CHAPTER 3 LOOPS & RUNNING TOTALS

This chapter examines the two **built-in loops** that are provided with **VBA** and formally considers the concept of **running totals** introduced in Chapter 2. You will also learn how to control the speed of program execution using a **delay loop** in which the amount of delay is adjusted via a **scroll bar** control.

3.1 Built-in Loops

The use of loops was noted in Chapter 2 to be particularly efficient when repetitive operations are involved. This will be even more true once the incremental analysis of physics problems is considered in Chapter 6 since up to a dozen calculations and operations may need to be repeated anywhere between 10,000 to 5 million times. Although it is possible to continue using the **Do-it-Yourself** loop approach of Chapter 2, two built-in loops are provided that simplify matters : **For…Next** loops, and **Do…Until/ While** loops. The syntax for these looping statements is given in **Appendices 14 & 15** .

The loop constructed in **Exercise #3 pg 2-16 Chap 2** consisted of the code :

```
Dim I As Integer
I = 0:  Row = 4

Beginning:  I = I + 1
    Worksheets("Sheet1").Cells(Row, 3) = I
    Row = Row + 1
If I <= 10 Then GoTo Beginning
```

This can be replaced by either of the following (code specific to the loop mechanism is presented in **bold**)

Do…Until loops

```
Dim I As Integer
I = 0

Do
I = I + 1
Worksheets("Sheet1").Cells(Row, 3) = I
Row = Row + 1
Loop Until I > 10
```

Exit condition

For…Next loops

```
Dim I As Integer


For I = 1 to 10

Worksheets("Sheet1").Cells(Row, 3) = I
Row = Row + 1
Next I
```

[Note : The incrementing step of a **Do** loop is often placed at the end but in this example that would alter the numbers written out.]

> The simple loops of Chapter 2 use a **continuation condition** in the **If** statement at the end of the loop; looping continues provided the condition *is* true. In contrast, the **Do ... loop** uses an **exit condition**; it loops until the condition *becomes* true.

The **For…Next** loop is simpler since it neither requires initialization of loop index **I**, nor the statement **I = I + 1** to increment the loop index (which is done automatically by the **Next I** statement). That being said, the main virtue of the **Do** loop is that complex mathematical expressions can be included in the **exit condition** of the **Loop Until** statement, and this can be a huge advantage in physics applications.

Example # 1 : Mark addition and averaging. The marks of 5 students on two tests (each out of 25) will be added, and the overall group average determined. A **Message Box** warning will be made if the mark on any test exceeds 25, and that particular student will not be counted in average.

```
Option Explicit

Private Sub cmdAdd_Click()
Dim Mark1 As Double, Mark2 As Double
Dim Sum As Double, Total As Double
Dim I As Integer, Row As Integer, N As Integer

Total = 0
Row = 4
N = 5

For I = 1 To 5
   Mark1 = Worksheets("sheet1").Cells(Row, 3)
   Mark2 = Worksheets("sheet1").Cells(Row, 4)
   Sum = Mark1 + Mark2
   If Mark1 > 25 Or Mark2 > 25 Then MsgBox "Check marks, one is > max": _
                                    N = N - 1: Sum = 0: GoTo Skip
   Worksheets("sheet1").Cells(Row, 5) = Sum
Skip:

   Total = Total + Sum
   Row = Row + 1
Next I

Worksheets("sheet1").Range("C10") = "Average mark = " & Format(Total / N, "0.00")
End Sub

Private Sub cmdClear_Click()
Worksheets("sheet1").Range("E4:E8").ClearContents
Worksheets("sheet1").Range("C10").ClearContents
End Sub
```

	A	B	C	D	E
1					
2					
3	Add Marks		Test 1	Test 2	Sum
4			15.5	16.75	32.25
5			8	14.25	22.25
6			20.25	23.75	44
7	Clear Output		23	15.25	38.25
8			10.25	12.75	23
9					
10			Average mark = 31.95		

Initialize **Total** marks

Input marks in box.

Label & value appear automatically

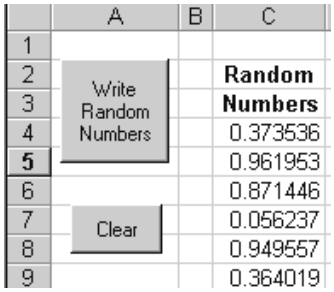
When a mark is rejected that student is not included in the overall average **N = N - 1: Sum = 0** and the write-out of that **Sum** is skipped.

Running (cumulative) total of all students marks.

A **running total** is used to keep track of a cumulative value. In this case the **Sum** of each student's test marks is added to the **Total** of all the previous students' marks. The value of **Total** was initialized to **0** outside the loop since a starting value is needed. The first time through the loop the value of **Sum** is **32.25** which is added to the existing value of **Total** to give a cumulative value of **Total = 32.25**. The next time through the loop the second student has a contribution **Sum = 22.25** which is added to the cumulative value of **Total** to produce **Total = 54.5** , and so on. Each time, the current value of **Total** is read from storage, added to the latest **Sum**, and then the result is stored back into **Total**.

Note that **Sum** does not need to be initialized since it is the sum of two defined values **Mark1** and **Mark2**. If **Total** were not initialized its value would automatically be set to zero when it is "created" by the **Dim** declaration; however, it is good programming practice to initialize all values to avoid situations in which a globally used variable retains a non-zero value.

