Numerical Methods for Ordinary Differential Equations

By Brian D. Storey

1. Introduction

Differential equations can describe nearly all systems undergoing change. They are ubiquitous is science and engineering as well as economics, social science, biology, business, health care, etc. Many mathematicians have studied the nature of these equations and many complicated systems can be described quite precisely with compact mathematical expressions. However, many systems involving differential equations are so complex, or the systems that they describe are so large, that a purely mathematical representation is not possible. It is in these complex systems where computer simulations and numerical approximations are useful.

The techniques for solving differential equations based on numerical approximations were developed before programmable computers existed. It was common to see equations solved in rooms of people working on mechanical calculators. As computers have increased in speed and decreased in cost, increasingly complex systems of differential equations can be solved on a common PC. Currently, your laptop could compute the long term trajectories of about 1 million interacting molecules with relative ease, a problem that was inaccessible to the fastest supercomputers just 5 or 10 years ago.

This chapter will describe some basic methods and techniques for the solution of differential equations using your laptop and MATLAB, your soon to be favorite program. We will review some basics of differential equations, though this will not be mathematically formal: this you will learn in your math and physics courses. Next we will review some basic methods for numerical approximations and introduce the Euler method (the simplest method). We will provide a lot of detail of how to write 'good' algorithms using the Euler method as an example. Next we will discuss error approximation and discuss fancier techniques. We will then discuss stability and 'stiff' equations. Finally we will unleash the full power of MATLAB and look into their built in solvers.

2. Describing physics with differential equations

Just to review, a simple differential equation usually has a form

$$\frac{dy}{dt} = f(y,t) \tag{2.1}$$

where dy/dt means the change in y with respect to time and $f(y,t)$ is any function of y and time. Note that the derivative of the variable, y, depends upon itself. There are many different notations for d/dt, common ones include $\dot{y}$ and y'.

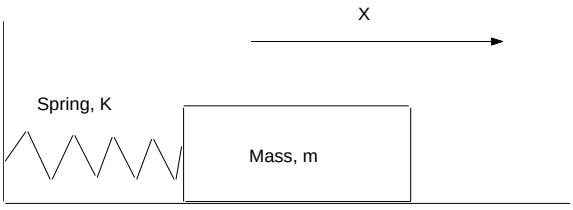

Figure 1. Mass-spring system we will be analyzing. The mass, m, is on a frictionless surface so it can freely slide back and forth in the x direction. Gravity acts normal to the motion of the mass.

A simple example of a system described by differential equation is the motion of mass on a spring, see figure 1. In your physics course you remember Newton's second law which says that

$$F = ma \tag{2.2}$$

where F is the force exerted on mass m and a is acceleration. Springs come in many shapes and sizes, but many obey a simple linear relation, that the force exerted by the spring is proportional to the amount that it is stretched, or

$$F = -kx \tag{2.3}$$

where K is called the spring constant and x is the displacement of the spring from it's equilibrium state.

Equating the above expressions lead to the expression

$$ma = -kx. \tag{2.4}$$

Remembering that acceleration is the second derivative of position and we have a differential equation,

$$m\frac{d^2x}{dt^2} = -kx \tag{2.5}$$

When solving differential equations numerically we usually like to work with systems of equations that involve only first derivatives. This is convenient because the same program can be generalized to solve any problem. In the above example, the second order system is transformed quite easily using the relationships

$$a = \frac{dv}{dt} \tag{2.6}$$

$$v = \frac{dx}{dt}. \tag{2.7}$$

where v is the velocity.

Using the above relationships we can easily rewrite equation 2.5 as two equations,

$$\frac{dv}{dt} = -\frac{k}{m}x \tag{2.8}$$

$$\frac{dx}{dt} = v. \tag{2.9}$$

The reason for rewriting the equations as a system of two coupled equations will become clear as we proceed. We say that the equations are coupled because the derivatives of velocity are related to the position and the derivative of position is related to the velocity.

> **Exercise: DEQ 1** Show that equations 2.5 is satisfied by $x(t) = x_0 cos(\sqrt{k/m}t)$ for the initial condition $x(t=0) = x_0$ and $v(t=0) = 0$. This states that the spring is pulled back to the position x_0 and released from rest. Plot the solution with MATLAB.

3. Dimensionless equations

Before implementing a method to solve differential equations, let us take a side trip into dimensionless equations. While this section may seem weird at first, over time you will start to realize the value of rescaling your equations so that they have no physical dimensions.

When constructing solutions to differential equations it is always convenient to recast the equations into a form that has no dimensions, i.e. the equations do not depend upon parameters that have units such as meters, seconds, etc. The reason is that rescaling the equations will always reduce the number of free parameters that can be varied. Referring back to equations 2.9 & 2.8 we might think that we have four free parameters: k, m, and x_0. When we remove the dimensions we find that there are no free parameters, i.e. all systems behave the same. Physically, looking parameters with no dimensions makes sense; nature cannot depend upon the units that people have created.

To create non-dimensional equations we are simply scaling variables by constants. Let us use the * superscript to denote variables with no dimensions. The equations have three variables, position, velocity, and time. To create a non-dimensional variable we simply perform the transformation

$$x(t)^* = \frac{x(t)}{x_0} \tag{3.1}$$

where the time dependent position is scaled by the initial condition. We could use any length scale that we like (the size of your foot would work just fine), however the initial position is a convenient scale because then the initial condition in dimensionless variables is always $x^* = 1$. Now let us rewrite the equations using the following non-dimensional variables

$$v(t)^* = \frac{v(t)}{\bar{v}} \tag{3.2}$$

$$t^* = \frac{t}{\bar{t}} \tag{3.3}$$

where $\bar{v}$ and $\bar{t}$ are the velocity and time scale, constants that we will leave general for now. Applying these transformations to equation 2.9 yields

$$\frac{x_0}{\bar{t}} \frac{dx(t)^*}{dt^*} = \bar{v} v(t)^*. \tag{3.4}$$

Since we left the velocity and time scale arbitrary we can set them to be anything that we would like. It is easy to see that Equation 3.4 would be convenient if

$$\bar{v} = \frac{x_0}{\bar{t}}. \tag{3.5}$$

This scaling would make equation 2.9 become

$$\frac{dx(t)^*}{dt^*} = v(t)^*. \tag{3.6}$$

in dimensionless form, the same as it was in dimensional form.

Applying the non-dimensional scaling to equation 2.8 yields,

$$\frac{\bar{v}}{\bar{t}}\frac{dv(t)^*}{dt^*} = -\frac{kx_0}{m}x(t)^*. \tag{3.7}$$

recalling that $\bar{v} = x_0/\bar{t}$ and collecting all the constants on the right-hand-side of the equation yields.

$$\frac{dv(t)^*}{dt^*} = -\frac{k\bar{t}^2}{m}x(t)^*. \tag{3.8}$$

Since the time scale, $\bar{t}$, was arbitrary we see that it would be convenient if $\bar{t} = \sqrt{k/m}$. This scaling results in a system of equations with no free parameters!

$$\frac{dv(t)^*}{dt} = -x(t)^* \tag{3.9}$$

$$\frac{dx(t)^*}{dt} = v(t)^*. \tag{3.10}$$

with initial conditions

$$x(t=0)^* = 1 \tag{3.11}$$

$$v(t=0)^* = 0 \tag{3.12}$$

What this non-dimensional scaling means is that we can solve the system of equations once. We can plot the solution in non-dimensional form and anyone can find dimensional solutions by multiplying the solution by a constant. Instead of solving the equations for each choice of parameters, we only have to solve it once.

