Introduction to Visual BASIC for Applications: Part I

The purpose of this lab is to help you become familiar with Visual BASIC (VB) for Applications, the built-in macro language for Microsoft Excel spreadsheets, Word documents, and many other applications. VB is a powerful programming language, and since it is built into the spreadsheet, all of the spreadsheet functions are available to the programmer along with many of the features of the standalone VB compilers. VB provides an easy way to write sophisticated simulation models of environmental and biological processes using the Windows graphical user interface.

This laboratory exercise is not likely to turn you into a computer programmer. It should, however, get you far enough so that you can understand BASIC computer programs, write simple functions and programs, and modify programs that are provided to you.

Becoming Familiar with Visual BASIC

VB is the macro language of Excel, and is therefore built into each copy of Excel. We will assume that you have some familiarity with Excel and Windows. To get started on VB, open a spreadsheet and then click on Help. If the “Office Assistant” is active, hide it first (Right click on the icon, select “options”, and disable “Use the Office Assistant”). Now click on Microsoft Excel Help, select “Microsoft Excel Visual Basic Reference” from the table of contents. Read the information on VB and try to become familiar with its general operation.

Recording a Visual BASIC Macro

Read the help section on recording a macro (“Answer Wizard” tab, search for “Create a macro”). Recording a macro is a good way to see how to generate VB commands that you may want to use in your programs. Now record a macro to cut values from one section of a spreadsheet and paste them to another. Select Tools, Macro, Record New Macro, then OK. The cursor will return to the spreadsheet with a little stop button appearing at the right of the sheet. Now go through the steps as you would normally carry them out. Select the cells, cut them out, move to a new location on the spreadsheet, and paste them. Then press the stop button. Now activate the VB editor (Alt-F11) and look at the code in Module1.

Functions and Subroutines

VB programs are made up of functions and subroutines. Functions start with the name Function, and end with End Function. Subroutines start with Sub and end with End Sub. Functions, in general, are used to return a value resulting from a computation, while subroutines are used to execute a series of program instructions. Functions that you write as macros can be accessed directly from cells in the spreadsheet, just as you access the built-in functions in Excel (such as SQRT, SUM). Subroutines are run from the spreadsheet by selecting Tools, Macro, choosing the name of the subroutine from a list, and then choosing Run. The code in your generated subroutine above consists of instructions for selecting, cutting and pasting information in the spreadsheet. This code shows one way of addressing locations in the spreadsheet. We will show others later. Try generating code with the macro recorder for other common spreadsheet activities and then check what is generated. The other subroutines will be added to Module1 and called Macro2, Macro3, etc.

Writing and Running a Function in Visual Basic

Enter a few numbers in adjacent columns of your spreadsheet that could represent maximum and minimum air temperatures ($T_{max}$ and $T_{min}$). We will now write a function that takes the pairs of values and
computes the thermal time for that day. The thermal time is the difference between average air temperature and base temperature if the average is above base temperature, and is zero if the average temperature is equal to or below base temperature. From the main menu bar, select **Insert** and then select **Module** to open a new module and enter the following code into the module:

```vba
Option Explicit

Function DDI(Tmax As Single, Tmin As Single, Tb As Single) As Single
    Dim dd As Single
    dd = 0.5 * (Tmax + Tmin) - Tb
    If dd <= 0 Then DDI = 0 Else DDI = dd
End Function
```

The name of the function is DDI (*day degree increment*). You could name it anything you want. The “Option Explicit” is not required in VB, but it is a good habit to use it in all of your modules. This forces the computer to check that all variables you use have been declared in a Dim statement or the function argument list. Using this statement will save a lot of debugging time. The variables in the augment list (in parentheses after the function name) are $T_{\text{max}}$ and $T_{\text{min}}$ and the base temperature. These values are passed to the function when it is called. We have also declared the variable type (Single) of each variable, and of the value the function returns.

The following table shows the supported data types, including their storage sizes and ranges.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Storage Size</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>2 bytes</td>
<td>True or False</td>
</tr>
<tr>
<td>Integer</td>
<td>2 bytes</td>
<td>−32,768 to 32,767</td>
</tr>
<tr>
<td>Long</td>
<td>4 bytes</td>
<td>−2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>Single</td>
<td>4 bytes</td>
<td>−3.402823E38 to −1.401298E−45 for negative values; 1.401298E−45 to 3.402823E38 for positive values</td>
</tr>
<tr>
<td>Double</td>
<td>8 bytes</td>
<td>−1.79769313486232E308 to −4.94065645841247E−324 for negative values; 4.94065645841247E−324 to 1.79769313486232E308 for positive values</td>
</tr>
<tr>
<td>String</td>
<td>1 byte per char</td>
<td>0 to approximately 2 billions</td>
</tr>
<tr>
<td>Variant</td>
<td>16 bytes +</td>
<td>Any numeric value up to the range of a Double or any character text</td>
</tr>
</tbody>
</table>

