Numerical integration for solving differential equations

After integration, it is natural to consider how to find numerical solutions to differential equations on the computer. Simple equations of motion lead to 2nd order differential equations. In the case of constant acceleration, we see that:

$$v = at + v_o, \tag{1}$$

so,

$$x = \frac{1}{2}at^2 + v_0 t + x_0. \tag{2}$$

However, if the force and hence acceleration is related to the position or the velocity in any way, then we cannot just to simple integration in closed form (or at least not simple integration of the form that one typically learns in a first calculus class). [1]

There are still plenty of such problems that can be solved in closed form. Let's consider two problems that come up a lot in a first classical mechanics course – the ideal spring which follows Hooke's Law, and the simple pendulum. In an introductory classical mechanics course, we will usually use the small angle approximation, and say that $\sin\theta = \theta$, which then reduces the math for a pendulum's angular motion to being basically the same as the math for a spring's linear motions. It's a pretty good approximation, but it's not quite right, and the differences are something that you could measure in a lab fairly easily.

So, the equations of motion of a pendulum are:

$$\alpha = \frac{-g}{l}\sin\theta = \frac{d\omega}{dt}, \tag{3}$$

and

$$\frac{d\theta}{dt} = \omega. \tag{4}$$

For if we make the small angle approximation that $\sin\theta = \theta$, then we get:

$$\frac{d^2\theta}{dt} = \frac{-g}{l}\theta, \tag{5}$$

which we can solve by inspection to give a sine wave with frequency $\sqrt{g/l}$.

If we don't make the small angle approximation, there is no closed form solution to the differential equation.

[1] Historically, these problems were solved by perturbation analyses – one would solve a problem that was almost the problem that one wanted to solve, and then figure out how to make small changes to the solution based on the small deviations from the soluble problem. This kind of approach is still often useful for two purposes – (1) it can sometimes guide an efficient approach to finding the numerical solution to the problem and (2) it can sometimes produce simple formula which are approximately correct, and which can guide understanding of what really happening in a way that the outputs of a thousand computer simulations often cannot.

There are a few approaches to solving differential equations on a computer. The simplest, by far, is Euler's method, which is basically like the rectangular rule that we talked about for solving simple integrals last week. Instead of trying to solve a second order differential equation, what we do is we split it into two first order differential equations, and then solve each of them. We look at the force law to solve the equation to get the velocity, and then we look at the velocity to get the equation for the position.[2]

Let's lay this out in abstract terms. We have two variables, x and t. We want to solve for x as a function of t. Let's let the second derivative of x be a function of x itself, and maybe also of the first derivative of x. We define a new variable v to be $\frac{dx}{dt}$.

We just set:

$$x_{i+1} = x_i + \frac{dx}{dt} \times \Delta t \tag{6}$$

$$v_{i+1} = v_i + \frac{dv}{dt} \times \Delta t \tag{7}$$

$$\frac{dv}{dt} = f(x, v) \tag{8}$$

We can now solve this on the computer, as long as we can write down the expression for $f(x, v)$. As an example of how to do this, I have written a program to solve the pendulum problem with Euler's method.

```
   /* Pendulum program */
#include<math.h>
#define g 9.8 /* sets g for use throughout the program*/
#define deltat 0.001 /*sets a timestep value for use throughout the
program*/
#include <iostream>
#include <fstream>
using namespace std;

   float euler(float x,float xdot)
{
/*This function just increments a variable by its derivative times
*/
/*the time step.  This is Euler's method, which is a lot.*/
/*like the rectangular rule for evaluating a simple integral.  */
x=x+xdot*deltat;
return(x);
}

   void read_start_vals(float *theta, float *length)
{
cout << "What is the initial value of theta in degrees?\n";
```

[2]For the pendulum, really, we're looking at torques, angles, and angular velocities, of course, but the math is applicable to a wide range of problems.

```
cin >> *theta; /*theta is a pointer */
/*The next line converts theta into radians */
*theta=*theta*M_PI/180.0; /*M_PI is pi defined in math.h */
/* An asterisk is needed for theta because we want the value*/
/* at the memory address of the pointer, not the address itself*/
cout << "What is the length of the pendulum in m?\n";
cin >> *length; /* *length is a pointer */
return;
}

   float angular_acceleration(float theta,float length)
{
float alpha,omega;
alpha=-(g/length)*sin(theta);
/* This is just first year physics, but before we make the small
angle approximation.  */
return alpha;
}

   main()
{
int i,imax;
float alpha,theta,omega,length,t,kineticenergy,potentialenergy,totalenergy,velocity;
ofstream outfile;
/*Above lines define the variables*/
/*Mass is ignored here, since it falls out of all equations*/
/*To get energy in joules, etc.  would have to add a few lines of code*/
outfile.open("./pendulumoffset"); /* opens a file for writing the output*/
t=0;
omega=0;
/*The two lines above make sure we start at zero.*/
read_start_vals(&theta,&length); /*Reads initial offset and length of
pendulum */
for (i=0;i<50000; i=i+1) /*Starts a loop that will go through 50000
time steps */
{
alpha=angular_acceleration(theta,length);/*Computes angular accel.
*/
omega=euler(omega,alpha);/*Changes the angular velocity by Euler meth.
*/
theta=euler(theta,omega);/*Changes angle by Euler meth.*/
potentialenergy=g*length*(1.0-cos(theta));/*Computes grav.  pot en.*/
velocity=omega*length;/* Computes linear velocity*/
kineticenergy=0.5*velocity*velocity;/*Computes kin.  en */
totalenergy=potentialenergy+kineticenergy;/*Computes total energy*/
t=t+deltat;/*Increments the time*/
```

```
outfile << t <<" " << theta <<" " << omega <<" " << kineticenergy <<"
" << potentialenergy <<" " << totalenergy << endl;/* Prints out the
key variables*/ /*Note:  keeping track of the total energy is a good
way to test a*/
/*code like this where the energy should be conserved*/
}

   fclose(fp);/*Closes the file*/
}
```

