CHAPTER # 6 INCREMENTAL ITERATION APPLIED TO TRAJECTORIES

Objects that experience a constant resultant force and constant acceleration, such as a projectile moving through the air (ignoring drag), or a puck moving on an inclined air table, have a motion that can be easily modeled by the standard kinematics equations. A great many situations of interest, however, are characterized by the presence of forces that vary and which therefore cannot be analyzed using the kinematics equations. Two alternatives exist : the work-energy approach, and the differential equation solution to the dynamics equation. Unfortunately, the energy approach does not easily lend itself to constructing actual trajectories nor to dealing with varying forces other than position dependent conservative ones that are modelled by potential functions, and the differential equation approach is restricted to simple situations and geometries. A third alternative, **incremental iteration**, is available which, although relying on the computational aid of a computer, is much more flexible than the other two from a practical point of view.

The following three chapters focus on incremental iteration. Chapter 6 explores incremental iteration applied to 1-D situations involving air resistance, spring forces, and variable thrusts, and to two simple, constant acceleration 2-D problems. A number of useful programming tools are developed such as automatic reversal of force directions for friction and air resistance, modelling time variable forces and masses during the flight of a rocket, making program execution interactive, and using hidden loops to reduce data output. Chapter 7 examines more complicated 2-D dynamics problems subject to varying X and Y accelerations. Chapter 8 applies incremental iteration to numerical integration.

6.1 Incremental Iteration

Incremental iteration is a technique for analyzing a large scale situation, which would otherwise be difficult to analyze, by breaking it into a number of smaller parts. The contribution of each smaller part is determined and is added back to re-construct the original situation. The analysis is based on two conditions : the principle that the whole is simply the sum of the parts (superposition), and the fact that the behaviour of the infinitesimal parts can be described by fundamental relationships that are theoretically known and easy to work with. Incremental iteration is simply a computational means of applying the limit concept of calculus to create "sufficiently small" contributions which ensure the error in the approximations cancels out.

In physics there are two general situations in which incremental iteration can be applied, and in two entirely different ways :

i) The first involves the study of change in a system as it moves from one state to another such that the overall change is simply the sum of a number of smaller changes. The examples of this situation considered here and in the next chapter all relate to the motion of an object and the construction of its **trajectory**[1]. The state of the system at any time is defined by its location and velocity (**xf, Vfx**) and is subject to change described by the acceleration (which accounts for the net force acting

---

[1] Provided the laws, relationships, or rules that describe small scale change are known, it should be possible to model the change of state of any type of system in the same way. Indeed, the concept of a trajectory can usefully be extended beyond mechanics. In the very general sense, a trajectory might be viewed as a description of how a system responds to change in terms of the different states that it moves through. It is therefore possible to conceive of the trajectory of financial, economic, biological, chemical, behavioural, etc systems, and so it would not be inappropriate to attempt to apply an incremental analysis to such systems.

on the particular mass) . Each motion is divided into a large number of incremental motions which are analyzed separately and then linked together to form the overall motion.

ii) The second situation involves determining the physical effect produced by some large scale object as a sum of the contributions of the smaller components of which it is comprised. For example, charge causes electric field and mass causes gravitational field[2]. The field due to a continuous distribution of charge or mass can be obtained by mathematically chopping the object into a large number of small elements of charge or mass. The contribution of each element to the field is determined as that of a point charge or point mass and then all contributions are summed together to form the total field. The use of incremental iteration here is nothing more than an application of **numerical integration** to integrals developed in the usual theoretical analysis encountered in your Electricity & Magnetism course. Applications of this type are presented in chapter 8.

Since in both cases the change or the object are being broken down into smaller parts, analyzed, and then combined back together, I'll loosely refer to the technique as a "**Chop & Sum**" method.

6.2 Incremental Iterative Analysis of Trajectories

The need for incremental iteration in the study of an object's motion arises when the acceleration of the object is not constant so that the standard kinematics equations cannot be applied.

Standard kinematics equations for uniform (constant) acceleration

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

The acceleration will vary if the force is not constant due to either a changing magnitude or changing direction, or if the mass is not constant. Some examples of variable forces and varying masses are :

a) **variable magnitude forces** :
   - position dependent forces (spring, gravity, Coulombic).
   - velocity dependent forces (air resistance - - also known as drag).
   - time dependent forces (any force that is changed with time such as due to altered engine power in a car, or variable thrust in a rocket).

b) **variable direction forces** :
   - gravitational or Coulombic forces on objects moving in two dimensions through space (except near the surface of the Earth where gravity is assumed to point downward.)
   - circular motion problems such as the pendulum, or roller coaster, or any situation where a tension, normal force, or the like, acts on an object moving over a curved path.

   [In general, with the exception of projectile motion situations, 2-D problems will contain forces that change orientation.]

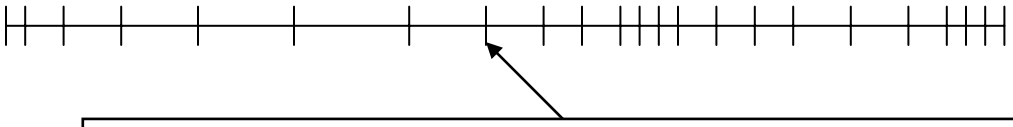
c) **variable mass situations** :(a rocket engine/jet engine which burns fuel and ejects mass)

---

[2] In fact the relationship doesn't necessarily need to be **causal**. One of the properties of distributed mass is rotational inertia (moment of inertia) which, as will be demonstrated in Chapter 8, can be easily calculated using incremental iteration

The incremental iterative analysis of motion involves allowing the object to move for only small[3] intervals of time Δt so that it travels over correspondingly small distances. Within such small intervals the variation in the acceleration can be ignored since the over- and under-estimation errors cancel out, or can be reduced to tolerable levels by making **Δt** sufficiently small. In the limit, as **Δt** approaches zero, any desired level of accuracy can be achieved. Since the acceleration is effectively constant within the small intervals, the standard kinematics equations are now valid and may be applied. [On the macro scale the equations cannot be applied; on the micro scale they can.]

As the equations are applied for a short period of time, and then re-applied for the next interval, a link must be made such that the final values of position and velocity at the end of one interval become the initial values at the start of the next interval. The process is known as matching the **boundary conditions** and it ensures the continuity of the motion[4]. A schematic representation of a 1-D motion would show it chopped up into a collection of small displacements as depicted below. [**Note**: **Δt** is assumed to be constant for each of the displacements. Acceleration is occurring when successive displacements are increasing, and deceleration occurs when they decrease.]



Boundary Conditions
At the end of each interval (each iteration), the final position and velocity are used to define the initial position and velocity for the next interval.

$$X_o = X_f \quad \text{and} \quad V_{ox} = V_{fx}$$

---

[3] Small is definitely a relative term here. In certain applications we'll be moving for 10 to 100 **microseconds**, while for spaceflight applications we may be moving for 10 to 100 **seconds**.

[4] Matching the boundary conditions for the position and velocity functions obscures the fact that both the position and velocity functions are nothing more than running totals with change terms, $\mathbf{(1/2)a\Delta t^2}$ and $\mathbf{a\Delta t}$, included. Refer to **section 6.6** .

The iterative process just described can be broken down into a number of programmable steps. For a general 2-D motion in the X-Y plane subject to both X and Y components of net force the flow chart appears as follows :

Set Initial Conditions { = Launch conditions }

$[X_o , Y_o , V_{ox} , V_{oy}]$

START ITERATION

FnetX = some value
FnetY = some value

a_x = FnetX/ mass

a_y = FnetY/ mass

Calculate Final Position & Velocity after a time Δt

$$X_f = X_o + V_{ox}\,\Delta t + (\tfrac{1}{2})\,a_x\,\Delta t^2$$
$$Y_f = Y_o + V_{oy}\,\Delta t + (\tfrac{1}{2})\,a_y\,\Delta t^2$$
$$V_{fx} = V_{ox} + a_x\,\Delta t$$
$$V_{fy} = V_{oy} + a_y\,\Delta t$$

Notice that the motion equations are using the value "Δt", and not "t".

