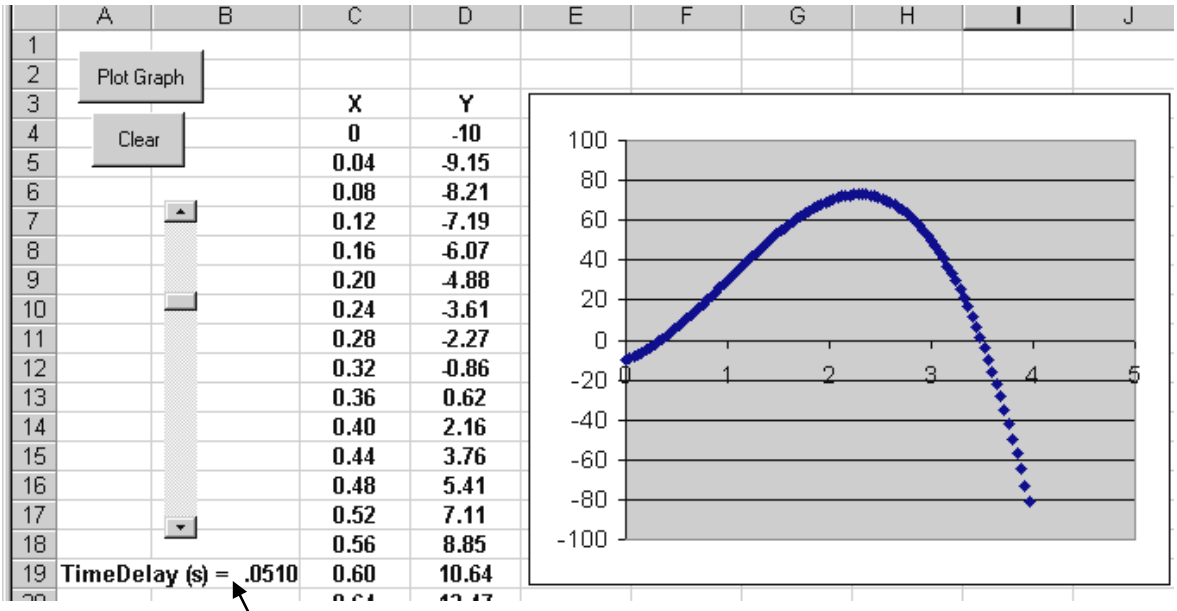
CHAPTER # 5 GRAPHS

This chapter extends the graph plotting program developed in **Example # 4** Chapter 3 for functions of the form **Y = f(X)** . The first application slows down each iteration to produce the illusion that the graph is animated as additional points are added to the data set. The second application demonstrates how to add a graph programmatically (using code statements rather than going through the **Chart Wizard**). The third application reduces the set of plotted points to a moving collection of five points which will be referred to as a "**comet tail**". Comet tail displays will be particularly useful in future applications involving 2-D dynamics. A fourth application examines how the trajectories of objects can be simulated using kinematics equations. The fifth application develops a brute force technique for curve fitting which together with the hanging weight lab **Application # 3** of Chapter 3 lays the groundwork for future projects that select optimum values from within a set (such as using experimental velocity-position data to find the drag coefficient and rolling friction of a car). Finally, the chapter concludes with two approaches for creating a surface plot of a function of two variables **Z = f(X, Y)** which was first introduced in **Example # 7** Chapter 3.

5.1 Animated Graphs Y = f(X) : Application # 4 a)

In certain of our applications, generally when we're following the motion of an object, it will be helpful to slow down the operation of the loop so that the points are added one-by-one to the graph with a small delay between each addition. This is accomplished by using the **TimeDelay** subroutine encountered in Chapter 3 **Example # 5** and a **DoEvents** statement which together produce the illusion that the graph is animated. Recall that a **DoEvents** command interrupts program execution to allow other events to be processed (in this case, the plotting of each point on the worksheet graph, and in other cases to allow the user to interact with the program as it executes). Without the **DoEvents** command the graph would only be plotted in completed form once the loop has been exited. A **scroll bar** (slider) is again used to control the amount of the **Delay**.

Save **Example # 4** in **Chapter 3** under a new name and add a scroll bar control : Name = **scbDelay** . Go into the **Properties** window of the Scroll Bar and set **Max = 200** .



Do not type. Appears automatically.

Add the code indicated in **bold**; the code in *italics* should already be present.

```
Option Explicit
Dim Delay As Double

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

Delay = scbDelay.Value / 1000

Row = 4

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
DoEvents
Call TimeDelay(Delay)
Row = Row + 1
Next X

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 scbDelay_Change()
Range("A19") = "TimeDelay (s) =  " & Format(scbDelay / 1000, "0.0000")

End Sub

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

The "**Call**" statement causes subroutine **TimeDelay** to be executed for the amount of **Delay** passed through the argument.

DoEvents interrupts program execution and causes the graph points to be plotted immediately; otherwise nothing appears until the loop is finished. Adding the code line **Range("A1").Select** would have the same effect as it shifts the program focus to the worksheet at which time the graph is also updated.

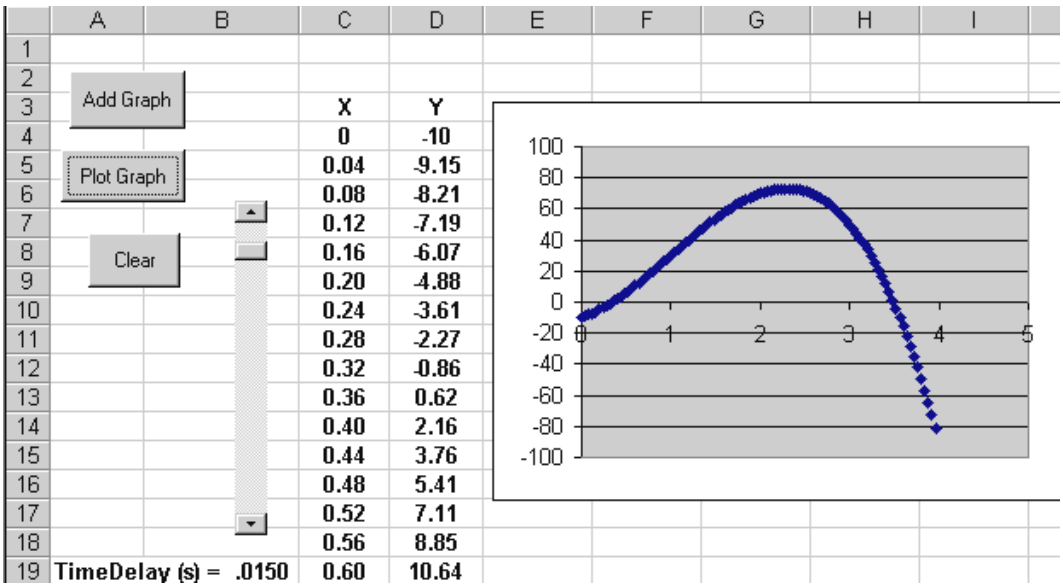
Same as earlier **TimeDelay**

To avoid the annoyance of **AutoScaling** of the graph axes, right-click on the X axis [when the cursor displays **Value (X) Axis**], select **Format ⇒ Scale** and set **Min = 0** and **Max = 5**. Repeat to set the Y axis **Min = - 100** and **Max = + 100** .

5.2 Adding Graphs Programmatically : Application # 4 b)

This application demonstrates how to add a graph programmatically, specify its type, and adjust certain properties such as the axis **Max** and **Min** values. This technique was originally used in simulations to prevent a moving object from going off the graph by changing the graph **Max** and **Min** "on the fly". [Notwithstanding the pleasure to be had being able to control a graph, I'm now of the view that the benefits are not really worth the clutter due to the additional code -- which tends to intimidate novice programmers who complain that it's difficult to "see what's happening".]

Save **Application # 4 a)** under a new name, delete the existing graph, and add a command button : **Name = cmdAddGraph**, **Caption = Add Graph** . Add the code in **bold** given below.