Example # 2 : Random number generation in a variable loop. This program generates a list of random numbers between 0 and 1 using the **Rnd( )** function described at the end of the chapter and in **Appendix 18**. The number of values to be produced is specified by means of an **Input Box**. A **variable loop** upper limit is used so that the number of iterations can be adjusted.




```
Option Explicit
Dim Row As Integer

Private Sub cmdRandom_Click()
Dim Random_number As Double, I As Integer, N As Integer

N = InputBox("Enter total random numbers to be generated", _
                    "Random numbers between 0 and 1")
Row = 4

For I = 1 To N
   Random_number = Rnd()
   Worksheets("sheet1").Cells(Row, 3) = Random_number
   Row = Row + 1
Next I

End Sub

Private Sub cmdClear_Click()
Range(Cells(4, 3), Cells(Row – 1, 3)).ClearContents
End Sub
```

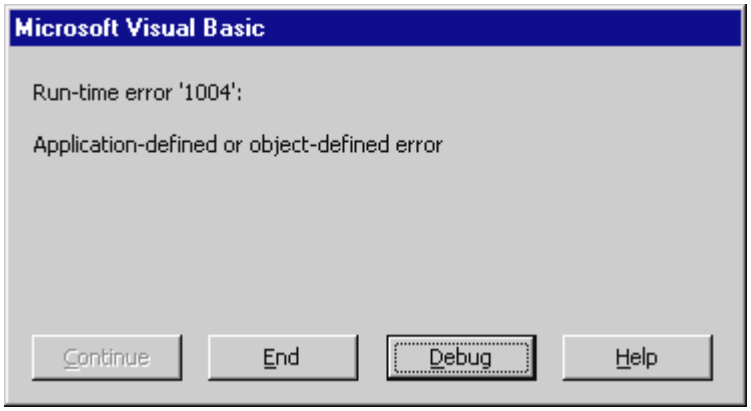
The number of loops can be varied.

Rnd() is a built-in function that generates random numbers between 0 and 1. The argument parentheses are optional. If an argument is included, the way in which the random numbers are generated can be changed. Refer to the end of the chapter; note the reference to the **Randomize** command.

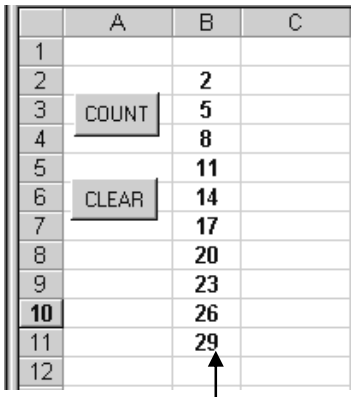
Code for variable ClearContents. The last row containing a random number varies depending on the value of **N**. **Row – 1** is used since **Row** is always increased just before the loop is exited (after the last write). As noted on page 2-15 an alternative approach is :

Range("C4 : C" & Row –1).ClearContents

Caution : The error message shown on the right will result if the **Clear** button is pressed **after** the following circumstances : a) if the user creates a list of random numbers and then resets the program after encountering an error message while making modifications, or b) if the file is closed without clearing the output and then reopened, or c) if an **End** statement is encountered during code execution. In each of these cases the value of **Row** becomes zero as the program is restarted so that **Cells(Row - 1, 3)** refers to the **-1** row which, of course, does not exist. [Also watch out for situations in which the value of **Row** is zero.]



Example # 3 : Counter. Suppose we wished to create a counter that counts by threes starting with the value **2** so that the sequence **2, 5, 8, 11, 14, 17, …** is produced. The basic process is to add **3** to a **running (cumulative) total** that keeps track of the counted values. The repetitive counting is controlled by either a **For … Next** loop or a **Do … Loop** .




```
Option Explicit

Private Sub cmdCount_Click()
Dim Count As Double, I As Integer

Count = 2

For I = 1 To 10
    Count = Count + 3
Next I
End Sub
```

Initialization : Sets starting value of **Count**.

Running total

Note : The code given on the left is not sufficient to produce the output given above. The complete code is given on the next page.

When the program first encounters the **For** statement, loop index **I** is set to a value of **1**. The statement **Count = Count + 3** takes the current value of **Count** (which was initialized to **2**), adds **3**, and then stores the resultant value of **5** back into memory location **Count** . The **Next I** statement increments index **I** by **1** so that its new value is **2** and program control is "looped back" to the **For** statement. The statement **Count = Count + 3** looks up the current value of **Count** (which is now **5**) in its storage location and adds **3** to give a new value of **8** which is then entered back into memory location **Count**. The **Next I** statement again increments index **I** by **1** producing a new value of **3** and program control is "looped back" to the **For** statement. This looping action continues until the loop for which **I = 10** has been executed and then the program moves out of the loop and down to the **End Sub** statement.

In this example index **I** is referred to as a **"dummy" index** since it doesn't enter into any of the computations; its purpose is merely to control loop operation and ensure that the loop is executed **10** times. [The observant student will notice that the output displayed seems to indicate that the loop was only executed **9** times since **2 + 9 x 3 = 29** . This will be discussed shortly].

Alternative **Do … Loop** approaches.

```
Count = 2
I = 1

Do
Count = Count + 3
I = I + 1
Loop Until I >= 11
```

The 3 code lines used in the **For …Next** loop could be replaced with the 5 line **Do … Loop** on the left. The obvious disadvantage is that loop index **I** does not automatically have its initial value set to **1** (which must be done with the statement **I = 1**) nor is the index automatically incremented by **1** (this is done with the line **I = I + 1**). Note that the program does not execute the 11-th loop.

```
Count = 2
Do
Count = Count + 3
Loop Until Count >= 30
```

Dummy index **I** could be eliminated altogether if the approximate final value of **Count** is known. This demonstrates that the **Do** loop does not really require an index, it simply keeps looping until the exit condition is satisfied.

To view the count sequence a write statement is required that returns the values of **Count** back to the worksheet; the location of the write statement within the program code will affect the last value of the sequence that is displayed. The complete code for this exercise is given below : existing code is in *italics*, new is shown in **bold**.

```
Option Explicit

Private Sub cmdCount_Click()
Dim Count As Double, I As Integer
Dim Row As Integer

Count = 2
Row = 2
For I = 1 To 10
    Worksheets("Sheet1").Cells(Row, 2) = Count
    Count = Count + 3
    Row = Row + 1
Next I
End Sub

Private Sub cmdClear_Click()
Range(Cells(2, 2), Cells(11, 2)).ClearContents
End Sub
```

In order to display the starting value of **2** the write statement has been placed before the actual count operation : **Count = Count + 3**. This means that the **Count** value generated in each loop is not actually written on the worksheet until the next loop, and more importantly, the final value is never displayed. How could the program be modified to show all counted values as well as the starting value?

Infinite Loops : Improper programming, such as not providing an exit condition for a **Do… Loop** that can actually be satisfied, can lead to infinite loops in which execution is stuck within a loop. If your screen appears frozen, you may be trapped in such a loop. Try **Ctrl + Break** , which interrupts loop execution, before **Ctrl + Alt + Del** . [Remember to press the **Reset/Stop** button before leaving the **VB Editor**, or click in and out of **Design Mode** on the worksheet.]