Store (X_f , Y_f) on the Worksheet (or in an Array)

Set Initial Conditions at start of the next iteration to Final Conditions at the end of this iteration

$$Y_o = Y_f$$
$$X_o = Y_f$$
$$V_{oy} = V_{fy}$$
$$V_{ox} = V_{fx}$$

Boundary Conditions

END ITERATION & RETURN TO START
{Exit Loop at Specified Condition}

Plot Stored Positions (X, Y) (if not automatically plotted)

"If you've done one, you've done them all !!" (almost)

Incremental iteration programs are essentially the same in all situations. What does change is the description of the net force which must clearly be tailored to the specific environment, plus, of course, any "bells & whistles" that you may decide to add.

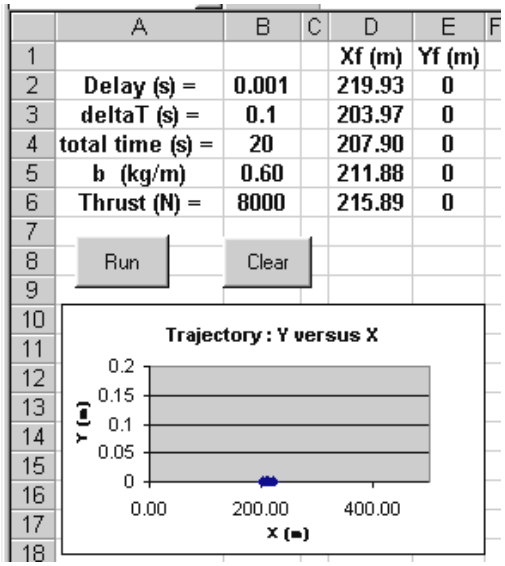
Time saving manoeuvre : I'm a slow typist and have found it useful to create three incremental iteration "shell" programs : one for 1-D X motion, one for 1-D Y motion, and a third for 2-D X/Y motion. (My shells contain hidden loops which you'll appreciate later in this chapter offer more flexibility.) In some of the following applications I've distinguished between new and existing code based on 3 different shells.

6.3 Incremental Iteration Applied to 1-D Situations:
Application # 8 : Horizontal Motion of Rocket Car with Drag

This application uses incremental iteration to construct the trajectory (**Y vs X**) of a rocket car moving along a frictionless horizontal surface in the presence of air resistance and acted on by a constant horizontal thrust. **Air resistance** (**drag**) will be modelled by the expression **b*Vox^2** . The net force acting on the rocket car is therefore : **Fnetx = Thrust – b * Vox ^ 2** . The car is assumed to move only to the right since there is no provision for reversing the direction of the drag (refer to the problem that follows). **Data :** mass of car = 2000 kg, Xo = 0 m, Vox = 0 m/s, b = 0.6 kg/m , deltaT = 0.1 s , Thrust = 8000 N, Delay = 0.001 s. (Also try deltaT = 0.01 s & Delay = 0 s)

```
Option Explicit
Dim Delay As Double

Private Sub cmdRun_Click()
Dim Xo As Double, Vox As Double, Xf As Double, Vfx As Double, mass As Double
Dim ax As Double, t As Double, deltaT As Double, Fnetx As Double
Dim row As Integer, Total_time As Double
Dim b As Double, Thrust As Double

Delay = Worksheets("sheet1").Range("B2")
deltaT = Worksheets("sheet1").Range("B3")
Total_time = Worksheets("sheet1").Range("B4")
b = Worksheets("sheet1").Range("B5")
Thrust = Worksheets("sheet1").Range("B6")

mass = 2000: Xo = 0: Vox = 0
row = 2
t = 0

Do
   Fnetx = Thrust – b * Vox ^ 2
   ax = Fnetx / mass

   Xf = Xo + Vox * deltaT + 0.5 * ax * deltaT ^ 2
   Vfx = Vox + ax * deltaT

   Xo = Xf
   Vox = Vfx
   t = t + deltaT

   Call TimeDelay(Delay)
   Worksheets("sheet1").Cells(row, 4) = Xf
   Worksheets("sheet1").Cells(row, 5) = 0

   DoEvents: DoEvents
   row = row + 1
   If row = 7 Then row = 2

Loop Until t >= Total_time

End Sub
```

Delay must be declared as a global variable since it is used in more than one subroutine.



Note that **Δt** is used in the position and velocity functions.

The final position and velocity at the end of each iteration become the initial position and velocity for the next iteration.

Time is "created" or defined using a cumulative total.

Since the car only moves horizontally the **Y** position is always zero.

DoEvents allows the data points to be displayed as they're generated; the second **DoEvents** is sometimes necessary to stop the display from being "jerky".

If used to create the comet tail.

```
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()
 Worksheets("sheet1").Range("D2:E6") = 0
End Sub
```

Save this program as a shell for future applications involving 1-D incremental iteration in the X direction.

Note : The choice of **Δt = 0.1** seconds is very likely too large to ensure that the net force is constant over each interval, and so considerable error will result. Try smaller values of **Δt**, but set **Delay = 0**.

An alternative approach is to construct the iterative loop using a **For … Next** as :

```
For t = 0 To Total_time Step deltaT
……
Next t
```

While arguably more straightforward in the current application, this approach is risky since novice programmers often end up corrupting the value of **t** when using the "hidden loops" (introduced later in the chapter) to reduce the amount of data (**section 6.5**) . In addition, the ability to prescribe other exit conditions makes the **Do…Loop** more attractive.

Overflows : No provision has been made for allowing the drag force to change direction (it always acts to the left); this means that the car can only be allowed to move to the right. If a user inadvertently applies a negative thrust to the car (starting from rest) it will begin to accelerate and move to the left. The drag force erroneously will also be acting to the left which will increase the net force (rather than decrease it). A positive feedback situation results in which the drag force acts to increase the speed of the car thereby increasing the drag and thence increasing the velocity at an even greater rate. Overflows for either the velocity or net force will rapidly occur.

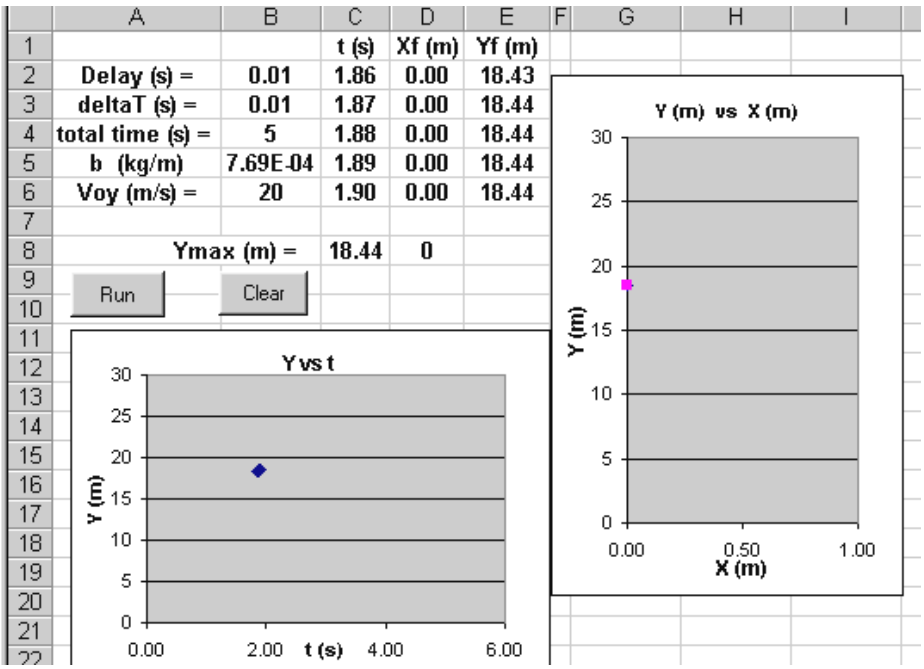
Application # 9 : Projectile Launched Vertically Upward with Drag

