3.5 The DoEvents Loop : Interacting With the Program as it Runs

Example # 5 wrote a set of integers down a column with a fixed time delay set by a scroll bar in between each number. You may have wondered whether it's possible to change the delay between numbers "on-the-fly"; that is, while the program is executing. What would be required is that the program become interactive in a very primitive way. When a control is activated execution normally continues until the **End Sub** line is reached so that the "focus" (or attention) of the processor is entirely on the particular routine. The screen cursor appears as a bee (or some other symbol) to indicate that the machine is busy working, and during this period the screen appears frozen to the user since nothing can be entered into the cells, and neither the VBA controls nor the worksheet menus and icons operate.

The **DoEvents** command very briefly interrupts program operation to allow the processor to check if other tasks need to be attended to; it literally means to "do any other events that have occurred"[1]. The interruption allows the VBA controls to "come alive", or be responsive, for a short moment, which provides the user with an opportunity to activate other controls which change values used in the subroutine that was interrupted. For example, the user might adjust the scroll bar in **Exercise # 5** to alter the time delay so that **Delay** will have a different value after the **DoEvents** interruption is over. Unfortunately, although the user sometimes gets lucky, the "window of opportunity" provided by a single **DoEvents** command is generally too small to give the user much time to act. Multiple **DoEvents** commands, though, can be strung together in a loop with the purpose of providing the user with a longer window of opportunity in which they can activate the scroll bar or button. [Even multiple **DoEvents** loops will not be sufficient to allow the user to type a new value into a cell, the window is simply not long enough.] With **DoEvents** present the "busy bee" cursor will intermittently change to the active arrow when hovered over a control.

```
For JJ = 1 To 50
  DoEvents: DoEvents
Next JJ
```

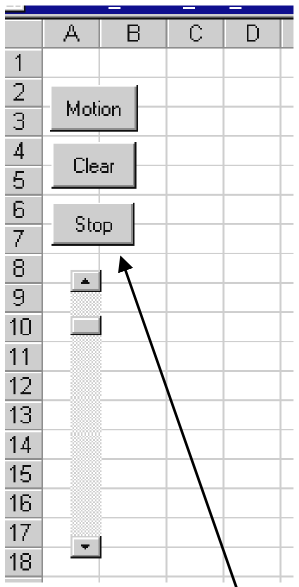
Writing values back onto the worksheet with a delay in between is not an action that requires a **DoEvents** command; however, if these values were to be plotted to create an animated graph [as with **Application # 4a)** , Chapter 5] the **DoEvents** command is essential to allow the graph to be updated. Without **DoEvents** the graph would be plotted all at once after subroutine execution had been completed.

In summary : a single **DoEvents** command must be included to allow a graph to be updated as the points are produced, and a multiple **DoEvents loop** is required if the user is to be allowed to change program parameters via activating controls.

A multiple **DoEvents** loop is introduced next in the **Random Colour Generator**, **Example # 9**.

[1] Remember that "event" has a well defined meaning; controls are triggered, or activated, by events such as clicking, double-clicking, moving a slider, moving the mouse, pressing a key, etc. Updating a graph is also considered as an event that needed to be tended to.

Example # 9 : Random Colour Generator. This exercise uses the random number generator function **Rnd ( )** twice during each iteration through a loop to create random integers between 3 an 20. One of these integers is assigned to variable **Row** to specify a row location; the second random integer is assigned to variable **Column** to specify a column location. The two random integers thus determine a random location of some cell in the range (3,3) to (20, 20). Each randomly selected cell is filled with the number **I** of an index **that** is also used to set the **ColorIndex** of the cell. A scroll bar sets the amount of **Delay** In each loop. A description of the **Rnd ( )** function is available at the end of the chapter and in **Appendix 18**.



A **Stop** button is included since not every user is aware of **Ctrl + Break** .

Highlight columns **C** through **T** (click and hold the **C** label and drag over to the **T** label. Release the left mouse button and click on the right button : select **Column width …** from the menu that appears and set the column width to **4**.

Open and save a new workbook. Add 3 buttons :
Name = **cmdMotion** , Caption = **Motion** ;
Name = **cmdClear** , Caption = **Clear** ;
Name = **cmdStop** , Caption = **Stop** .

Add a scroll bar : Name = **scbDelay** and set its **Max** property to **40**.

Add the following code in the appropriate subroutine :

```
Option Explicit
Dim Delay As Double, bstop As Boolean

Private Sub cmdStop_Click()
bstop = True
End Sub

Private Sub scbDelay_Change()
Delay = scbDelay / 40
Range("A22") = "Delay (sec) =  " & Delay
End Sub

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

Private Sub cmdClear_Click()
Range(Cells(3, 3), Cells(20, 20)).Clear
End Sub
```

Same subroutine as in **Example # 5.**

```
Private Sub cmdMotion_Click()
Dim JJ As Integer, upper As Integer, lower As Integer
Dim row As Integer, column As Integer, I As Integer, J As Integer

upper = 20
lower = 3
```

Upper and **lower** define the limits of the random number generator.

```
J = 1
I = 1
bstop = False

Do
```

This expression produces random numbers in the range between **upper** & **lower**. Refer to the **Rnd Function** description at the end of the chapter.

```
    row = Int((upper - lower + 1) * Rnd + lower)
    column = Int((upper - lower + 1) * Rnd + lower)
```

Function **Int ( )** takes the <u>integer part</u> of the argument.

```
        For JJ = 1 To 50
        DoEvents
        Next JJ
```

This loop repeatedly interrupts program execution using the **DoEvents** command to allow the user to change the value of the **Delay**, or to press the **Stop** button and terminate loop execution.

```
    Delay = scbDelay / 40
    Call TimeDelay(Delay)
    Cells(row, column) = I
```

The randomly generated **row** and **column** values specify which cell is to be filled.

```
        Cells(row, column).Select
        Selection.Interior.ColorIndex = I
```

You might want to experi-ment with using the **RGB** function to colour the cell.

```
    Range("A22") = "Delay (sec) =  " & Delay
    J = J + 1
    I = I + 1
    If I = 55 Then I = 1
Loop Until J = 1000 Or bstop = True

End Sub
```

Index **J** controls the total iterations.

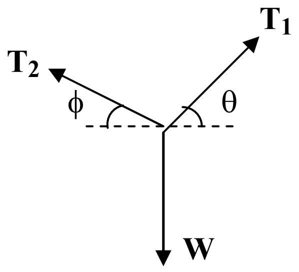
Index **I** sets the interior colour of the cell. The max value is 55 so that **I** must be periodically reset.

The **bStop = true** condition will occur if the **Stop** button has been pressed which allows early exit of the loop.