We will usually use Single for floating point data and Integer for integers. Occasionally, where numbers might be very large or small, or where high precision is required, we will use Double. Long is used for integers outside the range of the Integer type. String is used to store character strings. If you do not specify the type of a variable it is automatically typed as Variant. That uses a lot more memory than the other data types, and the program runs slower, so you should give the type of all variables and functions. The “Dim dd
as Single” statement shows how variables are declared in the body of the function or subroutine. The remaining code performs the calculation and returns the result.

After you finish entering the code, go to the spreadsheet into which you entered the temperature values. Assume the first set of temperatures is in cells A2 and B2. In cell C2 enter the function as =DDI(A2, B2, 10), which tells the computer to run the function DDI using the values of A2 and B2 for $T_{\text{max}}$ and $T_{\text{min}}$ and 10 for the base temperature. The result of the calculation will appear in C2. Now copy this cell down to the remaining cells. Verify by hand calculation that the correct numbers are generated. Try some temperatures for which DDI should be zero and verify that it is.

A simple function like DDI would not be difficult to implement as a regular cell formula in a spreadsheet, so you may wonder why we go through the hassle to make it a macro. The important thing to understand is that DDI could be made arbitrarily complicated, far beyond the capability of the cell calculations in the spreadsheet and it would work the same way as it does with this simple function.

Writing and Running a Subroutine in Visual BASIC

Subroutines do not return values; they just run the program steps within them when they are called. Subroutines can call functions and other subroutines. They are run either from the Tools menu, or by placing buttons on the spreadsheet that call the subroutine. We will write a subroutine, which reads a value from cell A1, writes the numbers 1 to 10 in cells B1:B10, and writes the product of the number read in and the numbers 1 to 10 in cells C1:C10. The subroutine can be placed in the same module with your function, and could look like this:

```vbnet
Sub ReadWrite()
    Dim I As Integer, a As Single
    Worksheets("Sheet2").Activate
    a = [A1]
    For I = 1 To 10
        Cells(I, 2) = I
        Cells(I, 3) = I * a
    Next I
End Sub
```

Note that the subroutine has an empty argument list. The parentheses must be there whether or not arguments are passed. The Worksheets("Sheet2").Activate statement tells the subroutine which worksheet to operate on. The worksheet folder name must correspond exactly to the name given here. There are two ways to access the cells of the spreadsheet. One is to place the cell name in brackets, such as [A3] or [X7]. The other is to use the keyword Cells with the row and column in parentheses. Both methods have been used here. Each works both for reading and writing. a = [A1] places the contents of cell A1 in a, and [A1] = a places the contents of a in A1.

Enter the subroutine, place a number in cell A1 of Sheet2, select Tools, Macro, then select the name of the subroutine from the list by clicking on it, and select Run. The subroutine should run and place the correct numbers in the spreadsheet. Again, this is a trivial example, but if you can do this you can do any of the reading from and writing to spreadsheets that needs to be done.

You should also try setting up a button to run the subroutine. If the Drawing toolbar is not visible, go to View, select Toolbars, and check the box for Forms. Go to Sheet2, select the Create Button icon, and make a rectangular button on the spreadsheet. A menu will appear asking which subroutine you want to assign. Pick the ReadWrite subroutine and OK. You can edit the text on the button now by choosing it with
the mouse and writing the text you want, or you can change it later by right-clicking the mouse on the button. When you are finished, click elsewhere on the sheet, and then go back to the button. A little hand will appear. Click on the button and the program will run. Note: you will need to erase the old numbers to see it work. You can also assign macros to pictures, drawings, clipart and any other object elements that can be embedded in the worksheet (more on objects later).

**Arrays and Subscripted Variables**

If we were interested in having the computer store the amount of rain for each day of a year, it would be inconvenient to specify 365 different variable names as locations for these values. Visual BASIC provides an Array type that allows us to specify a single name for these variables, with subscripts to identify the values for the different days. If the variable name is p (for precipitation), then the array would be p(d), where d is the day number (1..365). To provide sufficient space in memory for these arrays, we need to specify their dimensions and types at the beginning of the program using the DIM statement. Go to your help index again, and read about the DIM statement. The dimension statement for recording precipitation values would be “DIM p(366) as Single”. When no lower value is specified, BASIC assumes it to be 0. This would provide 367 locations in which to store precipitation values.