The 1-D projectile motion of an object launched vertically with a velocity **Voy** in the presence of air resistance can be simulated using incremental iteration by describing the net force as :

Fnety = – W – Sgn(Voy) * b * Voy ^ 2

Programming the Reversal of Force Directions
Unlike the rocket car, the object here reverses direction once it reaches maximum height meaning that the drag force direction must be reversed. A simple method of reversing a force direction due to a change in direction of motion is to apply the **Sgn( )** function. **Sgn( )** is a function that determines the sign of the argument (the velocity in this case) returning + 1 when the argument is positive or zero and –1 if the argument is negative. Since drag always acts opposite to the velocity the inclusion of a minus sign, **– Sgn(Voy)** , produces the correct direction of the drag. [If the object was only moving upward , and the motion stopped at maximum height, then the **– Sgn(Voy)** could be removed.] This technique will also be applied to reversing the direction of friction when the object changes direction in **Application # 10** .

The **Y vs X** trajectory of the projectile will move up and down the Y axis. A graph of **Y vs t** has been added to highlight the distinction between the two types of graph.



Save this program as a shell for future applications involving 1-D incremental iteration in the Y direction.

Air Resistance : A simplified model of drag has been used **Drag = b*v^2** ; the complete expression is **Drag = 0.5*ρ*A*Cd *V^2** where ρ is the density of air (1.29 kg/m^3), **A** is the cross-sectional area of the object, and **Cd** the drag coefficient. Comparing the two equations indicates that the value of **b** is **0.5*ρ*A*Cd** . For a baseball **Cd** = 0.284 [see **Serway & Beichner, Physics for Scientists and Engineers**, 5th Edition, pg 169] and **A** = 4.2 x 10^{-3} m^2 which produces a value of **b** = 7.69 x 10^{-4} kg/m . Without drag the maximum height should be 20.4 m compared with the value of 18.44 m found here. **Update** : In the 6th Edition, pg 167, **Serway & Jewett** use ρ = 1.20 kg/m^3 for the same problem, which is a more commonly accepted value.

```
Option Explicit
Dim Delay As Double

Private Sub cmdRun_Click()
Dim Yo As Double, Voy As Double, Yf As Double, Vfy As Double, mass As Double
Dim ay As Double, t As Double, deltaT As Double, Fnety As Double
Dim row As Integer, Total_time As Double
Dim b As Double

Delay = Worksheets("sheet1").Range("B2")
deltaT = Worksheets("sheet1").Range("B3")
Total_time = Worksheets("sheet1").Range("B4")
b = Worksheets("sheet1").Range("B5")
Voy = Worksheets("sheet1").Range("B6")

mass = 2: Yo = 0
row = 2
t = 0

Do
   Fnety = –mass * 9.81 – Sgn(Voy) * b * Voy ^ 2
   ay = Fnety / mass

   Yf = Yo + Voy * deltaT + 0.5 * ay * deltaT ^ 2
   Vfy = Voy + ay * deltaT

   Yo = Yf
   Voy = Vfy
   t = t + deltaT

   Call TimeDelay(Delay)
   Worksheets("sheet1").Cells(row, 3) = t
   Worksheets("sheet1").Cells(row, 4) = 0
   Worksheets("sheet1").Cells(row, 5) = Yf

   DoEvents : DoEvents
   row = row + 1
   If row = 7 Then row = 2
Loop Until t >= Total_time
End Sub

Private Sub cmdClear_Click()
Worksheets("sheet1").Range("C2:E6") = 0
End Sub
```

The **TimeDelay** subroutine is the same as used in the previous application.

The location of the maximum height can be plotted on the **Y vs X** graph by including the code :

```
Dim Ymax as double
Ymax = 0        ' locate before the loop
If Yf > Ymax Then Ymax = Yf     ' locate inside the loop
Worksheets("sheet1").Range("C8") = Ymax       ' locate inside the loop
```

A second series is then added to the graph plotting **Ymax** and the value **0** for the X coordinate.

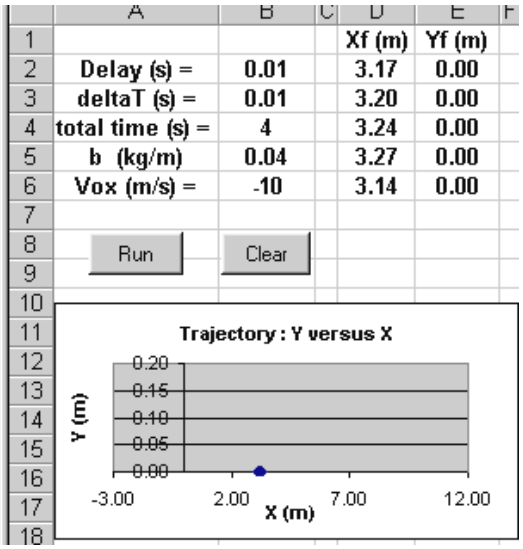
Application # 10 : Block Bouncing Off Spring on Horizontal Surface with Friction & Drag

A 2 kg block is initially located at Xo = 3 meters and is travelling at –10 m/s (= Vox) toward a spring (k = 40 N/m). The unstretched location of the spring is at X = 0; the left end of the spring is attached to a wall somewhere on the negative X axis. The spring is not attached to the block. A constant frictional force **Ff** of 8 newtons acts on the block. Construct a simulation that displays the trajectory of the block.

The situation is described by a net force : **Fnetx = –k * Xo –Sgn(Vox) * Ff – Sgn(Vox) * b * Vox ^ 2** with the block in contact with the spring, and **Fnetx = -Sgn(Vox) * Ff – Sgn(Vox) * b * Vox ^ 2** otherwise. Use b = 0.04 kg/m.

```
Option Explicit
Dim Delay As Double

Private Sub cmdRun_Click()
Dim Xo As Double, Vox As Double, Xf As Double, Vfx As Double, mass As Double
Dim ax As Double, t As Double, deltaT As Double, Fnetx As Double
Dim row As Integer, Total_time As Double
Dim b As Double, Ff As Double, k As Double

Delay = Worksheets("sheet1").Range("B2")
deltaT = Worksheets("sheet1").Range("B3")
Total_time = Worksheets("sheet1").Range("B4")
b = Worksheets("sheet1").Range("B5")
Vox = Worksheets("sheet1").Range("B6")

mass = 2: Xo = 3: k = 40: Ff = 8
row = 2
t = 0

Do
   Fnetx = –Sgn(Vox) * Ff – Sgn(Vox) * b * Vox ^ 2
   If Xo < 0 Then Fnetx = –k * Xo –Sgn(Vox) * Ff – Sgn(Vox) * b * Vox ^ 2

   ax = Fnetx / mass

   Xf = Xo + Vox * deltaT + 0.5 * ax * deltaT ^ 2
   Vfx = Vox + ax * deltaT

   Xo = Xf
   Vox = Vfx
   t = t + deltaT

   Call TimeDelay(Delay)
   Worksheets("sheet1").Cells(row, 4) = Xf
   Worksheets("sheet1").Cells(row, 5) = 0

   DoEvents: DoEvents
   row = row + 1
   If row = 7 Then row = 2

Loop Until t >= Total_time
End Sub
```