Once the execution of any program begins it will continue until all instructions have been completed (unless the user presses **Ctrl + Break**). However, it will occasionally be useful to be able to change a parameter value during the course of program execution without causing the program to stop. For example, in this program the user may wish to vary the value of **Delay** by adjusting the scroll bar. The <u>**DoEvents** command momentarily interrupts program execution to allow the processor to check if any other events require attention</u>. Unfortunately, the interruption caused by a single **DoEvents** command is so brief that it is unlikely to coincide with the user's changing of the scroll bar slider. To increase the probability of this happening, a loop that causes the execution of fifty **DoEvents** has been included. Similarly, pressing the **Stop** button has no effect without the presence of **DoEvents**; and as it stands, you may have to click on the **Stop** a half dozen times before the program responds. Repeatedly clicking on a cell on the worksheet will also eventually halt the program.

Application # 3 : Two Variable Search for Optimum Angles in a Hanging Weight Lab

Translational Equilibrium occurs when $\Sigma\ Fx = 0$ and $\Sigma\ Fy = 0$ for an object in which the forces are concurrent (so that rotational effects can be ignored). One of the more basic equilibrium problems is that of a hanging weight supported by two cables and which has the force diagram shown below.



$$\boxed{\begin{aligned} \Sigma\ F_x &= T_2 \cos\phi - T_1 \cos\theta = 0 \quad \ldots(1) \\ \Sigma\ F_y &= T_2 \sin\phi + T_1 \sin\theta - W = 0 \quad \ldots(2) \end{aligned}}$$

An experimental simulation of this problem uses hanging masses connected over two pulleys to create the two tensions, plus a hanging mass connected to the central knot to represent the hanging weight **W**. The first part of such an experiment usually consists of verifying that the system is indeed in equilibrium by adding up the force components as : $\Sigma\ F_x = T_2 \cos\phi - T_1 \cos\theta$ and $\Sigma\ F_y = T_2 \sin\phi + T_1 \sin\theta - W$. Unfortunately, this experiment can be quite sensitive to measurement error in the two angles[1] . One of the purposes of the application that follows, apart from being a good example of nested loops used to search for an optimum value, is to allow students to input their experimental data and then use the program to find the angles that produce the smallest errors. One or two degree errors in angles can lead to surprisingly large % deviations in certain situations.

Program Methodology : Since the values of the three hanging masses are quite accurate, the experimental error is generally due to friction associated with the pulleys (which is difficult to model), and measurement error in the two angles. The error leads to values of $\Sigma\ F_x$ and $\Sigma\ F_y$ (referred to as the <u>remainders</u>) that are not zero. The application searches over all combinations of values of θ and ϕ (near the experimental values) to find the combination that produces the smallest remainders. The strategy is to evaluate the total % deviation arising from $\Sigma\ F_x$ and $\Sigma\ F_y$ for the range of values of θ and ϕ, and to identify the values that give the minimum total % deviation.

Unfortunately, since the sum of the X and Y force components should theoretically be zero the % deviations cannot be calculated in the usual fashion since the theoretical zero in the denominator of the % deviation expressions would result in an undefined (or infinite) % deviation. Instead, two alternative approaches will be used. The Y equation can be rearranged as $T_2 \sin\phi + T_1 \sin\theta = W$ so that the the Y components of the tensions will be <u>compared directly to the value of hanging weight W</u> (which is assumed to be correct). Defining Ty as the sum of the Y tension values : $Ty = T_1 \sin\theta + T_2 \sin\phi$ gives a Y component % deviation that can be expressed as :

$$\text{Y \% deviation} = \frac{(Ty - W)}{W} \times 100 \qquad \ldots(3)$$

[1] single degree errors in sine and cosine arguments at small and large angles respectively can produce significant variations in values (due to the steep slope of the functions in these regions).

In the X direction there are only two components that should add to zero. Rearranging this equation gives $T_2 = T_1 \cos\theta / \cos\phi$. The deviation can now be taken between the right hand side and the left hand side (which is assumed to be accurately known).

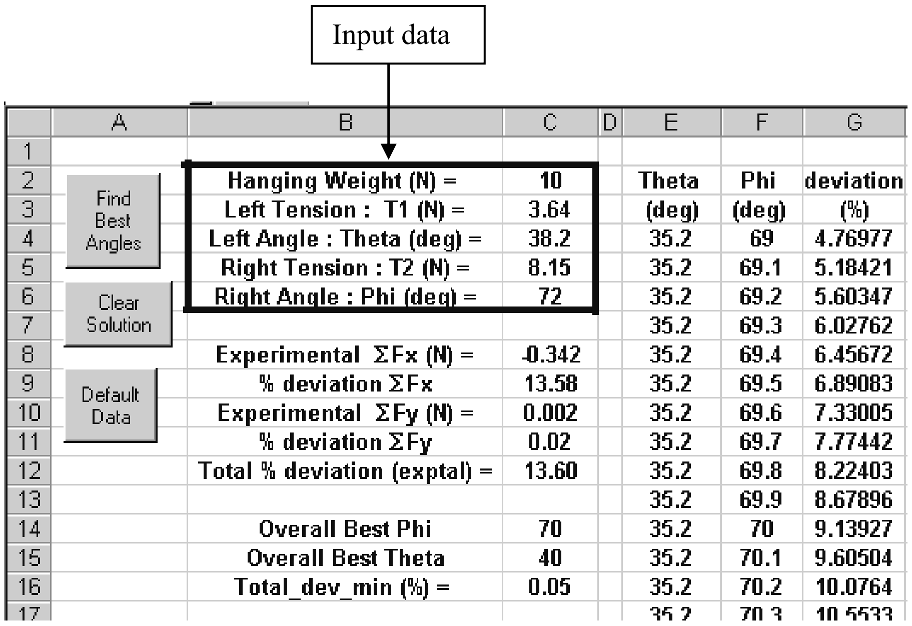
$$\text{X \% deviation} = \frac{(\text{T1} \cos\theta / \cos\phi - \text{T2}) \times 100}{\text{T2}} \qquad \ldots(4)$$

The total % deviation will be formed from the magnitude of the individual % deviations as [2] :

Total deviation = Abs(X deviation) + Abs(Y deviation)

The absolute value of the individual deviations has been taken for 2 reasons : first to prevent negative deviations from been interpreted as being small (in physics a negative deviation simply means a value is less than what it's being compared to), and more importantly to prevent a negative deviation from canceling out a positive deviation and making the result appear falsely close to zero.