4. Taylor Series

When solving equations such as 2.8 & 2.9 we often have information about the initial state of the system and would like to understand how the system will evolve with time. We need a way to integrate this equation forward in time, given the starting state. At the heart of such approximations is the Taylor series.

Consider an arbitrary function and assume that we have all the information about the function at the origin (x=0) and we would like to construct an approximation for $x > 0$. As an example let's construct an approximation for the function e^x, given only information at the origin. Let's assume that we can create a polynomial approximation to the original function, $\hat{f}$, i.e.

$$\hat{f} = a + bx + cx^2 + dx^3 + ... \tag{4.1}$$

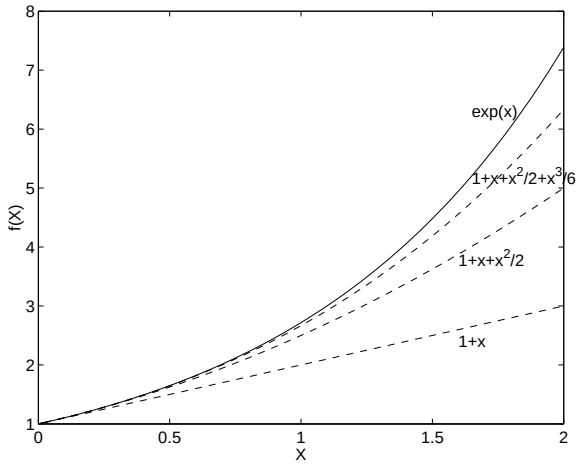

Figure 2. Taylor series approximation of the function e^x. We have included the first three terms in the expansion, it is clear that the approximation is valid for larger and larger x as more terms are retained.

where we will solve for the unknown coefficients a, b, c, d, etc.

The simplest approximation might be to take the derivative and extrapolate the function forward, precisely we mean,

$$\hat{f} = f(x=0) + \left.\frac{df}{dx}\right|_{x=0} x, \tag{4.2}$$

where the notation $df/dx|_{x=0}$ means you take the derivative of the function with respect to x and then evaluate that derivative at the point $x = 0$. Since $e^0 = 1$, this approximation for our test case reduces to

$$\hat{f} = 1 + x. \tag{4.3}$$

This approximation to the original function is plotted in figure 2. We see that the approximation works well when x is small and deviates further away, this is expected. By using the extrapolation technique we have simply stated that the value of the function and its derivative must be the same in both the real function and the approximation.

We can improve the approximation by matching the second derivative at the origin as well, i.e.

$$\hat{f}(x=0) = a = f(x=0) \tag{4.4}$$

$$\left.\frac{d\hat{f}}{dx}\right|_{x=0} = b = \left.\frac{df}{dx}\right|_{x=0} \tag{4.5}$$

$$\left.\frac{d^2\hat{f}}{dx^2}\right|_{x=0} = 2c = \left.\frac{d^2f}{dx^2}\right|_{x=0} \tag{4.6}$$

if we continued this approximation to higher and higher derivatives we would obtain the expression

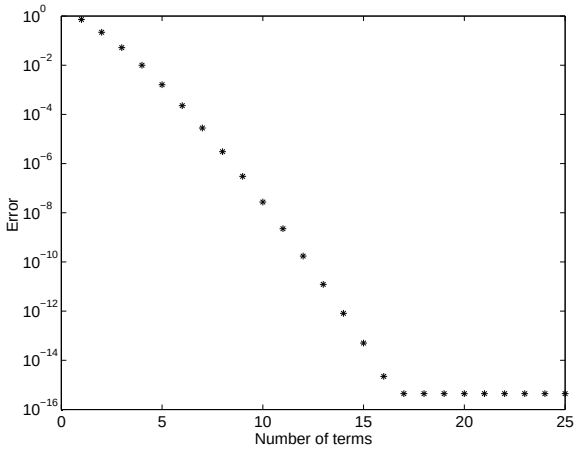

Figure 3. Error in the Taylor series approximation of the function e^x as more terms are included in the sum. We see that the fundamental accuracy limit is hit at the roundoff error of $\epsilon = 10^{-16}$ when 15 terms are included in the Taylor series. The error is computed as the difference between the true function and the Taylor approximation at $x = 1$.

$$\hat{f}(x) = f(x=0) + x \left.\frac{df}{dx}\right|_{x=0} + \frac{x^2}{2} \left.\frac{d^2f}{dx^2}\right|_{x=0} + \frac{x^3}{6} \left.\frac{d^3f}{dx^3}\right|_{x=0} + ... \frac{x^n}{n!} \left.\frac{d^nf}{dx^n}\right|_{x=0} \quad (4.7)$$

Applying approximation 4.7 to our example function shows that the approximation improves as more terms are included. See figure 2 where we have plotted the series up to terms of x^3. The error for the approximation in the exponential function with the Taylor series as more terms are included is plotted in Figure 3.

Exercise: DEQ 2 Write out the first 4 terms of the Taylor series for the function $f(x) = sin(x)$. Plot the true function and the approximation as each term is added on the interval $0 < x < \pi$. Compute and plot the error in the approximation at each order.

Exercise: DEQ 3 Write out the first 4 terms of the Taylor series for the function $f(x) = ln(x)$ about the point $x = 3$. Plot the true function and the approximation as each term is added on the interval $3 < x < 4$. Compute and plot the error in the approximation at each order.

5. Your first numerical method: Euler's method

When solving an equation such as the motion of the mass on the spring it is often described as an initial value problem. This means that you often know the initial state of the system and the equations tell you how the system will evolve with time. The initial state of the system might be that you pull back the spring, hold the block at rest, and let go. The initial condition is then $x = x_0$ and $v = 0$, the spring is displaced but held still until you release.

Applying this initial condition to the equations, 3.9 & 3.10 shows that the instant that your release the spring

$$\left.\frac{dv}{dt}\right|_{t=0} = -1 \tag{5.1}$$

$$\left.\frac{dx}{dt}\right|_{t=0} = 0. \tag{5.2}$$

This simply means that the acceleration of the mass is negative (the spring is contracting) but the position of the mass is not yet changed. The important thing to note from the above equation is that you know the value of the function (position and velocity are given from the initial condition) and you know the value of their derivatives from the equation.

This problem is then very similar to the Taylor series considered in the previous section. We know information about the function at an given time and we want to use this information to predict later times. The simplest way to construct an approximation simply use the derivative information to propagate the solution forward in time, i.e. keep only the first term of the Taylor series.

$$v(t = \Delta t) = v(t = 0) + \Delta t \frac{dv}{dt}_{|t=0} \tag{5.3}$$

$$x(t = \Delta t) = x(t = 0) + \Delta t \frac{dx}{dt}_{|t=0}. \tag{5.4}$$

These expression simply mean that we are using the value of the derivative to extrapolate the initial condition to a new time Δt. We know from figure 2 that the smaller the time that we extrapolate, the better the approximation will be.