```
Option Explicit
Dim Delay As Double
Dim Graph As ChartObject

Private Sub cmdAddGraph_Click()
Range("C4:D5") = 0
Set Graph = ActiveSheet.ChartObjects.Add(195, 30, 250, 175)
Graph.Chart.ChartType = xlXYScatter
Graph.Chart.SetSourceData Source:= Sheets("Sheet1").Range("C4:D103")

Graph.Activate
ActiveChart.Legend.Select
Selection.Delete

With Graph.Chart.Axes(xlCategory)
   .MinimumScale = 0
   .MaximumScale = 5
End With
```

A **Chart Object** has been given the imaginative name **Graph**.

Lower case "**L**", not the numeral **1**

Add a space

Delete if you're using a French version of **Excel**.

Indicates the source of the data.

xlCategory refers to the **X** axis.

Lower case "**L**", not the numeral **1**

```
With Graph.Chart.Axes(xlValue)
   .MinimumScale = -100
   .MaximumScale = 100
End With

End Sub

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

xlValue refers to the **Y** axis.

The code statement that adds a chart and gives it the name **Graph** is :

Set Graph = ActiveSheet.ChartObjects.Add(195, 30, 250, 175)

The syntax of the **ChartObjects.Add(195, 30, 250, 175)** is :

ChartObjects.Add(Left, Top, Width, Height)

where **Left** and **Top** are measured from the left side and top of the screen. [The screen has a coordinate system associated with it which by convention has its origin at the top left of the screen with positive down and to the right.]. All distances are given in "points" where there are 72 points per inch (or about 28.35 points per centimeter).

A "**work-around**" is code that needs to be added to work around a software bug. An interesting example is present in the first line of **cmdAddGraph_Click()** subroutine. The values in cells **C4:D5** are set to zero to ensure that there are some values (zero is a value !) for the **Chart Wizard** to act on. If these cells were left empty, with the entire range **C4:D103** empty, any future number that appeared in cell **D4** would also magically appear in the title on the graph. This may be observed by commenting out the line with a single quote,

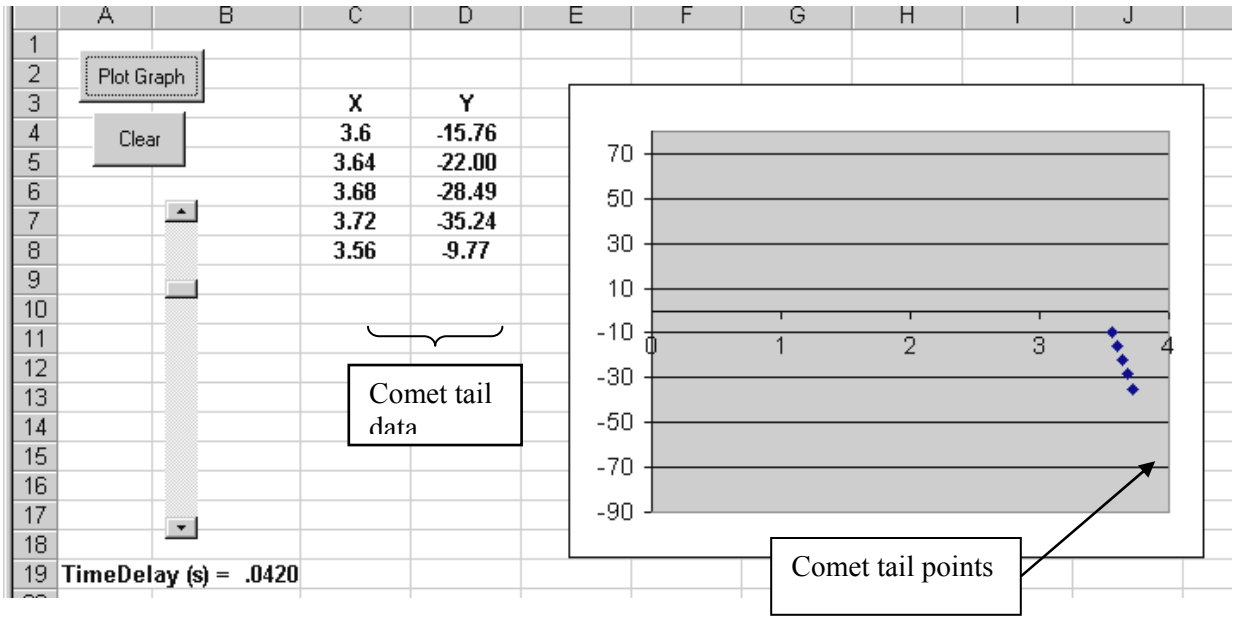
```
Private Sub cmdAddGraph_Click()
'  Range("C4:D5") = 0
```

and deleting the entries in cells **C4:D103**. Now press the **Add Graph** button followed by the **Plot Graph** button.. [Of course you could always press the **Plot Graph** button first to generate the data, and then press the **Add Graph** button, but this would eliminate the animation effect.] Another approach would be to zero the data field in the **Clear** button using **Range("C4:D5") = 0** .