The worksheet for this application is set up in the following way :



	A	B	C	D	E	F	G
1							
2		Hanging Weight (N) =	10		Theta	Phi	deviation
3		Left Tension : T1 (N) =	3.64		(deg)	(deg)	(%)
4		Left Angle : Theta (deg) =	38.2		35.2	69	4.76977
5		Right Tension : T2 (N) =	8.15		35.2	69.1	5.18421
6		Right Angle : Phi (deg) =	72		35.2	69.2	5.60347
7					35.2	69.3	6.02762
8		Experimental ΣFx (N) =	-0.342		35.2	69.4	6.45672
9		% deviation ΣFx	13.58		35.2	69.5	6.89083
10		Experimental ΣFy (N) =	0.002		35.2	69.6	7.33005
11		% deviation ΣFy	0.02		35.2	69.7	7.77442
12		Total % deviation (exptal) =	13.60		35.2	69.8	8.22403
13					35.2	69.9	8.67896
14		Overall Best Phi	70		35.2	70	9.13927
15		Overall Best Theta	40		35.2	70.1	9.60504
16		Total_dev_min (%) =	0.05		35.2	70.2	10.0764
17					35.2	70.3	10.5533

[2] Although adding percentages can lead to skewed results when the relative size of the quantities being considered is different (that is, if the terms are not weighted), the simplicity achieved here in doing so cannot be ignored. However, users should avoid situations in which both supporting cables are close to vertical, or close to horizontal, as this can lead to large deviations in the other components (although the tension values themselves are small).

```
Option Explicit

Private Sub cmdEquilibrium_Click()
Dim T1 As Double, Theta_Exptal As Double, Theta As Double, W As Double
Dim T2 As Double, Phi_Exptal As Double, Phi As Double, Ty As Double
Dim X_deviation As Double, Y_deviation As Double, Average As Double
Dim Sum_Fx As Double, Sum_Fy As Double
Dim Total_dev As Double, Total_dev_min As Double
Dim Theta_best As Double, Phi_best As Double, Row As Integer
Const Factor As Double = 3.14159 / 180

W = Worksheets("Sheet1").Range("C2")
T1 = Worksheets("Sheet1").Range("C3")
Theta_Exptal = Worksheets("Sheet1").Range("C4")
T2 = Worksheets("Sheet1").Range("C5")
Phi_Exptal = Worksheets("Sheet1").Range("C6")

Total_dev_min = 1000
Row = 4

For Theta = (Theta_Exptal - 3) To (Theta_Exptal + 3) Step 0.1

    For Phi = (Phi_Exptal - 3) To (Phi_Exptal + 3) Step 0.1

        Ty = T1 * Sin(Theta * Factor) + T2 * Sin(Phi * Factor)
        Y_deviation = (Ty - W) * 100 / W                              Equation (3)

        X_deviation = (T1 * Cos(Theta * Factor) / Cos(Phi * Factor) - T2) * 100 / T2
                                                                      Equation (4)
        Total_dev = Abs(Y_deviation) + Abs(X_deviation)
        If Total_dev < Total_dev_min Then Total_dev_min = Total_dev:
                                               Theta_best = Theta: Phi_best = Phi
        Worksheets("Sheet1").Cells(Row, 5) = Theta
        Worksheets("Sheet1").Cells(Row, 6) = Phi
        Worksheets("Sheet1").Cells(Row, 7) = Total_dev
        Row = Row + 1
    Next Phi
Next Theta

Worksheets("Sheet1").Range("C14") = Phi_best
Worksheets("Sheet1").Range("C15") = Theta_best
Worksheets("Sheet1").Range("C16") = Total_dev_min
                                                           Left side of equation (1)

Sum_Fx = T2 * Cos(Phi_Exptal * Factor) - T1 * Cos(Theta_Exptal * Factor)
Sum_Fy = T1 * Sin(Theta_Exptal * Factor) + T2 * Sin(Phi_Exptal * Factor) - W
Worksheets("Sheet1").Range("C8") = Sum_Fx
Worksheets("Sheet1").Range("C10") = Sum_Fy                 Left side of equation (2)
```

The following code writes out the X, Y, and total % deviations of the experimental data.

```
Ty = T1 * Sin(Theta_Exptal * Factor) + T2 * Sin(Phi_Exptal * Factor)
Y_deviation = (Ty - W) * 100 / W
Worksheets("Sheet1").Range("C11") = Y_deviation

X_deviation = (T1 * Cos(Theta_Exptal * Factor) / Cos(Phi_Exptal * Factor) - T2) * 100 / T2
Worksheets("Sheet1").Range("C9") = X_deviation

Total_dev = Abs(Y_deviation) + Abs(X_deviation)
Worksheets("Sheet1").Range("C12") = Total_dev

End Sub
```

The approach presented here can be modified to include rotational equilibrium and hunt for the two optimum angles of a simple crane lab (angled beam of non-zero mass, hanging weight , angled supporting cable, contact force with X and Y components).

Future : An interesting addition to this program is to construct a surface plot of **Total_dev** versus **Phi** and **Theta**.

Problems **[Chapter 3]**
In many of these exercises you may find it exciting to add a delay loop so that the computations appear to be happening in real time, and not in almost instantaneous computer time. However, unless specifically requested, delay loops are optional. **Clear** buttons should be added to all problems.

The problems of Chapter 1 can be programmed using loops to accept multiple sets of data. In most cases the input data can be replaced by code that automatically varies the independent variable so that a view of the general behaviour is obtained when the results are graphed.

1. Construct a program that displays a counted value on the worksheet starting at some decimal value (specified by an **Input Box**) and increasing in decimal steps (specified by the user on the worksheet). A running total should be written out beside each number. The counting loop should be exited when the running total exceeds a value that can be changed by the user, or when a specific number of values have been counted. Create two versions of this program: one using a **Do** loop, the other using a **or…Next** loop. Refer to **Example # 3** (pg 3-4) Add a delay loop once each program is working.

2. Enter 10 numbers in a column on a worksheet. Design a program that reads each number and adds it to a running total. The running total should be written back on the worksheet beside each number. The user may not want to add all the numbers; the number of values to be added should entered via an **Input Box**. Change the colour index to a different value in each cell in which the running total appears (do not exceed integer index values greater than 55). Add a delay loop once the program is working and use a variable clear (pg 3-3) to clear only the running total of the values added.

3. Write a series of random numbers (between your choice of two positive decimal values) down a column. [*You will have to modify the random number generator to produce decimal values in a certain range.*] Keep a running total of the numbers and stop the process once the sum exceeds a total value that has been read-in from the worksheet. Write the running total beside each new number that is produced. Construct two versions of the program : one using a **For…Next** loop, the other a **Do** loop (which is the easiest). The **Do** loop versions should use an exit condition and not a dummy index. Change the colour index to a different value in each cell in which the running total appears (do not exceed integer index values greater than 55). Add a delay loop once each program is working.

4. Modify the **Do** loop version of problem # 3 to allow different values of **upper** and **lower** (used in the Random Number Generator) to be read of the worksheet. Save the program under a new name and then modify it to allow the user to select whether the random numbers will be integer or decimal. [Hint : read a **Boolean** value off the spreadsheet where, perhaps, **True** means integer, and then use an **If** to check its value, or, use the response from a Yes/ No **message box function** to detect (as in **Example # 1**, Chapter 2, pg 2-5) which type of random values are desired. Single outcome or multiple outcome **If**s can be used in either approach.] Add a delay loop once the program is working.