To continue integrating this equation forward in time we repeatedly apply this approximation at each time step interval, Δt.

$$v^{N+1} = v^N + \Delta t \frac{dv}{dt}^N \tag{5.5}$$

$$x^{N+1} = x^N + \Delta t \frac{dx}{dt}^N, \tag{5.6}$$

where the notation X^N means the position of the mass at time step N. Equations 5.5 and 5.6 are simply iterative equations. Everything on the right hand side of the equation is known, these are values of the position and velocity at the current time step. The new value is simply updated and the equations are propagated forward in time.

6. Implementing Euler's method in MATLAB, in detail

Now we will solve the equations for the mass on the spring using Euler's method and the non-dimensional equations. In this section for simplicity we will drop the * notation for non-dimensional variables. From the prior section we found that the iteration equation for the position and velocity was given as:

$$v^{N+1} = v^N + \Delta t \frac{dv}{dt}^N \tag{6.1}$$

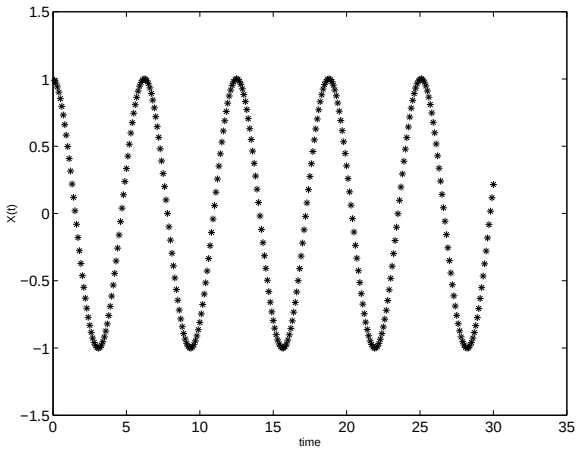

Figure 4. Result of applying Euler's method to the mass spring system.

$$x^{N+1} = x^N + \Delta t \frac{dx}{dt}^N, \tag{6.2}$$

Substituting equations 3.10 & 3.9 in to the above equations yields

$$v^{N+1} = v^N - x^N \Delta t \tag{6.3}$$

$$x^{N+1} = x^N + v^N \Delta t. \tag{6.4}$$

with the initial conditions $x^0 = 1, v^0 = 0$. To implement this method in MATLAB you could type the following commands into an m-file and run it. The comments are placed so you will know what each line is doing. You should type this up and make sure that you can get the same result!

```
clear;    %% clear exisiting workspace
x = 1;    %% initial condition
v = 0;    %% initial condition
dt = 0.1;    %% set the time step interval
time = 0;    %% set the time=0

figure(1);  %% open figure
clf;        %% clear the figure
hold on;    %% accumulate contents of the figure

for i = 1:300   %% number of time steps to take
   v = v - dt*x    %% Equation 6.3
   x = x + dt*v    %% Equation 6.4
   time = time + dt
   plot(time,x,'*');
end
```

Try this out and see if you can get the result shown in Figure 4.

You may have noted that we cheated a little bit. Equation 6.4 was actually implemented as

$$x^{N+1} = x^N + v^{N+1}\Delta t, \tag{6.5}$$

you should be able to convince yourself of this by reviewing the lines of code.

Let's take a different approach and implement the equations as specified by equation 6.4. Instead of using separate variables for position and velocity we will create a list of numbers (a vector) where each element of the list corresponds to a variable that we are solving for. This representation will become valuable as we increase the size of the system and have many variables to solve for. Try typing in the example below and understand what each line of the program is doing. The program is nearly the same as the last only the position and velocity are stored in a vector, y, rather than as individual variables.

```
clear;    %% clear exisiting workspace
y(1) = 1;    %% initial condition, position
y(2) = 0;    %% initial condition, velocity
dt = 0.1;    %% set the time step interval
time = 0;    %% set the time=0

figure(1);  %% open figure
clf;        %% clear the figure
hold on;    %% accumulate contents of the figure

for i = 1:300   %% number of time steps to take
   dy(2) = -y(1)    %% Equation for dv/dt
   dy(1) = y(2)     %% Equation for dx/dt
   y = y + dt*dy  %% integrate both equations with Euler
   time = time + dt
   plot(time,y(1),'*');
end
```

If you typed in the above program you should get the result shown in Figure 5 It seems that the method has gone haywire, and indeed it has. Soon we will cover error estimation and be able to prove why the first method worked and the second one did not. We will find that the error in Euler's method is growing in time. The effect of the error can be seen by reducing the time step and increasing the number of time steps by a factor of ten. The comparison with $\Delta t = 0.1$ and $\Delta t = 0.01$ are compared in Figure 6. When we cover error analysis later we will find later that Euler's method is THE WORST POSSIBLE METHOD WE COULD USE. It is however, very simple, and very intuitive so we will continue to use it for now. Figure 6 represents a powerful idea, convergence. We will find that as we continue to decrease the time step then the solution will converge.

Before we continue with the analysis of error, let's look at the programming aspects of the code that we have written and try to make it more flexible. A good way to think how to make the program more flexible (i.e. for solving different systems of equations) is to ask yourself the question, what would be different and what would be the same in our program? Euler's method is general, and therefore much of that loop would be the same. Computing the derivatives is dependent on each system of equations. The initial conditions are also dependent on the system.

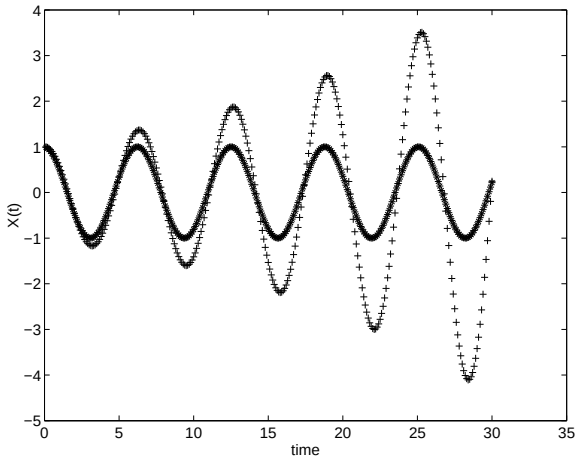

Figure 5. Result of applying 'true' Euler's method to the mass spring system. We left the solution from Figure 4 for comparison. Clearly, the previous method worked much better.

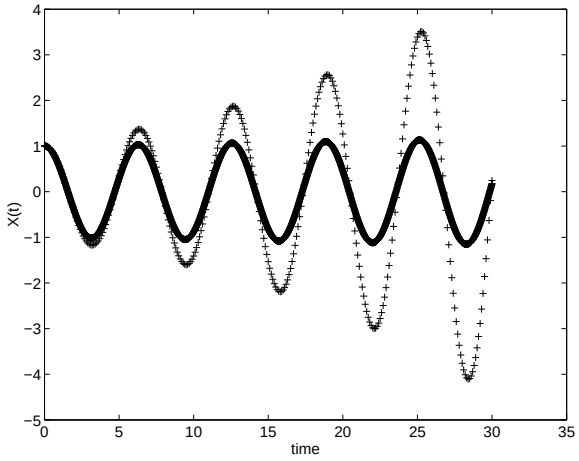

Figure 6. Comparison of Euler's method with $\Delta t = 0.1$ and $\Delta t = 0.01$

Let's split this algorithm into a few separate functions so that we can reuse pieces of the Euler algorithm.

