1

Numerical Methods for Differential Equations

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 for hundreds of years and there are many well-developed solution techniques. Often, systems described by differential equations are so complex, or the systems that they describe are so large, that a purely analytical solution to the equations is not tractable. It is in these complex systems where computer simulations and numerical methods are useful.

The techniques for solving differential equations based on numerical approximations were developed before programmable computers existed. During World War II, it was common to find rooms of people (usually women) working on mechanical calculators to numerically solve systems of differential equations for military calculations. Before programmable computers, it was also common to exploit analogies to electrical systems to design analog computers to study mechanical, thermal, or chemical systems. As programmable computers have increased in speed and decreased in cost, increasingly complex systems of differential equations can be solved with simple programs written to run on a common PC. Currently, the computer on your desk can tackle problems that were inaccessible to the fastest supercomputers just 5 or 10 years ago.

This chapter will describe some basic methods and techniques for programming simulations of differential equations. First, we will review some basic concepts of numerical approximations and then introduce Euler's method, the simplest method. We will provide details on algorithm development using the Euler method as an example. Next we will discuss error approximation and discuss some better techniques. Finally we will use the algorithms that are built into the MATLAB programming environment.

The fundamental concepts in this chapter will be introduced along with practical implementation programs. In this chapter we will present the programs written in the MATLAB programming language. It should be stressed that the results are not particular to MATLAB; all the programs in this chapter could easily be implemented in any programming language, such as C, Java, or assembly. MATLAB is a convenient choice as it was designed for scientific computing (not general purpose software development) and has a variety of numerical operations and numerical graphical display capabilities built in. The use of MATLAB allows the student to focus more on the concepts and less on the programming.

1.1 FIRST ORDER SYSTEMS

A simple first order differential equation has general form

$$\frac{dy}{dt} = f(y,t) \tag{1.1}$$

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'.

One of the simplest differential equations is

$$\frac{dy}{dt} = -y. \tag{1.2}$$

We will concentrate on this equation to introduce the many of the concepts. The equation is convenient because the easy analytical solution will allow us to check if our numerical scheme is accurate. This first order equation is also relevant in that it governs the behavior of a heating and cooling, radioactive decay of materials, absorption of drugs in the body, the charging of a capacitor, and population growth just to name a few.

To solve the equation analytically, we start by rearranging the equation as

$$\frac{dy}{y} = -dt \tag{1.3}$$

and integrate once with respect to time to obtain

$$ln(y) = -t + C \tag{1.4}$$

where C is a constant of integration. We remove the natural log term by taking the exponential of the entire equation

$$e^{ln(y)} = e^{C-t} \tag{1.5}$$

which finally can be written as

$$y = Ce^{-t}. \tag{1.6}$$

You can check that this answer satisfies the equation by substituting the solution back into the original equation. Since we obtained the solution by integration, there will always be a constant of integration that remains to be specified. This constant (C in our above solution) is specified by an initial condition or the initial state of the system. For simplicity of this chapter, we will proceed with the initial condition that $y(t = 0) = 1$, yielding C=1.

1.1.1 Discrete derivative

You should recall that the derivative of a function is equivalent to the slope. If you plotted the position of a car traveling along a long, straight, Midwestern highway as a function of time, the slope of that curve is the velocity - the derivative of position. We can use this intuitive concept of slope to numerically compute the discrete derivative of a known function.

On the computer we represent continuous functions as a collection discrete, sampled values. To estimate the slope (derivative) at any point on the curve we can simply take the change in rise divided by the change in run at any of the closely spaced points, a and b,

$$\frac{dy}{dt} = \frac{y_a - y_b}{t_b - t_a}. \tag{1.7}$$

We can demonstrate this concept of the numerical derivative with a simple MATLAB script.

Program 1.1: Exploring the discrete approximation to the derivative.

```
t = linspace(0,2,20);       %% define a time array
y = exp(-t);                 %% evaluate the function y = ê(-t)
plot(t,y);                   %% plot the function
hold on
dydt = diff(y)./diff(t);     %% take the discret derivative
plot(t(1:end-1),dydt,'r--'); %% plot the numerical derivative
plot(t,-y);                  %% plot the analytical derivative
```

This program simply creates a list of 20 equally spaced times between 0 and 2 and stores these numbers in the variable `t`. The program then evaluates the function $y = e^{-t}$ at these sample points and plots the function. Using the MATLAB `diff` command, we can evaluate the difference between neighboring points in the arrays y and t, which is used to compute an estimate of the derivative. The `diff` command simply takes the difference of neighboring points in a list of numbers $[y(1), y(2), ...y(n)]$ as $diff(y) = [y(2) - y(1), y(3) - y(2), ...y(n) - y(n-1)]$. The resulting list is one element shorter than the original function. Finally, in the script we plot the numerical and analytical function for the derivative. The plot that results from the script is shown in Figure 1.1. We see derivative is approximate, but appears to be generally correct. We will explore the error in this approximation in the exercises below and more formally in a later section.

1.1.2 Euler's method

We can use the numerical derivative from the previous section to derive a simple method for approximating the solution to differential equations. When we know the the governing differential equation and the start time then we know the derivative (slope) of the solution at the initial condition. The initial slope is simply the right hand side of Equation 1.1. Our first numerical method, known as Euler's method, will use this initial slope to extrapolate and predict the future. For the case of the function $dy/dt = -y$, $y(t = 0) = 1$, the slope at the initial condition is $dy/dt = -1$. In Figure 1.2 we show the function and the extrapolation based on the initial condition. The extrapolation is valid for times not to far in the future (i.e. $t < 0.2$), but the estimate eventually breaks down.