Caution: Unless there is a need for altering certain graph characteristics or varying the size of the data set to be plotted "on the fly", it is generally safer to avoid adding graphs programmatically. Graph names disappear once a file is closed even though the graph itself may not have been deleted. If the file is then reopened, any attempt to alter a graph property without first re-adding the graph will fail since the program doesn't recognize the graph name. Of course, this will also happen if an attempt is made to change a graph property before the graph has actually been added.

5.3 Comet Tail Graphs : Application # 4 c)

Application # 4 a) is modified to reduce the set of plotted points to a moving collection of five points. Each newly calculated point replaces the oldest point in the collection so that a cluster of 5 points is tracked. The appearance of the display is what I'll refer to as a "**comet tail**" [often similar to a caterpillar moving across the screen]. The comet tails acts to represent a moving object during simulations. It is also useful for eliminating clutter when tracking an object's trajectory that crosses back over itself. In certain situations, such as the simulation of the simple harmonic motion of a pendulum, the display would appear inanimate after one cycle's worth of points were plotted if a comet tail were not used. In addition, the comet tail usually gives the user enough information about past performance to predict the immediate future location. This will be useful in the 2-D dynamics application in which a virtual spaceship is moved about the 2-D plane.



Save **Application # 4 a)** under a new name : add the code indicated in **bold**; the code in *italics* should already be present.

```
Option Explicit
Dim Delay As Double

Private Sub cmdGraph_Click()
Dim X As Double, Y As Double, Row As Integer
Delay = scbDelay.Value / 1000

Row = 4

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
DoEvents
```

```
Call TimeDelay(Delay)
Row = Row + 1

If Row = 9 Then Row = 4

Next X

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 scbDelay_Change()
Range("A19") = "TimeDelay (s) =  " & Format(scbDelay / 1000, "0.0000")
End Sub

Private Sub cmdClear_Click()
Range(Cells(4, 3), Cells(8, 4)).ClearContents
End Sub
```

The **comet tail** is created by this **If** statement. To begin with, the first data point is entered in row **4**. Once **5** points have been added (rows **4** through **8**), the **Row** index which has become **9** is reset to **4** by the **If** statement so that the next point replaces the oldest point (in row **4**). The points in rows **5**, **6**, **7**, and **8** in turn become the oldest points and are subsequently replaced, and the process repeats itself.

5.4 Plotting Trajectories

The techniques recently examined for plotting and animating graphs used loops that were "driven" by varying the independent variable and can be applied to simulations of trajectories of moving objects. A trajectory shows the actual path of motion in the X-Y plane (although it will sometimes also be useful to plot Y-t and X-t graphs to obtain information about the behaviour of position with time). The physics of the situation provides kinematics (motion) equations that describe an object's position and velocity as functions of time given its initial state (initial position and initial velocity) and the change which it experiences (caused by net force and described by acceleration). Although Y versus X graphs will be plotted, the loops will be driven by varying time which controls the X and Y positions.
[Refer to **parametric equations** in your math text.]

Application # 5 2-D Projectile Motion Under Constant Acceleration

The 2-D motion of a projectile in freefall can be completely described by the X and Y position and velocity functions. As freely falling motion occurs when the only force present is the gravitational force (which acts entirely in the negative Y direction) it follows that the X component of acceleration is zero. The relevant X and Y equations become :

Y Direction Freefall
Uniform (constant) acceleration

$$Y_f = Y_o + V_{oy}\,t + \frac{1}{2}a_y\,t^2 \quad \ldots. (1)$$

$$V_{fy} = V_{oy} + a_y\,t \quad \ldots.(2)$$

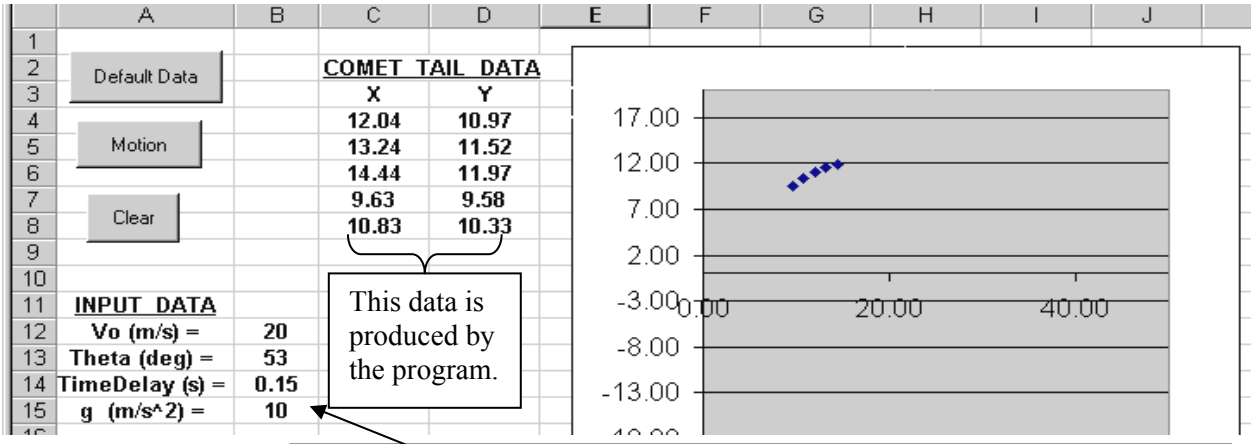
X Direction Freefall
Uniform (constant) velocity

$$X_f = X_o + V_{ox}\,t \quad \ldots. (3)$$

$$V_{fx} = V_{ox} \quad (a_x = 0;\ \text{const. vel}) \quad \ldots. (4)$$

Notice that the position and velocity functions are constructed from a knowledge of three parameters : the initial position, the initial velocity, and the acceleration (really the net force in disguise). These equations are valid only when the acceleration is constant.

The trajectory can be plotted by evaluating **Yf** and **Xf** at different times inside a loop that writes comet tail values back onto the worksheet. Open a new Workbook and save it under an appropriate name. Add 3 command buttons : Name = **cmdDefault**, Caption = **Default Data** ; Name = **cmdMotion** , Caption = **Motion** ; Name = **cmdClear**, Caption = **Clear** .



