An introduction to programming in Visual Basic
AGEC 637 - Summer 2010

I. Introduction
The purpose of this tutorial is to provide you with the basic tools needed to write very simple programs in Visual Basic (VB). They are designed for version of VB that comes with MS Office 2003. If you’re running a more recent version, these instructions may not work in exactly the same way.

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. In particular, I find the book by Albright to be quite useful. Fortunately, for the work we’ll be doing in this class, the vast majority of these tools will be unnecessary and the next several pages should be enough for you to all or almost all the programming that you will need.

As you’re working through this sheet, 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.

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 must turn in the quiz before you turn 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 in AGEC 637.

II. Overview
VB has much in common with virtually all 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 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, taking hours of work with them.

IV. First step - your first program for writing output
2. Open Excel, then go to the Tools menu, click on Options, then the Security Tab. Click on the Macro Security button then select Medium from the list of options. Indicate OK. From now on you will be prompted when you open a spreadsheet with macros attached. Sometimes you need to exit Excel and then start it again before you can start using VB.
3. Start with a blank worksheet and save it (e.g., VBindro.xls).
4. Load the VB editor (alt-F11).
5. From the Tools menu in VB, choose the options and on the Editor page, select “Require Variable Declaration.” (Tools, Option 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, just like you had to do in Matlab with the syms command before using a variable.

6. 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.

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

8. In the 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

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

9. Let’s type a couple of simple lines of code (see below)

This program first initializes \( x \), specifying that it has to be an integer. Then it gives it a value, then it writes that value into the spreadsheet in the first row and second column of the spreadsheet that you have open.

10. There are a number of ways to run a program. To start, I suggest you always run the program one line at a time by pressing the F8 button. This will allow you to carefully 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.

11. Important. 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

12. 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.

13. 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 very 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 with a value of 1.0. This might be stored in your computer as 0.99999999999999999999, which is close enough for most purposes; it’s 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. (To keep integers and real numbers straight, I usually start my integer variables with an \( i \), or \( j \) or an \( n \) or \( m \), such as \( i\text{x} \) or \( n\text{x} \).

14. **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 program (a Sub), its value will local, so that only one Sub is ready for that variable. However, if you put the Dim statement at the top of your Module, above all of the Subs, then it will be global variable. This means two things, first you don’t have to Dim the variable in the individual subs. More importantly, it also means that if you call one sub from another, the value of global variables will be passed between the two subs. Otherwise, if you go from one Sub to the another, the value of the variable is lost.

   Except in special cases you will want all your variables to be global, above all of your Subs.

15. In the code above we initialized \( x \) as an integer. Now let’s initialize a variable named \( a \):
   ```vba
   Dim a As Double
   ```
   This initializes a variable named \( a \). 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.
16. 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 lots of comments, explanations that help the outside reader and user 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 (ignored by the compiler).

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

17. See step #1.

VIII. Stepping through a program and watching your variables

18. 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 after the line x=637 has run, it will take on a value of 637.

19. If you right-click on a variable you will be given a list of option. 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 without having to use the mouse.
IX. Simple variable manipulation

20. You can easily assign values to variables, e.g.,
   \[ a = 6.21 \]
   \[ b = 3.0 \]
   \[ c = a + b \]
   Note that VB usually automatically changes 3.0 to 3#, indicating that this will be zero to 16 digits of precision.

21. Suppose instead we want \( b \) to be a 2×1 vector. We could dimension \( b \) differently, say
   \[ \text{Dim } b(1 \text{ to } 2) \]
   This means that The meaning of this is 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.

22. If you wanted to create a 2-dimensional array (a matrix), you simply add another pair of indices, as in \[ \text{Dim } b(1 \text{ to } 2, 1 \text{ to } 1) \] 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, rather than 1-dimensional arrays. You can also have 3 or higher-order arrays e.g. \( b(1 \text{ to } 2, 1 \text{ to } 5, 1 \text{ to } 3) \).

23. If you want your arrays to start numbering from 1 instead of 0, you can add the command \[ \text{Option Base 1} \]
   directly below the \[ \text{Option Explicit} \] command at the top of your program. If you do this, then you can concisely write \( b(2, 1) \) instead of \( b(1 \text{ to } 2, 1 \text{ to } 1) \).

24. It is often useful to allow the dimensions of your program to be determined by the variables themselves. To do this, you must use the \text{ReDim} \] command. For example:
   \[ \text{Dim } b \]
   \[ n = 8 \]
   \[ \text{ReDim } b(1 \text{ to } n, 1 \text{ to } 1) \]

   If you wanted to restrict the \( b \) to be an integer or real, then the following syntax is required:
   \[ \text{Dim } b() \text{ as double} \]

25. Although a variable can only be initialized with a \text{Dim} \ statement once in a single program (either in the program or in a global statement), they can be redimensioned with a \text{ReDim} \ statement as many times as you like. Each time, the variable’s values are set to zero.
X. Loops

26. We’ll use lots and lots of loops in our problems. Two Simple ways to write a loop:

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

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

Before you run the code above you have to Dim 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 the value of a be 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.

27. 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 loose 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 will pause the program and give you the option of stopping or debugging. 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.

