An introduction to programming in Visual Basic

AGEC 642 - 2022

I. Introduction

The purpose of this tutorial is to provide you with the basic tools needed to write very simple programs for numerical analysis in Visual Basic (VB). VB is extremely powerful and can create nice user interfaces and do lots of fancy formatting. There are a number of references that can help you learn these tools. If you want a book you can borrow my copy of the book by Albright. But these days, I just google to find the answers I need. For the programs required for this class, the vast majority of these tools will be unnecessary and this tutorial covers everything that you need to do basic programming to solve dynamic programming problems. While powerful, however, VB is quite slow. For small programs this is not an impediment, but if you're interested in solving large problems, use the programming skills you learn here and then learn a more efficient language such as Matlab or Fortran.

As you work through this tutorial, make sure you understand what you're doing. You'll need to follow these or similar steps many times in the future. If you understand instead of just repeating, you'll be much happier in the long run. Not every piece of example code in these notes will work by itself. For example, there may be an example that makes use of a variable x, but if your program has not defined that variable or given it values, you may get nonsensical results. This is intentional. You need to think about what you're doing, not just copy it blindly.

There is a ***quiz*** at the end of these notes will test to see if you have learned the basics of programming in VB. **All students using VB for PS#3 should complete this quiz before working on in problem 2.** You may choose to look at the quiz first – if you can solve it, you probably have all the knowledge you need for the programming assignment.

II. Overview

VB has much in common with many other programming languages. A VB program is a series of commands that creates and manipulates variables. VB programs are also called Macros. Several different programs (called Subs in VB) can be in a single file. These Subs can act separately or they can be interconnected. With few exceptions, all your commands must be contained in a Sub, i.e., after a line that opens a sub and before the End Sub line that that ends the sub.

Unlike the command-prompt version of Matlab, a VB program does not run until you tell it to. Further, and very importantly, VB does not give you any output unless you explicitly tell it to put the output into an Excel spreadsheet.

III. A word of warning!!!

1. Save your work every 5-10 minutes. There is usually no autosave working in Excel and even if it is, don't trust it. *VB programs frequently crash*; unless you've saved your file, you can easily lose hours of work.

IV. First step - your first program for writing output

2. Open Excel. If you do not see the Developer menu, you need to activate it. Under the File menu, click Options, then Customize Ribbon. On the right you will see a list of the menus that are visible. Be sure that Developer is checked, then click OK.

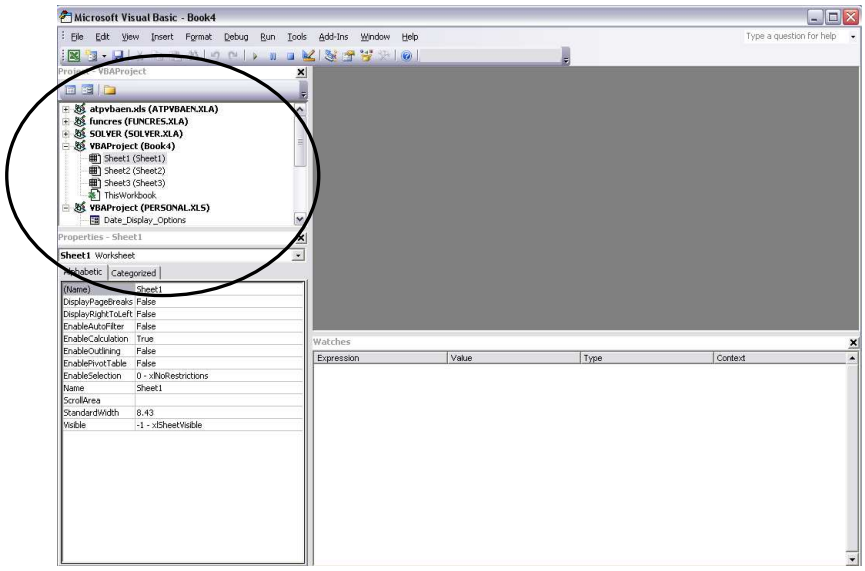
3. By default, Excel is usually set up to not allow subs to run for security reasons. You need to turn this off.
   From the Developer menu, click on Macro Security. On the Macro Settings tab, select Disable all macros except digitally signed macros. This seems to work. If your computer is not recognizing your macros, you may need to switch to the last option, Enable all macros, save and close the file, then re-open it.

4. Start with a blank Excel worksheet and, using Save As, and choose the option for a Macro-Enabled Workbook. Give it any name you like (e.g., something like "VBintro.xlsm").