**Modeling Insect Population Dynamics—An Example of Arrays in Models**

The use of arrays and matrix algebra are ideal for modeling population dynamics of organisms. If \( n(t) \) is an array that represents the number of organisms at time \( t \), then the number at a later time, \( t+1 \), can be computed from

\[
    n(t+1) = \Lambda \, n(t)
\]

The array, or vector, \( n \), consists of the number of females in each age class, and the array, or matrix, \( \Lambda \) (called a Leslie matrix), is the survival-replacement matrix. Multiplying the number of females in each age class at time \( t \), by the survival-replacement matrix, gives the number of females in each age class at the next time step. We need only model the female population because they carry the entire reproductive potential for the population.

We will work through, and then program a very simple example. Assume a female insect population consists of only 2 age classes, juveniles and adults. Assume that 60% of juveniles survive to become adults, and that each adult female produces 3 female juveniles within a time step. The survival-replacement matrix and beginning age class vector are (there are 100 adults and 10 juveniles at the start):

\[
    \Lambda = \begin{pmatrix} 0 & 0.6 \\ 3 & 0 \end{pmatrix} \quad n(0) = \begin{pmatrix} 100 \\ 10 \end{pmatrix}
\]

Using the rules of matrix multiplication, and applying Eq. 1, we can easily compute the numbers of females in each age class after one time step. The number of new adults would be \( 0 \times 100 + 0.6 \times 10 = 6 \), and the number of new juveniles would be \( 3 \times 100 + 0 \times 10 = 300 \). For the numbers in the next time step, matrix \( \Lambda \) would operate on the new numbers 6 and 300 to give 180 and 18. You can see already that the population fluctuations are pretty wild. A few more steps would convince you that the general population level will continue to increase without bound.

Now write a program to compute insect populations over many generations. It uses the ideas we have already discussed, but extends them. A possible implementation is as follows:
Sub Leslie()
    Dim nt(2) As Single, ntplus1(2) As Single, lambda(2, 2) As Single
    Dim I As Integer, J As Integer, K As Integer, sum As Single
    Worksheets("Sheet3").Activate
    lambda(1, 1) = [A1]: lambda(1, 2) = [B1]
    lambda(2, 1) = [A2]: lambda(2, 2) = [B2]
    nt(1) = [D1]: nt(2) = [E1]
    For K = 1 To 20
        Cells(5 + K, 1) = K
        For I = 1 To 2
            sum = 0
            For J = 1 To 2
                sum = sum + lambda(I, J) * nt(J)
            Next J
            ntplus1(I) = sum
        Next I
        Cells(5 + K, 1 + I) = ntplus1(I)
        nt(I) = ntplus1(I)
    Next I
    Next K
End Sub

The Dim statement defines three arrays: nt is the array of female numbers at time \( t \), ntplus1 is the array of population numbers at time \( t+1 \), and lambda is a 2×2 array for the survival-replacement matrix. In a two-dimensional array, the first index is the row number, and the second is the column number. The program first reads values to the Leslie matrix and initial values to the population vector from Sheet3 of the spreadsheet. The headings and initial values are recorded at respective cells in Sheet3, and the program enters four nested For loops. The first (\( k \)) is the number of generations. The next two loops perform the matrix multiplication. The equation inside the J loop may cause some confusion to people who are new to programming since it appears to violate the rules of algebra. You should remember that the equal sign in BASIC (and several other programming languages) is an assignment operator, not a statement of equality. Thus, the statement means “add the contents of sum to the product of \( nt(J) \) and \( lambda(I, J) \), and store the result in sum.” Since we set sum to zero before entering the loop, it will just contain the sum of the two products when the loop finishes. This comes up again when \( ntplus1(I) \) is assigned the value in sum, and again when \( nt(I) \) is assigned the value in \( ntplus1(I) \). This has the effect of updating the values in \( nt \) to start the calculation for the next generation.

Make sure you understand the program (ask questions if you do not), and then use it to predict population for 20 or more generations. Be sure to put the values for the survival-replacement matrix and the starting populations in Sheet3 before you run the program. Check a few values with hand calculations to be sure the program is doing what you expect it to. Once the program is working, try changing the values in the Leslie matrix so that, say, 40% survive, or there are only 2 juvenile females produced per adult female.

As a final note, a model with just two age classes is not something you need a computer to solve, but this model easily expands to any larger number of age classes, just by increasing the sizes of the arrays.

**Lab 2 Assignment**

Complete the program, including adding codes to sum the adults and juveniles to obtain a total, and make sure it runs properly. Email to the class TA an Excel file containing the VB codes, the outputs in both tabular form (generated by the codes) and graphical form (which you will prepare) from executing the Insect Population model. Also, provide line-by-line interpretations of the codes. (25 pts)