Another way to think about the extrapolation concept, is imagine you are in a car traveling on a small country highway. You see sign stating the next gas station is 10 miles away, you look at your speedometer and it says you are traveling 60 miles per hour. By extrapolation, you might predict that you will be at the gas station in 10 minutes. The extrapolation assumes you will continue at your current speed (derivative of position) until you reach the next gas station. If there is no traffic and your cruise control is working this assumption will be accurate. However, your prediction will not be accurate if there are many stop lights along the way, or you get stuck behind a large,

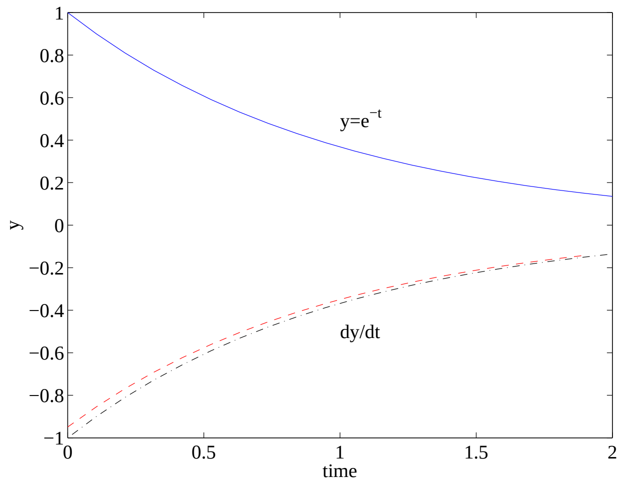

Fig. 1.1 Graphical output from running program 1.1 in MATLAB. The plot shows the function $y = e^{-t}$, the derivative of that function taken numerically and analytically.

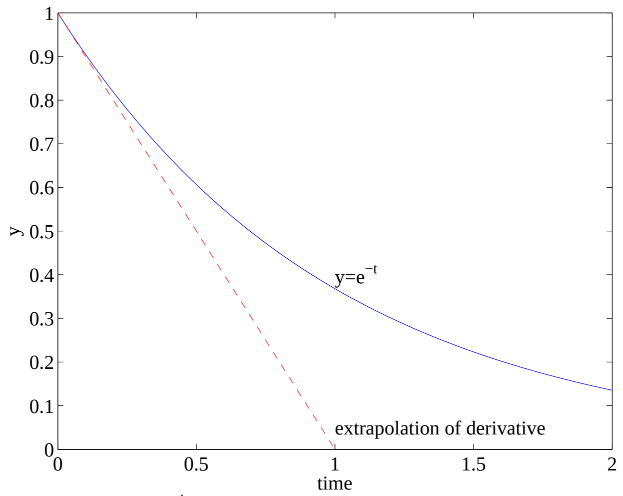

Fig. 1.2 Extrapolation of the function $y = e^{-t}$ based on the initial condition, $t = 0$. For very short times, the estimate is quite good, but clearly breaks down as we go forward in time.

slow-moving truck. Extrapolation is a good guess for where the system might be in a the near future, but eventually the predictions will break down except in the simplest systems.

Since our predictions far into the future are not accurate, we will take small steps while the extrapolation assumption is good, figure out where we are and then extrapolate forward again. Using our equation $dy/dt = -y$ and initial condition, we know the value of the function and the slope at the initial time, $t = 0$. The value at a later time, t_1, can be predicted by extrapolations as

$$y_1 = y_0 + \left.\frac{dy}{dt}\right|_{t=0} (t_1 - t_0). \tag{1.8}$$

where the notation $dy/dt|_{t=0}$ means the derivative of y evaluated at time equals zero. For our specific equation, the extrapolation formula becomes

$$y_1 = y_0 - y_0(t_1 - t_0). \tag{1.9}$$

This expression is equivalent to the discrete difference approximation in the last section, we can rewrite Equation 1.9 as

$$\frac{dy}{dt} = \frac{y_1 - y_0}{t_1 - t_0} = -y_0. \tag{1.10}$$

Once the value of the function at t_1 is known we can re-evaluate the derivative and move forward to t_2. We typically call the time interval over which we extrapolate, the time step $\Delta t = t_1 - t_0$. Equation 1.9 is used as an iteration

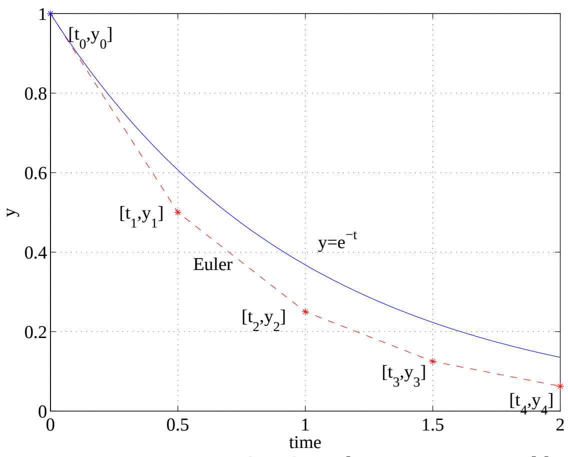

Fig. 1.3 Graphical output from running program 1 in MATLAB. The points connected by the dashed line are the results of the numerical solution and the solid line is the exact solution. The time step size is $t = 0.5$. This large time step size results in large error between the numerical and analytical solution, but is chosen to exaggerate the results. Better agreement between the numerical and analytical solution can be obtained by decreasing the time step size.

equation to simply march forward in small increments, always solving for the value of y at the next time step given the known information. This procedure is commonly called Euler's method.

The result of this method for our model equation using a time step size of $\Delta t = 0.5$ is shown in Figure 1.3. We see that the extrapolation of the initial slope, $dy/dt = -1$, gets us to the point (0.5,0.5) after the first time step. We then re-evaluate the slope, which is now $dy/dt = -0.5$ and use that slope to extrapolate the next time step to $t = 1$ where we land at (1,0.25). This process repeats. While the error in Figure 1.3 seems large the basic trend seems correct. As we make the time step size smaller and smaller the numerical solution comes closer to the true analytical solution.