5. Load the VB editor (alt-F11). The VB editor is an interface where you can create and edit subs and run them.

6. From the Tools menu in VB, choose the Options and on the Editor page, select "Require Variable Declaration." (Tools, Options 2nd box on the Editor page). Click OK. This means that every time you use a new variable, you need to explicitly introduce that variable with a Dim command. (This is just like in Matlab, where a syms command must be used before using a variable can be used on the right-hand-side of an expression.)

7. Make sure that you are Viewing the Project Explorer and make sure that your current project is highlighted in the project explorer. If the Project Explorer is not visible, press Ctrl-R to view it.



8. Using the Insert menu, Insert a Module (*not a class module*). A module is a work space where you can write your programs. The new module you create will appear under your file in the Project Explorer window (on the left). The words Option Explicit should appear at the top of module screen. You can have as many modules as you like and can interact between modules in other programs. But for now, just stick with one.

9. You can also associate VB code with Microsoft Excel Objects (e.g., a sheet or the workbook). This seems to work OK, but I prefer to place them in a Module, where they can easily be found and saved independently of a particular Excel file.

10. In the VB editor, type the words
    sub FirstProgram
    and press [enter]. This will create your first subroutine. Note that the editor automatically completes the lines with () and writes End Sub. It should look something like this

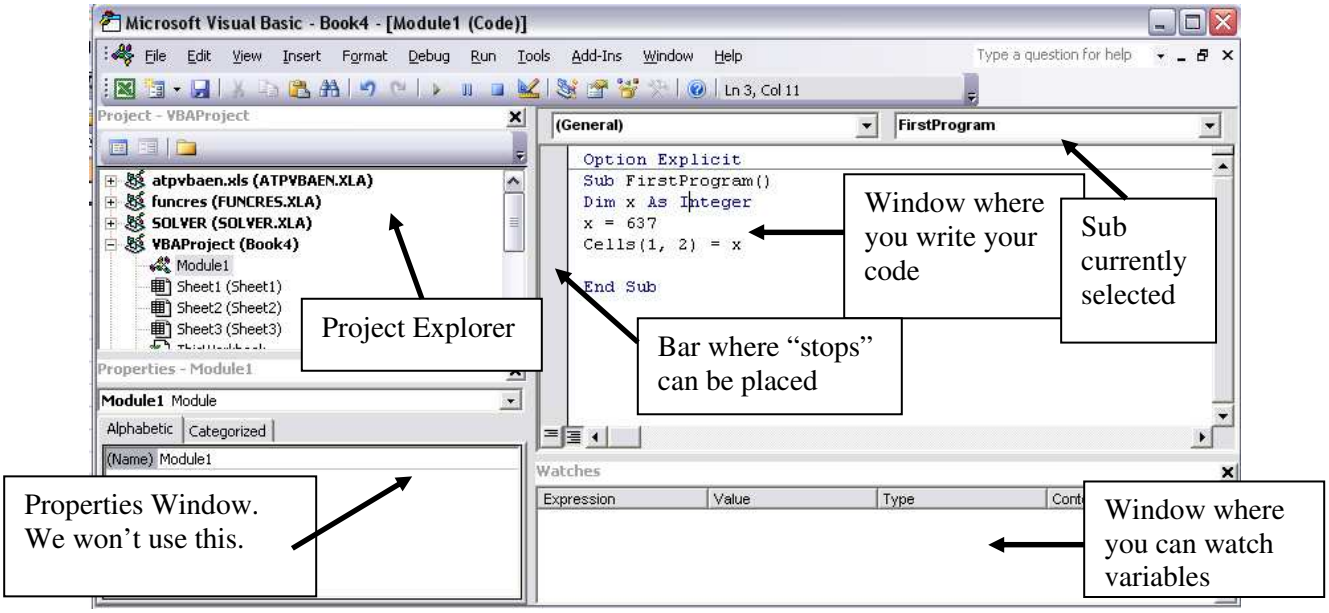
```
Sub FirstProgram()

End Sub
```

    A Sub is a computer program. Unlike in some other languages, in VB there is no difference between the main program and a subroutine. Most Subs can be run by themselves or called by other subs.

    Tip: The help menu in the VB editor (F1) is quite useful and typically has a lot of examples.

V. Running a program

11. The parts of the screen are important. You may want to refer back to the image below later in the notes as we introduce ideas like placing stops and watching variables. Some of these windows may not be visible. You can always activate any of these windows from the View menu.