5. Modify the **Do** loop version of problem # 3 to display the numbers in a rectangular array 5 columns wide. Enter the running total values in a second array situated to the right of the first array. [**Example # 8** shows how to read values from a 2-D array.] Add a delay loop once the program is working. The **Clear** button should only clear up to the final entry, and not beyond.

6. In **Example # 4 (pg 3-6)** both the interval (**0** to **4**) and step size **0.04** are fixed in the **cmdGraph** subroutine. It would clearly be useful to be able to change these parameters on the spreadsheet without having to directly access the program. Save the file under a new name. Use variable names to define the starting and ending points of the **For … Next** loop as well as the step size. It's probably best to read these 3 values off the spreadsheet rather than using an **InputBox** which, although fun, is a more time consuming during execution. Plot an interesting function of your own choosing over some range. It may be necessary to right click on the horizontal axis to set a convenient minimum value.

7. Enter 20 numbers between −1,000 and + 1,000 in a column. a) Write a program that determines the minimum number in the set, and writes this number somewhere on the worksheet. b) Enhance the program by adding a time delay loop controlled by a slider, and bold, or add colour, to the cell containing the number currently being examined.

8. Modify problem # 7 to find the number closest to a specified target number read-in off the worksheet. One approach is to use a single outcome **If** to compare each **Number** to the **Target** value. **If Abs(Number – Target) < Small Then Small = Abs(Number – Target): Closest = Number : …etc…** The absolute value (or magnitude) is used to prevent large negative differences from appearing to be small. The initial value of **Small** would be conveniently large so that at least one closest value is found. Observe that the value of **Small** is updated with each closer value that is found.

9. Enter a set of different numbers between −50 and + 50 in a **2-D** 5 x 6 array (that is, 5 rows down and 6 columns across giving a total of 30 numbers). Write a program that identifies the maximum and minimum numbers in the array, and also displays the cell location of each. When the program has finished running it should highlight the two numbers in some way (bold or colourized cell). [**Example # 8 pg 3-14** shows how to read values from a 2-D array.] Once working, modify the program to find and identify the value closest to a specified value.

10. Modify problem # 9 to find the number in the array that is **closest** to a user prescribed number.

11. This program consists of two parts : in the first part a rectangular array is filled with random numbers, then in the second part the array is searched for the minimum value. Use the **Random Number generator** to fill a rectangular array of cells 5 columns wide by 6 rows deep with random decimal numbers between −2.5 and + 4.8. [**Example # 8** shows how to read values from a 2-D array.] Search the array for the smallest number and **Select** the cell in which it appears so that a box appears around the cell [refer to **Example # 8 chapter 3 pg 3-14**]. Use a running total to add up all the random numbers, write the total somewhere on the worksheet [future : use a static variable to write different totals into a column and then Bin sort and plot the distribution. **Chap_1_Store_Salary** Excel file has example of static variable]

12. **Sorting into Bins**. A set of receipts (bills) at a grocery store are to be sorted into different categories : 0 < to ≤ \$20, \$20 < to ≤ \$50, \$50 < to ≤\$100, and > \$100. Develop a program that sorts receipts into the 4 categories and which writes out the totals for each category onto the worksheet. The categories are known as "bins" (imagine that you're working in a hardware store and are sorting screws and nails into different bins). You will need to declare a different bin variable for each category and then use a single outcome **If** to update the total in each bin. For example : **If Receipt > 0 And Receipt<= 20 Then Bin1 = Bin1 + 1** . Enter 40 values of grocery bills onto the spreadsheet as input data. Highlight the cells containing the category labels (0 < to ≤ \$20 etc.) and the category totals,

and use the **Chart Wizard** to display a column graph. Modify the working program to handle a variable number of input values. [Note : a second type of common sorting involves arranging the numbers from highest to lowest, or vice versa, which is a little more difficult to program than what you're doing here.]

Note ; Column (really "Bar") charts constructed on older versions of Excel will sometimes not put the bin categories on the X axis, and instead simply give the numbers 1, 2, 3, … etc. To obtain bin descriptions such as $10 \leq$ to < 20, right click on the chart area, select **Source data**, select the **Series** tab, and then enter the location of the bin descriptions in the **Category (X) Axis Labels** box by clicking on the icon at the right end of the box, and then dragging over the particular cells, and finally closing the box.

13. **Mark Distribution**. Construct a program that determines the distribution of class marks between 0 and 100 in intervals of 10 : 0 to < 10 , $10 \leq$ to < 20, ….., $80 \leq$ to < 90 **,** $90 \leq$ to ≤ 100. The program needs to consider each mark, determine the range (or "bin" – refer to problem # 12) that it falls in, and add 1 to the bin total for that range. Write out the frequency, or number of marks, in each bin on the worksheet and plot a column graph. Modify the program so that the user can input a variable number of marks.

14. **Theoretical distribution of outcomes for a pair of dice**. Two six sided dice are to be used in a school fundraising event. {Each die has a different number of dots between 1 and 6 on each side.} Numbers between 2 and 12 representing the possible outcomes of the thrown dice are painted on a table; players place bets on the numbers and hope for the best. An observant student realizes that certain numbers are more likely to occur than other numbers and decides to write a small program to determine the frequency distribution of the possible outcomes. Develop a program that uses two nested loops, one for each die, to exhaustively determine all possible outcomes. The program should take each outcome and sort it into the appropriate "bin" (that keeps track of the frequency with which each value occurred - refer to problem # 12). Plot a column chart using the final distribution.

15 a) **Problem # 14** gives the theoretical distribution of possible outcomes when two dice are thrown. Design a program that uses two random number generators (one for each die) to produce the outcome of each toss. [The random number generators should produce integers between 1 and 6.] Read the number of dice throws, N, off the worksheet. Plot a column chart of the frequency distribution. Qualitatively compare the frequency distribution for a particular N with the theoretical distribution.

b) Save a working copy under a new name and add an outer loop that varies N systematically and which writes out N and the distribution into separate columns of the worksheet. Add a single column chart to view how the experimental distributions converge to the theoretical distribution as N increase (display all the distributions on the same chart). Note: The bin variables will need to be declared **Long** if the number of throws is to exceed about 180,000.

16. Find the integer values of **a**, **b**, and **c** that satisfy the relationship $\mathbf{a}^2 + \mathbf{b}^2 = \mathbf{c}^2$. Hint : use 3 loops to search over 50 values of **a**, and **b**. The search interval for **c** should start at the value of the greater of **a** or **b** and should extend to at about **a + b** . Write out only those values of **a**, **b**, and **c** for which the relationship is true. Option : Count the number of distinct solutions noting that a second solution is always produced by interchanging the values of a and b [3,4,5 and 4,3,5].