Alternative code for specifying the value of **Fnetx** is given on the next page.



```
Private Sub cmdClear_Click()
 Worksheets("sheet1").Range("D2:E6") = 0
End Sub
```

The programming of **Fnetx** used above defined a default value which was changed if the block happened to be in contact with the spring. The **If…Then…Else** structure shown below is an alternative which might be attractive if the expressions for **Fnetx** are large and complex. This approach would not always, and unnecessarily, evaluate the default expression and thus execution should be quicker (depending on the time requirements of the **If** vis-a-vis the **Fnetx** algebra).

```
If Xo < 0 Then
   Fnetx = –k * Xo  – Sgn(Vox) * Ff – Sgn(Vox) * b * Vox ^ 2
Else
   Fnetx = –Sgn(Vox) * Ff – Sgn(Vox) * b * Vox ^ 2
End If
```

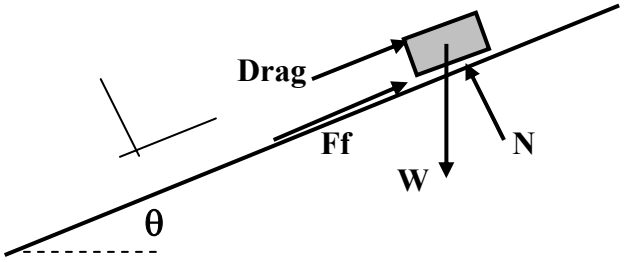
Caution : Choose the physics and simulation parameters carefully. A poor choice of physics parameters (such as mass and b here) can interact very unfavourably with a poor choice of simulation parameter Δt .For example, An ill-advised combination of a large Δ**t** (= 0.1 s) and an unrealistically large **b** (= 6 kg/m) in the presence of a small mass can lead to some "wonky" results. The initial velocity of – 10 m/s will produce a huge drag acting on the small mass for the relatively long time interval of 0.1 s. Large accelerations result in even larger velocities and drag forces. The direction of **Fnetx** flip-flops and an overflow will occur after a mere 0.9 seconds ! Try it. [A choice Δ**t** = 0.06 s is small enough for this **b** and mass to allow the motion to decay to zero velocity.]

Be mindful of the underlying calculus principles. The use of the standard kinematics equations is only valid provided the acceleration is constant over each interval. The overflow described in the previous box can be traced to a value of **Fnetx** that was, in reality, varying significantly over the 0.1 s interval, but which was held fixed in the simulation. The solution is to recall from Cal class that the error in the approximation can be made manageable provided that Δt is chosen sufficiently small. In the limit, as Δ**t** approaches zero, the simulation will approach reality.

A sophisticated version of these incremental iteration programs would allow dynamic assessment of Δ**t** to either decrease or increase the value depending on the variation of **Fnet** in the previous interval relative to the mass (of course, inspection of the change in acceleration would automatically account for the mass). The concept of dynamic assessment found an interesting application in the stock market modelling described in the **NOVA** episode (on **PBS**) : **Trillion Dollar Bet.** A stock market hedge fund (**LTCM**) created by inspired academics used dynamic hedging to offset risk; this video is always shown at the end of April in my Electronics class, check with me for the date if you're interested. Note the reference to rocket science and the idea of "continuous time", or appropriately small Δ**t**. (This is one of the best **NOVA** episodes and is well worth watching.)

Application # 11 : Oscillatory motion of a block bouncing off spring at bottom of incline with friction and drag.

This simulation considers a block moving on an incline plane (angle θ) under the influence of gravity, friction, air resistance, and a spring lying at the bottom of the incline (but not attached to the block). A rotated coordinate system, with X positive up the incline, has its origin located at the un-stretched free end of the spring. The block oscillates as it moves up and down the incline periodically striking and compressing the free end of the spring. [Modifications are presented which convert the X data of the rotated coordinate system into X and Y data relative to a normal upright coordinate system.]
Data : θ = 50°, mass of block = 2 kg, Xo = 4 m, Vox = –10 m/s, uk = 0.3, k = N/m, b = 0.04 kg/m . deltaT = 0.01 s, and Delay = 0s .



Spring & spring force not shown due to difficulties in drawing.

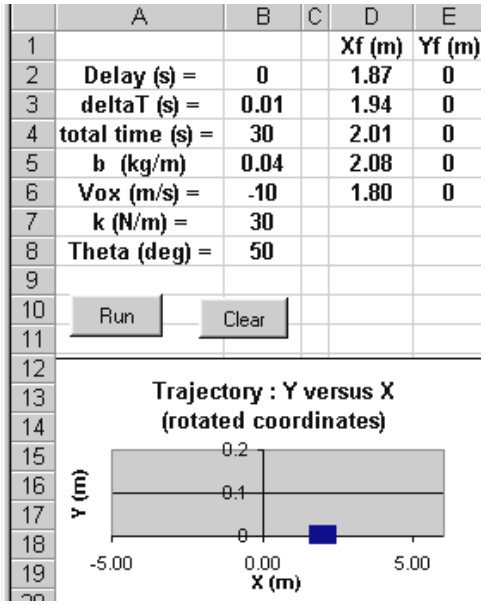
The situation is described by a net force :

Fnetx = –Sgn(Vox) * Ff – Sgn(Vox) * b * Vox ^ 2 – W * Sin(Theta) – k * Xo (block in contact with spring)
Fnetx = –Sgn(Vox) * Ff – Sgn(Vox) * b * Vox ^ 2 – W * Sin(Theta) (otherwise)

As before, the **–Sgn(Vox)** attached to the friction **Ff** and drag $\mathbf{bVox^2}$ terms accounts for the reversal of force direction when the block changes direction. The component of weight down the incline appears as **– W * Sin(Theta)** , and the spring force is modeled as **– k * Xo** . Remember that the description of the spring force as **– k * Xo** automatically accounts for the correct direction of the spring force if the positive direction of **Xo** is away from the spring (up the incline in this situation). A negative **Xo** resulting from the compression of the spring leads to a positive value for **– k * Xo** which is consistent with an upward spring force in the present circumstances. Since the spring is not attached to the block there is never any extension of the spring; however, if there were extension the accompanying positive value of **Xo** would result in a negative value for the term **– k * Xo** consistent with the downward direction of the force. [Note: the **Sgn( )** function could be used to describe the reversal of the spring force direction but the intrinsic sign of **Xo** would have to be suppressed using the **Abs( )** function : **– Sgn(Vox)* k *Abs(Xo)** . The **– k * Xo** description is clearly more elegant!]

```
Option Explicit
Dim Delay As Double

Private Sub cmdRun_Click()
Dim Xo As Double, Vox As Double, Xf As Double, Vfx As Double, mass As Double
Dim ax As Double, t As Double, deltaT As Double, Fnetx As Double
Dim row As Integer, Total_time As Double
Dim b As Double, k As Double, Theta As Double, Ff As Double, W As Double
```

```
Delay = Worksheets("sheet1").Range("B2")
deltaT = Worksheets("sheet1").Range("B3")
Total_time = Worksheets("sheet1").Range("B4")
b = Worksheets("sheet1").Range("B5")
Vox = Worksheets("sheet1").Range("B6")
k = Worksheets("sheet1").Range("B7")
Theta = Worksheets("sheet1").Range("B8") * 3.14159 / 180

mass = 2: Xo = 4
row = 2
t = 0
W = mass * 9.81
Ff = 0.3 * W * Cos(Theta)        ' Ff = uk*W*cos(theta)
```




```
Do
   If Xo < 0 Then
      Fnetx = –Sgn(Vox) * Ff – Sgn(Vox) * b * Vox ^ 2 – W * Sin(Theta) – k * Xo
   Else
      Fnetx = –Sgn(Vox) * Ff – Sgn(Vox) * b * Vox ^ 2 – W * Sin(Theta)
   End If

   ax = Fnetx / mass

   Xf = Xo + Vox * deltaT + 0.5 * ax * deltaT ^ 2
   Vfx = Vox + ax * deltaT

   Xo = Xf
   Vox = Vfx
   t = t + deltaT

   Call TimeDelay(Delay)
   Worksheets("sheet1").Cells(row, 4) = Xf
   Worksheets("sheet1").Cells(row, 5) = 0

   DoEvents: DoEvents
   row = row + 1
   If row = 7 Then row = 2

Loop Until t >= Total_time

End Sub

Private Sub cmdClear_Click()
 Worksheets("sheet1").Range("D2:E6") = 0
End Sub
```

Compare this **If…Then…Else** approach to including the spring force with that of the previous application.