This data is produced by the program.

Notice that the amount of **Delay** is being read-in and is not set by a slider.

Enter the code for the position function in **Function Position(Xo, Vox, ax, t)** under **Option Explicit**. or in a **Module** if you wish to use, or test , the function on the worksheet.

```
Option Explicit

Function Position(Xo, Vox, ax, t)
Position = Xo + Vox * t + 0.5 * ax * t ^ 2
End Function
```

Refer to **Appendix 16** for a description of Function routines.

Add the code below to **cmdMotion** .

```
Private Sub cmdMotion_Click()
Dim X As Double, Y As Double, Row As Integer
Dim Vo As Double, Theta As Double, Vox As Double, Voy As Double
Dim Xo As Double, Yo As Double, t As Double
Dim g As Double, ax As Double, ay As Double, Delay As Double
Const Factor As Double = 3.1415926535897 / 180

' Read-in Input Data
Vo = Range("B12")
Theta = Range("B13") * Factor
Delay = Range("B14")
g = Range("B15")

' Set parameter values
ay = -g: ax = 0
Xo = 0: Yo = 0
Vox = Vo * Cos(Theta)
Voy = Vo * Sin(Theta)

Row = 4

For t = 0 To 4 Step 0.1

    X = Position(Xo, Vox, ax, t)
    Y = Position(Yo, Voy, ay, t)

    Cells(Row, 3) = X
    Cells(Row, 4) = Y
    DoEvents
    DoEvents
    Call TimeDelay(Delay)
    Row = Row + 1
    If Row = 9 Then Row = 4
Next t

End Sub
```

Observe that although a plot of **Y** versus **X** is being produced, the loop variable here is time which is used to determine **X** & **Y** from the position functions.

Notice that the same position function is used with different argument variables. In fact, it is only the values of the argument variables are passed to the Function, and not the variable names.

The addition of a second **DoEvents** eliminates a "jerky" graph display that sometimes occurs.

The comet tail is produced by the **If** statement which causes the oldest data to be overwritten.

Set the default data in **cmdDefault**.

```
Private Sub cmdDefault_Click()
Range("B12") = 20
Range("B13") = 53
Range("B14") = 0.15
Range("B15") = 10
End Sub
```

Next add the code to the **cmdClear** button, and then construct the **TimeDelay** subroutine.

```
Private Sub cmdClear_Click()
Range("C4:D8") = 0
End Sub

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

Same subroutine used earlier.

Add an **XY (Scatter)** graph using the **Chart Wizard** .

To avoid constant resizing of the graph due to **AutoScaling** you'll need to right-click on the graph axes and set the **Min** and **Max** values. For the set of input parameters given here you should use the following **Max** and **Min** values : **X** axis **0 to 50**, **Y** axis **–18 to 20**.

The trajectory can also be plotted by constructing the actual trajectory function : **Y = f(X)** and then programming this function into the loop of Application # 4 c). The two position functions are first simplified by dropping the subscript "f " and by assuming that Xo = 0 and Yo = 0 :

$$Y = V_{oy}\, t + \frac{1}{2} a_y t^2, \qquad X = V_{ox}\, t$$

Solving the X equation for " t " and eliminating " t " in the Y equation gives :

$$Y = \left[\frac{V_{oy}}{V_{ox}}\right] X + \left[\frac{1\, a_y}{2\, V_{ox}^2}\right] X^2 \qquad \ldots. (5)$$

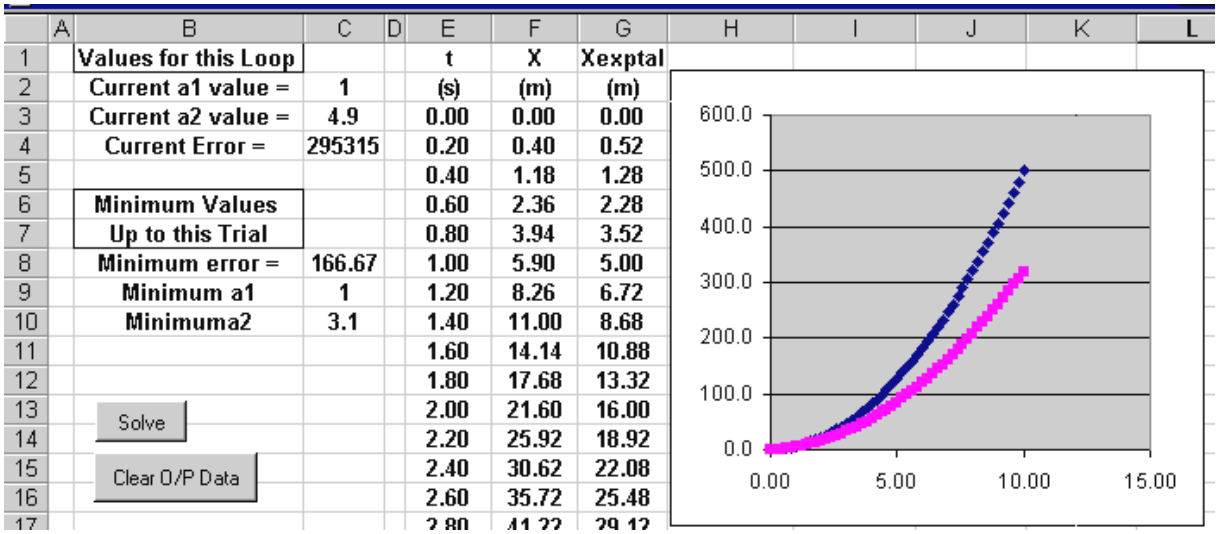
Freefall trajectories are termed parabolic because equation (5) is 2nd order, not because the Y(t) position function is quadratic !!

Caution : Many students confuse the Y-t graph of a 2-D projectile with its Y-X trajectory because of the similar shapes. In fact they are two separate entities. In your kinematics labs, the actual trajectory was generally used to produce the Y-t , X-t, and V_y-t graphs. Fitting functions to these graphs then gave the initial conditions Xo , Yo**, Vox, and Voy, and the acceleration. The simulations that we'll be developing are created in exactly the opposite sense. We'll start with a knowledge of the initial conditions and the acceleration [as determined from the net force], and use these parameters to define the position and velocity functions. The actual trajectory is then determined from the two position functions. In the lab we were studying the motion that had occurred; here we're creating the motion. [**Note that although the origin was set at some data point value, Xo & Yo as determined from the fitted function will be slightly different]

5.5 "Brute Force" Curve Fitting : Application # 6

Most curve fitting programs, such as the **ChartWizard**'s **trendline**, use some form of least squares minimization involving matrix inversions. A much less elegant approach is presented here to fit a second order function to a set of data. A general 2nd order function of time has the form $x(t) = a_o + a_1 t + a_2 t^2$. The process of fitting a function to the data means to find the constants a_o , a_1 , and a_2 that define the function which is closest to the trend described by the data. In the example below the experimental data was created assuming $a_o = 0$, so for simplicity we'll ignore a_o . The values of a_1 and a_2 are determined by "**exhaustive search**". Two loops are used to systematically vary the values of a_1 and a_2 over given ranges. For each set of a_1 and a_2 the associated fitted function is evaluated at each of 100 points and compared to the value of the experimental data. The square of the error between the fitted function value and each data value is summed over the 100 points. The values of a_1 and a_2 that give the minimum error become the "best" fit. An improved best fit can usually be found by narrowing the search region for a_1 and a_2. [Note : the assumption here is that the data is available over a range of values of independent variable "t" that can be conveniently accessed in a loop. If the data is not available at easily described intervals of the independent variable, a dummy index could be used to access the experimental data from an array.]

A graph of the points generated by each trial fitted function is plotted along with a graph of the original data so that it is possible to watch the "hunt" as the various fitted functions approach the actual data. This method has been used in an application that finds the drag coefficient and rolling friction acting on a car given experimental velocity-position.



Open and save a new workbook. Add 2 buttons : Name = **cmdSolve** , Caption = **Solve** ; Name = **cmdClear** , Caption = **Clear O/P Data**. Add the code given below.