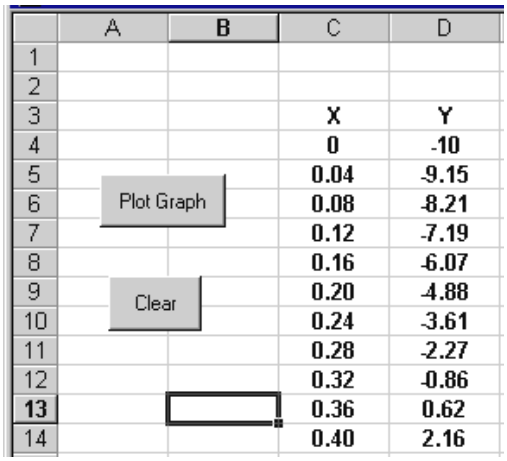
Examples : A **Do…Loop** that counts by twos starting with the number 3 will never reach any even values. If the loop is to be exited when **Count** reaches 10 it will never happen. In general, it's always safer to exit using a " >=" condition rather than an " =" condition.

A **Do…Loop** that is exited when the loop index reaches or exceeds a certain value will run continuously if the statement that increments the index is missing.

If the loop index in either a **Do…Loop** or a **For ..Next** loop is initialized to some value inside the loop, the exit condition will never be reached since the variable is stuck at the particular value.

Large File Sizes : If your program gets caught in an infinite loop that writes values back onto the worksheet you may end up with a column of numbers that extends down 20,000 lines or more. This will produce a large file that may not fit on your diskette. You will need to use **.Clear** and not just **.ClearContents** to completely eliminate the output.

Example # 4 : Function Evaluation & Graph Plotting. Evaluating a function **Y = f(X)** over a range of values of the independent variable **X** is easily accomplished using a loop. This program uses a **For ….Next** loop to change variable **X** over 100 different values starting at 0 and increasing in pre-set increments of **0.04** . Each time the function is evaluated the values of both **Y** and **X** are written back onto the worksheet in columns **C** and **D** . Finally, the **ChartWizard** is used to add an **XY (Scatter)** graph. In this example the function **$Y(X) = -10X^3 + 30X^2 - 10$** will be used, although any function whatsoever could be substituted.



	A	B	C	D
1				
2				
3			X	Y
4			0	-10
5			0.04	-9.15
6			0.08	-8.21
7			0.12	-7.19
8			0.16	-6.07
9			0.20	-4.88
10			0.24	-3.61
11			0.28	-2.27
12			0.32	-0.86
13			0.36	0.62
14			0.40	2.16

Open and save a new workbook. Add two buttons : Name = **cmdGraph** , Caption = **Plot Graph** ; Name = **cmdClear**, Caption = **Clear**. Type in the titles **X** and **Y** into cells **C3** and **D3**.

Go into **Design Mode**, double click on each control, and add the code given below.

```
Option Explicit

Private Sub cmdGraph_Click()
Dim X As Double, Y As Double, Row As Integer
Row = 4
```

Instead of using a dummy index the loop is being controlled by the value of the independent variable **X**

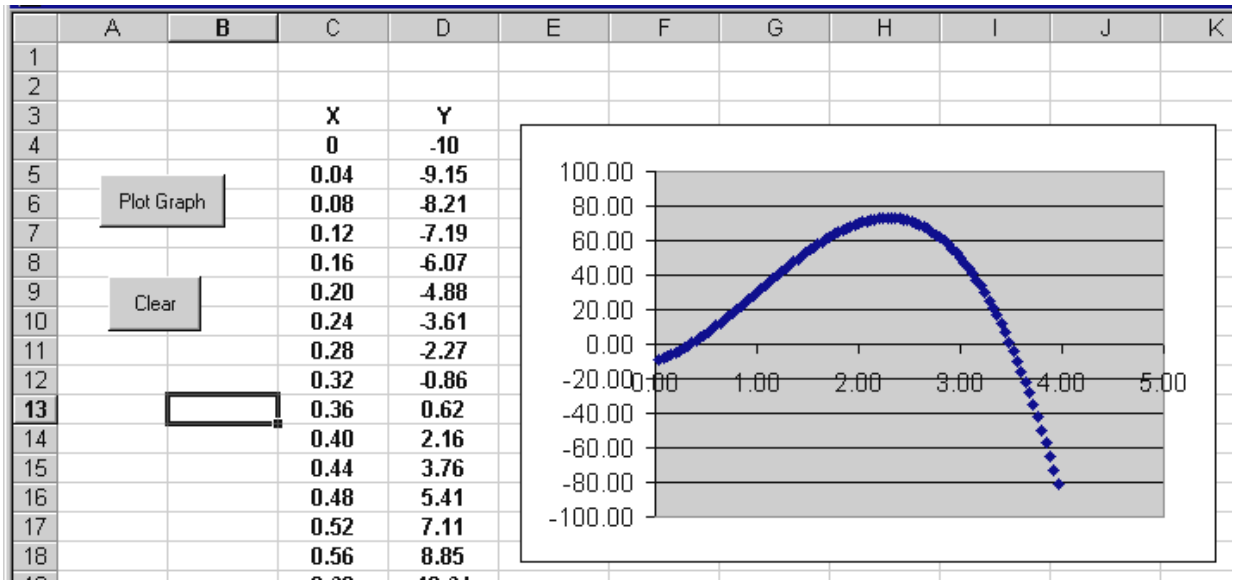
```
For X = 0 To 4 Step 0.04

    Y = -10 + 20 * X + 30 * X ^ 2 - 10 * X ^ 3

    Cells(Row, 3) = X
    Cells(Row, 4) = Y
    Row = Row + 1
Next X

End Sub
```

The **Step** keyword is used to increase the loop index by values other than **1**. Here **X** will increase by the amount **0.04** over the range **0** to **4**. When **Step** is omitted the default increment is **1**.

```
Private Sub cmdClear_Click()
Range(Cells(4, 3), Cells(103, 4)).ClearContents
End Sub
```

Note that there are 100 rows starting with row 4 and ending on row 103.