First, create an m-file called euler.m and enter the following code for a general Euler solver.

```
function euler(dydtHandle,y,dt,steps)
   clf;
   time = 0;

   for i =1:steps
```

```
    dy = feval(dydtHandle,y,time);
    y = y + dy*dt;
    time = time+dt;
    plot(time,y(1),'.');
    hold on
  end
```

This code is nearly the same as that used in the previous section. The only difference is now we have made the Euler solver a function, rather than part of one big program. We would like to use the same Euler solver for many different systems of equations, the above form allows us to generalize. You should have read the MATLAB section on functions in the text so it should be clear what this code is doing. The input arguments are the handle to a function which computes the derivatives given the current values (dydtHandle), the vector of current values (y), the time step size (dt), and the number of time steps (steps). The only new programming feature might by the passing of a function handle to a function. This is a very common programming trick that is very useful. The general idea is that we would like our Euler solver to work for any general system of equations. Therefore a MATLAB function which is computing the derivatives must be an input argument to the Euler solver. You should read the MATLAB help on function handles and the feval command to really understand what this code is doing. Just as we pass variables to functions we can pass functions to functions.

Next we will write our main function which sets the initial conditions and creates the function that computes the derivatives. Type the following code into a file called spring.m.

```
function spring()
   y = zeros(2,1)   %% initialize to zero
   y(1) = 1;    %% initial condition, position
   y(2) = 0;    %% initial condition, velocity
   euler(@derivs,y,0.01,3000)

function dy = derivs(y,time)
   dy = zeros(2,1);
   dy(2) =-y(1);   %% dv/dt = -x
   dy(1) = y(2);   %% dx/dt = v
```

This code is composed of a main program, *spring*, and the function that computes the current values of the derivatives, *derivs*. The format @derivs in the euler function call means pass the handle of the function derivs. You should read about function handles in the MATLAB book and help documentation. Now if we wanted to solve a different set of equations, the function euler could stay the same and the function spring and derivs could be rewritten for the new system.

We will do one final modification before we move on to the next section. The functions above works fine, however it could become quite confusing if you had many variables in the system of equations. While it may be easy to remember that index 1 into the y array means position and index 2 is velocity, when the number of variables is increased dramatically it is hard to keep track! One possible way to fix this is to create variable names that refer to the indices into the list of numbers.

Since you may want these index variables in other functions, it is convenient to pack the variables together in a data structure. Note that we will be cleaning up the Euler solver so that all the data at each time step is returned from the function.

```
function spring()
   y = zeros(2,1)   %% initialize to zero
   in.X = 1;     %% index for position, X
   in.V = 2;     %% index for velocity, V
   y(in.X) = 1;    %% initial condition, position
   y(in.V) = 0;    %% initial condition, velocity
   [T, Y] = euler(@derivs,y,0.01,in,3000);
   plot(T,Y(:,in.X));

function dy = derivs(y,time,in)
   dy = zeros(2,1);
   dy(in.V) =-y(in.X);   %% dv/dt = -x
   dy(in.X) = y(in.V);   %% dx/dt = v
```

Let's also clean up the Euler solver a little bit as well so that the function will output all the data at each time step into a big box of numbers, *data* and *t*.

```
function [t,data] = euler(dydtHandle,y,dt,in,steps)
   time = 0;
   t = zeros(steps,1);
   data = zeros(steps,length(y));

   for i =1:steps
      dy = feval(dydtHandle,y,time,in);
      y = y + dy*dt;
      time = time+dt;
      t(i) = time;
      data(i,:) = y';
   end
```

All the programs shown in this section are essentially equivalent. The last one that we have written is really no simpler (and maybe even more complicated) than the first. The power in the way we have developed the final program is that it is easy to change for different equations. You now have a very general method for solving any system of differential equations using Euler's method. The way we solved this problem is not unique. There are an infinite number of ways to implement this code. We have tried to work toward a program that is easy to modify and easy to understand. There might be even cleaner and easier programs that provide more flexibility and easier reading, can you come up with one? You should make sure that you understand the programs created in this section before moving on. You should type them up, run them, change them, and experiment with them to help make sense of all these ideas.

> **Exercise: DEQ 4** Lorentz Attractor. Lorentz proposed a system of differential equations as a simple model of atmospheric convection. He hoped to use the equations to aid in weather prediction. By accident

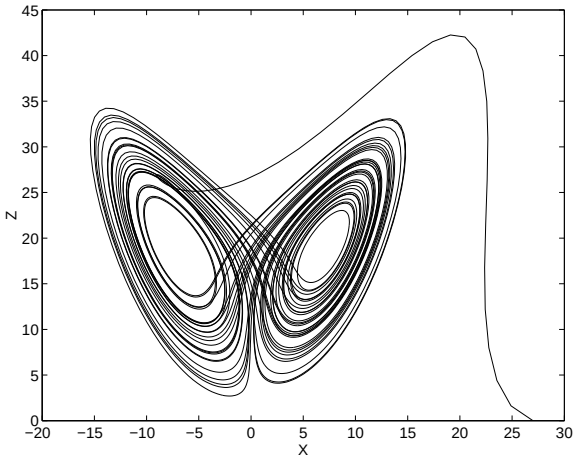

Figure 7. Plot of X vs. Z produces the Lorentz 'butterfly'

he noticed that he when he solved his equations numerically, he got completely different answers for only a small change in initial condition. He also noticed that while the variables plotted as a function of time seemed random, the variables plotted against each other showed regular patterns. You will use your Euler solver to reproduce some of Lorentz's results. The equations Lorentz derived were:

$$\frac{dx}{dt} = 10(y - x) \tag{6.6}$$

$$\frac{dy}{dt} = x(20 - z) - y \tag{6.7}$$

$$\frac{dz}{dt} = xy - \frac{8}{3}z \tag{6.8}$$

We will not discuss the derivation of these equations but they were based on physical arguments relating to atmospheric convection. The variables x, y, z represent physical quantities such as temperatures and flow velocities, while the numbers 10, 20, and $8/3$ represent properties of the air.

Code up these equations using your Euler solver and explore their behavior. Plot the time series for the variable X for an arbitrary initial condition. Save the plot and show the behavior for a slightly changed (1 %) initial condition. Plot the variable x vs. z. You will know you are on the right track if that plot looks like Figure 7. Adjust the time step and see if the solution is converging.