```
Option Explicit

Function Exptal_Position(t)
Exptal_Position = 2 * t + 3 * t ^ 2
End Function
```

Appendix 16 presents guidelines for creating **Custom Functions**.

```
Private Sub cmdSolve_Click()
Dim a1_min As Double, a2_min As Double, Current_Minimum As Double
Dim a1 As Double, a2 As Double, x As Double, t As Double, Row As Integer
Dim bFlag As Boolean, Error As Double

bFlag = True
Current_Minimum = 500000#
t = 0
Row = 3

    For a1 = 1 To 10 Step 0.1

    Cells(2, 3) = a1

        For a2 = 1 To 10 Step 0.1

        Cells(3, 3) = a2
        Error = 0
        Row = 3

            For t = 0 To 10 Step 0.1
                x = a1 * t + a2 * t ^ 2
                Error = Error + (x - Exptal_Position(t)) ^ 2
                Cells(Row, 5) = t
                Cells(Row, 6) = x

            If bFlag = True Then Cells(Row, 7) = Exptal_Position(t)

                Row = Row + 1
            Next t

        DoEvents
        bFlag = False

            If Error < Current_Minimum Then
                Current_Minimum = Error
                a1_min = a1
                a2_min = a2
                Range("C8") = Current_Minimum
                Range("C9") = a1_min
                Range("C10") = a2_min
            Else
            End If
        Range("C4") = Error
        Next a2
    Next a1
End Sub
```

The result of typing **5E05** . Any reasonably large starting value could be used.

The **Exptal_Position(t)** function is evaluated over 100 time values in the inner time loop for each of the 10,000 combinations of **a1** & **a2**. To prevent needless repeated writing of the same values back on the worksheet and **If** statement is used. **bFlag = true** only when the function evaluation loop is first executed. The experimental data is thus entered into the cells only once.

The two outer loops select values of a_1 & a_2 to test in the third inner loop.

Evaluates fitted function **x(t)** constructed from the latest values of **a1** & **a2**

The **DoEvents** statement allows the graph to be updated on the worksheet.

If the error for the present set of **a1** & **a2** is less than the current minimum value, make the present set the new minimum.

```
Private Sub cmdClear_Click()
Range("E3:H300").ClearContents
End Sub
```

Program Structure : Two loops are used to select all possible values of **a1** and **a2** within the range and step size specified in the loop **For** statements. A value of **a1** is set in the outer loop, and then the inner loop cycles over all values of **a2**. A third inner loop evaluates the fitted function described by the current choice of **a1** and **a2** over 100 values of the independent variable (time). For each of the 100 time values, the error between the current fitted function being "tested" and the value of the known experimental function is determined, squared, and stored in the running total **Error** variable. [The error is squared to avoid negative under-estimation errors from canceling positive over-estimation errors and giving a false minimum.] The programming of the inner loop is not very efficient. Although the **bFlag** approach prevents the same values of **Exptal_Position(t)** from being repeatedly re-written on the workseheet, the value of **Exptal_Position(t)** is nevertheless calculated over the same 100 times for every set of **a1** & **a2 .**

As the search process can extend over several minutes (primarily because each fitted function is displayed on the graph) it is sometimes desirable to interrupt the search. Program execution can be halted by **Ctrl + Break** and selecting **End**. An alternative is to add a **Stop** button with the code :

```
Private Sub cmdStop_Click()
bStop = True
End Sub
```

Boolean variable **bStop** needs to be declared under **Option Explicit** to define it as a **Public** variable that can be used in more than one subroutine.

```
Option Explicit
Dim bStop As Integer
```

bStop is then initialized at the same time **bFlag** is in the **cmdSolve** routine.

```
bStop = False
bFlag = True
```

The following code is added just above **Next t** in the innermost loop :

```
DoEvents
DoEvents
If bStop = True Then GoTo Out
```

The **DoEvents** interrupt program execution to check for other events, such as the user pressing the **Stop** button in this case.

Out is the name of a code line that is added just before **End Sub**.