12. Look at the Sub entitled FirstProgram in the image above. This program initializes x, specifying that it has to be an integer. Then it sets the value of x to 637. Then it writes that value into the cell in the first row and first column of the spreadsheet that you have open. *Type this simple 3 line program.*

13. There are a number of ways to run a program. To start, I suggest you always run the program one line at a time. First, make sure your cursor is inside the Sub you want to run. Then, start pressing the F8 button. Each time you press F8 a new command is run and you can watch the program progress. Try it.

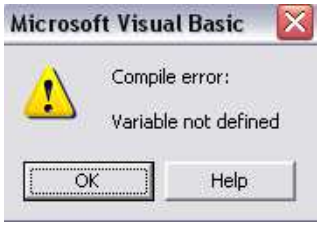
    You can also run the program all at once by pressing the F5 button or clicking on the icon at the top of the screen. Finally you can run the program in pieces by placing a "stop" by clicking on the grey bar to the left of your code, then running the program with the F5 button up to that point.

    Try all of these approaches. Go back to Excel (Alt F11) and you should see 637 in cell B1. Then change the program to put the number 63 in cell C2.

14. ***Important Tip***. Always build your program up in pieces, making sure that each piece runs before you proceed to work on the next step.

VI. Simple variable manipulation

15. Because of the Option Explicit statement in your program, every variable that you use must be explicitly defined with a Dim statement. What a Dim statement does is tells VB that you will be using that variable. If you forget to dimension a variable, when you attempt to run your program VB will tell you that you didn't warn it that the variable was coming by giving you an error message that looks like this.



16. There are four main types of variables that we will use:
    Double – double precision, a pseudo real number (16 zeros after the decimal point),
    Integer – A number that only takes on integer values, used for counting.
    String – A variable that is used to carry text, such as a name
    Variant – A variable that change type during your program.

    It is important to understand the difference between these variable types. Computers make small numerical errors fairly regularly. For example, suppose you've defined a variable x as real number (double) with a value of 1.0. After some manipulations, the computer might store this as 0.99999999999999999999, which is close enough for most purposes; it is as good as 1.0 to you and me. But it looks like a very different number to a computer. If you want to find the first element in an array, for example, if you ask the computer to give you element number x, it won't know what to do; the 0.999^{th} element does not exist. Hence, it is important to always distinguish between integers and real numbers in your programs and use the right variable type for the task at hand.

    How might this affect you? Suppose your program includes a line like,

    ```
    if x = 7 then …
    ```

    Most of the time this will work when x should equal 7.0. But then, all of a sudden, it may make a mistake because of machine error. So what should you do? Either, only use inequalities with real numbers, e.g.

```
if x > 6.999 and x < 7.001 then …
```

or, if possible, use integers

```
if ix = 7 then …
```

Tip: To distinguish between integers and real numbers straight, I usually start my integer variables names with the letter i, or j, such as ix or jStage, and variables that I will use to define dimensions of a problem with an n or m, such as nx or mT.

17. **Local vs. Global variables**
    VB code consists primarily of one or many subroutines, such as the one you've created above. If you Dim a variable inside a Sub, its value will *local*, so that only one Sub is prepared for that variable. However, if you put the Dim statement at the top of your Module, before all of the Subs, then it will be *global variable*. This means two things. First a global variable has been defined for all the subs; you cannot Dim it again within a Sub. Second, it means that the value of global variables will be passed between the subs. Local variables, in contrast, must be redefined (Dim) in each sub and its value will start over in each one. Except in special cases you will probably want all your variables to be global, i.e., dimensioned above all of your Subs.

18. In the code above, we initialized x as an integer. Now let's initialize a variable named a:
    ```
    Dim a As Double
    ```
    Note that as you start typing the word Double a list will appear that gives you all the different types of variables you could use. You can use the Tab button to auto-complete your line of code.