17. A **Fibonacci sequence** is a series of numbers in which each number is the sum of the preceding two (except for the first two) **0, 1, 1, 2, 3, 5, 8, ….** Develop a program that calculates a portion of the series (to be specified by the user) and writes it into a column on your worksheet. Hint : the program needs to keep track of 3 numbers : the previous 2 numbers which are then added to give the next value in the sequence. Once the latest value has been written onto the worksheet the 2 older values are updated in preparation to producing the next value [the latest value becomes the first oldest, and the first oldest is bumped to become the second oldest]. Add a second column that determines the running total of the series values. Plot a graph of sequence, and a second graph of the running total. Try a **Google** search to find some interesting applications including in botany and astronomy.

18. **The Egyptian Chessboard**. As the story goes, a king in ancient Egypt wished to reward a subject for some contribution by allowing the person to choose their own reward. The king was surprised, and mildly offended, when the subject chose what appeared to be a very modest present (and not the gold that the king assumed would be chosen !). The subject's reward consisted of an amount of grain to be determined by imagining a chess board with one grain of wheat on the 1st square, two grains on the 2nd square, 4 grains on the 3rd square, 8 grains on the 4th square, ... and so on (doubling every square). a) Construct a program writes out the grain contained in each square in a single column. Use two columns : 1 through 64 for the squares, and the second column holding the corresponding grain value. b) write the values back into an 8 x 8 array of cells on the chess-board (enlarge the cell width and/or use scientific notation.

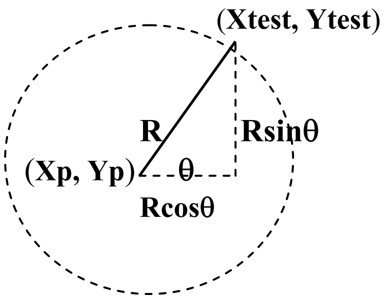
19. (removed June, 2004)

20. **Examples** # **5** and # **9** in this chapter introduced the concept of a delay loop. Out of curiosity, it would be interesting to determine how many iterations must be made through the delay loop to produce specific amounts of **Delay**. Construct a program consisting of a single loop with loop index **Delay** that increases from 0 to some maximum value (your choice) in steps of 0.1 seconds. Call the **TimeDelay** subroutine from inside the single loop. The **Timedelay** subroutine of **Examples # 5** and **# 9** should be modified by adding an index that tracks the total number of iterations; also add a statement to write out the **Delay** and number of iterations just before the **TimeDelay** sub is exited. Use the **Chart Wizard** to plot a graph of **Delay** versus number of iterations and observe if there's a linear region as might be expected. [The process of counting the number of iterations is, of course, Heisenbergian since the index used to count the loops adds extra operations, and hence extra delay, to the original loop.]

25. **Equipotential Lines :** This application will plot an equipotential line for a distribution of 3 charges beginning at an arbitrary starting point (**Xp, Yp**). A modified **polar search** (pg 3-16, section 3.5.3) is made about the point (**Xp , Yp**) by searching over a number of test points (**Xtest, Ytest**) on the circumference of a circle of fixed radius **R**. A loop is used to vary angle **Theta** in uniform increments such that the test points are determined as :

Xtest = Xp + R * Cos(theta * Factor)
Ytest = Yp + R * Sin(theta * Factor)

{**Factor** converts **theta** to radians}



The objective of each search is to find the point (**Xbest, Ybest**) with the potential closest to the starting potential **Vstart** . A closest value search is constructed from a single **If** statement as described in section 3.5.1 pg 3-15. Once the best point has been identified the search moves to that point, which becomes the new (**Xp , Yp**) , and begins again. A second, outer loop moves the search from best point to best point.

Since equipotential lines form closed loops there will actually be two possible points on each circle that have a value equal to **Vstart**. This unfortunately means that the search may get caught shifting back and forth between the two points on the very first search circle. Two prevent this all searches except the very first will be restricted to less than 360^o.

The best points are written onto the worksheet and plotted on an **XY(Scatter)** graph along with additional series to indicate the charge locations.

A program flow chart is presented at the end of this problem.

Pre-Lab (Complete this section *before* you come to the lab)

a) Add the following declarations in the **General Declarations** area under **Option Explicit** :

```
Dim Q1 As Double, Xq1 As Double, Yq1 As Double
Dim Q2 As Double, Xq2 As Double, Yq2 As Double
Dim Q3 As Double, Xq3 As Double, Yq3 As Double
Const k As Double = 9000000000#
```

b) Construct a custom function **Potential(X, Y)** that evaluates the potential due to 3 charges (given above) at any point (X, Y). Since the charge values and locations are only available once read-in, there is not much benefit to defining the function in a module (as it cannot be easily evaluated directly on the spreadsheet).

```
Potential = k * Q1 / (Sqr((X - Xq1) ^ 2 + (Y - Yq1) ^ 2)) + _
                k * Q2 / (Sqr((X - Xq2) ^ 2 + (Y - Yq2) ^ 2)) + _
                  k * Q3 / (Sqr((X - Xq3) ^ 2 + (Y - Yq3) ^ 2))
```

c) Create a button which will contain the search for the equipotential points. Add the declarations :

```
Dim R As Double, small As Double
Dim Vstart As Double, V As Double
Dim Xtest As Double, Ytest As Double,  Xp As Double, Yp As Double
Dim Xbest As Double, Ybest As Double, Vbest As Double, Angle As Double
Dim theta As Double, Theta1 As Double, Theta2 As Double, delta_Theta As Double
Const Factor As Double = 3.14159265897 / 180
Dim Number_points As Double, I As Integer, Row As Integer
```

Add statements to read-in the following data from the worksheet :

Xq1 = 1.05 m, Yq1 = 1.05 m, Q1 = 1E–08 C
Xq2 = 1.05 m, Yq2 = 2.05 m, Q2 = – 2E–08 C
Xq3 = 2.05 m, Yq3 = 1.05 m, Q3 = – 6E–09 C

Xp = 1.9 m, Yp = 2.25 m, R = 0.2 m, delta_Theta = 1 degree
Number_of_points = 50

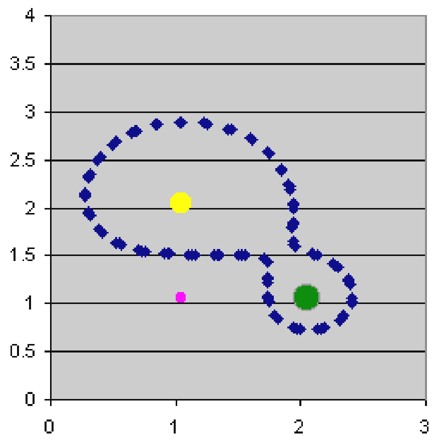
Test your potential function : Use the statement **Vstart = Potential(Xp, Yp)** and write the value somewhere on the worksheet; it should be – 189.59 volts.