A simple example of MATLAB script that will implement Euler's method is shown below. This program also plots the exact, known solution as a comparison.

Program 1.2: Euler's method for the fi rst order equation.

```
clear;                        %% clear exisiting workspace
y = 1;                        %% initial condition
dt = 0.5;                     %% set the time step interval
time = 0;                     %% set the start time=0
t_final = 2;                 %% end time of the simulation
Nsteps = round(t_final/dt);  %% number of time steps to take, integer
plot(time,y,'*');             %% plot initial conditions
hold on;                      %% accumulate contents of the  figure

for i = 1:Nsteps                  %% number of time steps to take
      y = y - dt*y;               %% Equation 1.9
      time = time + dt            %% Increment time
      plot(time,y,'*');           %% Plot the current point
end
t = linspace(0,t_final,100);     %% plot analytical solution
y = exp(-t);
plot(t,y,'r')
xlabel('time');                   %% add plot labels
ylabel('y');
```

1.1.3 Evaluating error using Taylor series

When solving equations such as 1.2 we typically have information about the initial state of the system and would like to understand how the system evolves. In the last section we used the intuitive approach of discussing extrapolation. We simply used information about the slope, to propagate the solution forward. In this section we will place the extrapolation notion in a more formal mathematical framework and discuss the error of these approximations using the Taylor series.

Consider an arbitrary function and assume that we have all the information about the function at the origin ($t = 0$) and we would like to construct an approximation around the origin. Let's assume that we can create a polynomial approximation to the original function, $\hat{y}$, i.e.

$$\hat{y} = a + bt + ct^2 + dt^3 + ... \tag{1.11}$$

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

The simplest approximation for the $\hat{y}$ would be to use the method of the last section to match the derivative of the true and approximated functions, precisely we mean,

$$\hat{y} = y(t = 0) + \left.\frac{dy}{dt}\right|_{t=0} t, \tag{1.12}$$

where the notation $dy/dt|_{t=0}$ means take the derivative of the function with respect to t and then evaluate that derivative at the point $t = 0$.

We can improve the polynomial approximation by matching the second derivative of the real function and the approximate function at the origin, i.e.

$$\hat{y}(t = 0) = a = y(t = 0) \tag{1.13}$$

$$\left.\frac{d\hat{y}}{dt}\right|_{t=0} = b = \left.\frac{dy}{dt}\right|_{t=0} \tag{1.14}$$

$$\left.\frac{d^2\hat{y}}{dt^2}\right|_{t=0} = 2c = \left.\frac{d^2y}{dt^2}\right|_{t=0}. \tag{1.15}$$

If we continued to match higher derivatives of the true and approximated functions we would obtain the expression

$$\hat{y}(t) = y(t = 0) + t\left.\frac{dy}{dt}\right|_{t=0} + \frac{t^2}{2}\left.\frac{d^2y}{dt^2}\right|_{t=0} + \frac{t^3}{6}\left.\frac{d^3y}{dt^3}\right|_{t=0} + ...\frac{t^n}{n!}\left.\frac{d^ny}{dt^n}\right|_{t=0} \tag{1.16}$$

which is known as the Taylor series. Taylor series are covered in most Calculus text where you can find more detail, examples, and generalizations.

To test this series we will return to our model function $y = e^{-t}$. Substituting this function in Equation 1.16 yields the approximation as

$$\hat{y}(t) = 1 - t + \frac{t^2}{2} - \frac{t^3}{6} + ...(-1)^n\frac{t^n}{n!} \tag{1.17}$$

The simplicity of this expression is due to the fact that all derivatives of y evaluated at the origin are 1. In Figure 1.4 we plot the first few terms of the series in comparison to the true function. We see that the approximation works well when t is small and deviates for large values of t. We also find that more terms included in the Taylor series results in better agreement between the true and approximate functions.

Now we return to the context of the initial value problem. In the initial value problem we want to move from the initial condition to the time $t = \Delta t$, a short time into the future. Therefore we evaluate the Taylor series at a time Δt into the future.

$$y(\Delta t) = y(0) + \Delta t\left.\frac{dy}{dt}\right|_{t=0} + \frac{\Delta t^2}{2}\left.\frac{d^2y}{dt^2}\right|_{t=0} + \dots \tag{1.18}$$

which can be rearranged as

$$\frac{y(\Delta t) - y(0)}{\Delta t} = \left.\frac{dy}{dt}\right|_{t=0} + \frac{\Delta t}{2}\left.\frac{d^2y}{dt^2}\right|_{t=0} + \dots \tag{1.19}$$

which looks like Euler's method presented in the last section with the exception of the extra term on the right hand side. This extra term is the error of using Euler's method. Technically, the error is a series including terms to infinity of higher powers of t (denoted by the repeating dots). We retain only the first under the assumption that Δt is small and therefore, the first error term is dominant. When $\Delta t = 0.1$ then $\Delta t^2 = 0.01$, $\Delta t^3 = 0.001$, and so on. We only need to retain the first neglected term in the Taylor series to understand the error, when Δt is small. Equation 1.18 shows that the error in Euler's method will *scale* with Δt. By the scaling, we mean that if we halve the time step then we halve the error. In later sections we will introduce methods that exhibit better scaling, if we halve the time step then the error might decrease as a higher power of Δt. Knowing the scaling allows one to check that

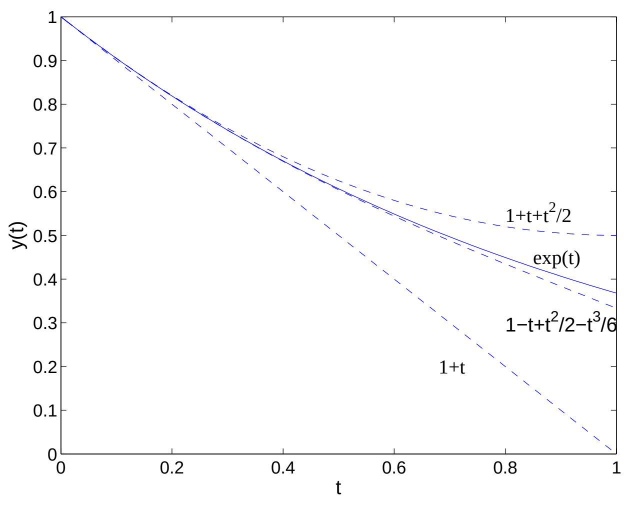

Fig. 1.4 Taylor series approximation of the function e^{-t}. We have included the fi rst three terms in the Taylor expansion. It is clear that the approximation is valid for larger and larger x as more terms are retained.