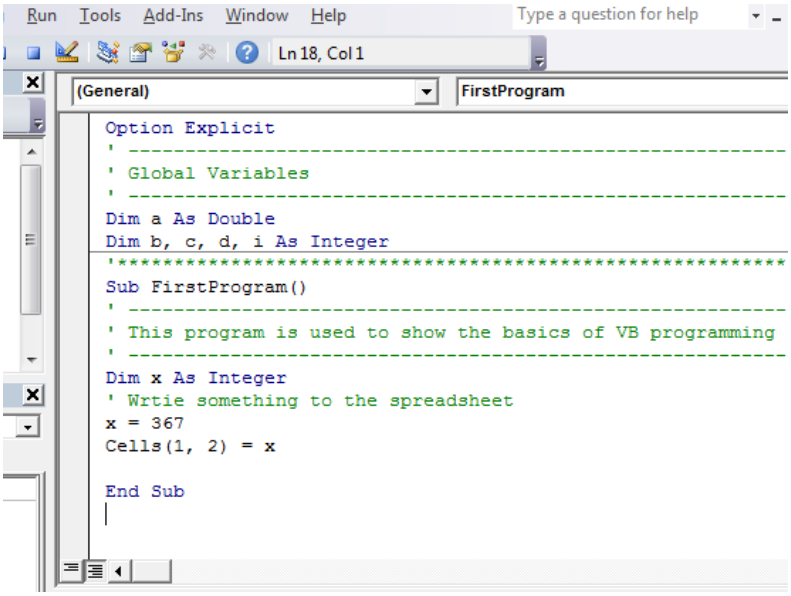
19. If you don't explicitly identify the type of variable, then it becomes a Variant. In my experience VB does a pretty good job of choosing correctly. If a variant type variable is specified as a scalar, it can be Redimensioned to create a multidimensional variable (like a matrix).
    Note that you can dimension a bunch of variables in a single line. E.g.,
    ```
    Dim b, c As Double, d, i As Integer, n As Integer
    ```
    In this case b and d would be variants, c would be a double, and i would be an integer.

VII. Formatting and comments

Good computer code contains comments that help you, the outside reader, and the user to understand what is going on. If you get in the habit of keeping your programs clear as you write them, you will save lots of time later. In VB, any time you write a single apostrophe, the rest of the line will be treated as a comment and ignored when the program is run

Here's what my program looks like so far with comments and notation to divide pieces of the

code.



20. See step #1.

VIII. Stepping through a program and watching your variables

21. Place your cursor inside the sub, then press the F8 key. This will highlight the first line of your code. Press F8 again and it will step to the next line. You can always step through your program line by line in this way. If you wave the cursor over the variable x you will see that when the program starts it has a value of 0, and then will change after you have given it a value.

22. If you right-click on a variable you will be given a list of options. Choose Add Watch and a list should appear at the bottom of the screen with the variables that you're watching. If the list is not visible, from the View menu, choose Watch Window. Now you can see variables' values as you step through the code without having to use the mouse.

IX. Simple variable manipulation

23. You can easily assign values to variables, e.g.,
    a = 6.21
    b = 3.0
    c = a + b
    Note that as you are typing VB usually automatically changes 3.0 to 3#, indicating that this will be zero to 16 digits of precision.

24. Suppose we want b to be an array with two elements. To accomplish this, we need to dimension b differently, say
    Dim b(1 to 2)
    This means that b is now a one-dimensional array with 2 elements, b(1) and b(2).

    The reason we have to write "1 to 2" instead of just "2" is because the default in VB is to start the index with 0. If you typed instead
    Dim b(2)
    then b would have three elements, b(0) , b(1) and b(2).

25. To create a two-dimensional array (a matrix), you simply add another range of indices, as in Dim b(1 to 2, 1 to 1) for a matrix with 2 rows and 1 column (i.e. a vector). When using matrix operations, you must define your vectors as matrices with 1 column. For example, to define b to be a vector with two rows and one column, you would write
Dim b(1 to 2, 1 to 1).
You can also have 3 or higher-order arrays e.g. b(1 to 2, 1 to 5, 1 to 3, 1 to 14).

26. The best way to think of a VB array is as a container, with discrete boxes (see image below). For example, the array defined by the command Dim b(1 to 2, 1 to 3). would have six elements: b(1,1), b(1,2), b(1,3), b(2,1), b(2,2), and b(2,3). You cannot use real numbers to refer to these cells; the command b(1.5, 2.3) doesn't make any sense. Similarly, b(x, y) wouldn't make any sense if x and/or y are not dimensioned integers.

27. To avoid the need to always dimension your arrays so that they start at 1, you can add the command
Option Base 1
directly below the Option Explicit command at the top of your program. If you do this, then Dim b(2, 1) would be the same as Dim b(1 to 2, 1 to 1).