7. Error Estimates: Local Error

Let us return to the Taylor series approximation and use it to estimate the error in the approximation to the derivative. If we assume that we have all the data (f and

it's derivatives) at t=0, then the value of the function at time $t = \Delta t$ is given as

$$f(\Delta t) = f(t=0) + \Delta t \left.\frac{df}{dt}\right|_{t=0} + \frac{\Delta t^2}{2}\left.\frac{d^2 f}{dt^2}\right|_{t=0} + \frac{\Delta t^3}{6}\left.\frac{d^3 f}{dt^3}\right|_{t=0} + ... \frac{\Delta t^n}{n!}\left.\frac{d^n f}{dt^n}\right|_{t=0} \tag{7.1}$$

Rearranging this equation yields,

$$\frac{f(\Delta t) - f(t=0)}{\Delta t} = \left.\frac{df}{dt}\right|_{t=0} + \frac{\Delta t}{2}\left.\frac{d^2 f}{dt^2}\right|_{t=0} + \frac{\Delta t^2}{6}\left.\frac{d^3 f}{dt^3}\right|_{t=0} + ... \frac{\Delta t^{n-1}}{n!}\left.\frac{d^n f}{dt^n}\right|_{t=0} \tag{7.2}$$

Since Δt is small then the series of terms on the right hand side is dominated by the term with the smallest power of Δt, i.e.

$$\frac{f(\Delta t) - f(t=0)}{\Delta t} = \left.\frac{df}{dt}\right|_{t=0} + \frac{\Delta t}{2}\left.\frac{d^2 f}{dt^2}\right|_{t=0} + ... \tag{7.3}$$

Therefore, the Euler approximation to the derivative is off by a factor proportional to Δt. The good news is that the error goes to zero as smaller and smaller time steps are taken. The bad news is that we need to take very small time steps to get good answers. In the next section we will see that obtaining better accuracy does not require much extra work.

We call the error in the approximation to the derivative over one time step the local truncation error. This error occurs over one time step and can be estimated from the Taylor series, as we have just shown.

Exercise: DEQ 5 Write a MATLAB program to solve

$$\frac{dy}{dt} = y \tag{7.4}$$

using Euler's method. Use the initial condition that $y(t=0) = 1$, and solve on the interval $0 < t < 1$. The exact solution to this equation is $y(t) = e^t$. Try solving with a time step of 0.25, 0.1, 0.05, and 0.01. Plot the error between the numerical solution and the analytical solution ($y = e^t$) as a function of Δt. The result you should obtain is shown in Figure 8. Can you explain why the departure between the actual and predicted error grows at large Δt.

Let's return to our original problem of the spring and see if we can understand why the first method that we tried worked so much better than the true Euler method in equation 5. If you recall the first implementation worked quite well because it was not a true Euler method. The first algorithm that we tried can be written as

$$x^{N+1} = x^N + v^N \Delta t \tag{7.5}$$

$$v^{N+1} = v^N - k x^{N+1} \Delta t \tag{7.6}$$

Advancing to the next time step the equation for x becomes

$$x^{N+2} = x^{N+1} + v^{N+1} \Delta t \tag{7.7}$$

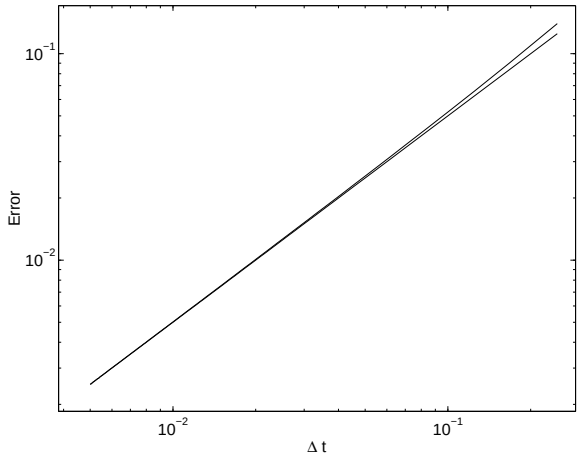

Figure 8. Error as the time step is changed for Euler's method applied to $dy/dt = y$. Both the error and the function $error = \Delta t/2$ are plotted. One can see from Equation 7.3, that the error in the approximation is just as predicted, with slight departure for large time steps. Note that we are plotting the difference between the true and numerical solution at $t = 1$.

which can be rewritten as

$$x^{N+2} = x^N + v^N \Delta t + \Delta t(v^N - k\Delta t(x^N + v^N \Delta t)). \tag{7.8}$$

Collecting terms we obtain

$$x^{N+2} = x^N + v^N 2\Delta t - kx^N \Delta t^2 - k\Delta t^3 v^N \Delta t. \tag{7.9}$$

using the fact that $-kx^N = d^2x/dt^2$ and $v^N = dx/dt$ we can write the expression for x^{N+2} as

$$x^{N+2} = x^N + 2\Delta t \frac{dx}{dt}^N + \Delta t^2 \frac{d^2x}{dt^2}^N + \Delta t^3 \frac{d^3x}{dt^3}^N . \tag{7.10}$$

The first 3 terms on the right hand side of this equation are the Taylor series for the point x^{N+2}: the Δt^3 term does not cancel. If we expand the velocity we will find the same result. This method is formally accurate to terms of order Δt^2 rather than Δt of the true Euler method. The method that was the easiest to program, in this case turned out to be a good one. Unfortunately, in general the easier the method the less accurate!

8. Error Estimates: Global Error

In the last section we looked at how the error changed as we varied the step size. We also used the Taylor series to estimate the discretization error associated with the scheme that we were using. In addition to error at each time step, there is a global error associated with the accumulation of errors at each time step. Refer back to the spring problem and our attempt to apply the Euler method to the motion, Figure 6. It is clear that in this case the error is accumulating. If we look at the results

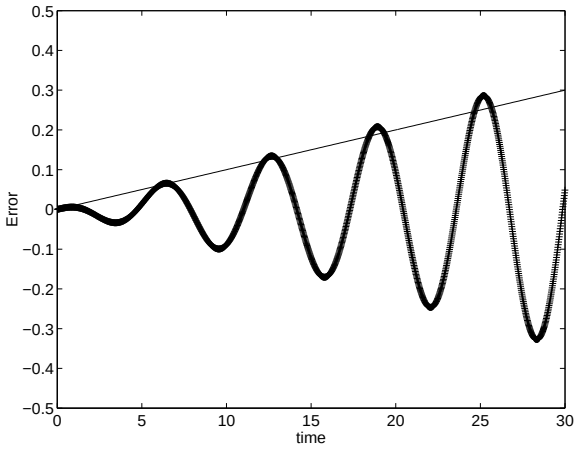

Figure 9. Error between numerical and true solution as a function of time for the mass on spring system. The straight line shows $t\Delta t/2$. We see that the error is primarily a simple sum of the truncation error at each time step. The non-linear behavior seen in the error is due to the fact that the error depends on the solution, i.e. at long times the error will grow exponentially.

for one cycle of the oscillation with the large step size we might think that the solution has some error, but we might think that error is acceptable - it seems that the position is off by 10%. At the end of 4 oscillations the large step size is clearly giving results that are not acceptable. Physically we know that the system should conserve energy and clearly the numerical solution is artificially gaining energy. The accumulation of error is known as global error.

In Figure 9 we show the global error $(y^n - y_{exact}(t_n))$ as a function of time for the mass-spring system. We find that the global error of the numerical approximation is well estimated by the simple sum of the local truncation errors. We find that the situation is somewhat worse since the local truncation error depends on the second derivative of the function itself (see equation 7.3). Therefore, as the error accumulates and the solution grows, even more error is introduced into the approximation. At long times we find exponential growth of the error. It is clear that accumulation of global error can be a severe problem for Euler's method.