Input/ Output Bound

The value of **Delay** was set to zero in this application because the program is **output bound** which means that it was slowed down by all the updating of output values that was occurring on the worksheet. [On my older machine it took 85.47 seconds to simulate 30 seconds of time in the program with Delay = 0.] A program is said to be **input/output bound** when the volume of reading from and writing to the worksheet results in a natural slowdown of program execution. The situation is further exacerbated when the **DoEvents** command is used to force the continual update of the animated graph. Remember, without the **DoEvents** command the graph points would only be plotted once execution had been completed, and since a comet tail is being used only the very last set of values would be plotted.

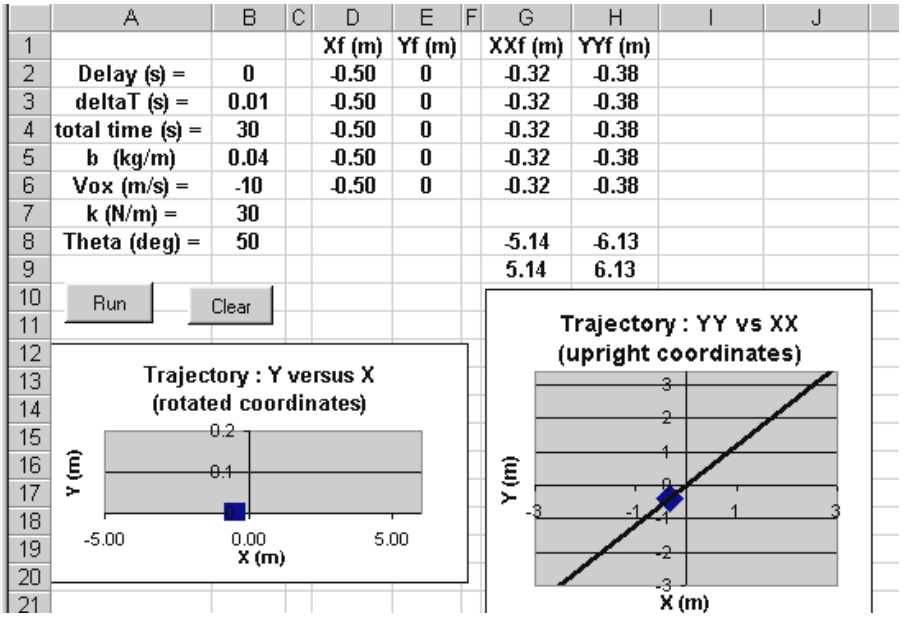
Hidden loops will be introduced later to reduce the amount of output (**section 6.5**).

Displaying Motion on an Incline. Rotated coordinates are used to simplify the analysis of problems involving inclined planes. Unfortunately the data that is produced is relative to an X axis lying along the incline; this means that the resultant motion on the animated graph is horizontal. The actual inclined trajectory can be recovered by taking components of the X locations as shown in the bolded code below.

```
Worksheets("sheet1").Cells(row, 4) = Xf
Worksheets("sheet1").Cells(row, 5) = 0
Worksheets("sheet1").Cells(row, 7) = Xf * Cos(Theta)
Worksheets("sheet1").Cells(row, 8) = Xf * Sin(Theta)
```

Existing code.

The actual trajectory points are presented in the XXf and YYf columns where the incline is represented by a 2nd data series of two points and the comet tail points were formatted in a larger font size to represent the block (right click on the data point).



A rotated coordinate system was selected since it simplified modeling the spring force; however, it became clear in the Winter 2005 semester that for multi-region problems (such as a block launched by a spring down an incline, along a horizontal surface, and into a vertical, circular track that it might fall off) the use of a single, upright coordinate system is desirable. Should you be interested in such problems contact me at : **kenton@champlaincollege.qc.ca**

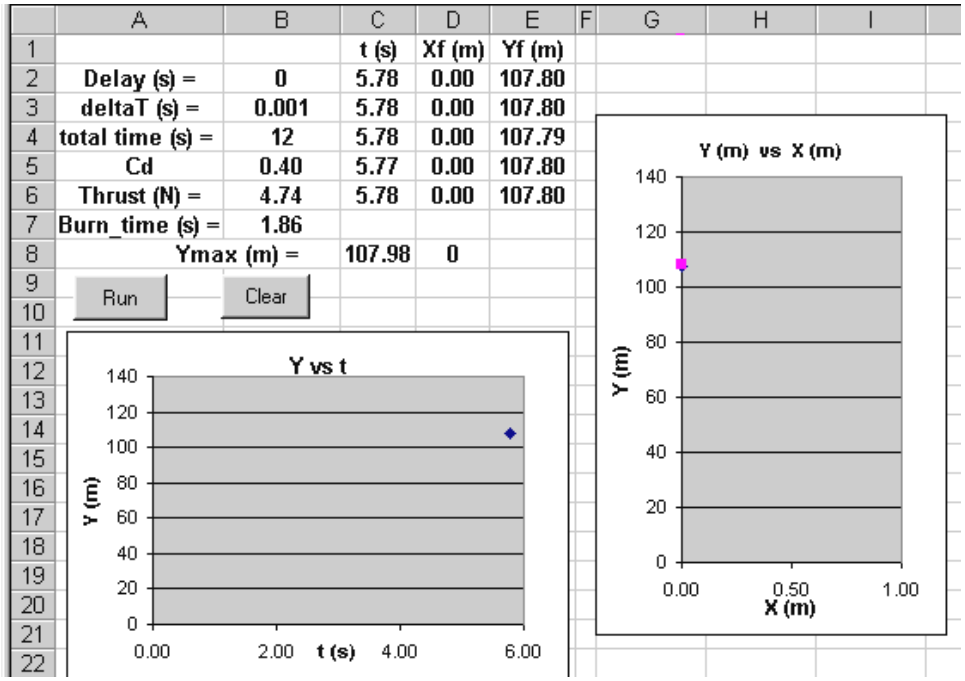
Application # 12 : Vertical rocket with drag, constant mass & thrust during engine burn

Incremental iteration is applied to simulate the 1-D vertical motion of a 149 gram **Estes Super Shot** rocket (www.estesrockets.com) powered by a C6-5 motor that produces an average thrust of 4.74 N for 1.86 seconds (www.thrustcurve.com) [Type in this site name; the automatic link has been contaminated by an "s" that is appended to the end of the name] . For the time being both the mass of the rocket and the thrust are assumed constant. The diameter of the rocket is 3.4 cm and a value of Cd = 0.40 is used. Also check www.aerorocket.com /aerodrag.html for Cd values (look closely at the sample dialog boxes). ρ (air) = 1.29 kg/m^3 . www.A3MAQ.qc.ca will put you in contact with local (Montreal) rocketeers and has a few useful links. [The 149 grams includes the 125 gram rocket plus the C6-5 engine.]