28. It is often useful to allow the dimensions of your program to be dynamic so that you can determine the size of your arrays during your program, perhaps based on other variables. You cannot do this directly using a Dim command; you must use the ReDim command. For example:

```
Dim b( ), c() ' This creates a variable b and tells VB that it will be an array
n = 8
ReDim b(1 to n,1 to 1)     ' this defines a row vector with n elements.
ReDim c(1 to n)            ' this defines a one dimensional array with n elements.
```

If you wanted to restrict the b so that its contents must be real numbers, then the following syntax is required:
Dim b() as double

29. Variables can only be initialized with a Dim statement one time in a single program (either in the program or in a global statement), but they can be redimensioned with a ReDim statement as many times as you like. Each time an array is redimensioned, the values in the array are set to zero.

X. Loops

30. We will use lots of loops in our problems. Here are two simple ways to write a loop:

```
'  ------------------------------------------------------------------------
'  The first loop
'  ----------------------------------------
   i = 1
   Do Until i > 4
      b(i, 1) = i*0.3#
      i = i + 1
   Loop

'  ----------------------------------------
'  A second loop
'  ----------------------------------------
   i = 1
   a = 2#
   For i = 1 to  4
      a = a * a
   Next i
```

Note how the lines between "Do" and "Loop" are indented so that it is easy to see where a loop begins and ends. The same convention is useful for " if " statements.

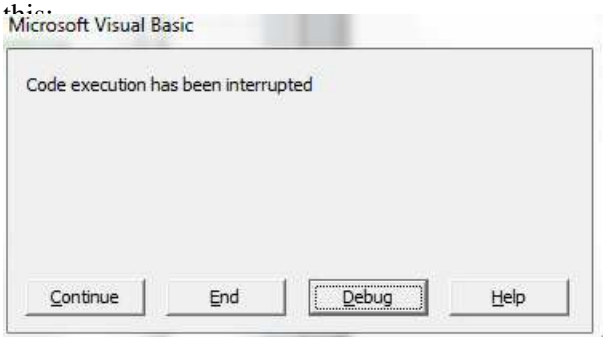
IMPORTANT: Use indentations like this for programs submitted in this class.

Before you run the code above you have to Dim the variable b so that it has the appropriate size and dimension.
What will the array b look like when the first loop is complete? What will be the value of the variable a when the second loop is done?

Note: There is something that can be a bit confusing about these loops: i and a appear on both the right and left sides of some equations. This does not make sense in algebra, but it is perfectly o.k. in computer programming. The right-hand side is treated as happening before the left-hand side, so the value of the variables i and a may be different on the right and left. For example, if the computer enters the line of code i = i + 1 with i=8, then before the line is executed i=8, and after the line is executed, i=9.

31. *Infinite loops and their termination*. Suppose, in the first loop above we accidentally wrote i = i - 1. In this case the termination criterion, i> 4, would never be reached since i would be declining forever. The program would, therefore, theoretically continue running forever. Eventually Excel would crash and you'd lose all your work.

    You can stop a program in the middle using the key stroke *ctrl-Break* (the Break key is usually located at the top right of your keyboard). This *should* pause the program and give you the option of stopping or debugging. When you do this you will be shown a message like this:



    Usually you will want to press debug your code, since that will take you to the point in your program where it currently is and, by rolling the mouse over variables, you can see their values and figure out what is going on. This doesn't always work however, so be careful; you may end up closing the program, losing all of your work.
    Hence, pay attention to step #1 on page 1.

    Below we'll see how you can catch infinite loops before they happen.

32. *Summing up numbers.* Suppose you want to add up all the numbers in a one-dimensional array, say X, which has elements numbered 1 to nx. It would be nice if there were a simple sum function that would do this, but as far as I know there is no such function in VB; you have to do it by yourself. The way you do this is by using a loop:

```
‘  Global Variables
Dim SumX
Dim  X(), i, nx

Sub ProgramToSub
‘ Redimension X as an array with elements, 1, 2, … up to nx
ReDim X(nx)

… A bunch of operations that give us values for the elements of X would be here

 ‘ Now loop from 1 to nx, adding up the values of X
SumX = 0
for i = 1 to nx
     SumX = SumX + X(i)
next i
```

    Summing up the elements of the array X(·)

    At the end of this loop, SumX will be equal to the sum of all the elements of X.