Now that we have discussed the impact of numerical truncation error we will begin to look at schemes that are designed to be more accurate.

9. Midpoint method

When numerically solving differential equations, we want to find the best estimate for the 'effective slope' across the time step interval. So far we have used the value of the slope at the start of the interval since this is the only location where we have any information about the function. Consider figure 10 where we have plotted the function $f(t) = e^t$ and various approximations for the derivative to shoot across the interval $0 < t < 1$. We clearly see that the simple Euler method shoots too low. If we somehow knew the value at the endpoint of the shooting interval ($t = 1$) and

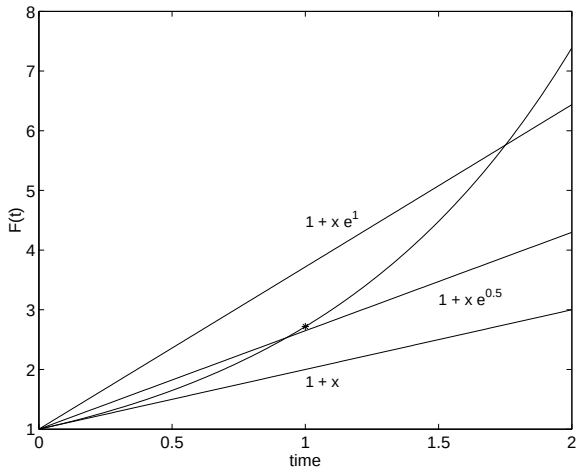

Figure 10. Midpoint approximation for $f(t) = e^t$

used that value, we would shoot too high. If we conjecture that using a value from the midpoint of the interval might be better representation of the effective slope of across the interval, we get a much better answer. The reason why can be derived quite easily (and seen in Figure 10). The approximation that looks good in this figure is

$$f(\Delta t) = f(t=0) + \Delta t \left.\frac{df}{dt}\right|_{t=\Delta t/2} \tag{9.1}$$

Expanding the derivative of f with respect to time using a Taylor series yields

$$\left.\frac{df}{dt}\right|_{t=\Delta t/2} = \left.\frac{df}{dt}\right|_{t=0} + \frac{\Delta t}{2}\left.\frac{d^2 f}{dt^2}\right|_{t=0} + \frac{\Delta t^2}{4}\left.\frac{d^3 f}{dt^3}\right|_{t=0} + ... \tag{9.2}$$

Substituting this expression into equation 9.1 yields

$$f(\Delta t) = f(t=0) + \Delta t \left(\left.\frac{df}{dt}\right|_{t=0} + \frac{\Delta t}{2}\left.\frac{d^2 f}{dt^2}\right|_{t=0} + \frac{\Delta t^2}{4}\left.\frac{d^3 f}{dt^3}\right|_{t=0} + ... \right) \tag{9.3}$$

We see that the first three terms of the right-hand-side exactly match the Taylor series approximation. Therefore the error of the approximation is on the order of Δt^2. The problem with this method is that this method requires knowing data in the future. The problem can be remedied with a simple approximation, we will use the Euler method to shoot to an approximated midpoint. We will estimate the derivative there and then use the result to make the complete step. Specifically, the midpoint method works as follows.

$$y^{N+1/2} = y^N + \frac{\Delta t}{2}\frac{dy}{dt}^N \tag{9.4}$$

$$y^{N+1} = y^N + \Delta t \frac{dy}{dt}^{N+1/2}. \tag{9.5}$$

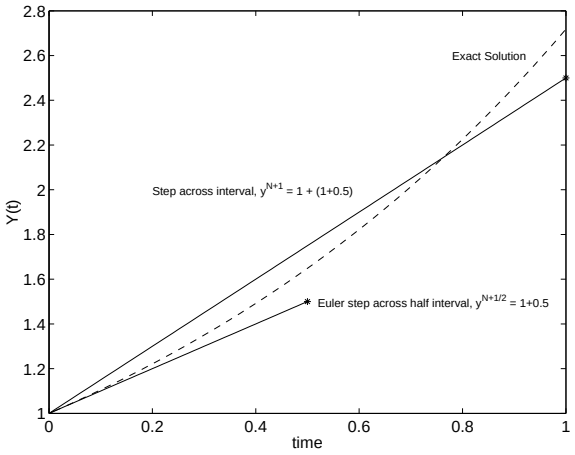

Figure 11. Example of the midpoint method for $y(t) = e^t$, where $\Delta t = 1$. First we take an Euler step to the midpoint of the interval. The 'effective slope' for the interval is computed at this approximate midpoint location. The midpoint slope is used to take the full step across the interval.

The first step applies Euler's method halfway across the interval. The values of $y^{N+1/2}$ and $t = \Delta t/2$ are used to recompute the derivatives. The values of the estimated midpoint derivatives are then used to shoot across the entire domain. A schematic is shown in Figure 11 for the equation $dy/dt = y$ using a large time step of $\Delta t = 1$ in order to amplify the effects. The initial condition is $y(t = 0) = 1$, so therefore $dy/dt^0 = 1$ via the governing equation. Applying the governing equation and the initial condition

$$y^{N+1/2} = 1 + 1\Delta t/2 \tag{9.6}$$

$$y^{N+1} = 1 + \Delta t(1 + \Delta t/2) \tag{9.7}$$

> **Exercise: DEQ 6** Write a MATLAB program similar to the Euler solver that applies the midpoint method. The program should be general so that you can apply it to any system of equations. The program should also follow the same usage as the Euler solver so that in your programs you could easily switch between methods.
>
> Use your midpoint solver to solve your mass-spring system. Compare the result to the Euler solver. Change the time step and assess the error of the approximation by comparing to the exact solution.

10. Runge-Kutta Method

There are many, many, many different schemes for solving ODEs numerically. Some of them exist for a good reason, some are never used, some were relevant when computers were slow, some just aren't very good. However, different types of equations are better suited for different methods, we will discuss this more in the next section. The basic ideas, however are similar to the ones that we have already presented: the

schemes try to minimize the amount of error in estimating the slope that propagates the solution forward.

One of the standard workhorses for solving ODEs is the called the Runge-Kutta method. This method is simply a higher order approximation to the midpoint method. Instead of shooting to the midpoint, estimating the derivative, the shooting across the entire interval - the Runge-Kutta method takes four steps, shooting across one quarter of the interval, estimating the derivative, then shooting to the midpoint, and so on. We will not provide a formal derivation of the Runge-Kutta algorithm, instead we will present the method and implement it.

The general ODE that we are solving is given as,

$$\frac{dy}{dt} = f(y,t). \tag{10.1}$$

The Runge-Kutta method can be defined as:

$$k1 = \Delta t f(t^N, y^N) \tag{10.2}$$

$$k2 = \Delta t f(t^N + \Delta t/2, y^N + k1/2) \tag{10.3}$$

$$k3 = \Delta t f(t^N + \Delta t/2, y^N + k2/2) \tag{10.4}$$

$$k4 = \Delta t f(t^N + \Delta t, y^N + k3) \tag{10.5}$$

$$y^{N+1} = y^N + \frac{k1}{6} + \frac{k2}{3} + \frac{k3}{3} + \frac{k4}{6} \tag{10.6}$$

One should note the similarity to the midpoint method discussed in the previous section. Also note that each time step requires 4 evaluations of the derivatives, i.e. the function f.