Test your program. Once data is produced use the **ChartWizard** to add an **XY (Scatter)** graph. Remember to highlight the cells only containing the data (but not the titles **X** and **Y**) . Delete the graph legend. Notice that the **ChartWizard** requires that the horizontal axis data be placed in the left column. [You might, of course, specify the **data range** to the **Chart Wizard** and locate the data however you wish. Refer to **Appendix 29**.]



	X	Y
4	0	-10
5	0.04	-9.15
6	0.08	-8.21
7	0.12	-7.19
8	0.16	-6.07
9	0.20	-4.88
10	0.24	-3.61
11	0.28	-2.27
12	0.32	-0.86
13	0.36	0.62
14	0.40	2.16
15	0.44	3.76
16	0.48	5.41
17	0.52	7.11
18	0.56	8.85

You may find it necessary to right-click on the graph axes to set the maximum and minimum values of each axis for certain functions and/or certain intervals (in its existing form the graph axes are on **AutoScale**).

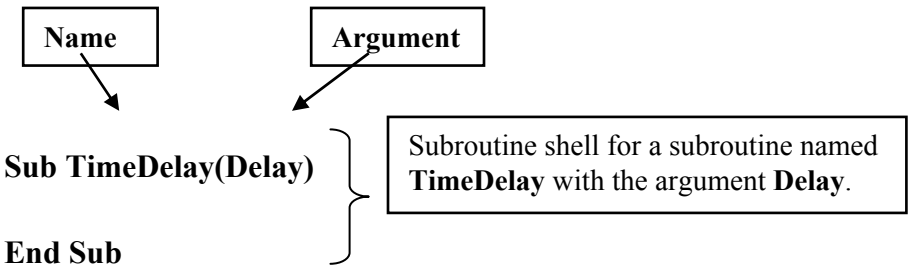
3.2 Using a Delay Loop to Slow Down the Speed of Program Execution

3.2.1 Stand-Alone Subroutines

Not all subroutines are associated with controls; in fact in complex programs there are often more subroutines that exist by themselves than those tied to controls. A subroutine that "stands alone" is used to perform certain functions that may be needed at different points in the main program. Rather than repeating the same code, which may be lengthy, a separate subroutine is created and accessed via a **Call** statement. The benefits are that the main program is reduced in size and becomes more readable and organized, and the functionality of the various parts usually becomes clearer. Creating a separate subroutine is likewise useful when the same operations are required in different controls of the same application. Stand-alone subroutines are frequently break down a project when a team is developing a program; the various components can be identified, developed, and tested independently by team members as subroutines, and then called from the main program which usually takes on the appearance of a very general **block diagram**. [A block diagram shows the functional parts of the process without giving the details; it gives the general structure without specific content. Those of you taking the Electronics course might consider the block diagram of the typical power supply.]

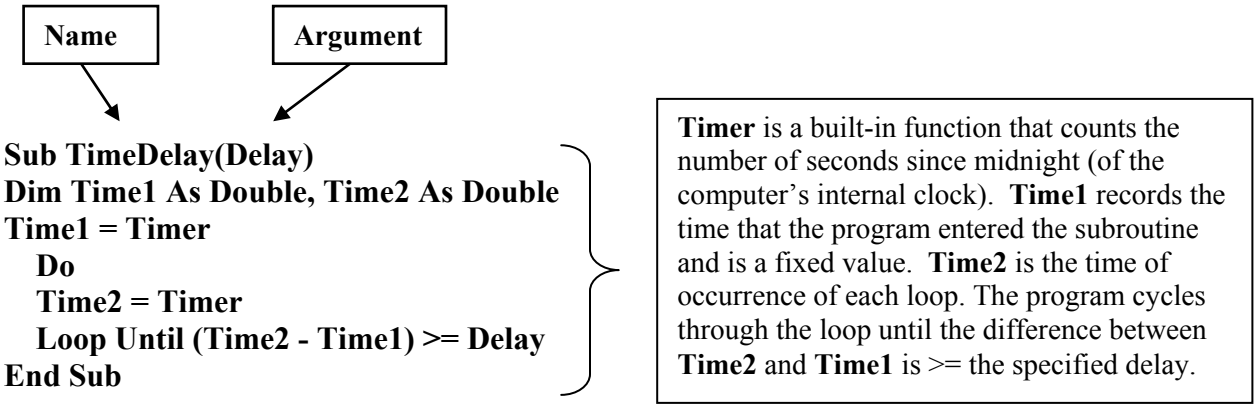
A good example is the spherical lens ray diagram application which you may later encounter. Subroutines **Detect** and **Refract** are constructed to detect when the ray is entering a different medium, and to determine the refracted angle. Although each subroutine is used only once in the main program, the removal of about 20 lines of somewhat complicated code which is replaced by two quite descriptive statements **Call Detect( )** and **Call Refract( )** makes the program considerably more readable. Similarly, you may encounter a coalescing matter application in which 3 dense masses are allowed to move together under their mutual gravitational attraction. The periodic collisions that occur are modelled in a separate **Collision** subroutine of approximately 50 lines which is called in 3 **If** statements. In this case the main program appears much less intimidating with the **Call Collision( )** reference than with the actual code.

A stand- alone subroutine is created within a shell with the name and arguments at the beginning and **End Sub** at the end (recall the book front & rear cover analogy). The major difference with control associated subroutines is that there is no event appended to the name line (such as **_Click( )**).



3.2.2 Delay Loops

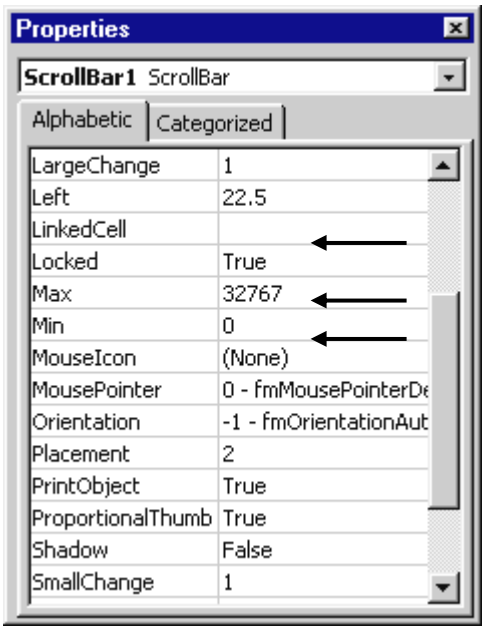
Program execution can be slowed down by "calling" a **delay loop**. The effect of introducing delay will allow the user to follow certain events, such as reading and writing, on the worksheet. This gives the illusion of "watching" the computer operate. Later, delay loops will be used to animate trajectory plots that will allow us to follow an object's motion in the X-Y plane. A typical delay loop is found in **Example # 5** which uses the following subroutine to waste time, or create delay, by causing execution to occur in a loop that does nothing but loop, not unlike a dog chasing its tail. Once an amount of time **Delay** has elapsed the loop is exited. The value of **Delay** will be controlled by a **Scroll Bar** (although it could be read-in directly from the worksheet).