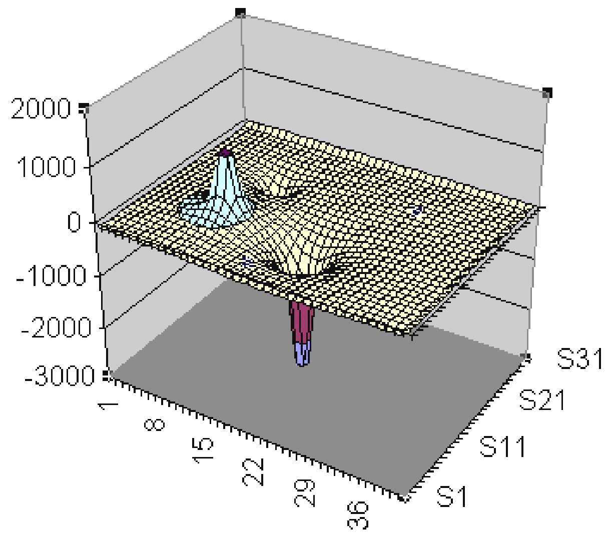
Add the two loops : [The code denoted by the dots **…….** will be determined by you in the lab.]

```
For I = 1 To Number_points

      small = 1E+20
      For theta = Theta1 To Theta2 Step delta_Theta
         ………
         Xtest = Xp + R * Cos(theta * Factor)
         Ytest = Yp + R * Sin(theta * Factor)

         ……..
         If Abs(V – Vstart) < small Then small = Abs(V – Vstart): Xbest = Xtest: _
                                              Ybest = Ytest: Vbest = V: Angle = theta
      Next theta
   …….
   …….
Next I
```

In the Lab

d) Carefully consider the program structure described on the next page, and complete the programming. You will need to add some moving write statements and a few other commands. The output should appear as shown below on the left :





e) **Refer to Example # 7 pg 3-13.** Create a button which will write out the values of the potential funct-ion into a 30 x 40 array of cells on the worksheet over the range X = 0 to 2.9 m and Y = 0 to 3.9 m. You will need to shift the read-in statements for the charge values and locations into this subroutine. Add a surface plot to display the function (shown on the above right). **Note : Charges must not be located on any of the points at which the function is evaluated (otherwise a singularity will occur).** *Charge magnitudes do not necessarily correlate with the size of a "spike".*

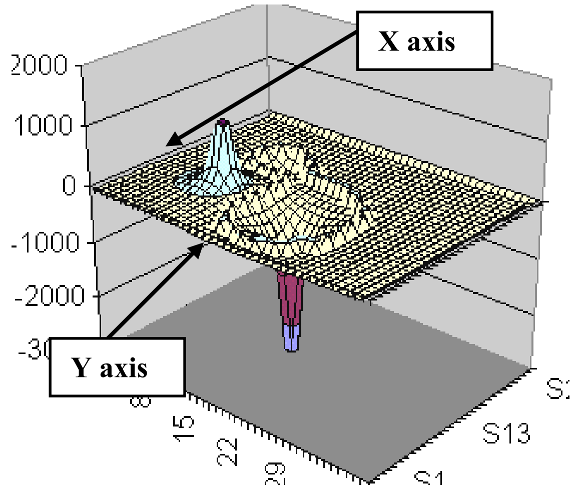
f) Save the working program under a new name and modify it so that a maximum of **5** equipotential lines may be plotted on the same graph.

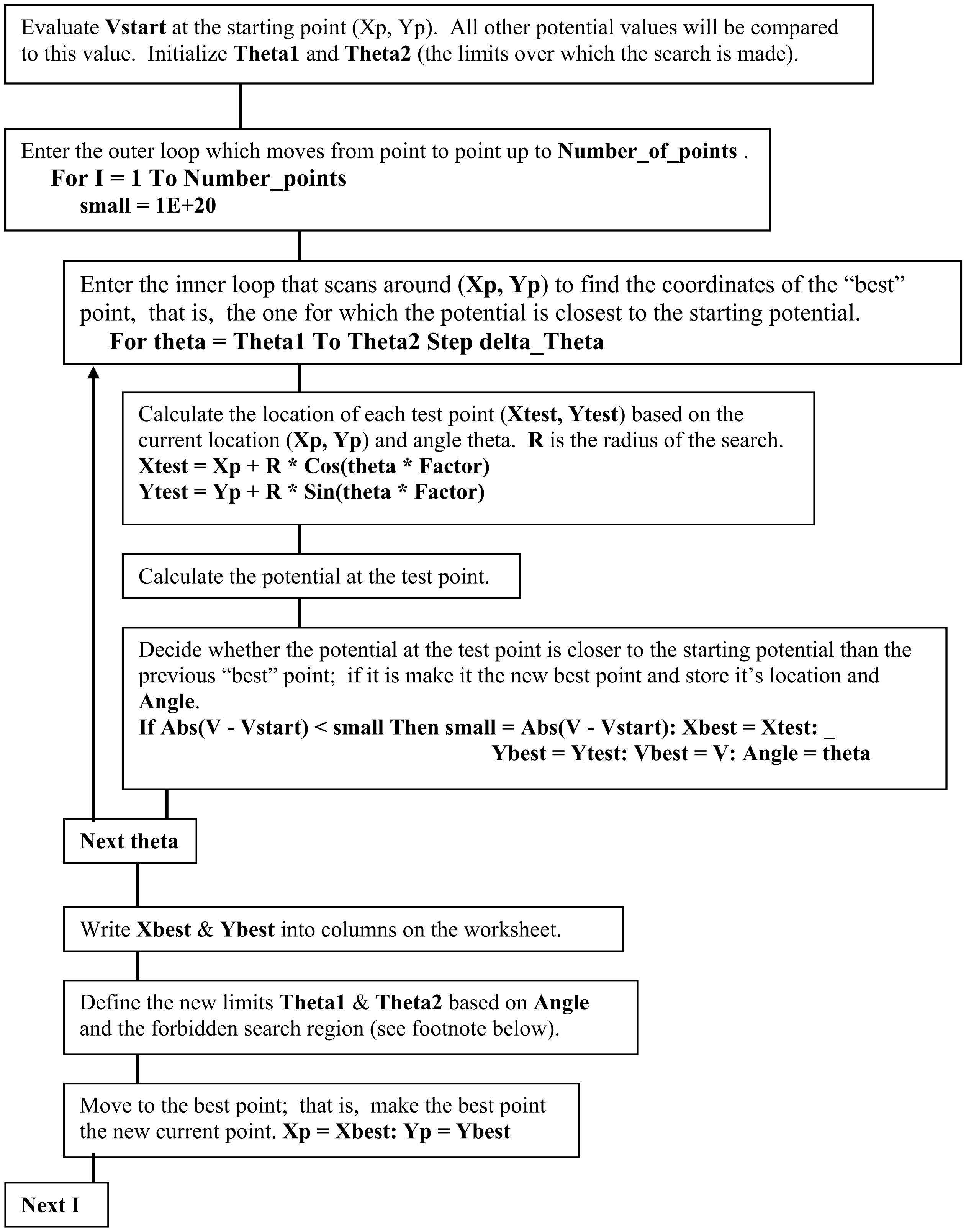
g) Modify the potential function so that it accepts up to about 8 charges and use it to model an interesting charge distribution [check the note in part e)]. If your Electricity & Magnetism class performed an equipotential lab try modelling one of the more complex configurations and compare the program results with the lab results (if not, ask me and I'll provide some data). Check a number of other physics textbooks in the library for possible equipotential line plots.