Since we have only given the equations to implement the Runge-Kutta method it is not clear how the error behaves. Rather than perform the analysis, we will compute the error by solving an equation numerically and compare the result to an exact solution as we vary the time step. To test the error we solve the model problem, $dy/dt = -y$, where $y(0) = 1$ and we integrate until time $t = 1$. In Figure 12 we plot the error between the exact and numerical solutions at $t = 1$ as a function of the time step size. We also plot a function $f = C\Delta t^4$ on the same graph. We find that the error of the Runge-Kutta method scales as Δt^4. This is quite good - if we double the resolution (half the time step size) we get 16 times less error!.

> **Exercise: DEQ 7** Implement the Runge-Kutta method by writing a function that works in the same way as your midpoint method and Euler solvers, only using this new algorithm.
>
> Use your Runge-Kutta solver to solve your mass-spring system. Compare the result to the Euler and midpoint solver. Compare the Runge-Kutta solution to the exact solution and plot the error on a log-log plot as you vary the time step size. How does this plot compare to one generated applying the midpoint method?

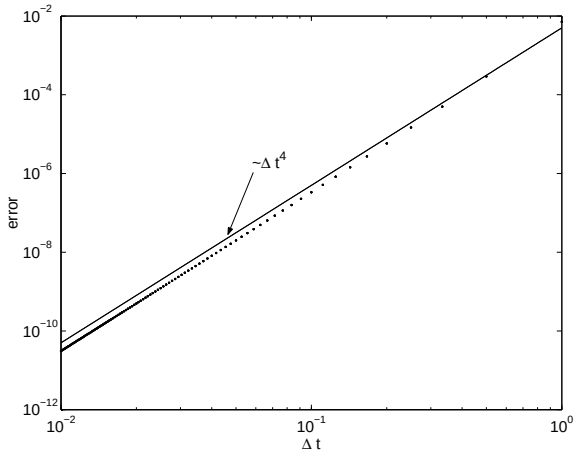

Figure 12. The error between the Runge-Kutta method and exact solution as a function of time step size. One the plot we also display a function that scales as Δt^4. We see that this fits the slop of the data quite well, therefore error in the Runge-Kutta approximation scales as Δt^4.

11. Stability and Stiff Equations: Backward Euler Method

So far we have only discussed accuracy of ODE solvers, but another important issue is stability. In many situations we find that the algorithms we have thus far discussed will be unstable unless the time step is very small. By unstable, we mean the solution will begin to oscillate or grow in an unphysical manner.

Consider the following set of coupled ODEs:

$$\frac{dy}{dt} = -100(x + y) \tag{11.1}$$

$$\frac{dx}{dt} = -x \tag{11.2}$$

Equations of this form are common in chemistry where x and y are chemical species and the coefficients in the equation (100 & 1) are reaction rates. It is common in chemical systems to have reactions with varying rates of progress. Stiff equations are have widely different time scales that can cause a numerical solution difficulty even though the 'fast' time scale might not be important for the final solution.

> **Exercise: DEQ 8** Set up equations 11.1 and 11.2 using your Euler solver and the initial conditions that $x(t = 0) = y(t = 0) = 1$. Set your time step to 0.001 and take 2000 time steps. Plot the $x(t)$ and $y(t)$ on the same graph.

If you completed the exercise then you should generate a plot that looks like Figure 13. This behavior is very common in chemical systems, notice that one chemical (Y) goes from +1 to -1 very rapidly, then undergoes a much slower increased. The behavior is indicative of the two times scales present in the equations, 1/100 and 1.

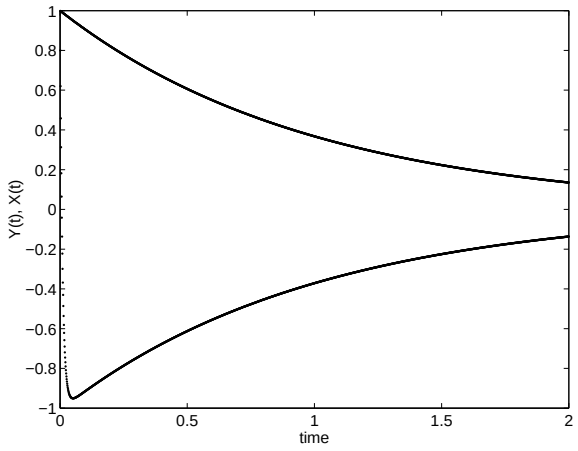

Figure 13. Solution to the stiff differential equations 11.1 and 11.2. Note the two time scales in the solution for y, there is the rapid decrease followed by the slow convergence to zero.

> **Exercise: DEQ 9** Set up equations 11.1 and 11.2 using your Euler solver and the initial conditions that $x(t = 0) = 1, y(t = 0) = -1$. Set your time step to 0.001 and take 2000 time steps. Plot the $x(t)$ and $y(t)$ on the same graph.

After completing this exercise you should notice that the plot looks very similar to Figure 13, only the observation of the fast time scale is not present. In this case we have set the initial condition such that the fast time scale does not influence the appearance of the solution to the equations. The fast time scale does influence the numerical solution, however. Try the following exercise:

> **Exercise: DEQ 10** Set up equations 11.1 and 11.2 using your Euler solver and the initial conditions that $x(t = 0) = 1, y(t = 0) = -1$. Set your time step to 0.02001 and take 2000 time steps. Plot the $x(t)$ and $y(t)$ on the same graph.

Once you complete this exercise you should obtain a plot that looks like Figure 14. The solution looks correct at early times, but then we find that the solution of y begins to oscillate across zero *after* the solution has come close to its final steady state solution of $x = y = 0$. The numerical solution is unstable. Try making Δt just a little bit larger and you will notice that the solution is wildly unstable.

So what has happened, why has the solution gone unstable? A simple way to view stability is consider the Euler method applied to the equation $dy/dt = -Cy, y(t = 0) = 1$. When we write out Euler's method for this equation we obtain

$$y^{N+1} = (1 - C\Delta t)y^N. \tag{11.3}$$

We know that the true solution of this ODE should decay from 1 to 0 on a time scale of $1/C$. It is easy to see that if $C\Delta t < 1$ then the iterative equation 11.3

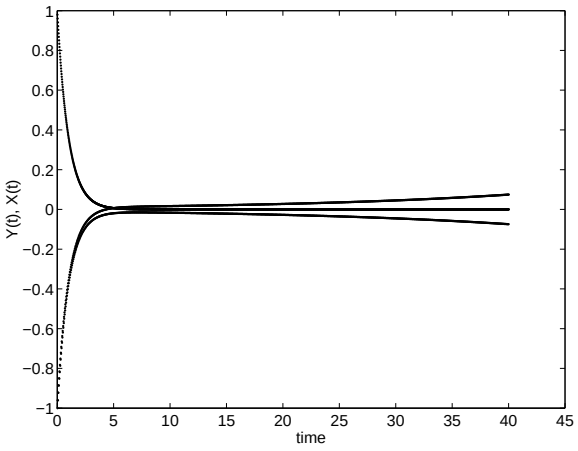

Figure 14. Solution to stiff equations showing unstable behavior.

has qualitatively the right behavior, each subsequent value of y is smaller than the last. If $C\Delta t > 1$ then we see that the sign of y will change with each iteration, i.e. the equations are behaving in a non-physical manner. Further, if $C\Delta t > 2$ then y will approach infinity as the number of iterations approaches infinity; each subsequent value of y is greater than the last. This stability problem is essentially what has happened in our example of the the two coupled equations. The stability requirements say that the equations must be integrated with the shortest time scale, regardless if that time scale is influencing the solution. This can be a serious limitation in many situations even on modern computers. In chemistry applications one often cares about the reactions that occur over several seconds where the fastest time scale in the system might be a few nano-seconds. It would require 10^9 time steps to solve this system, a lot of work even for a fast PC.