The net force description of the situation is :

Fnety = Thrust – mass * 9.81 – Sgn(Voy) * b * Voy ^ 2 for t <= 1.86 s
Fnety = – mass * 9.81 – Sgn(Voy) * b * Voy ^ 2 for t > 1.86 s

(That is, the thrust is zero after 1.86 s.)



The code below is a modification of Application # 9 – old code is in italics, **new code is bolded**.

Option Explicit
Dim Delay As Double

Private Sub cmdRun_Click()
Dim Yo As Double, Voy As Double, Yf As Double, Vfy As Double, mass As Double
Dim ay As Double, t As Double, deltaT As Double, Fnety As Double
Dim row As Integer, Total_time As Double
Dim b As Double, **Ymax As Double, Thrust As Double, Cd As Double, A As Double**
Dim Burn_time As Double

```
Delay = Worksheets("sheet1").Range("B2")
deltaT = Worksheets("sheet1").Range("B3")
Total_time = Worksheets("sheet1").Range("B4")
Cd = Worksheets("sheet1").Range("B5")
Thrust = Worksheets("sheet1").Range("B6")        ** note the change for this cell
Burn_time = Worksheets("sheet1").Range("B7")

mass = 0.149: Yo = 0: A = 3.14159 * (0.034 / 2) ^ 2
row = 2
t = 0: Ymax = 0

b = 0.5 * 1.29 * A * Cd
Do
   If t > Burn_time Then Thrust = 0
   Fnety = Thrust - mass * 9.81 - Sgn(Voy) * b * Voy ^ 2
   ay = Fnety / mass

   Yf = Yo + Voy * deltaT + 0.5 * ay * deltaT ^ 2
   Vfy = Voy + ay * deltaT

   Yo = Yf
   Voy = Vfy
   t = t + deltaT

   If Yf > Ymax Then Ymax = Yf
Call TimeDelay(Delay)
Worksheets("sheet1").Cells(row, 3) = t
Worksheets("sheet1").Cells(row, 4) = 0
Worksheets("sheet1").Cells(row, 5) = Yf
Worksheets("sheet1").Range("C8") = Ymax
DoEvents
row = row + 1
If row = 7 Then row = 2

Loop Until t >= Total_time

End Sub

Private Sub cmdClear_Click()
Worksheets("sheet1").Range("C2:E6") = 0
Worksheets("sheet1").Range("C8") = 0
End Sub
```

Update : The density of air should be 1.20 kg/m^3 and not 1.29 kg/m^3. Refer to the box at the bottom of pg 6-7

Complete Sets of Data While the comet tail approach is convenient for displaying trajectories, especially those that fold back on themselves when the object reverses direction or executes a periodic motion over the same points in space (as for a pendulum or orbiting satellite), it is nevertheless often useful to present the complete sets of data, especially for velocity, net force, and acceleration. The principal modification involves including a second index **row2** to describe the row location where the data will be written out. Unlike the comet tail **row** index the value of **row2** continues to increase with each new point that is generated. The following additions produce the set of graphs given below :

Dim row As Integer, **row2 As Integer,** *Total_time As Double*

row = 2: **row2 = 2**

Worksheets("sheet1").Cells(row, 5) = Yf
Worksheets("sheet1").Cells(row2, 7) = t
Worksheets("sheet1").Cells(row2, 8) = Vfy
Worksheets("sheet1").Cells(row2, 9) = Fnety
Worksheets("sheet1").Cells(row2, 10) = ay

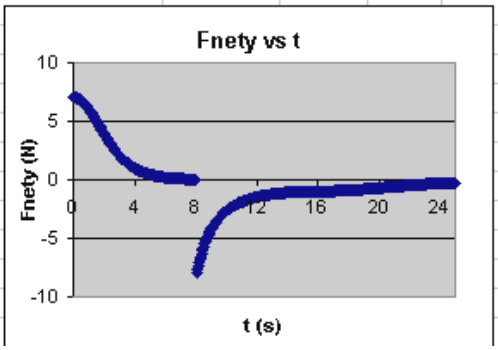
Modifications to **Clear** button code need to be made, but read the box below.

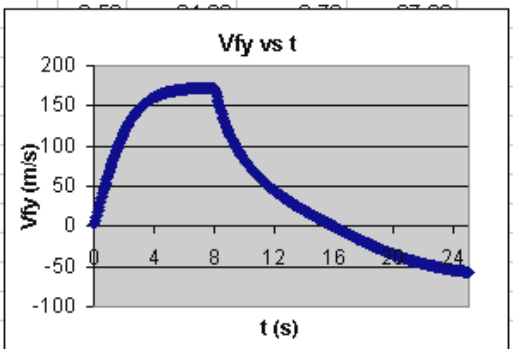
row = row + 1
row2 = row2 + 1

File Size Warning : The VBA applications that we've been constructing usually produce files of the order of 40 to 80 K. Programs that produce complete sets of data can produce files that are unwieldy (in excess of 1 Meg) if **Δt** is too small or if the simulation is allowed to run for too long. Unfortunately, the trade-off here is that **Δt** needs to be small to reduce the approximation error but this increases the amount of output when complete sets of data are being produced (as well as slowing down the program execution). Using the **.Clearcontents** method to clear the data before closing the file is not helpful since there appears to be vestigial presence of the cleared values; what does work is the **.Clear** method.

Parameter changes : To create some interesting features the total rocket mass has been reduced to 0.1 kg, Δt increased to 0.05 s, total time increased to 25 s, thrust increased to 8 N, and burn time increased to 8 s.

	A	B	C	D	E	G	H	I	J
1			t (s)	Xf (m)	Yf (m)	t (s)	Vfy (m/s)	Fnety (N)	ay (m/s^2)
2	Delay (s) =	0	24.80	0.00	1239.93	0.05	3.51	7.02	70.19
3	deltaT (s) =	0.05	24.85	0.00	1237.11	0.10	7.02	7.02	70.16
4	total time (s) =	25	24.90	0.00	1234.28	0.15	10.52	7.01	70.07
5	Cd	0.40	24.95	0.00	1231.45	0.20	14.02	6.99	69.93
6	Thrust (N) =	8	25.00	0.00	1228.60	0.25	17.50	6.97	69.73
7	Burn_time (s) =	8				0.30	20.98	6.95	69.47
8	Ymax (m) =		1543.00	0		0.35	24.44	6.92	69.16
9						0.40	27.88	6.88	68.79
10	Run	Clear				0.45	31.29	6.84	68.37





As was likely clear (and painful) with your mechanics labs, many students have difficulty with the intellectual analysis (or interpretation) of graphs. Among the features of these velocity and net force graphs are : a) a **"pseudo" terminal velocity** is almost reached between 6.85 and 8 seconds when the net force is close to zero (less than 0.1 N) and the velocity is essentially constant. [I'm using "pseudo" here due to the presence of the engine thrust which acts upward and is almost cancelled by the downward acting weight and drag], b) maximum height is detected on the Vy-t graph when Vfy = 0 , just before 16 s after which time the drag force reverses direction and acts upward, c) a true **terminal velocit**y is being approached sometime after about 25 seconds as the velocity becomes constant due to the zero net force resulting from the upward drag being exactly equal to the downward weight, d) the discontinuity of the velocity slope at t = 8 seconds is caused by the discontinuity in net force due to engine burn out, e) the curved shape of the velocity-time graphs (which would be linear for constant acceleration) can be traced to variation in net force caused by the variation in drag, f) the rate of change of the velocity, reflected in the slope of the Vy-t graph, decreases between 0 and 8 seconds due to the decreasing value of the net force.