The big disadvantage of the Euler method is that it tends to give answers which get worse and worse with time. This is actually true for all numerical integration schemes for solving differential equations for nearly any problem. Problems like this one, for which the solutions are periodic are less troublesome than problems for which the solution could go off in any directions. First – in this case, it's obvious that the solution will be periodic, even if the solution is not a sine wave, so we have a sanity check [3]

The next level up in complication is a method to do something sort of like what we did with Simpson's rule, and to try to do something to take into account the curvature of the function for which we are solving. There is a family of methods called *Runge-Kutta* methods which can be used for solving differential equations more precisely than the Euler method can in the same amount of computer time. In this class, we will use the "classical Runge-Kutta" method, which provides a good compromise among simplicitly, accuracy, and computational time needed. It is also sometimes called the RK4 method, since it is a 4th order method in the seconds that the errors per step scale with the step size to the 5th power, so the total errors scale with the step size to the 4th power.

Let's first lay out the method for the simplest case, where we are solving a first order differential equation:

$$\frac{dx}{dt} = f(t, x), \tag{9}$$

along with which we will need a boundary condition, that at t_0 $x = x_0$.

Now, we integrate this out with something analogous to Simpson's rule:

$$x_{i+1} = x_i + \frac{1}{6}\Delta t(k_1 + 2k_2 + 2k_3 + k_4), \tag{10}$$

[3] You may have also noticed that I computed and printed out the energy in this system to make sure that it was conserved which is another sanity check.

where the k_n values are given by taking:

$$\begin{aligned} k_1 &= f(t_i, x_i) \\ k_2 &= f(t_i + \frac{1}{2}\Delta t, x_i + \frac{\Delta t}{2} k_1) \\ k_3 &= f(t_i + \frac{1}{2}\Delta t, x_i + \frac{\Delta t}{2} k_2) \\ k_4 &= f(t_i + \Delta t, x_i + \Delta t k_3) \end{aligned} \tag{11}$$

What is going on in this set of equations is that a series of estimates of the derivative of x with respect to t are being made. The first is the Euler method's approximation. The second makes an estimate of the value of x half way through the time step, based on using the value from k_1, and just adds half a time step to the time. The third makes another estimate based on adding half a time step, but now using the value of the position based on k_2, and the final one is a best guess at what's happening at the end of the time step.

So, this method is great for first order differential equations, but how do we apply it to second order differential equations? Now things get a bit more complicated. Let's take the case that $\frac{d^2x}{dt^2} = f(t, x, \frac{dx}{dt})$, which will allow for solving cases with something like air resistance present. Now, we need to vreak the second order differential equation up into two first order differential equations:

$$\begin{aligned} \frac{dx}{dt} &= v \\ \frac{dv}{dt} &= f(t, x, v) \end{aligned} \tag{12}$$

Then, we need to solve these two equations simultaneously.

We can start with writing down the solutions for the two equations in terms of k_i values for the second derivative, and K_i values for the first derivative – i.e. we will add up the k values to integrate out v, and the K values to integrate out x.

This gives is:

$$\begin{aligned} v_{i+1} &= v_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)\Delta t \\ x_{i+1} &= x_i + \frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4)\Delta t, \end{aligned} \tag{13}$$

and now we need a prescription for evaluating the values of the k's and the K's.

This is again similar to in the first order equation case:

$$\begin{aligned} k_1 &= f(t_i, x_i, v_i) \\ k_2 &= f(t_i + \frac{1}{2}\Delta t, x_i + \frac{\Delta t}{2} K_1, v_i + \frac{\Delta t}{2} k_1) \\ k_3 &= f(t_i + \frac{1}{2}\Delta t, x_i + \frac{\Delta t}{2} K_2, v_i + \frac{\Delta t}{2} k_2) \\ k_4 &= f(t_i + \Delta t, x_i + \Delta t K_3, v_i + \Delta t k_3), \end{aligned} \tag{14}$$

while evaluating the values of K_i is simpler:

$$\begin{aligned} K_1 &= v_i \\ K_2 &= v_i + \frac{\Delta t}{2} k_1 \\ K_3 &= v_i + \frac{\Delta t}{2} k_2 \\ K_4 &= v_i + \Delta t k_3 \end{aligned} \tag{15}$$

This gives us a recipe that is a bit harder to turn into computer code than the Euler method, but harder only in the sense of being more work, rather than genuinely conceptually harder. At the same time, it's much more accurate that the Euler method, which is why it's worth the effort.

Other methods

Just as Simpson's rule often provides the best compromise among accuracy, time to program, and number of CPU cycles needed for integrals, the RK4 method often provides the best compromise for differential equations. At the same time, there will often be better approaches both for integrals and differential equations.

In this course, we won't go to the lengths of actually executing these "better" approaches, but since you may eventually find yourself in a situation where RK4 isn't good enough, I figured that I should at least give some buzzwords you can look up to find more sophisticated approaches. The first thing people often try is RK4 with adaptive step sizes – that is, instead of making Δt a constant, let Δt increase when the higher order derivatives of the function are small. This is an approach that is useful for dealing with cases where the RK4 method is "good enough" in the sense that it will converge to the right answer given enough computer power, but where you want to get a precise answer more quickly than you can get it with fixed step sizes. There are a range of more and less clever ways to compute optimal step sizes. A few commonly used more advanced approaches are the Gauss-Legendre method, and the Bulirsch-Stoler algorithm. If you run into a problem where you need a more advanced method, you may find it helpful to know the terminology, but we will not discuss these methods in class.