33. *Nested loops*. Suppose that you want to work with a function for all possible combinations of x^1 and x^2, e.g. $f(x_i^1, x_j^2)$ for all i, j. In this case you would *nest* your loops, one inside the other like this:

```
'Start of first loop
 for i = 1 to nx1
     x1 = x1array(i)
     ' Start of second loop
     for j = 1 to nx2
              x2 = x2array(j)

              do whatever you need to do with f(x1, x2)

     next j
     ' End of second loop
next i
' End of first loop
```

XI. Debugging

34. Suppose that you have a watch window like this

Watches

Expression	Value	Type	Context
a	6.21	Variant/Double	Module1.test
⊞ b		Variant/Variant(1 to 2, 1 to 1)	Module1.test

The ⊞ symbol indicates that b is an array. Clicking on a ⊞ will open up a dimension. Opening up both dimensions of b I see the following:

Watches

Expression	Value
a	6.21
⊟ b	
⊟ b(1)	
b(1,1)	1
⊟ b(2)	
b(2,1)	2

XII. Passing values back & forth to Excel

Except towards the beginning of these notes, for the most point we have created and manipulated variables in VB. Such variables only exist in the memory of the computer for the short time that your program is running – then they disappear. In Excel VBA you save your results by passing the numbers to an excel spreadsheet. Similarly, you can also read data from the spreadsheet.

35. The easiest way to write the output of your program into Excel is to include a line like
    `Cells(1, 1) = a`
    This command will place the value of a in the first row, first column of the currently open worksheet of your spreadsheet.

36. Insert such a line of code into your program, run it, then toggle over to your spreadsheet (alt-F11) and verify that it worked.

37. You can also read a variable from the spreadsheet in the same fashion, e.g.
    a=Cells(1, 1)

38. You will often find it convenient to write to spaces that vary within a loop. For example, you could write:
    cells(i,1) = x(i)
    cells(i,2) = SumX
    which would write the i^{th} element of x in row i, column 1 of the open sheet and SumX in row i, column 2.

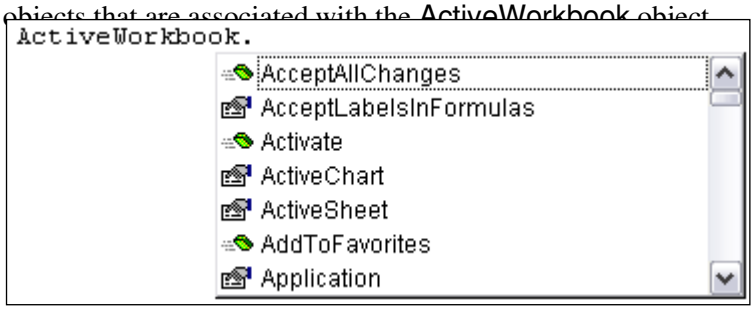
    To practice, in the loop created in step 32 above, to write out your output on rows 1 through n and in columns A and B.

XIII. Introduction to object oriented programming (not critical for AGEC 642)

39. The command cells(1,1) refers to the first row and column of the open worksheet. If you accidentally changed the worksheet that was open, then it would be looking at a different sheet. To avoid this problem, you may want to specify exactly which worksheet you're writing to. To do this, you need to define which *object* you'll be writing to. The cell, is an object in a worksheet, which is an object in a workbook. Using VB you can change the characteristics of any object.

Follow these steps very carefully to see how VB helps you work with objects.

First we identify that we'll be working with an object that is a member of a workbook object. (a) On a new line, type
ActiveWorkbook.
(including the period). Notice that when you type the "." a window pops up indicating the objects that are associated with the ActiveWorkbook object.



Using the arrows to scroll up and down this list, look for the object you want, or start typing and it will automatically move to the one you want. (b) Now start typing
Worksheets
as you type the program will automatically identify the correct field. As soon as the word is selected you can press [tab] and the editor will automatically complete the word. (c) Now type an open parentheses,
(
when you do this, instructions will appear on the appropriate characteristics of the current Worksheet object. In this case you'll see the bolded word ***Index*** which means that we have to identify which worksheet we want to choose. You can type the number 1, referring to the first worksheet in your file, or the word "sheet1" including the quotation marks (or any other name you might give a sheet). (d) Complete the statement
ActiveWorkbook.Worksheets(1).Cells(2, 1) = b(1,1)
and run the program (F5) and verify that cell A2 of the first worksheet now has a value.

40. Note that in the previous step we have to identify which element of b we want to use. Try eliminating the index on b, i.e. change it to
ActiveWorkbook.Worksheets(1).Cells(2, 1) = b
This will put the first element of b in cell A2. If your array b has more than one element, you will *not* be able to print the entire array in this manner. You should get an error since the program is trying to put an entire array into a single cell of the spreadsheet.

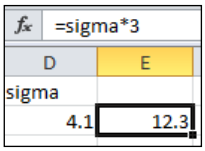
    This alternative command should work:
    ActiveWorkbook.Worksheets(1).Cells(2, 1) = b(1,1)
    Do you understand why the change helped your code to work?