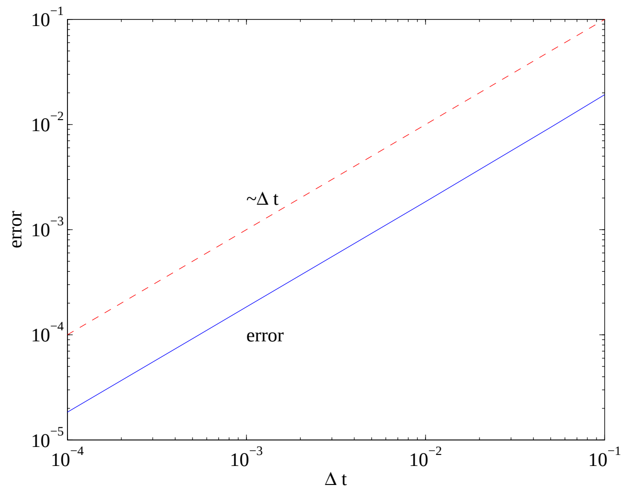

Fig. 1.5 Error of the Euler method as we change the time step size. For comparison we show the linear plot of Δt. Since the slope of the error curve matches the slope of the linear Δt curve we know that the error scales as Δt.

there result is behaving as expected. The knowledge of the scaling is also used in more complex methods that use adaptive control of the time step size to ensure that the solution is within acceptable error bounds.

One way to confirm the scaling of the numerical methods is to plot the error on a log-log plot. In Figure 1.5 we plot the error in applying Euler's method to our model equation as a function of the time step size. We also plot the line, error $= \Delta t$. We find that both lines have the same slope. A straight line on a a log-log plot means that the plot follows a power law, i.e error $\sim \Delta t^n$. The slope of the line provides the power. We find from this plot that our Euler method error is scaling linearly with Δt as the slopes of the two displayed curves match. This graphical result agrees with the prediction of the error using Taylor series.

To provide a little insight into how such graphical results can be easily generated we present the program that created Figure 1.5. The program is a simple modification of Program 1.2. In the program we simply loop the Euler solver for several time step sizes and store the values of the error for plotting. The error is defined as the difference between the true and numerical solutions at the end of the integration period. In this example we integrate until $t = 1$, but the time that we integrate until is irrelevant.

Program 1.3: Program to check error scaling in Euler method. The result is shown in Figure 1.5.

```
clear;         %% clear exisiting workspace
dt = [0.0001  0.0005 0.001 0.005 0.01 0.05 0.1 ];
for j = 1:length(dt)
   y = 1;                            %% initial condition
   time = 0;                         %% set the time=0
   t_final = 1.;                     %% final time
   Nsteps = round(t_final/dt(j));    %% number of steps to take
   for i = 1:Nsteps
      y = y - dt(j)*y;               %% extrapolate one time step
      time = time + dt(j);           %% increment time
   end
   X(j,2) = exp(-t_final) - y;       %% compute the error and store
   X(j,1) = dt(j);                   %% store the time step
end
loglog(X(:,1),X(:,2))        %% display on log-log plot
```

1.1.4 Programming and implementation

In this section we provide a few different ways to create the program that implements Euler's method. Program 1.2 is a perfectly acceptable way to solve Euler's equation. Since we will be interested in applying this same basic method to different sets of equations there is some benefit in breaking the program into functions. While the benefit with something as simple as Euler's method applied to first order systems is not substantial, this programming effort will become worthwhile as we proceed in the next section and deal with systems of equations. We proceed with a demonstration of several programs that are equivalent to Program 1.2. It is recommended to try these programs as you proceed through the chapter, making sure to understand each program before proceeding to the next. Consult the section on programming functions in MATLAB in order to understand the syntax of the programs presented.

One possible program is provided in Program 1.4. This program is makes use of different functions to keep the Euler algorithm separate so that it only needs to be programmed once, and then left alone for all other equations. Program 1.4 has the advantage in that the to solve a different system you do not need to modify lines of code that deal with the Euler method.

The program has the program broken into two functions. The functions can then be called from the command line. For example, the command `euler1storder(1,0.01,1)` means the function will solve the first order equation with an initial condition of 1, a time step size of 0.01, until a time of 1. The first function, `derivs`, contains the governing differential equation we would like to solve. The input to this function is the current value of y and time and the output of the function is the derivative, or the right-hand-side of the differential equation. The function `euler1storder` contains all the code directly related to Euler's method. The benefit of this arrangement is a different equation is easy to solve without the risk of corrupting the Euler method itself. You would never need to modify the code inside the function `euler1storder`.

Program 1.4: Use of functions to generalize the Euler program. All the code should be in one m-fi le named `modeleqns.m`.

```
%% derivative functions
function dy = derivs(time,y)
dy = -y;   %% governing equation

%% Euler Solver
function euler1storder(y,dt,t_final)
clf;
Nsteps = round(t_final/dt);   %% number of steps to take
time = 0;

for i =1:Nsteps
  dy = derivs(time,y);     %% compute the derivatives
  y = y + dy*dt;           %% extrapolate one time step
  time = time+dt;          %% increment time
```

```
    plot(time,y(1),'.');     %% plot current point
    hold on
  end
```

The next evolution of this program is to not embed the plotting into the program, but instead have the solution data returned to the user. This change is helpful as you may not always want to plot the data and you may want to do some further calculations or processing on the final result. Try typing the following program functions into a single m-file. Run the function at the command line as demonstrated.