Fortunately methods exist for solving stiff equations. A very simple way to make a system more stable is to use the Backward Euler method. This method uses information from the end of the time step interval to estimate the derivative, i.e.

$$y^{N+1} = y^N + \frac{dy}{dt}^{N+1} \tag{11.4}$$

Applying the Backward Euler technique to the equation $dy/dt = -Cy, y(t=0) = 1$ yields,

$$y^{N+1} = \frac{y^N}{(1 + C\Delta t)}. \tag{11.5}$$

It is easy to see from this equation that regardless of the time step size the system will always display the correct behavior: subsequent values of y will always be smaller than the last, and y will never go negative. Of course stability does not mean accuracy.

The difficulty with Backward Euler solvers is that they require information from the future and are therefore often more difficult (or impossible) to implement for non-linear equations. However, for linear equations such as the simple system that we are dealing with in this section it is quite easy to implement the method directly.

Exercise: DEQ 11 Solve equations 11.1 and 11.2 using a Backward Euler method with the initial conditions $x(t = 0) = 1, y(t = 0) = -1$. You do not need to write a general Backward Euler solver, just implement a method specific for these equations. Set your time step to 0.25 and take 40 time steps. Plot the $x(t)$ and $y(t)$ on the same graph. You should obtain a solution that looks quite reasonable. Notice that you can obtain the same solution at a fraction of the cost of the forward Euler.

There are a variety semi-implicit methods that are effective for solving stiff systems in both linear and non-linear equations. These semi-implicit methods are very important in the solution of ODEs but we will not cover them in this class.

12. Using MATLAB

As mentioned previously, the backward Euler method is not convenient as a general method since the discretized equations and ability to apply the method depend on the problem that you are solving. Many general methods exist for stiff equations, but they are all based on the general idea of the backward Euler method, using information from the future tends to stabilize the numerical method.

At this point it is worth introducing the ODE solvers that are built into MATLAB. These solvers are very general, employ adaptive time stepping (speed up or slow down when it needs to), and have the capability for handling stiff equations. So you ask, if MATLAB can do all this already then why did you make us write all these programs? Well, it is very easy to employ packaged numerical techniques and obtain bad answers, especially in complex problems. It is also easy to use a package that works just fine, but the operator (i.e. you) makes a mistake and gets a bad answer. It is important to understand some of the basic issues of ODE solvers so that you will be able to use them correctly and intelligently. On the other hand, if other people have already spent a lot of time developing sophisticated techniques that work really well, why should we replicate all their work. We turn to these 'canned' routines at this point.

You have already developed a your own Runge-Kutta solver. MATLAB has a solver that is called ODE45 and the usage of the function will be very similar to the routines that you wrote for the Euler method, midpoint method, and Runge-Kutta. The ODE45 command uses the same Runge-Kutta algorithm you developed, only the MATLAB version uses adaptive time stepping. At this point in this tutorial we are going to let you figure out how to use the ODE45 command. You can read the help (i.e. type `help ode45`), though that isn't the best help out there. You can also surf the MATLAB web page documentation and find some examples of using the ODE45 routine.

As mentioned, the `ode45` command uses adaptive step sizes to control the error. With these algorithms you specify the error (there is a default) and the algorithm adjusts the time step size to maintain this at a constant level. Therefore, with adaptive algorithms you cannot generate a plot of error vs. time step size. In general these adaptive algorithms work by comparing the difference between taking a step with two methods that have different orders (i.e. midpoint (Δt^2) and Runge-Kutta

(Δt^4)). The difference is indicative of the error, and the time step is adjusted (increased or decreased) to hold the error constant.

> **Exercise: DEQ 12** Through the MATLAB documentation, figure out how to use the ode45 command. Apply the ODE45 command to the spring equations that we discussed in previous sections. Compare the results to those obtained with Euler, Midpoint, and Runge-Kutta solver. Compare the error between the true and numerical solutions. Try to adjust the error tolerance on the ODE45 command and see that the tolerance and the true error roughly agree.

MATLAB has other ODE solvers in addition to the ODE45. You can read the help on the MATLAB web page about the different solvers. The two most common that you will use are ode45 and ode23s. ODE23s is optimized for solving stiff equations. Try the following exercise to really see the difference.

> **Exercise: DEQ 13** Apply the ODE45 and ODE23s command to equations 11.1 and 11.2. Compare the results obtained with both methods, specifically note how many time steps each method took. Compare the difference between the two methods for the same error tolerance.

13. Is your solution correct?

One of the big difficulties in using numerical methods is that takes very little time to get an answer, it takes much longer to decide if it is right. Usually the first test is to check that the system is behaving physically. Usually before running simulations it is best to use physics to try and understand qualitatively what you think your system will do. Will it oscillate, will it grow, will it decay? Do you expect your solution to be bounded, i.e. if you start a pendulum swinging under free gravity you would not expect that the height of the swing would grow.

We already encountered unphysical growth when we solved the mass-spring motion using Euler's method in Figure 5 and 6. When the time step was large we noticed unphysical behavior: the amplitude of the mass on the spring was growing with time. This growth in oscillation amplitude is violating the conservation of energy principle, therefore we know that something is wrong.

One simple test of a numerical method is to change the time step and see what happens. If you have implemented the method correctly (and its a good method) the answer should converge as the time step is decreased. If you know the order of your approximation then you know how fast this decrease should happen. If the method has an error proportional to Δt then you know that cutting the time step in half should cut the error in half. You should NEVER NEVER turn in results from a simulation where you do not check different time steps to make sure that your solution is converging. You should also note that just because the solution converges does not mean that your answer is correct.

The MATLAB routines use adaptive time stepping, therefore you should vary the error tolerance rather than the time step interval. You should always check the convergence as you vary the error. Plot the difference between subsequent solutions as you vary the error tolerance. Further results on checking convergence and examples of doing so will be given in the problem example supplement.

References

Boyce & DiPrima, 2001 *Elementary Differential Equations and Boundary Value Problems*, John Wiley and Sons. (This is your Math text).

Epperson, 2002 *An Introduction to Numerical Methods and Analysis* John Wiley and Sonce.

Press, Tuekolsky, Vetterling, & Flannery, 1992 *Numerical Recipes in C*, Cambridge University Press. See www.nr.com for free pdf version.

MATLAB Product Documentation *www.mathworks.com*.