Below we’ll see how you can catch infinite loops before they happen.
28. *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 \( n_x \). It would be nice if there were a simple sum function that would do this, but as far as I know there isn’t. So the way you do this is by using a loop. Here is some code that would do that

```
' Global Variables
Dim SumX, X(), i, nx
Sub ProgramToSub
' Redimension X as an array with elements, 1, 2, ... up to \( n_x \)
ReDim X(nx)

' carry out operations that give us values for the elements of \( X \)
' Now loop from 1 to \( n_x \), adding up the values of \( X \)
SumX = 0
for i = 1 to nx
    SumX = SumX + X(i)
next i

At the end of this loop, \( \text{SumX} \) will be equal to the sum of all the elements of \( X \).
```

29. See step #1 on the first page of this guide.

**XI. Debugging**

30. Suppose that you have a watch window like this

<table>
<thead>
<tr>
<th>Watches</th>
<th>Expression</th>
<th>Value</th>
<th>Type</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>6.21</td>
<td>Variant/Double</td>
<td>Module1.test</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>Variant/Variant(1 to 2, 1 to 1)</td>
<td>Module1.test</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Watches</th>
<th>Expression</th>
<th>Value</th>
<th>Type</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>6.21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b(1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b(1,1)</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b(2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b(2,1)</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**XII. Passing values back & forth to Excel**

Except for at the very beginning, to this point you have created and manipulated variables in VB. But they exist only for the short time that your program is running – then they disappear. You will want to save your work by passing the numbers to an excel spreadsheet. You can also read data from the spreadsheet.

31. 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.
32. Toggle over to your spreadsheet (alt-F11) and verify that it worked.

33. You can also read a variable from the spreadsheet in the same fashion, e.g.

   a = Cells(1, 1)

34. 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

in the loop created in step 28 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 637)

35. 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.

36. 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.

   ![ActiveWorkbook Menu](image)

37. You can use the up and down arrows to scroll up and down this list looking 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). 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.
38. 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

\[
\text{ActiveWorkbook.Worksheets(1).Cells(2, 1) = } b
\]

and see what happens. You should get an error, why? What are we now trying to put in a single cell of the spreadsheet? Try the alternative command

\[
\text{ActiveWorkbook.Worksheets(1).Cells(2, 1) = } b(1,1)
\]

Do you understand why the change helped your code to work? You can’t put an array in a single cell.

39. Now change the code to read “…Worksheets(6)…” and press run (F5). An error will appear like that discussed in 52. Why? In this case the “6” is the “index” that is problematic.

40. 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. “\( \beta_0 \)," instead of the cell address, e.g. “F23". In Excel (not VB), a named range is created using Insert menu select and the Name & Define commands. These can then be referred to in both VB and Excel using the name, rather than the cell address.

41. 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).

42. 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 has a lot of advantages.

43. 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

\[
\begin{align*}
\text{Range("n") = 2} \\
\text{Range("b") = 3} \\
\text{Range("b").Cells(2, 1) = 1}
\end{align*}
\]

44. Run the code and the spreadsheet should now include the following cells:

<table>
<thead>
<tr>
<th>n</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

(Note that the statement \( \text{Range("b").Cells(2, 1)} \) refers to the 2nd row and 1st column of the named range, \( b \). You are not, however limited to 1,1, through 2, 2. For example, \( \text{Cells(0, 0)} \), \( \text{Cells(1, 7)} \), or even \( \text{Cells(5, -1)} \), would also be valid. This will prove quite helpful as we will frequently want to use the number 0 as an index.)
45. You can also read from your worksheet. For example, replace the line
   \[
   n = 2, \text{ with} \\
   n = \text{Range}("n")
   \]
   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.

46. So, have you remembered step #1?

XV. **Some other issues**

47. It is frequently very useful to break up your programs into small bite size 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:

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

   You can then call this sub by writing
   ```vb
   Call UtilityFunction
   ```
   in your main program. This structure allows you to write general code that can easily be modified for the problem at hand.
   (note: in VB, Log takes the natural log, while in Excel, \text{log()} is the base 10 log, and \text{ln()} is the natural log.)

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

   ```vb
   Sub UtilityFunction()
       ' ----------------------------------------------- ---
       ' for positive value of z, u=ln(z+1)
       ' for all other values of z, u=0
       ' ----------------------------------------------- ---
       If z > 0 Then
           utility = Log(z+1)
       Else
           utility = 0
       End If
   End Sub
   ```

49. If you want to use a function from Excel on numbers in excel, you can do this by simply writing the \text{Application}. before the command. For example, to set \(c\) equal to the max of \(a\) & \(b\), use Excel’s max function,

   \[
   c = \text{Application} \cdot \text{max}(a, b)
   \]

   or,

   \[
   c = \text{Application} \cdot \text{max}(a, b)
   \]

50. If a line of your code is so long that you want to break it into two lines, place an underscore symbol, “\_\_”, at the end of the line (after a space) then continue the command on the next line. Try breaking a line in two for practice.
XVI. Running your programs directly from Excel

51. 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. First show the Forms Toolbar by using the View Menu and the Toolbars & Forms commands. Click on the button icon, then 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 for the macro (e.g., sub) that you want that button to be associated with, choose the one you want to use from the list. Then type the text that you want to appear on the button. Click outside the button and it is complete. You can later change the properties of the button by right-clicking on the button.

XVII. Common VB Error statements

52. A frequent error message is

![Error Message]

This occurs when the “subscript” (i.e. the index) “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.

Quiz

(This will not be graded, but you must turn it in before you can turn in your program for PS3)

1. Create a new spreadsheet with a VB module.

2. The spreadsheet must contain three named ranges, one named a, one named b, and one named results. Put the numbers 5 and -.5 in the ranges a and b.

3. Write a program that
   a. reads the parameters a and b from your spreadsheet,
   b. calculates the numbers $a*i + b*i^2$, for $i=1,2,\ldots,n$ and stores these in a $n\times1$ two-dimensional VB array which you have redimensioned using a ReDim statement.
   c. identifies the largest value from among the $n$ values calculated
   d. writes the results to the spreadsheet into a range named “results,” placing a “*” in the cell to the right of the one that has the maximum value.

4. Optional but recommended: Set up your spreadsheet so that you can run your program from the spreadsheet using a button.