The subroutine is entered, or activated, by the statement **Call TimeDelay(Delay)** in the main program which transfers program execution to the subroutine. Once the **End Sub** line in the subroutine is encountered execution is returned to the main program.

3.2.3 The Scroll Bar Control

Scroll Bar controls are activated by the event of changing the scroll bar slider which causes the code in the associated subroutine to be executed. In fact all of the programs created so far could have been located in scroll bars subroutines and run by moving the slider. The main appeal of a scroll bar control, however, is that it has a **.Value** property that is based on the location of the slider. This means that variables can be given values that are obtained from the current setting of a scroll bar, and when the scroll bar is changed the value of the variable is changed as well. The important properties, from our point of view at least, are the **Max**, **Min**, and **LinkedCell** properties. Both the **Max** and **Min** can be set to negative integer values. The **Linked Cell** property is interesting as it enters the scroll bar value into the worksheet cell with which it has been linked.



Although a scroll bar can only have integer values, decimal values can be created by dividing the output value by a suitable number. In **Example # 5** that follows the scroll bar has been named **scbDelay** ; **Max** has been set to 50 and **Min** left at 0. The value of variable **Delay** is obtained from the scroll bar as : **Delay = scbDelay.Value / 100** (since the **.Value** property is the default, the appendage **.Value** can be omitted). The resulting values of **Delay** range between 0 and 0.5 in steps of 0.01.

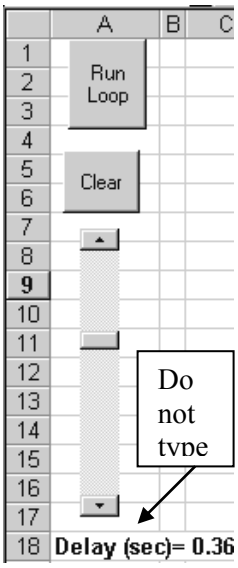
Example # 5 : **Adding a TimeDelay Loop**. This program modifies **Example # 4** from Chapter 2 with the addition of a delay loop so that the index value written in each coloured cell appears at a regular rate controlled by a **Scroll Bar** setting. The **Scroll Bar** has been named **scbDelay**. Its **Max** property (in the **Properties** window) has been set to **50** (the **Min** should be **0**). The main feature of a scroll bar control is that it has an *integer* value that depends on the slider location. If the slider is moved three-quarters of the way along, then the value of the scroll bar will be three-quarters of its maximum (assuming that the minimum value has been set to **0**). To obtain decimal values the scroll bar value has been divided by **100**.

Since variable **Delay** appears in more than one subroutine it must be declared globally under the **Option Explicit.**

```
Option Explicit
Dim Delay As Double

Private Sub cmdLoop_Click()
Dim I As Integer, Row As Integer
Row = 4
Delay = scbDelay.Value / 100
Worksheets("sheet1").Range("A18") = "Delay (sec)= " & Delay
Cells(1, 1).Select
```

Dividing by 100 converts the value of the scroll bar setting (which is between 0 and 50) to a value of **Delay** that is between 0 and 0.5.

Excel 97 "workaround". See pg 2-17




```
For I = 1 To 10
   Worksheets("Sheet1").Cells(Row, 3) = I
   Worksheets("sheet1").Cells(Row, 3).Interior.ColorIndex = 3
   Worksheets("sheet1").Cells(Row, 3).Font.Bold = True
   Worksheets("sheet1").Cells(Row, 3).HorizontalAlignment = xlCenter
   Call TimeDelay(Delay)
   Row = Row + 1

Next I
End Sub

Private Sub scbDelay_Change()
Delay = scbDelay.Value / 100
Worksheets("sheet1").Range("A18") = "Delay (sec)= " & Delay
End Sub
```

The **TimeDelay** subroutine is "called" meaning that execution is transferred to the set of code instructions contained in the subroutine named **TimeDelay**. In this case the value of the argument **Delay** is passed over to the **TimeDelay** subroutine. This statement could be replaced by the line : **TimeDelay** . That is, no "Call", no argument, and no parentheses.

TimeDelay is a subroutine, or set of instructions, that are not associated with any control. The code in the subroutine is executed by calling the subroutine from any other subroutine.

```
Sub TimeDelay(Delay)
Dim Time1 As Double, Time2 As Double
Time1 = Timer
   Do
   Time2 = Timer
   Loop Until (Time2 - Time1) >= Delay
End Sub
```

Timer is a built-in function that counts the number of seconds since midnight (of the computer's internal clock). **Time1** records the time that the program entered the subroutine and is a fixed value. **Time2** is the time of occurrence of each loop. The program cycles through the loop until the difference between **Time2** and **Time1** is >= the specified delay.