Variable Thrust Rockets: While the average thrust description of the force provided by the rocket engine works well (refer to problems 20 to 23), there will be occasions when it will be desirable to program the actual thrust variation with time. For example, the Estes C6 engine (data measured 25 March, 1995) (www.thrustcurve.com) ** exhibits about 5 basic regions : 0 to 0.192s, 0.192 to 0.248s, 0.248 to 0.475s, 0.475 to 1.821 s, and 1.821 to 1.860s. Appropriate functions can be fit to the data of each region and then nested **If** statements, or multiple single outcome **If**s as used below, can be included in the main loop to specify the actual thrust.

If t >= 0 And t <= 0.192 Then Thrust = F1(t)
If t > 0.192 And t <= 0.248 Then Thrust = F2(t)
If t > 0.248 And t <= 0.475 Then Thrust = F3(t)
If t > 0.475 And t <= 1.821 Then Thrust = F4(t)
If t > 1.821 And t <= 1.86 Then Thrust = F5(t)
If t > 1.86 Then Thrust = 0

where F1(t) through F5(t) are functions determined from "trendlines" fitted to the data.. The last **If**, which sets the thrust to zero, is essential because the last iteration point during the engine burn would otherwise leave a small **Thrust** value which would then be present throughout the remainder of the flight.

** The Estes data is obtained at the site www.thrustcurve.com; note, there is no "s" at the end of the name as the link will try to add.

Variable Mass Rockets : The mass of a rocket varies as the fuel is burned and expelled (which creates the thrust), and can be modelled in one of 3 ways :

a) In the same way as the varying thrust above by using either an average over the period of the burn, or as a function of time. Although the fuel burn doubtless has an initial spike like the thrust, it will be assumed that the burn rate **k** (kg/s) is constant [note that **k** = $\Delta m/\Delta t$, or dm/dt] so that mass varies as a linearly decreasing function : **M(t) = Mtotal – k t** where **Mtotal** is the mass of the mass of the rocket plus engine with fuel. One the engine has burned out the mass must be set to a value **Mfinal** which is the mass of the rocket plus the engine mass (without the fuel).
b) Alternatively, the mass can be modelled as a running total in which the value is reduced by an amount **k*deltaT** for each interval **deltaT** : **Mf = Mo – k*deltaT** where **Mf** is the mass at the end of the iteration and **Mo** is the mass at the start of the iteration. The boundary condition **Mo = Mf** must be included with the outer boundary conditions.
c) Finally, the recommended, and more compact, approach avoids the boundary condition of method b) and uses a single line of "computer algebra" to form the running total : **M = M – k*deltaT** (refer to **Implicitly Defined Boundary Conditions**, **section 6.6**).

As with method a), methods b) & c) require initialization of the starting mass outside of the main loop by means of either **Mo = Mtotal**, or **M = Mtotal** ; both of these approaches also require that the mass after the engine has burned out be set to the **Mfinal** value.

The model of the varying mass according to each of the three methods is :

Method a)

```
If t >= 0 And t <= 1.86 Then mass = 0.149 – k * t
If t > 1.86 And t <= 6.86 Then mass = 0.1382
If t > 6.86 Then mass = 0.1344
```

Method b)

```
Mo = 0.149
Do
   ay = Fnety / Mo
   Mf = Mo – k * deltaT
   If t > 1.86 And t <= 6.86 Then Mf = 0.1382
   If t > 6.86 Then Mf = 0.1344

   Yo = Yf
   Voy = Vfy
   Mo = Mf
Loop Until ……
```

Notice that variable **mass** has been replaced by **Mo** and **Mf**. Not all old code is shown.

Method c)

```
mass = 0.149
Do
   ay = Fnety / mass
   mass = mass – k * deltaT
   If t > 1.86 And t <= 6.86 Then mass = 0.1382
   If t > 6.86 Then mass = 0.1344

Loop Until ……
```

Determining the fuel burn rate, k, and the initial and final masses from the spec sheet data : The NAR data for the C6-5 Estes rocket engine [www.thrustcurve.com] gives a "Propellant Mass" of 10.8 grams ("Certified Values" section). Assuming that all the fuel is expelled in the 1.86 s burn time gives an average burn rate of $\mathbf{k = \Delta m/\Delta t} = 10.8/\ 1.86 = 5.81$ g/s $(= 5.81 \times 10^{-3}$ kg/s).

The mass of the cardboard engine shell plus nozzle plug is listed in the "Static Test Data" under "Mass After Firing" = 9.4 grams. The total initial engine mass is the sum of the propellant and the engine shell = 10.8 + 9.4 = 20.2 grams, which is consistent with the value given for a zero delay time engine. The delay times refer to a fuse that leads to a second explosive charge of about 4 grams (including clay plug) that blows the parachute out of the nose cone. The mass of the **Super Shot** rocket that we used (without the engine) was 125 grams so that **Mtotal** = 125 + 24 = 149 g (0.149 kg). The presence of the second explosive charge, which is assumed to burn instantaneously, leads to two values of the final mass : the mass after the engine burns out at 1.86 seconds is 149 – 10.8 (propellant) = 138.2 g, and the mass after the parachute charge fires 5 seconds later (at 1.86 + 5 = 6.86 seconds) is 138.2 – 3.8 = 134.4 g.

Specifying Forces as Functions of Time and Position

Forces such as the spring force, gravitational force (away from the Earth's surface), and the Coulombic force are naturally represented as functions of position in view their actual physical behaviour. However, other forces may be better represented as either functions of time or position depending on the situation and what data is available. For example, rocket engine thrust is only given as a function of time because the variation of the engine thrust with time is easily measured and is unique regardless of the mass of the object being propelled. An attempt to specify the data as a function of Y location (in the case of a 1-D vertical launch) would produce data that applied only to rockets having the same mass as the test vehicle. Different mass rockets would experience different thrust at the same altitude due to the effects of different accelerations on the motion (which would lead to a non-uniqueness of the specification).

In other situations, such as throwing an object with your hand and arm, a force-time specification would be inappropriate for different masses because of the physical constraints presented by your arm. Consider the description of a rock thrown vertically into the air in problem # **13** and a toy car launched horizontally in problem # **17** (pg 6-42). The force-position descriptions both assume that the arm provides a linearly increasing force for the first half of the throw, and a linearly decreasing force during the second half. The use of a force-time specification would result in an "elastic" arm that would either lengthen or shorten depending on the mass being propelled. For example, an object with a small mass would experience a very large acceleration and consequently move over large distances while the hand was still in contact (mathematically at least), and large mass objects might only move a few centimeters before the hand force erroneously zeroed out. The physical reality, of course, is that your hand-arm combination will pretty much move through the same distance for all masses. [The biomechanics of the situation would probably suggest that a family of force-position functions would be required since the force variation produced by the arm very likely changes for different masses. That is, the F(x) function for throwing a ping pong ball would not be the same as that for throwing a heavy rock.]

The weight of the object has been added to the vertical force in problem # **13** corresponding to the hand holding it in equilibrium at Y = 0, and a small force added to get it moving. A force component equal to the 3.5 N friction force has been added to the hand force in problem # **17** so that the toy car wouldn't move spontaneously due to friction at X = 0, and a small force has also been added to get the system moving. Note that the force provided by the hand is not the net force in these problems.

6.4 Making the Programs Interactive

When a program is executing the screen normally appears frozen except for the animated graph(s); that is, the user cannot add or change values in a worksheet cell, activate controls, perform any operations on the **Menu bar** etc., or otherwise engage **Excel**. This means that once a simulation starts nothing about it can be changed unless the program itself creates the change.

Applications can be made "interactive", or responsive to parameter changes made by the user, by adding a **DoEvents Loop** and then using either of the following approaches :

a) adding a **Pause button** that allows the user to completely stop program execution for a non-specified period of time in order to leisurely change parameter values to be read off the worksheet. Execution is then resumed by clicking on the original **Run** button,

or b) activating controls such as buttons and scroll bars that are programmed to change parameter values. This "**on-the-fly"** technique provides the user with a greater sense of real time interaction since the program appears to be continually running. While the resultant effect gives a primitive video game type responsiveness it can be more than a little nerve-wracking if the user has to digest a lot of changing simulation data (information overload as faced by fighter jet pilots; you'll appreciate this more with the Chapter 7 Space Flight simulator problem.)

