Sample numerical solution to a particle with position - dependent forces

Heavily revised by DMW, May 2012

The hard part of many programming problems is deciding HOW to *represent* what you' re interested in. Here we are interested in the time dependence of the altitude and velocity of a particle falling under gravity experiencing a drag force. This notebook mingles a LITTLE physics with mostly ideas about how to solve a mechanics problem by using small time steps. (A For loop makes sophisticated *Mathematica* users cringe, but it works.) Make sure you have read the PDF file named *NumericalAnalysis* before working on this noteboook.

Newton's law F = m a = m d^2 z/dt^2 is a *second order* ordinary differential equation in one dimension. If we wish (and follow Euler) we can write this as two coupled *first order* ODEs, one for x and one for v.

Idea : in a time Δt the particle has moved a distance v Δt and its velocity has changed by a Δt. So one way to accumulate data on the ' trajectory' is to make a list of data pairs (t, z) [time, altitude] and (t, v) [time, velocity]. **Warning:** Read Wikipedia on the "Euler method" to understand why (i) your results may depend (sometimes sensitively) on the time step Δt, (ii) the approach here is just about the most naive approach that will actually work; more sophisticated approaches would be much more stable and insensitive to Δt. NDSolve, for the numerical solution of an ODE with specified initial conditions, could be used to find the solution in a numerically sophisticated way.

Setup

```
Clear["Global`*"]
(* Clear the definitions of all non system
 variables.  DMW: This is important IF you want to run
    the Notebook from scratch WITHOUT killing the kernel first *)
```

A quick and dirty MODEL calculation to establish the IDEAS

We will next make a completely unmotivated and in general incorrect assumption about the acceleration OF THE PROBE (defined by d (vector velocity)/dt):

```
a[v_] := Sqrt[v];
```

(inpendent of the position). That is, the object is always speeding up. Note how the UNITS are wrong; but this is just a parody of reality. Obviously hellish things happen if the velocity goes negative. This shouldn' t happen because the probe is *falling.*

```
deltat = 2.; (* The time step that we will be using *)
```

```
zData = {{0., 1.}}; (* zData contains (time t, altitude z) data pairs. *)
```

```
vData = {{0., 0.}};
```

```
maxSteps = 10; (* This limits the number of steps in the For loop,
to make sure that we don't end up with an infinite loop *)
```

```
t = 0.; (* Initial value of the time *)
```

Here comes the juicy part, the core of the algorithm: each time we go through the loop, we are evaluating position, velocity, and acceleration at a new time. Here we use the velocity and acceleration at the *previous* instant in time to calculate the velocity at the new instant in time. Simultaneously, we add another (time, velocity) pair to the array vData.

```
? For
```

```
For[n = 1, n ≤ maxSteps, n++,
 t = t + deltat;
 AppendTo[zData, {t, zData[[n]][[2]] + vData[[n]][[2]] * deltat}];
 (* Now we calculate the new acceleration. We
  apply the acceleration function defined above. *)
AppendTo[vData, {t, a[zData[[n + 1]][[2]]] deltat}];
]
```

[Note that after the first AppendTo the list zData has one more element; we may as well use the acceleration at the updated velocity.] At this point we have finished the loop and can look at the history of the velocity and acceleration.]

```
TableForm[zData]
```

```
TableForm[vData]
```

```
ListPlot[zData, AxesLabel → {Time, Velocity}, PlotRange → {{0, 20}, {0, 100}},
 PlotStyle → {Thick, Blue, PointSize[.02]}, Joined → True, Mesh → All]
```

You get the idea : you have been able to directly construct a trajectory. Note how flat the curve starts off-- it takes a while for the acceleration to 'kick in'.

Exercise: a more realistic trajectory

```
forceGrav[y_] := gravConst mProbe mPlanet / (rPlanet + y) ^ 2;
(* Now we define the atmospheric force,
which is a function of both position and velocity *)
```

Evidently y is the distance above the Earth' s surface; next we include the drag force due to the atmospheric, which falls (if we assume fixed temperature) as the altitude rises because the atmosphere' s density falls approximately exponentially with height.

```
forceAtmo[y_, v_] := 0.15708 Exp[-y / y0] v^2;
```

Next come the parameters that we will use. Assume that all values are given in standard MKS units. You should recognize the gravitational constant. mProbe is the mass of the prob, mPlanet is the mass of the planet. y0 is a constant that defines the length scale of the atmospheric force which can be seen in the definition of forceAtmo. rPlanet is the radius of the planet. And finally h is the initial height of the Probe above the planet' s surface.

```
parameters = {gravConst = 6.67 * 10^(-11), mProbe = 10,
   mPlanet = 6.419 * 10^23, y0 = 11 100, rPlanet = 3390 * 10^3, h = 10^6};
```

Note : (1) If you were going to change these parameters, you could replace the = signs above by ->, indicating a replacement rule, but you'd need to put /. parameters after expressions you want to have numerical values; (2) you could have grabbed some of these from the PhysicalConstants package. For example

```
<< PhysicalConstants` (*Knows many physical constants,
 in MKS (meters, kg, sec) units *)
```

```
GravitationalConstant
```

So, here' s your task

With the information above, create a function for the instantaneous acceleration of the probe.
Then, with a For, While, or some other loop:

- Create one array with (time, position) data pairs
- Create one array with (time, velocity) data pairs
- Create one array with (time, acceleration) data pairs

Your For loop should exit out once the probe hits the planet. Plot the position, velocity, and acceleration as functions of time. Make the plot as nice as you can.