Program 1.5: Use of functions with function input/output. User is returned the solution to the governing equation. All the code should be in one m-file named `modeleqns.m`.

```
  %% derivative functions
  function dy = derivs(time,y)
  dy =-y;    %% govering equation

  %% Euler Solver
  function [t,data] = euler1storder(y,dt,t_final)
  time = 0;
  Nsteps = round(t_final/dt);      %% number of steps to take
  t = zeros(Nsteps,1);             %% initialize space for return array
  data = zeros(Nsteps,1);          %% initialize space for return array

  for i =1:Nsteps
    dy = derivs(time,y);      %% compute the derivatives
    y = y + dy*dt;            %% extrapolate one time step
    time = time+dt;           %% increment time
    t(i) = time;              %% store data for return
    data(i) = y;
  end
```

To run this program on the command line and plot the result,

```
  » [t,y] = euler1storder(1,0.01,1);
  » plot(t,y);
```

You will notice in program 1.5 that the solution and time data are allocated as zero arrays of the proper size and then filled in with data as Euler's method proceeds through the time steps. We showed in the MATLAB Primer that this allocation saves time, the array does not have to be reallocated in memory with each time step.

The final evolutionary change allows us to make the Euler solver an independent function. In the above example all the functions must exist in the same file. Since the Euler solver is general, it is useful in a separate file so that it need not always be included - just called. The program can be created once, and kept in a directory containing your own set of specially written MATLAB functions. In the MATLAB Primer we discussed the creation of functions and where the must reside to be found by MATLAB. In order to implement the Euler solver as an independent function, we do something that may seem a little unusual; we pass the name of the function containing the derivatives to the Euler solver. We set up the program in this form so that we can create one function containing the Euler solver, then several independent functions containing the systems that we are interested in solving.

Program 1.6.a: Euler solver program that needs to be created once and then may be applied to different first order systems. The function requires the initial condition, time step size, final time, and the handle to the derivative function. This program should be contained in a separate file called `eulersolver.m`.

```
  %% Euler Solver
  %% Place this code in a file called eulersolver.m
  function [t,data] = eulersolver(y,dt,t_final,derivs_Handle)
  time = 0;
  Nsteps = round(t_final/dt);
  t = zeros(Nsteps,1);    %%initialize data array
  data = zeros(Nsteps,1); %%initialize data array

  t(1) = time;     %% store intial condition
  data(1,:)  = y;
  for i =1:Nsteps
         dy = feval(derivs_Handle,time,y);
         y = y + dy*dt;
```

```
        time = time+dt;
        t(i+1) = time;
        data(i+1) = y;
end
```

Program 1.6.b: Form of the derivatives functions. In this context, the derivative function should be contained in a separate fi le named `derivs.m`.

```
%% derivative function
%% place in file derivs.m
function dy = derivs(time,y)
dy =-y;    %% govering equation
```

Program 1.6 can be executed from the command line or inside of another script

```
[t,y] = eulersolver(1,0.01,1,@derivs);
```

This final program is the easiest to modify. If you have a different set of equations, then you only need to write a new derivative function and set the initial conditions and integration parameters in the call to `eulersolver`. Again, we state the advantages of the final program in the case of the simple first order system may not be immediately apparent. As we proceed to systems of equations and more complex methods we will start to see the real advantages of the modularized programming. These same concepts in programming can be dealt with in any programming language, we have chosen MATLAB as our convention herein.

1.2 SECOND ORDER SYSTEMS: MASS-SPRING

Now that we have considered the first order system we will move on to second order systems. The differences between first and second order systems and their basic behaviors were detailed in Chapter 1. The simplest example of a second order system is a mass oscillating on a spring. A mass is attached to fixed spring where gravity is normal to the direction of motion, the spring is pulled back and held at rest, finally the mass is then released and oscillates.

The governing equations for this system can be derived using Newton's law

$$F = ma \tag{1.20}$$

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

$$F = -kx \tag{1.21}$$

where k is called the spring constant and x is the displacement of the spring from the equilibrium state. Equating the above expressions for the force lead to the expression

$$ma = -kx. \tag{1.22}$$

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

$$m\frac{d^2x}{dt^2} = -kx \tag{1.23}$$

The initial conditions for starting our spring system are that the spring is pulled back, held steady, then released. Mathematically, the initial condition is

$$x(t = 0) = x_0$$

and

$$v(t = 0) = 0$$

.

When solving differential equations numerically we usually like to work with systems of equations that involve only first derivatives. This is convenient because the numerical implementation can be generalized to solve any problem, regardless of size and order of the highest derivative. In the above example, the second order system is transformed quite easily using the relationships between velocity, position, and acceleration

$$a = \frac{dv}{dt} \tag{1.24}$$

$$v = \frac{dx}{dt}. \tag{1.25}$$

where v is the velocity. We can easily rewrite equation 1.23 as two equations,

$$\frac{dv}{dt} = -\frac{k}{m}x \tag{1.26}$$

$$\frac{dx}{dt} = v. \tag{1.27}$$

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. We will not detail methods for the analytical solutions of this equation. Your intuition for the system should be that the solution should be oscillatory. In our model there is no mechanism for damping (such as friction), therefore the energy of the system must be conserved. We can easily confirm that the exact solution to this problem is satisfied by

$$x = x_0 cos\left(t\frac{k}{m}\right) \tag{1.28}$$

1.2.1 Implementation of Euler's method for second order systems

From the initial condition and the equations, 1.26 & 1.27, we find that the instant that you release the spring

$$\left.\frac{dv}{dt}\right|_{t=0} = -x_0 \tag{1.29}$$

$$\left.\frac{dx}{dt}\right|_{t=0} = 0. \tag{1.30}$$

These expressions show 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 governing equation.