Both of these methods rely on interrupting program execution using the **DoEvents loop** (which was first introduced in the Chapter 3 **Colour Generator** exercise). Recall that a single **DoEvents** command very briefly interrupts code execution to check for other "events" that should be attended to; it was used to allow graphs to be updated with new points in animations. Unfortunately, the time of occurrence of the events that we wish to communicate to the program, either changing a value in a cell or activating a button or scroll bar, must occur synchronously with the instant that the **DoEvents** command is encountered during execution of the code -- which is not very likely to happen. Multiple **DoEvents** commands 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. Only the **Pause** approach allows cell values to be changed.]

```
For JJ = 1 To 40
  DoEvents: DoEvents
Next JJ
```

6.4.1 The Pause Button Approach

Two major modifications are made to the standard incremental iteration. Up to this point the initial or starting conditions of the simulations have been set in the **Run** subroutine so that each time this button is pressed the object is returned back to the starting point with the initial velocity, thrust, time, etc. In order to prevent this return to initial conditions when a pause has been made and **Run** re-activated, all initial conditions must be removed from the **Run** subroutine and collected into an **Initialize** subroutine. Since the variables describing the initial/launch conditions now appear in more than one subroutine they must be dimensioned globally under the **Option Explicit** . More importantly, since the values **Xo**, **Yo**, **Vox**, **Voy**, and **t** change with each iteration throughout the simulation it is necessary that they retain their values, which will only occur with a global declaration.

The second major modification is to change the exit condition on the main **Do…Loop** so that it also depends on the state of a Boolean variable **bStop** being **True**.

Loop Until t >= Total_time **Or bStop = True**

The value of **bStop** is set to **False** at the beginning of the **Run** routine. The user then can change the state of **bStop** by pressing a **Pause** button which contains the single line of code **bStop = True**. As soon as the **Pause** button is activated during one of the multiple **DoEvents** "windows" the main **Do…Loop** will be exited and the program stopped.

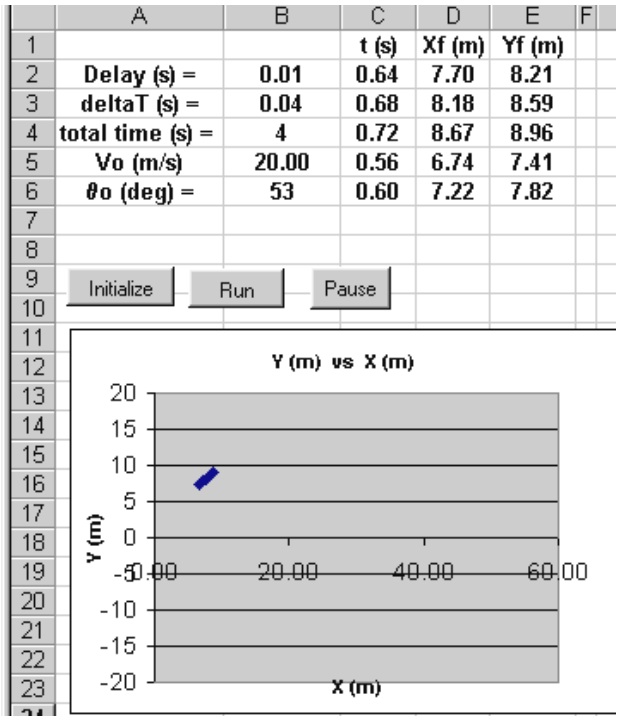
In actual fact, program execution is not really paused since the **Run** subroutine is executed down to the **End Sub** statement. What has happened is that the user is able to cause the program to exit the main loop by satisfying one of the exit conditions. When the user re-presses the **Run** button the initial iterative variables **Xo**, **Vox** have retained their values (they were declared globally), as have **t** and **row** so that program execution resumes from the point where it stopped.

Activating the Pause Button and Scroll Bar may not be easy: Actually activating a button or scroll bar during one of the **DoEvents** "windows" can be tricky. You made need to click on the button 2 to 6 times (but without double-clicking), and the scroll bar, for its part , may seem to have a mind of its own. Once you latch onto the slider you may stay latched onto it even though you've released the left mouse button, and "everywhere you move the mouse, the slider is sure to follow". These difficulties appear to be more prevalent with older versions of Excel.

Application # 13 : 2-D Projectile Motion Simulation

A 1 kg projectile is launched with a speed of 20 m/s at an angle of 53°. The simulation uses a **Pause** button approach to allow the user to stop program execution and, if desired, change either the **Delay**, **deltaT**, or the **total time**, but not **Vo** or **θo** . [The values of **Vo** and **θo** are only read-in in the **Initialize** routine, so that any change in values, which is not at all desirable, will not be registered when the **Run** button is re-activated.] All properties of the graph can be changed when the program is paused.

Save this program as a shell for future applications involving 2_D incremental iteration.




```
Option Explicit
Dim Delay As Double, bStop As Boolean, row As Integer
Dim Xo As Double, Vox As Double
Dim Yo As Double, Voy As Double
Dim Vo As Double, Theta As Double, t As Double

Private Sub cmdInitialize_Click()
Vo = Worksheets("sheet1").Range("B5")
Theta = Worksheets("sheet1").Range("B6") * 3.14159 / 180
Vox = Vo * Cos(Theta): Voy = Vo * Sin(Theta)
Xo = 0: Yo = 0
row = 2: t = 0

Worksheets("sheet1").Range("C2:C6") = 0
Worksheets("sheet1").Range("D2:D6") = Xo
Worksheets("sheet1").Range("E2:E6") = Yo
End Sub
```

Initialization statements & launch conditions.

The functions of the old **Clear** button are fulfilled by these write statements.

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

Pressing the **Pause** button changes the state of Boolean variable **bStop**, which allows the **Run** subroutine loop to be exited.

```
Private Sub cmdRun_Click()
Dim Xf As Double, Vfx As Double, ax As Double, Fnetx As Double
Dim Yf As Double, Vfy As Double, ay As Double, Fnety As Double
Dim deltaT As Double, Total_time As Double, mass As Double
Dim W As Double, JJ As Integer

Delay = Worksheets("sheet1").Range("B2")
deltaT = Worksheets("sheet1").Range("B3")
Total_time = Worksheets("sheet1").Range("B4")

mass = 1
W = mass * 9.81
bStop = False
ax = 0
```

bStop must be reset to **False** each time the **Run** routine is activated (otherwise the loop would be exited after only one iteration.

```
Do
   Fnety = –W
   ay = Fnety / mass

   Xf = Position(Xo, Vox, ax, deltaT)
   Yf = Position(Yo, Voy, ay, deltaT)
   Vfx = Velocity(Vox, ax, deltaT)
   Vfy = Velocity(Voy, ay, deltaT)
```

For simplicity, custom functions are used to define the position and velocity. Notice that the value of **deltaT** is being passed to the function definition routine.

```
   Xo = Xf
   Yo = Yf
   Vox = Vfx
   Voy = Vfy
   t = t + deltaT

   Call TimeDelay(Delay)
   Worksheets("sheet1").Cells(row, 3) = t
   Worksheets("sheet1").Cells(row, 4) = Xf
   Worksheets("sheet1").Cells(row, 5) = Yf

      For JJ = 1 To 60
      DoEvents: DoEvents
      Next J

   row = row + 1
   If row = 7 Then row = 2

Loop Until t >= Total_time Or bStop = True

End Sub
```