```
Out :
End Sub
```

Any numeral or string can be used to name a code line. Code can be added to the right of the colon.

5.6 Plotting Functions of Two Variables : Application # 7 a)

The Chart Wizard in general is aimed at business applications and is quite unscientific and unmathematical when it comes to plotting the surfaces associated with functions of two variables such as z = f(x, y). However, the results can still be magical. The main shortcoming of the **Excel Chart Wizard surface** plot is that it wants to plot "items" along the Y axis , "series of items" along the X axis, and only becomes quantitative when it plots actual values in the Z direction. Consequently, the graphs produced never have proper X and Y axes that represent real numbers. We'll work around this problem by establishing our own X and Y axes (but which are not be included in data of the final plot).

Two versions of this application are presented. **Application 6a)** uses a short VBA program to calculate the function values and to display the axis values. **Application 6b)** skips the program and operates entirely on the worksheet (except for the function definition). In both applications the surface graph is added manually; one of the assignment exercises at the end of the chapter requires you to add the graph programmatically to **Application 6a)** .

Open and save a new Excel workbook. Insert a **code module** from the **VB Editor** using either the **Insert** menu ⇒ **Module ,** or by right clicking a blank region of the **Project Explorer** ⇒ **Insert** ⇒ **Module** and define the following function :

```
Function Funct1(X, Y)
Dim A As Double, B As Double, C As Double
A = 0.5: B = -0.8: C = 0.2
Funct1 = A / ((X - 0.55) ^ 2 + (Y - 0.55) ^ 2) + _
         B / ((X - 1.05) ^ 2 + (Y - 1.45) ^ 2) + _
         C / ((X - 1.65) ^ 2 + (Y - 0.85) ^ 2)
End Function
```

Don't forget the **space** before the continuation underscore character !

Test the function by calling it in any cell on your worksheet. Do not use values of X & Y that are located near one of the singularities. Try = **Funct1(0.5, 0.7)** and compare with the value from the table below. If you obtain **#NAME** in the cell you probably defined the function in the **Sheet1 code window** by mistake, and not in a code **module**.

Add two buttons : **Name = cmdClear**, **Caption = Clear Data** ; **Name = cmdCalculate**, **Caption = Calculate Function Values.**

	A	B	C	D	E	F	G	H	I	J	K	
1												
2	Calculate Function Values	Y/ X		0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	
3												
4		0.1		0.777	1.008444	1.299167	1.63522	1.959992	2.168655	2.164083	1.94743	1
5		0.2		0.94	1.284714	1.769915	2.415319	3.146009	3.685659	3.67789	3.124442	2
6		0.3		1.106	1.601179	2.394665	3.669714	5.531139	7.323077	7.310066	5.494695	3
7	Clear Data	0.4		1.244	1.899995	3.096688	5.500638	10.69984	19.56175	19.54023	10.63905	5
8		0.5		1.311	2.075019	3.59758	7.24906	19.51492	99.47451	99.43928	19.41463	7
9		0.6		1.273	2.027853	3.538746	7.17584	19.42439	99.36392	99.30681	19.26051	6
10		0.7		1.131	1.758621	2.919825	5.279643	10.42511	19.22384	19.13203	10.15944	4
11		0.8		0.919	1.366179	2.099025	3.297406	5.063403	6.739803	6.593303	4.635458	2
12		0.9		0.68	0.957721	1.35534	1.887436	2.472623	2.828629	2.596837	1.786917	0
13		1		0.45	0.59306	0.767714	0.949442	1.068405	1.003565	0.641494	-0.0231	-
14		1.1		0.245	0.290747	0.322104	0.308081	0.193649	-0.0986	-0.65052	-1.51959	-

Enter the code given below :

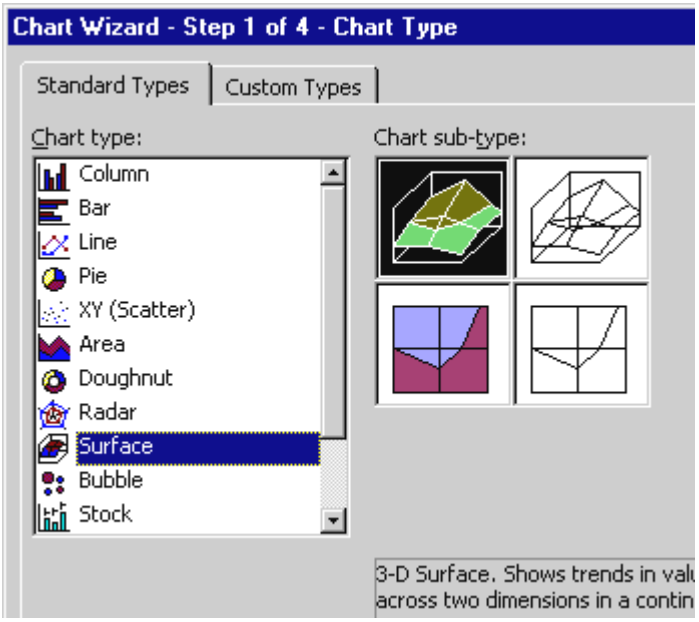
```
Option Explicit

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

Range("B2") = "X/ Y"
Row = 4

For Y = 0.1 To 2.1 Step 0.1

    Col = 4
    Worksheets("sheet1").Cells(Row, Col - 2) = Y

    For X = 0 To 2 Step 0.1

        If Row = 4 Then Cells(Row - 2, Col) = X
        Worksheets("sheet1").Cells(Row, Col) = Funct1(X, Y)
        Col = Col + 1
    Next X