41. Now change the code to read "…Worksheets(6)…" and press run (F5). An error will appear like that discussed below in step 55. Why? In this case the "6" is the "index" that is problematic.

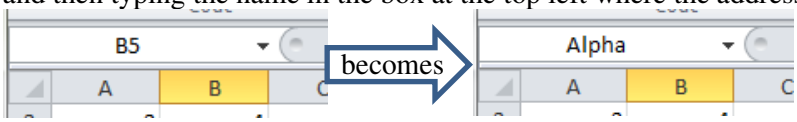
42. Change the code to read "…Worksheets("Sheet1")…" and verify that that works too. Using the name of your sheet is more stable since if you add worksheets, changing the order, your code still writes to the correct place.

XIV. Using named arrays in Excel

A named array in Excel allows you to refer to a cell or cells by a name, e.g. "beta," instead of the cell address, e.g. "F23". In Excel (*not* VB), a named range is created using the *Define Name* command on the *Formulas* ribbon. These names can then be referred to in both VB and Excel using the name, rather than the cell address. For example, this screenshot from Excel shows a cell named sigma in D2, and a formula in E2 with the equation 3*sigma, yielding a value of 12.3.



An alternative way to name a range is by simply selecting the cell or cells you want to name, and then typing the name in the box at the top left where the address usually appears, so that



There are some names that cannot be used; "c" and "r" are two that I know of. So when I want to use them, I use "cc" and "rr" instead. Names can refer to ranges of multiple cells too.

43. Create a named array consisting of a single cell named n and another named array consisting of a 2×2 array called b. (You'll need to use your mouse or the shift key to select the 4 cells in your 2×2 array).

44. In Excel, after creating a named cell with the name of "n" in any other cell you can refer to that value simply by typing its name. For example if A1 has a value of 7, and you give it a name of "n" then you can get an answer of 49 by typing "=A1^2" or "=n^2" Obviously, the second of these is easier to understand and you can type it without having to hunt down where "n" is placed.

45. One of the advantages of named ranges is that when working with VB you can refer to them easily. Toggle back to VB (alt-F11) and add the lines to your program

```
Range("n") = 2
Range("b") = 3
Range("b").Cells(2, 1) = 1
```

46. If you run the sequence of commands above, and the spreadsheet should now include the following cells:

n	b	
2	3	3
	1	3

(Note that the statement `Range("b").Cells(2, 1)` refers to the 2nd row and 1st column of the named range, b. It is convenient that you are not limited to 1,1, through 2, 2. For example, `Cells(0, 0)`, `Cells(1, 7)`, or even `Cells(5, -1)`, would also be valid. This can be quite helpful as we will frequently want to use the number 0 as an index and because it allows you to write output in an area around a named cell, without having to go to Excel to change the size of your named range.

47. You can also read from your worksheet. For example, replace the line
`n = 2`, with
`n=Range("n")`
This way you can use the worksheet to both input and output data.

Note that the range in Excel and the variable in VB are totally independent. Unless you explicitly place a number into Excel it will not be saved. Unless you read it into VB, VB will never know about it.

XV. Some other issues

48. It is frequently useful to break up your programs into pieces. For example, we might have a sub that evaluates the utility function. This is very easy in VB, particularly if all your variables are global. For example you might have a subroutine that simply reads:

```
Sub UtilityFunction()
   utility = Log(z)
End Sub
```

You can then call this sub by writing

```
Call UtilityFunction
```

in your main program. This structure allows you to write general code that can more easily be modified for the problem at hand.
(note: in VB, Log takes the natural log, while in Excel, log() is the base 10 log, and ln() is the natural log.)

Similarly, you can create stand-alone functions as follows

```
Function UtilityFunction(z)
   UtilityFunction = Log(z)
End Sub
```

You can then call this function just like you would any other:

```
utility = UtilityFunction(z)
```

49. We will also frequently use if statements. For example the utility function might be:

```
Function UtilityFunction(z)
'       ------------------------------------------------
'       for positive value of z, u=ln(z+1)
'       for all other values of z, u=0
'       ------------------------------------------------
        If z > 0 Then
                UtilityFunction = log(z+1)

        Else
                UtilityFunction = 0
        End If
End Function
```

50. Finding the maximum or minimum with VB, is easily done by checking each point that you evaluate. For example suppose there was some complicated function $u(x)$. We could find the value from the array x that maximizes $u(\cdot)$ as follows:

```
imax =0
umax = -99999
for ix = 1 to nx
        if UtilityFunction(x(ix)) > umax then
                umax = UtilityFunction(x(ix))
                imax = i
        end if
next ix
```

Note that in this code we initialize umax at a large negative number. That way we are assured that the maximum will be found, even if the maximum is less than zero.