To solve these equations with Euler's method, we simply apply Euler's method to both equations in our system simultaneously to predict the state of the system a short time in the future.

$$v_1 = v_0 + \Delta t\frac{dv}{dt}_0 \tag{1.31}$$

$$x_1 = x_0 + \Delta t\frac{dx}{dt}_0. \tag{1.32}$$

Substituting equations 1.27 & 1.26 in to the above equations and generalizing beyond the first time step yields

$$v_{N+1} = v_N - \frac{k}{m}x_N\Delta t \tag{1.33}$$

$$x_{N+1} = x_N + v_N\Delta t. \tag{1.34}$$

This subscript N refers to the N^{th} time step. It is assumed that the time step N is known and N+1 is the next unknown time step.

Without loss of generality, in the programs that follow we will simplify the constants in the equations by assuming that $k/m = 1$ and the $x_0 = 1$. In order to write a program that is extendable to larger systems, we will make use of MATLAB's whole array operations. The use of such operations is detailed in the MATLAB Primer. Instead

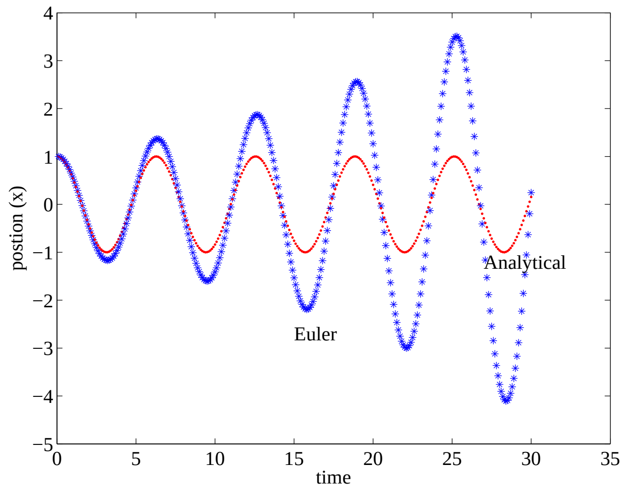

Fig. 1.6 Result of applying Euler's method to the mass spring system as done in Program 1.7. Clearly, the cumulative error is influencing the solution and contaminating the results. Euler's method shows that the amplitude of the oscillations is growing in time while the analytical solution shows that the amplitude is constant.

of creating separate variables for x and v we will store them in a 2 element, one-dimensional array. We write the equations as separate elements of a data array

$$\begin{bmatrix} x \\ \hline v \end{bmatrix}_{N+1} = \begin{bmatrix} x \\ \hline v \end{bmatrix}_{N} + \Delta t \begin{bmatrix} dx/dt \\ \hline dv/dt \end{bmatrix}_{N}. \tag{1.35}$$

When the array y is defined as

$$y = \begin{bmatrix} x \\ \hline v \end{bmatrix} \tag{1.36}$$

the resulting Euler's method becomes

$$y_{N+1} = y_N + \Delta t \frac{dy}{dt}_N \tag{1.37}$$

where the first element of the array y is x and the second element v. Substituting the governing equations into the above system becomes

$$\begin{bmatrix} x \\ \hline v \end{bmatrix}_{N+1} = \begin{bmatrix} x \\ \hline v \end{bmatrix}_{N} + \Delta t \begin{bmatrix} v \\ \hline -x \end{bmatrix}_{N}. \tag{1.38}$$

This array representation will become valuable as we increase the size of the system and have many variables. In this example of second order systems the saving in coding is not significantly reduced by the array storage of the data. However, as we get to later examples this storage will be more and more important.

First we demonstrate a straightforward implementation of Euler's method using the array storage for the data. The program below is similar in structure to Program 1.2, only we have a different system of equations and we make use of the the storage of two variables in the array y. The output of the program below will generate the result shown in Figure 1.6.

Program 1.7: Program to solver the mass spring system using Euler's method and the array storage for the two variables in the second order system.

```
clear;                 %% clear exisiting workspace
y(1) = 1;              %% initial condition, position
y(2) = 0;              %% initial condition, velocity
dt = 0.1;              %% set the time step interval
t_final = 30;          %% final time to integrate to
time = 0;              %% set the time=0
```

```
Nsteps = round(t_final/dt)  %% number of steps to take.

plot(time,y(1),'*');
hold on;              %% accumulate contents of the  figure

for i = 1:NSteps           %% 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),'*');
     plot(time,cos(time),'r.');
end
```

Program 1.7 is nearly the same as Program 1.2, the first program we wrote to implement Euler's method on first order systems. In the program we simply define the initial condition for position and velocity and store these values in the first and second elements of the array y. We then iterate the Euler method as before using the MATLAB whole array operations.

We can follow the same logic as was done in to reach Program 1.6 and create the Euler solver as a general function that can be called from anywhere. The new Euler solver is general in that it can be used on any *system* of first order differential equation.