Row = Row + 1
Next Y

End Sub

Private Sub cmdClear_Click()
Range("B2: W23").ClearContents
End Sub
```

Col must be re-initialized to 4 prior to filling each row from left to right.

The statements in un-bolded italics are used to write the X and Y axis values across row 2 and down column 2.

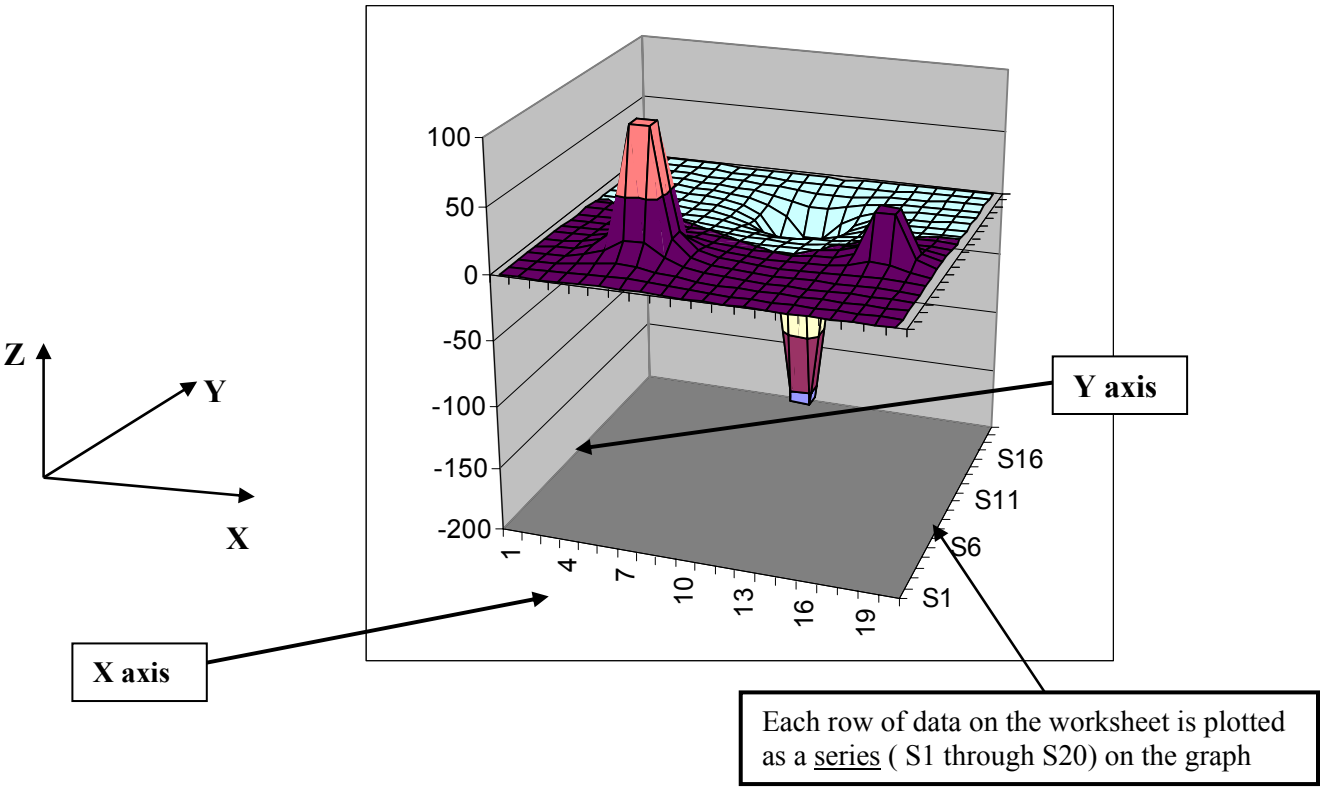
The inner **X** loop causes the function to be evaluated over the range of **X** values for one particular value of **Y**. These values are written across each row. The next value of **Y** is generated in the outer loop and the function values are calculated at different **X** values in the inner loop and are written across the next row down.

Run the program and check that the data agrees with the sample output. Highlight only the function values, and not the X and Y axis values, click on the **Chart Wizard**, select a **Surface** graph, use the default type, and delete the legend.



Notice that on the worksheet the positive X axis lies across row 2 and the positive Y axis down column 2. The label **Y/ X** in cell **B2** is effectively the origin. The function is evaluated at different X values across each row in the inner X loop (at a fixed value of Y set in the outer Y loop). The function must be evaluated in this manner if a right-handed plot is to be obtained as shown in the graph below.

The graph obtained should look like that shown below.

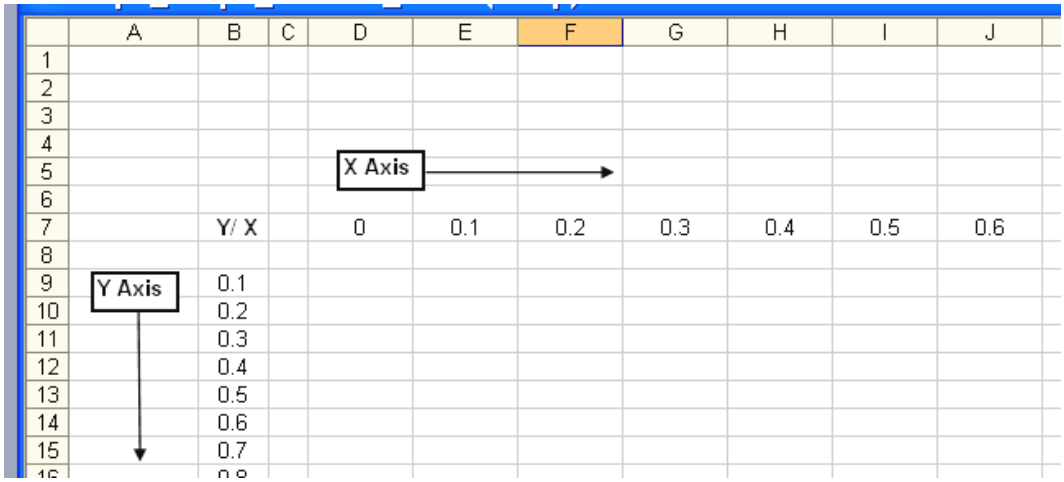


Activate the graph (click somewhere inside it) and hover the cursor at corner until a “corners” label appears beside the cursor. Click, hold and rotate the display. Be suitably impressed.

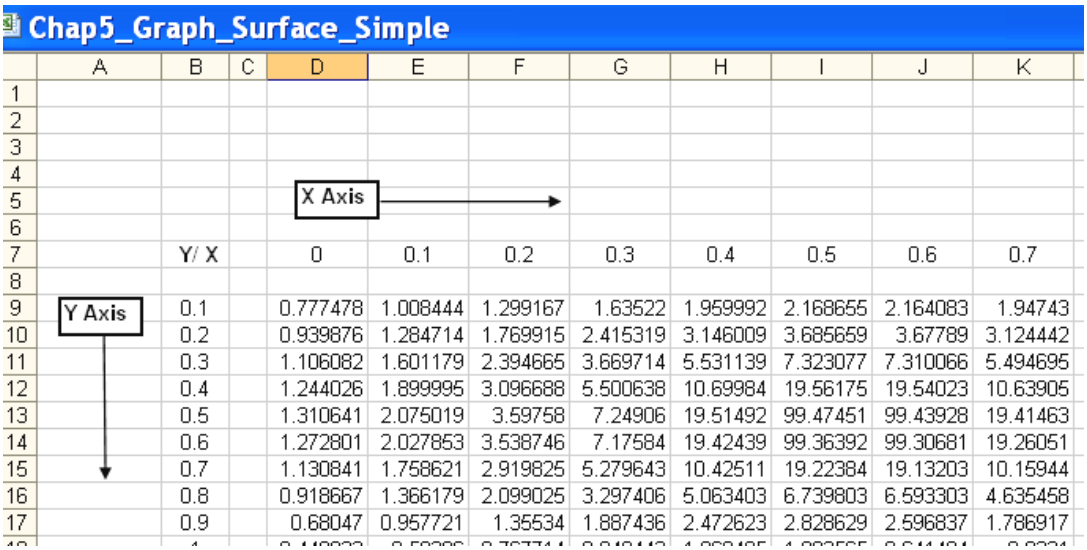
Application # 7 b) -- Plotting Functions of Two Variables *Without Using a VBA Program*

The only programming required here is entering the custom function **Funct1** in a code **Module**. Open and save a new workbook. Copy the **Funct1** routine from **Application # 6a)** and paste it into the **Module** window.

Set up the X and Y axis values at which the function will be evaluated. Use values of each variable between 0 and 1.9 and 0.1 and 2 in steps of 0.1 . The X axis is contained in the column down the left; the Y values appear in the row along the top. Do not type in these axis values: either use a formula and then **Fill**, or enter the first two values, highlight the two cells, hover the cursor at the corner of the second cell until it becomes a + and then drag the relationship over as many cells as desired.



Enter your function **Funct1** into the top left cell of the grid over which it is to be evaluated (**D9** in this example). You will need to use an absolute reference to the column containing the X values, and to the row containing the Y values as follows : = **Funct1(D$7, $B9)**. Copy { **Fill** } this formula over the field of cells down to the bottom right using **CTRL + D ,** keep holding the **CTRL**, and then press **R** .



Now use the **Chart Wizard** to add a surface plot.

Problems (Chapter 5) ALL GRAPHS SHOULD APPEAR ANIMATED

1. **Families of Curves.** The **Ideal Gas Law** [**PV = nRT**] has the form **PV = constant** for a particular temperature. Construct a program that will plot up to 5 different curves corresponding to 5 different constants on the same graph. The user should enter the value of the **constant** (or **T** if you wish) into a single cell; the curve for that value will be added to the family of curves when the **Run** button is activated. Each set of data should be written into a different pair of columns with the associated value of the **constant** (or **T**) written at the top. The data for the 6^{th} value entered should overwrite the 1^{st} set, and so on. The comet tail approach for overwriting data can be used to determine which column the current data will be written into; however, since the value of **Column** must be retained it needs to be declared globally and initialized in a separate subroutine **cmdInitialize**. For example, 5 sets of data starting in columns 5 and 6 and ending in columns 13 and 14 could be created initializing **Column = 5**, and adding **Column = Column + 2** : **If Column = 15 Then Column = 5** just before **End Sub** in the **Run** routine. The values of **Column** would cycle through 5, 7, 9, 11, 13, 5, 7, 9 …. Data would be written into columns **Column** and **Column + 1** . [***Substitute with February, 2005 version]