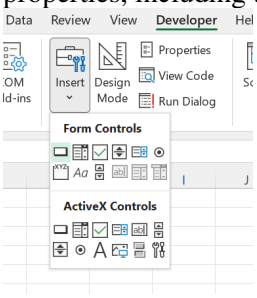
51. If you want to use a function from Excel on numbers in excel, you can do this by simply writing the Application. before the command. For example, to set c equal to the max of a & b, use Excel's max function,
    c = Application.Average(a, b)
    or,
    c = Application.max(a, b)

52. If you want to break a line of your code into two lines, then place an underscore symbol, "_", at the end of the line (after a space) then continue the command on the next line. For example:
    c = Application.Average(a+b*x^2 – c*x^3, _
    b+c*x^2 – c&x^3)

53. So, have you remembered step #1?

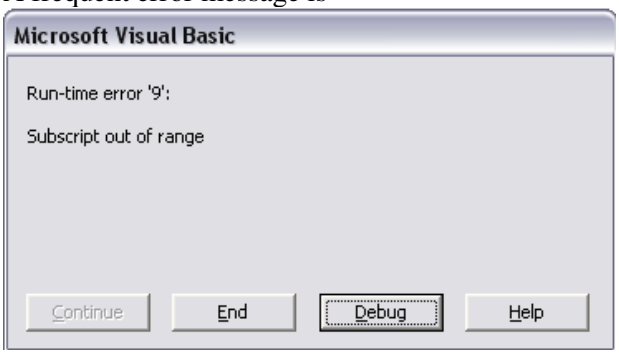
XVI. Running your programs directly from Excel

54. It is often convenient to insert buttons so that you can run your programs without having to go to VB. This is easily done.
    Toggle back to Excel. From the Developer ribbon (see image below), select the Insert menu, then choose the button icon, . Using a click-and-drag process, create a button of the size you wish on your spreadsheet. After defining the size of the button, Excel will prompt you to choose a macro, i.e. the sub you want to start when you push the button. Choose the one you want to use from the list. After that, you can right-click on the button and change its properties, including the text presented and the Sub with which it is associated.



XVII. Common VB Error statements

55. A frequent error message is



This occurs when the "subscript" (i.e. the index) is "out of range."
For example, the following lines of code would lead to this type of error:

```
Dim b(1 to 4), i
for i = 1 to 5
    b(i) = i
next i
```

If you step through this code using F8, you will see when i=5, the error message appears since b(5) does not exist.

After completing this tutorial, you should be able to do the following quiz in about 10 minutes.

Students using VB for their programming assignments in AGEC 642 must submit a spreadsheet that includes a correct and complete version of this quiz before working on the programming problems on problem set #3 or any programming Mastery Tests.

Quiz

(This will not be graded, but you must turn it in before you can take MT I.A.3)

1. Create a new spreadsheet with a VB module.

2. The spreadsheet must contain five named ranges, one named a, one named b, one named n, one named step, and one named results.

3. Put the numbers 5, -.1, 0.8, and 10 in the ranges a, b, step and n, respectively.

You must insert comments in your program that identify where your program accomplishes each of the tasks 4.a through 4.g.

4. Write a program that accomplishes the following tasks:
   a. Dimension a, b , step and n as global variables and i as a local variable.
   b. Read the parameters a, b, step and n from your spreadsheet.
   c. Define a new variable and redimension it using a ReDim statement to be an n×1, two-dimensional VB array.
   d. Using a loop, calculate the numbers a*(i*step)+b*((i*step)^2), for i=1,2,…,n.
   e. Store each of these values in the new variable that you created in step c.
   f. Identify the largest value from among the n values calculated.
   g. Write each of the n values to the spreadsheet below a range named "results," placing a "*" in the cell to the right of the one that has the maximum value.
   h. Save by copying and pasting your results into a separate worksheet 4 sets the results of runs in which each of the parameters, a, b, step and n, have been changed one at a time. The "*" must be repositioned automatically each time you run the program.

5. Optional but recommended: Set up your spreadsheet so that you can run your program from the spreadsheet using a button. You should be able to easily change the values of a, b, step and n to quickly see how your results change.`