Program 1.8.a: Euler solver program that needs to be created once and then may be applied to different systems. The function requires the initial condition, time step size, final time, and the handle to the derivative function. The function can handle systems of differential equations of any size. This program should be contained in a separate file called `eulersolver.m`.

```
%% Euler Solver
%% Place this code in a file called eulersolver.m
function [t,data] = eulersolver(y,dt,t_final,derivs_Handle)
time = 0;
Nsteps = round(t_final/dt);  %% number of steps to take.
t = zeros(Nsteps,1);
data = zeros(Nsteps,length(y));
t(1) = time;     %% store intial condition
data(1,:)  = y';
for i =1:Nsteps
       dy = feval(derivs_Handle,time,y);
       y = y + dy*dt;
       time = time+dt;
       t(i+1) = time;
       data(i+1,:)  = y';
end
```

Program 1.6.b: Form of the derivatives functions for the mass-spring system. In this context, the derivative function should be contained in a separate file named `derivs.m`.

```
%% derivative function
%% place in file derivs.m
function dy = derivs(time,y)
dy = zeros(2,1); %% initialize dy array and orient as column
dy(2) =-y(1);    %% dv/dt = -x
dy(1) = y(2);    %% dx/dt = v
```

We can run the above program by typing in at the MATLAB prompt or embedding into another program or script the following commands.

```
[T,Y] = eulersolver([1;0],0.1,3,@derivs);
plot(T,Y(:,1));  %% plot position
plot(T,Y(:,2));  %% plot velocity
```

A final programming tip is to define variables that correspond to the position in the data array, y, for the different variables. This definition will help you keep the equations straight since once the system becomes large it is difficult to remember if y(1) is position or velocity. We can define integers corresponding to the indices into the array so instead of typing $y(1)$ we have $y(X)$. There are other possible solutions, we provide one simple implementation.

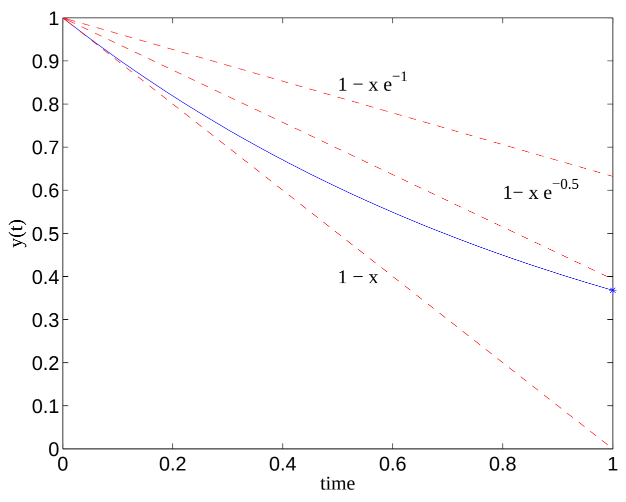

Fig. 1.7 Midpoint approximation for $y(t) = e^-t$. The fi gure shows the function, plus extrapolations using the slope evaluated at the beginning of the interval, the end of the interval, and at the midpoint. The slope at any point on the curve is e^-t.

```
%% derivative function
%% place in file derivs.m
function dy = derivs(time,y)
dy = zeros(2,1); %% initialize dy array and orient as column
X = 1;
V = 2;
dy(V) =-y(X);    %% dv/dt = -x
dy(X) = y(V);    %% dx/dt = v
```

All the programs shown in this section are equivalent in function. The advantage in the way we have developed the final program is that it is easy to change for any system of equations of any size. You now possess a very general program for solving any system of differential equations using Euler's method. The manner in which we implemented these programs is not unique. There are an infinite number of ways to implement such a 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?

1.3 MIDPOINT METHOD

When numerically solving differential equations, what we really want to do is find the best estimate for the average slope across the time step interval. If we want to know how long it will take to drive from Boston to New York, we must know what our average speed is over than interval accounting for driving time on open highway, traffic, getting pulled over by highway patrol, and stops for gas. So far we have used the initial value of the derivative to extrapolate across time step interval since this is the only location where we have any information about the function. We saw the folly in this assumption in the previous section when we found that the oscillations in the mass-spring system grew in amplitude over time. In this section we show a method to obtain a better estimate of the average slope by using the slope of the function at the midpoint of the interval.

Consider figure 1.7 where we have plotted the function $y(t) = e^{-t}$ and various approximations for the derivative to shoot across the interval $0 < t < 1$. We have used the value of the derivative at the beginning, end, and midpoint of the interval. We see that the simple Euler method based on the initial and final slope are quite far off, while the extrapolation based upon the midpoint approximation is much better. The midpoint works better for this specific case, but we can also prove that the midpoint is a better representation of the average slope for the interval. The extrapolation based upon the midpoint slope is given as

$$y(\Delta t) = y(0) + \Delta t \left.\frac{dy}{dt}\right|_{t=\Delta t/2}. \tag{1.39}$$

Expanding the derivative of y with respect to time using a Taylor series and evaluating this approximation at the interval midpoint yields

$$\left.\frac{dy}{dt}\right|_{t=\Delta t/2} = \left.\frac{dy}{dt}\right|_{t=0} + \frac{\Delta t}{2}\left.\frac{d^2y}{dt^2}\right|_{t=0} + \frac{\Delta t^2}{4}\left.\frac{d^3y}{dt^3}\right|_{t=0} + ... \tag{1.40}$$

Substituting this expression into equation 1.39 provides

$$y(\Delta t) = y(t=0) + \Delta t\left(\left.\frac{dy}{dt}\right|_{t=0} + \frac{\Delta t}{2}\left.\frac{d^2y}{dt^2}\right|_{t=0} + \frac{\Delta t^2}{4}\left.\frac{d^3y}{dt^3}\right|_{t=0} + ...\right). \tag{1.41}$$

We see that the first three terms of the right-hand-side exactly match the Taylor series approximation. The error is not introduced until we get the terms of order Δt^3. With the simple Euler's method based on the initial condition the first error term was of order Δt^2. Using the midpoint value as the estimate of the slope for the interval is a better approximation than using the initial value.

The difficulty with using the midpoint in that we only know the state at the start of the interval, the slope at the midpoint is unknown. The difficulty can be remedied with a simple approximation, we will use the Euler method to shoot to an approximated midpoint. We will estimate the midpoint derivative at this location and then use the result to make the complete step from the initial condition. Specifically, the midpoint method works as follows.

$$y_{N+1/2} = y_N + \frac{\Delta t}{2}\left.\frac{dy}{dt}\right|_{N} \tag{1.42}$$

$$y_{N+1} = y_N + \Delta t \left.\frac{dy}{dt}\right|_{N+1/2}. \tag{1.43}$$

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.