2. This problem constructs an animation of two pulses on a string that approach one another. A travelling wave pulse starting from the origin and moving in the positive X direction is described by the function **Amplitude1 = 20 * Exp(-((X - t) ^ 2))** -- a redundant pair of brackets essential to an Excel 97 spreadsheet version of this program have been left in. A second pulse starting at X = Xmax and moving to the left is described by the function : **Amplitude2 = 20 * Exp(-((Xmax - X - t) ^ 2))** -- which also includes redundant brackets not essential to the VBA version of this problem. The program should write the amplitudes of the two waves on the spreadsheet as a function of X (at steps of about 0.4 m); the sum of the two amplitudes should also be displayed. All data should be displayed; do not use a comet tail. Use the **Chart Wizard** to plot the results. **Hint:** use an outer loop that iterates over time, and an inner loop that iterates over distance. The location of the pulses for each value of time will be described by the inner loop. The animated effect occurs as time is varied in the outer loop. Try locating the **DoEvents** in the outer loop, and then in the inner loop. The graph should extend a little beyond **Xmax** in order to see the complete right pulse. Choose Xmax = 20 m, Tmax = 15 s, and Δt = 0.4 s to begin with. Adjust the value of Δt. [The original idea for this problem was presented by Ole Haugland in **The Physics Teacher**, January, 1999, Vol. 37, pg 14 and was the inspiration behind the development of this course.]

3. Add **If** statements to **Application # 4a)** that determine the maximum and minimum value of the function, and the value of independent variable for which they occurred. Modify the code so that the user can specify the "window" over which the function is evaluated. [The **Autoscale** feature of the graph will likely be annoying; a more sophisticated approach is to add the graph programmatically as in **Application # 4 b)** and then use the macro recorder to determine how to programmatically set the graph axes **Max** and **Min**.]

Equations (1) & (2) will be useful in problems # 4 to 6.

For objects undergoing **uniform(constant) acceleration** the position-time and velocity-time functions are :

$$X_f = X_o + V_{ox}\,t + \frac{1}{2}a_x t^2 \quad \ldots(1) \qquad V_{fx} = V_{ox} + a_x t \quad \ldots(2)$$

4. A 2,000 kg car experiences a constant net force of 10,000**i** Newtons. The car has an initial velocity of Vox at a location Xo, both of which are read in from the worksheet along with the net force. Construct a simulation of the motion that plots the trajectory Y vs X (note that Y = 0 for the entire motion). Also plot graphs of X vst t and Vx vs t.

5. A 2,000 kg rocket powered snow plow is moving in a straight line (along the positive X axis) and is subject to a constant net force of + 48,000 **i** N as it travels 400 meters. It then enters a large snow bank that is 600 meters long and which results in an unrealistically constant net force of – 32,000 **i** N on the plow. Plot graphs of **a** vs **t**, **Vx** vs **t**, and **X** vs **t** . Use a single coordinate system located at the initial position. And adjusted times for the 2nd region.

6. A speeding car (Vo = 25 m/s, a = 3 m/s^2) passes a police car that is at rest but which immediately starts accelerating at 5 m/s^2. Construct a program that evaluates the position and velocity of each car at 0.1 s intervals and which writes the information onto the worksheet. Add two animated graphs of the complete data: one showing the position of both cars as a function of time, the other showing the velocity of both cars as a function of time. Also add a comet tail Y vs X trajectory graph showing both cars on the same graph (note : Y = 0 for both cars at all times). Select the origin to be at the initial location of the police car so that Xo = 0 m for both vehicles. Add code that determines the time and location for which the cars are closest together and their velocities at that moment. Choose values of the input parameters which describe two approaching vehicles that are decelerating. Compare with the theoretical solution. Consider the modifications to allow a delayed start for one of the vehicles.