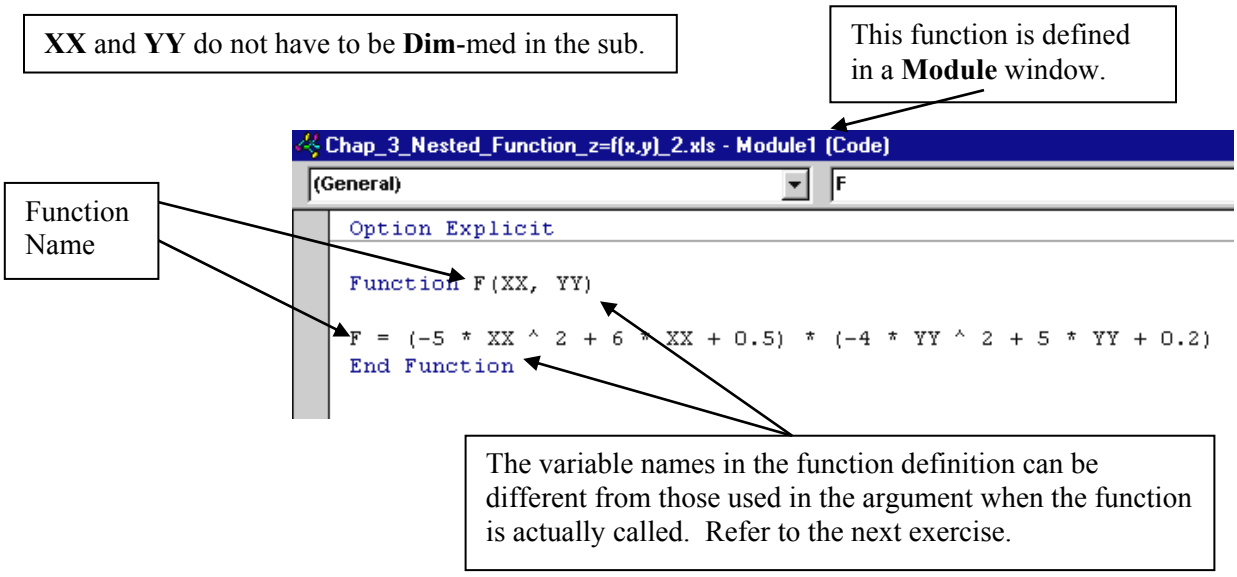
```
Private Sub cmdClear_Click()
Range(Cells(4, 3), Cells(14, 3)).Clear
Range("A2").Select
Worksheets("sheet1").Range("C3:C13").Interior.ColorIndex = xlNone
End Sub
```

Move the scroll bar and observe the change in the value of **Delay**.

3.3 Custom Functions & Nested Loops

A **custom function** is a user defined function (meaning that it's a function that you've created). If the custom function is only to be used by subroutines in the Code window, it can be defined in the **Code Window** along with the rest of the subroutines. If , however, the function is to be used both on the Worksheet and in the Code window then it must be entered in a **Module** [**VBA Editor ⇒ Insert Menu ⇒ Module** , or right -click on a blank area of the **Project Explorer ⇒ Insert ⇒ Module**]. (Note: defining the function in a **Module** is useful as it can be tested in a cell on the worksheet.)

A custom function is "registered" by typing the keyword **Function** followed by the function name including any arguments in parentheses. The **End Function** line, which is the last line of any function, should appear automatically once the first line is entered. The function itself is entered by forming an equation with the function name only appearing on the left side (without the arguments or parentheses!). The function $\mathbf{z = (-5x^2 + 6x + 0.5)(-4y^2 + 5y + 0.2)}$ has been given the function name "**F**" in the example below :



Certain names are reserved names and cannot be used in custom functions; for example, **Sin** is reserved for the built-in sine function, and if you attempt to define functions **A1**, **B1**, **C1**, **AB1**, **FF1**, **GG1** etc. for use on the spreadsheet an error will occur since these names are reserved for cell names.

Custom functions differ from stand-alone subroutines as they are used to return a single value that is assigned to the variable on the left side of the equation in which the function is used. The next example uses the custom function defined above in the statement : **Z = F(X, Y)**

Nested loops are loops that are inside other loops. They are particularly useful in changing two or more variables in a systematic way such as when a search is being performed over 2 or more sets of values, or when a function is being evaluated over a range of values (as in the next exercise).

Example # 6 : Evaluating a function of 2 variables using two nested loops. The value of the function $z = (-5x^2 + 6x + 0.5)(-4y^2 + 5y + 0.2)$ will be calculated and written out for **X : 0 → 1** and **Y : 0 → 1** at increments of **0.1** . The function has been defined in a module as shown on the previous page. The function is "called" by the statement **Z = F(X, Y)** .

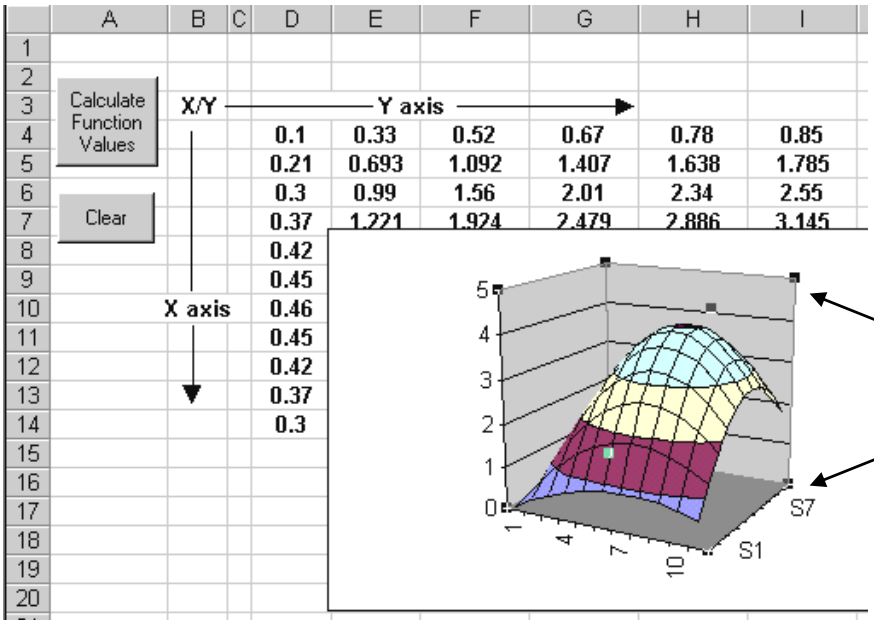
```
Option Explicit

Private Sub cmdCalculate_Click()
Dim X As Double, Y As Double, Z As Double
Dim Row As Integer

Row = 4

For X = 0 To 1 Step 0.1

   For Y = 0 To 1 Step 0.1
      Z = F(X, Y)
      Worksheets("Sheet1").Cells(Row, 4) = X
      Worksheets("Sheet1").Cells(Row, 5) = Y
      Worksheets("Sheet1").Cells(Row, 6) = Z
      Row = Row + 1
   Next Y

Next X

End Sub

Private Sub cmdClear_Click()
Range("D4:F124").ClearContents
End Sub
```

The variable names in the argument can be can different from those in the function definition since only values are passed through the argument.

