hill, New York (2000).
3. Monson, L., *Using Microsoft Excel 97*, Que Corporation, Indianapolis (1997).
4. Walkenbach, J., *Microsoft Excel 2000 Power Programming with VBA*, IDG Books Worldwide, Inc., Foster City, CA (1999). There are numerous other books on Excel by the same author, including 2003, 2007 and 2010 versions of this one.
5. Orvis, W.J., *Excel for Scientists and Engineers*, Sybex (1996).
6. Halberg, B., Kinkopf, S., Ray, B. et al., *Special Edition - Using Microsoft Excel 97*, Que Corporation, Indianapolis (1997).
7. Chapra, S.C., *Power Programming with VBA/Excel*, Pearson Education, Upper Saddle River, NJ, 2003.

8. Albright, S.C., *VBA for Modelers – Developing Decision Support Systems with Microsoft Excel*, Duxbury – Thomson Learning, Pacific Grove, CA 2001.
9. Bullen, S., Bovey, R. and Glenn, J., *Professional Excel Development: The Definitive Guide to Developing Applications Using Microsoft® Excel and VBA®*, The Addison-Wesley Microsoft Technology Series, 2005.
10. Ribando, R.J. and Galbis-Reig, V., "Convective Heat and Mass Transfer from a Runner Using Some Advanced Spreadsheet Features," *Computers in Education Journal*, Vol. VIII, No. 4, Oct-Dec. 1998, pp. 22-28.

This document was originally published as:

Ribando, R.J., "An Excel/Visual Basic for Applications (VBA) Primer," *Computers in Education Journal*, Vol. VIII, No. 2, April-June 1998, pp. 38-43.

You may cite that as a reference.

8/5/02 Most Recently Revised: 6/4/2018