h) Save the working program under a new name and modify the program so that the equipotential line is also visible on the function plot. Do this only for the original program that displays a single equipotential line.

The bumps represent the equipotential line. Note that the X and Y axes are unorthodox.

Note : The displays have not been converted to a right-hand coordinate system.





Evaluate **Vstart** at the starting point (Xp, Yp). All other potential values will be compared to this value. Initialize **Theta1** and **Theta2** (the limits over which the search is made).

Enter the outer loop which moves from point to point up to **Number_of_points** .

```
    For I = 1 To Number_points
        small = 1E+20
```

Enter the inner loop that scans around (**Xp, Yp**) to find the coordinates of the "best" point, that is, the one for which the potential is closest to the starting potential.

```
    For theta = Theta1 To Theta2 Step delta_Theta
```

Calculate the location of each test point (**Xtest, Ytest**) based on the current location (**Xp, Yp**) and angle theta. **R** is the radius of the search.

```
Xtest = Xp + R * Cos(theta * Factor)
Ytest = Yp + R * Sin(theta * Factor)
```

Calculate the potential at the test point.

Decide whether the potential at the test point is closer to the starting potential than the previous "best" point; if it is make it the new best point and store it's location and **Angle**.

```
If Abs(V - Vstart) < small Then small = Abs(V - Vstart): Xbest = Xtest: _
                                        Ybest = Ytest: Vbest = V: Angle = theta
```

Next theta

Write **Xbest** & **Ybest** into columns on the worksheet.

Define the new limits **Theta1** & **Theta2** based on **Angle** and the forbidden search region (see footnote below).

Move to the best point; that is, make the best point the new current point. **Xp = Xbest: Yp = Ybest**

Next I

The limits **Theta1** & **Theta2** can be defined relative to **Angle** as : **Theta1** = **Angle – some range** to **Theta2 = Angle + some range** (where "**some range**" might typically have a value of about 35^{o}).

26. **Planar Refraction.** This simulation allows a user to follow up to 5 rays of light emanating from a common point which are refracted by a planar surface.

Pre-Lab (Complete this section ***before*** you come to the lab)

a) Program a button that draws a single light ray at an angle read-in from the worksheet. Make the following declarations in the **General Declarations** area so that the variables are defined globally (although not immediately required, this will become necessary as the program evolves).

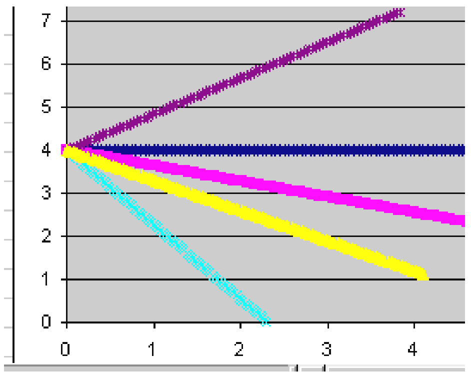
```
Dim Angle As Double, X As Double, Y As Double, deltaR As Double
```

X and **Y** represent the current location of the light ray. **deltaR** is the distance it moves at an angle **Angle**. Try a value of **0.05** meters for **deltaR** .

Place the following code inside a loop that performs about 140 iterations. The values of X and Y need to be written back onto the worksheet and graphed.

```
X = X + deltaR * Cos(Angle)
Y = Y + deltaR * Sin(Angle)
```

Although the diagram on the right shows a collection of rays, you will only be producing one for the time being.



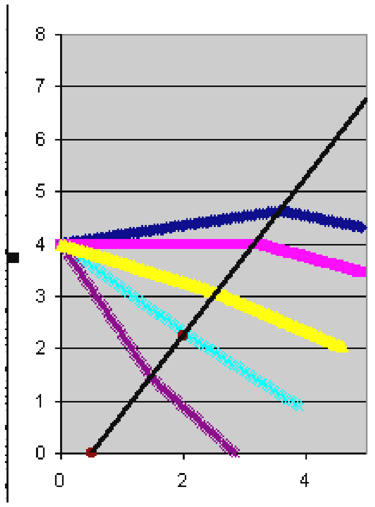
2. Test data (Theoretical solution) : construct a linear equation to describe a boundary between air and glass (use a positive sloped interface as shown in the diagram below). Consider a ray travelling at + 20^{o} (w.r.t. the + X axis). Determine the incident angle for the ray (from simple geometry), and using Snell's law find the refracted angle. Calculate the final angle at which the ray is moving in the glass (w.r.t. the + X axis). Repeat for a ray initially moving at about – 30^{o}.

During the Lab

3. Add code that allows up to 5 rays to be plotted; one each time the main run button is clicked. The user should be able to keep adding rays such that the 6th ray overwrites the 1st, and the 7th overwrites the 2nd, etc. Read the angles in from a single cell; write each value above the X & Y data produced for the particular ray.

4. Add a separate data series to the graph that represents 2 points on the interface. Fit a straight line to the points (extend the line forward in the **Options** tab of the **Add Trendline**, or choose 2 points well off the graph).

5. Consider the program structure described on the next page. Create code that detects the interface and construct a separate subroutine that refracts the ray according to Snell's law. Write out the refracted angle (in degrees) and compare with the theoretical solution.



Program Structure (This is not a flow chart)

Create a moving ray using a loop and the statements :

```
X = X + deltaR * Cos(Angle)
Y = Y + deltaR * Sin(Angle)
```

Write the location of the ray into 2 columns on the worksheet and plot an **XY (Scatter)** graph.

Detect the Interface :
Check if the new point is inside the second medium. If the boundary between the air an the medium is described (in 2-D) by the equation **Y = a x + b** then the ray will have crossed into the medium if **Y <= a x + b**. [Use a positively sloped interface for the time being, I haven't checked refracted angle expressions for negative sloped planar interfaces yet.] Once a ray has passed into the second medium it will always satisfy the inequality. Since we don't wish to erroneously make refraction calculations at each point in the second medium it is necessary to detect only the very first point inside the new medium. One approach is to use a **Boolean** variable that is initially **False** and remains **False** until the interface is detected at which time it is set to **True** to then prevent further refraction calculations from occurring.