	A	B	C	D	E	F
1						
2						
3				X	Y	Z
4	Calculate Function Values			0	0	0.1
5				0	0.1	0.33
6				0	0.2	0.52
7	Clear			0	0.3	0.67
8				0	0.4	0.78
9				0	0.5	0.85
10				0	0.6	0.88
11				0	0.7	0.87
12				0	0.8	0.82
13				0	0.9	0.73
14				0	1	0.6
15				0.1	0	0.21
16				0.1	0.1	0.693
17				0.1	0.2	1.092
18				0.1	0.3	1.407
19				0.1	0.4	1.638
20				0.1	0.5	1.785
21				0.1	0.6	1.848

In the inner loop the value of **Y** is varied while the value of **X** is held constant (examine the output). Once **Y** is varied over the range **0** to **1**, the inner loop is exited and the value of **X** is increased by **0.1** in the **outer loop**. With this new value of **X** the inner loop again varies the value of **Y** over the range **0** to **1**, and so on.

Example # 7 : **Surface Plots**. The function evaluation of **Example # 6** will be reorganized so that there is an implicit **X-Y** coordinate system on the worksheet such that each cell represents an increase of **0.1** in the **X** and **Y** values. The value of **Y** is increasing across the columns, and the value of **X** is increasing down the rows. Only **Z** values will be displayed over a rectangular array. With the data in this form a **Surface** plot of the function can be made.



The X and Y axes on the have meaningless values (items & series). Refer to chapter 5.

Click on the graph area so that black squares appear at the corners. Clicking and holding any of these squares allows the display to be rotated in 3-D.

```
Option Explicit

Private Sub cmdCalculate_Click()
Dim X As Double, Y As Double, Z As Double
Dim Row As Integer, Column As Integer

Row = 4

For X = 0 To 1 Step 0.1

   Column = 4
   For Y = 0 To 1 Step 0.1
      Z = F(X, Y)
      Worksheets("Sheet1").Cells(Row, Column) = Z
      Column = Column + 1
   Next Y
Row = Row + 1
Next X

End Sub

Private Sub cmdClear_Click()
Range("D4:N14").ClearContents
End Sub
```

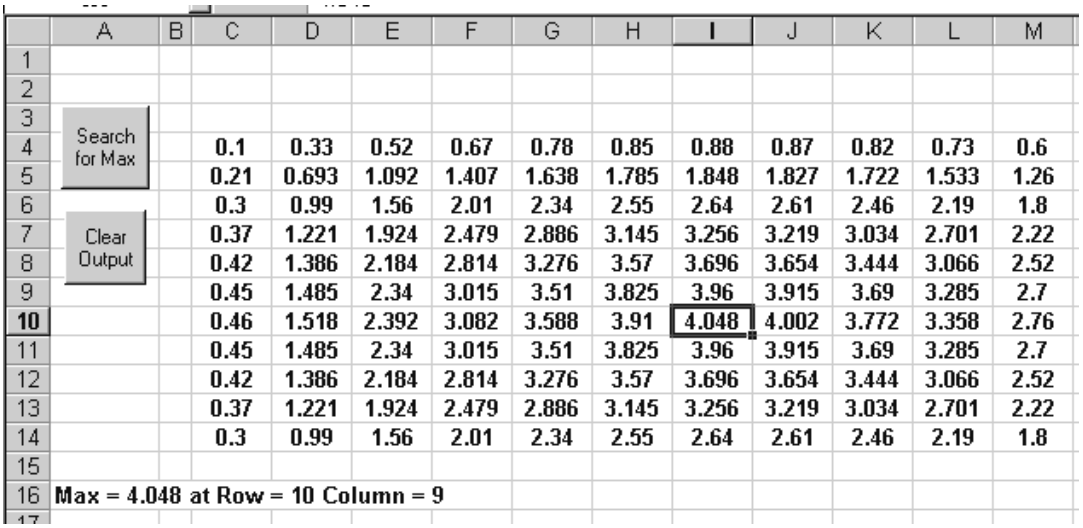
The single adjustment here is that a **Column** index has been included which allows the data to be displayed as a rectangular array.

The changing inner loop **Column** index causes the **Z** values to be written across a single row. The value of **Row** is constant in the inner loop. When one row has been filled the **Row** index is incremented to move down to the next row, and the **Column** index is reset to the left hand value of **4**. An electron beam scanning across an oscilloscope, or TV, could be modeled in this way.

Note : This example has varying Y values generated across each row which does not produce a plot a right-handed coordinate system. A better method is presented in Chapter 5, section 5.6 (pg 5-14).

3.4 Searches

Example # 8 : Searching for a maximum value. This program will search an array of data (copied from the previous surface plot exercise) and use an **If** statement to find the maximum value.



	A	B	C	D	E	F	G	H	I	J	K	L	M
1													
2													
3													
4			0.1	0.33	0.52	0.67	0.78	0.85	0.88	0.87	0.82	0.73	0.6
5			0.21	0.693	1.092	1.407	1.638	1.785	1.848	1.827	1.722	1.533	1.26
6			0.3	0.99	1.56	2.01	2.34	2.55	2.64	2.61	2.46	2.19	1.8
7			0.37	1.221	1.924	2.479	2.886	3.145	3.256	3.219	3.034	2.701	2.22
8			0.42	1.386	2.184	2.814	3.276	3.57	3.696	3.654	3.444	3.066	2.52
9			0.45	1.485	2.34	3.015	3.51	3.825	3.96	3.915	3.69	3.285	2.7
10			0.46	1.518	2.392	3.082	3.588	3.91	4.048	4.002	3.772	3.358	2.76
11			0.45	1.485	2.34	3.015	3.51	3.825	3.96	3.915	3.69	3.285	2.7
12			0.42	1.386	2.184	2.814	3.276	3.57	3.696	3.654	3.444	3.066	2.52
13			0.37	1.221	1.924	2.479	2.886	3.145	3.256	3.219	3.034	2.701	2.22
14			0.3	0.99	1.56	2.01	2.34	2.55	2.64	2.61	2.46	2.19	1.8
15													
16	Max = 4.048 at Row = 10 Column = 9												

```
Option Explicit

Private Sub cmdSearch_Click()
Dim Z As Double, Max As Double, Row_Max As Integer, Col_Max As Integer
Dim Row As Integer, Column As Integer

Max = – 1 E +27

For Row = 4 To 14
    For Column = 3 To 13
        Z = Worksheets("Sheet1").Cells(Row, Column)
        If Z > Max Then Max = Z: Row_Max = Row: Col_Max = Column
    Next Column
Next Row

Worksheets("Sheet1").Range("A16") = "Max = " & Max & " at Row = " & _
                                    Row_Max & " Column = " & Col_Max
Worksheets("Sheet1").Cells(Row_Max, Col_Max).Select
End Sub

Private Sub cmdClear_Click()
Worksheets("Sheet1").Range("A16").ClearContents
End Sub
```

Max needs to be initialized to a value that is not inadvertently the largest of the set. If we were searching over very small values, even this small starting value of **Max** would need to be reduced.

Notice that the loop indices are no longer **X** & **Y** values, but are row & column locations used to read each cell whose value is being examined.

When a new **Max** value has been detected it is made the **Max** for further comparison (using **Max = Z**), and its **Row** & **Column** locations are stored in **Row_Max** & **Col_Max**.

The cell with the maximum value is selected so that a box appears around it in the array.

3.4.1 Searches for Maximum, Minimum, and Closest Values

There are 3 fundamental searches that will be of use in our simulations : the search for the **maximum value**, **minimum value**, and **closest value**. Each search requires an initial comparison value, a loop (or loops) to vary the values searched, a single outcome **If** statement, and replacement of values such that the previous best solution is replaced by the latest best solution. One of the important issues here relates to the significance of negative signs which have an entirely different meaning for scalars as opposed to vectors. The searches described in this chapter will be assumed to concern scalar quantities so that a negative value is always smaller than a positive value (–1000 is less than + 5). The search statements will be modified in Chapter 6, section 6.7 to account for vector quantities such as displacement, velocity, net force, acceleration, and momentum.

Search for maximum. Exercise # 8 presented a search for the maximum value in an array of numbers using the statement below in which each number to be examined is stored in variable **Z**.

If Z > Max Then Max = Z: Row_Max = Row: Col_Max = Column

The value of **Max** was initialized to **Max = – 1 E +27** outside both loops. Each value of **Z** that exceeds the current **Max** then replaces the old value in the statement : **Max = Z** . This replacement process causes a continual updating of the maximum. The corresponding row and column location of each of the best solutions is also stored in variables **Row_Max** and **Col_Max** . The choice of the "large" negative number **Max = – 1 E +27** (which is really a very small number) for a starting value is significant. The initial value for the *maximum* must be selected *smaller* than the typical expected values for the situation in order that the starting value is not inadvertently the largest value. For example, an arbitrary starting value of **Max = 10** in this situation would be greater than all of the actual data and would become the maximum by default. Similarly, if we were searching for the largest electric potential in the neighbourhood of a number of negative charges and started with **0** volts, this value would always be the maximum since all potentials in this situation are negative numbers. [Likewise for the true gravitational potential which is by definition negative.]

Search for minimum : *Minimum* values are found by reversing the inequality of the maximum search and starting at a *large* initial value so that the starting value is not inadvertently the smallest. The minimum value in Example # 8 could be found using :

If Z < Min Then Min = Z

where **Min** is initialized outside both loops to **Min = + 1E 20** .

Search for closest value. The value closest to a specific target value is found by taking the difference between each test value and the target value to find which test value produces the smallest difference (which at best can be zero). The starting difference needs to be *large* to prevent it from being the closest.

If Abs(Test_Value – Target_Value) < Small Then Closest = Test_Value : _
Small = Abs(Test_Value – Target_Value)

Small and **Closest** were initialized to **Small = +1E+27: Closest = 222222** (anything large relative to the numbers involved) before any search loop is entered. Absolute values are required to prevent "large" negative differences such as – 1000 from appearing to be small. You can verify that the difference **Abs(Test_Value – Target_Value)** accurately treats both positive and negative test values and target values. The starting value of **Closest** is a "dummy" value which indicates a failed search if it appears at the output.

3.4.2 Radar Searches

A recent course project focused on developing a guidance system for a robot that was moving autonomously through a maze; that is, not being controlled by a driver but moving according to some algorithm which typically might involve targeting the farthest "visible" point. One approach is to develop a **radar search** subroutine that sends out a **virtual probe** in a variety of directions and determines which direction allowed the probe to move the farthest before encountering a barrier. The process of launching a virtual probe at different angles is more conveniently described by polar coordinates (r, θ) rather than the usual Cartesian coordinates (X, Y). The description "radar search" is applied due to the similarities with a radar device which sends out electromagnetic pulses and detects which pulses are reflected back indicating the presence of an object. The time interval between when the pulse was sent and when it returns is used to establish the distance to the target ; obviously the angle at which the pulse was sent defines the angular location of the target. The presence of a wall in a maze can be detected via an **If** statement similar to a search for a maximum value.(Refer to the Ball-in-a-Box application of Chapter 6).

The idea of sending out a probe and detecting the presence of a target is used in the planar refraction problem at the end of this chapter.

3.4.3 Polar Searches

Polar searches are operationally the inverse of radar searches. A radar search selects a particular angle and then launches a probe outward along that angle increasing the distance "**r**" from the origin until the "target" is detected. The process is repeated as the angle is varied. A polar search selects a particular distance **r** and then scans, typically over 360^o, at the fixed **r**. The value of **r** is systematically increased and the search proceeds over 360^o for each new value of **r**. [A good analogy to a polar search is to consider a search-and-rescue plane that is searching around a base camp in a forest. The plane flies concentric circles at ever increasing distances from the base camp.]

Problem # 25 in this chapter uses a modified polar search (for only a single value of **r**) to create an equipotential plot for a distribution of charges. The same technique can be applied to produce topographical contour lines in the case of gravitational fields.