Refraction :
Call a "refraction" subroutine that converts the angle ("**Angle**") of the ray before encountering the interface into its new value inside the second medium. The cosine version of Snell's Law will be used with incident and refracted angles measured with respect to the tangent to the surface (and not the perpendicular). [Although unnecessary here, this approach is useful in the case of spherical lenses for which the slope of the tangent is easily found from the derivative.] These expressions will be derived later. Construct code that determines the angle "**slope**" of the interface [hint: use the **Atn()** function)].

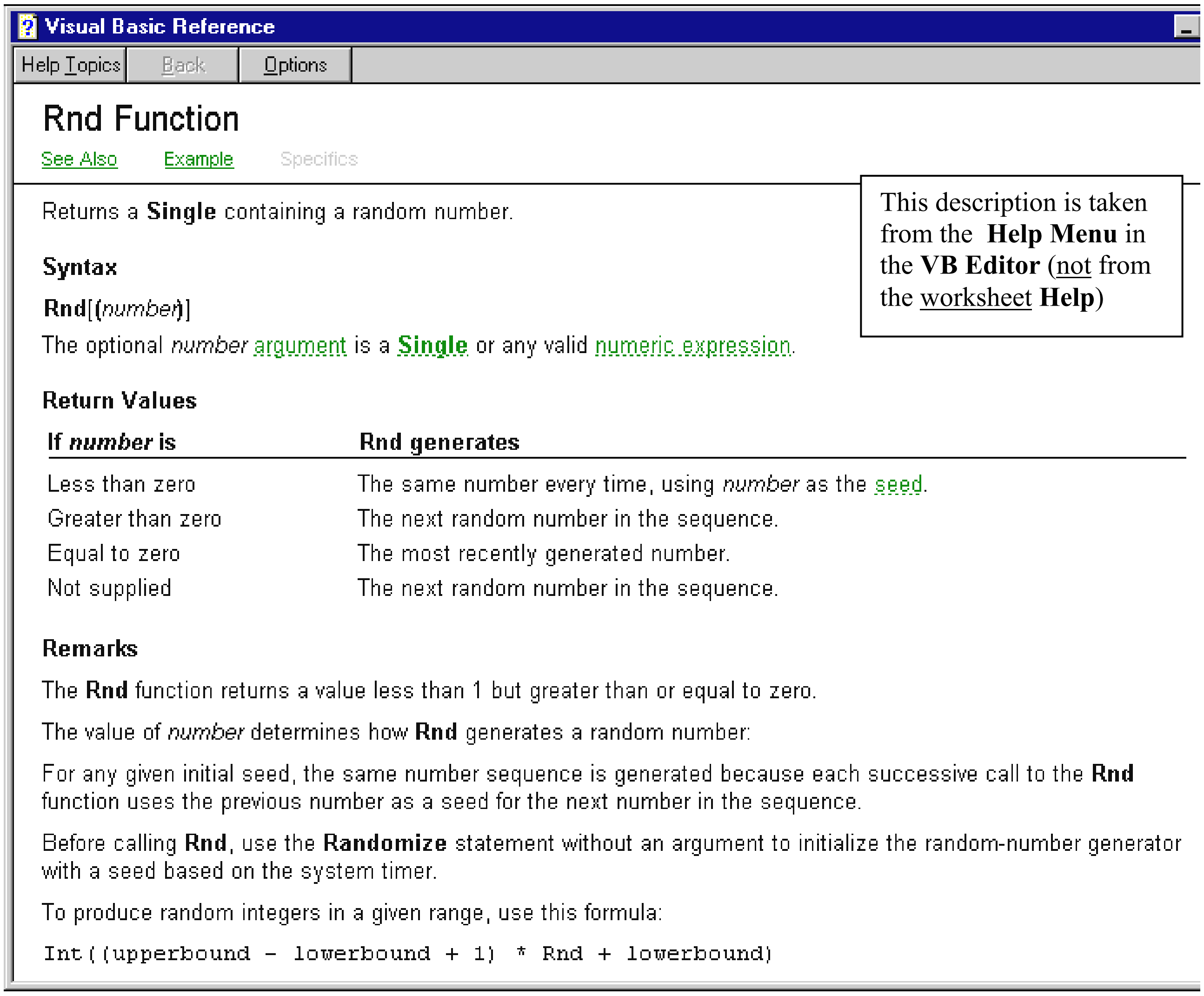
The following code can be used to determine the **Angle** that a ray moves in a refracted medium given the **Angle** that it moves in the incident medium. **Beta1** & **Beta2** are the angles of the ray w.r.t. the interface which are used in the Snell's Law computation (in cosine form).
For positive sloped interfaces :

```
    If Angle >= 0 Then Beta_1 = Abs(slope) - Abs(Angle)
    If Angle < 0 Then Beta_1 = Abs(slope) + Abs(Angle)
    Beta_2 = Application.WorksheetFunction.Acos(n1 * Cos(Beta_1) / n2)
    Angle = Abs(slope) - Abs(Beta_2)
```

[note that the **Abs(slope)** is redundant sine the slope is positive.]
For negative sloped interfaces :

```
    If Angle >= 0 Then Beta_1 = Abs(slope) + Abs(Angle)
    If Angle < 0 Then Beta_1 = Abs(slope) - Abs(Angle)
    Beta_2 = Application.WorksheetFunction.Acos(n1 * Cos(Beta_1) / n2)
    Angle = Abs(Beta_2) - Abs(slope)
```

[note : the negative sloped interface expressions have yet to be tested.]



Rnd Function

This description is taken from the **Help Menu** in the **VB Editor** (<u>not</u> from the <u>worksheet</u> **Help**)

Returns a **Single** containing a random number.

Syntax

Rnd[(*number*)]

The optional *number* argument is a **Single** or any valid numeric expression.

Return Values

If ***number*** is	**Rnd generates**
Less than zero	The same number every time, using *number* as the seed.
Greater than zero	The next random number in the sequence.
Equal to zero	The most recently generated number.
Not supplied	The next random number in the sequence.

Remarks

The **Rnd** function returns a value less than 1 but greater than or equal to zero.

The value of *number* determines how **Rnd** generates a random number:

For any given initial seed, the same number sequence is generated because each successive call to the **Rnd** function uses the previous number as a seed for the next number in the sequence.

Before calling **Rnd**, use the **Randomize** statement without an argument to initialize the random-number generator with a seed based on the system timer.

To produce random integers in a given range, use this formula:

```
Int((upperbound - lowerbound + 1) * Rnd + lowerbound)
```