To fully illustrate this method we detail one time step for the for the equation $dy/dt = -y$ using a large time step of $\Delta t = 1$ in order to illustrate more clearly the method. 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 the formula

$$y_{N+1/2} = 1 - \frac{\Delta t}{2} \tag{1.44}$$

extrapolates using Euler's method to the midpoint. Via the governing equation we reevaluate the slope at the midpoint to be

$$\left.\frac{dy}{dt}\right|_{N+1/2} = -\left(1 - \frac{\Delta t}{2}\right) \tag{1.45}$$

Using this slope we then shoot all the way across the interval

$$y_{N+1} = 1 + \Delta t\left(1 + \frac{\Delta t}{2}\right) \tag{1.46}$$

The schematic for these step is shown in Figure 1.8 using $\Delta t = 1$.

Now we will write a MATLAB function similar to the Euler solver that applies the midpoint algorithm. The program should be general so that you can apply it to any system of equations. The program should also follow the same inputs and outputs as the Euler solver so that in your programs you could easily switch between methods. The program should also expect the same format for the derivative functions such that the same derivatives can be used in the midpoint or Euler solver.

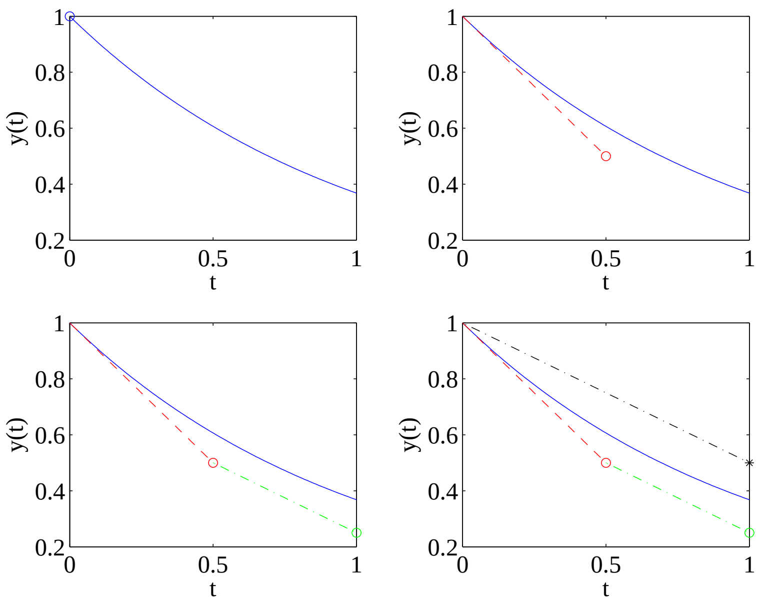

Fig. 1.8 Example of the midpoint method for $y(t) = e^{-t}$, where $\Delta t = 1$. The steps are shown in the four fi gures, going left to right, then down. The fi rst image shows the exact solution. At this time we only know the initial condition. In the second image we use the Euler method to extrapolate to the midpoint (dashed line). In the third image we fi nd the slope as if we were going to continue with Euler's method using the half step size. Finally, in the fourth frame we use the midpoint slope to shoot across the interval.

Program 1.9: General midpoint solver that has the same usage as the the Euler solver created earlier. This program should be created and placed in a fi le called `midpointsolver.m`.

```
%% Midpoint Solver
function [t,data] = midpointsolver(y,dt,t_final,derivs_Handle)
time = 0;
Nsteps = round(t_final/dt)  %% number of steps to take
t = zeros(Nsteps,1);
data = zeros(Nsteps,length(y));

t(1) = time;     %% store intial condition
data(1,:)  = y';
for i =1:Nsteps
  dy  = feval(derivs_Handle,time,y);   %% evaluate the initial derivatives
  yH  = y + dy*dt/2;                    %% take Euler step to midpoint
  dy  = feval(derivs_Handle,time,y);   %% re-evaluate the derivs
  y   = y + dy*dt;                     %% shoot across the interval
  time = time+dt;                      %% increment time
  t(i+1) = time;                         %% store for output
  data(i+1,:)  = y';                     %% store for output
end
```

We can now test this program by using the same derivative function and construct as used in Program 1.6. Create the function that computes the derivatives for the spring equations and rename the file `derivs_spring`. We will start naming the derivative files by more descriptive names as we will start accumulating more functions. We can compare the midpoint solver to the Euler solver by typing in at the MATLAB prompt or embedding into another program or script the following commands.

```
[T,Y] = eulersolver([1;0],0.1,3,@derivs_spring);
plot(T,Y(:,1),'r--');
[T,Y] = midpointsolver([1;0],0.1,3,@derivs_spring);
plot(T,Y(:,1),'b');
```

The result of this exercise is shown in Figure 1.9.

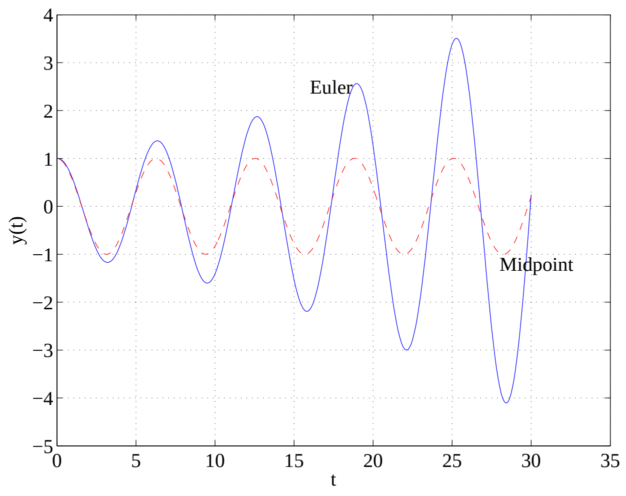

Fig. 1.9 Comparison of the solution of the mass spring system using the midpoint method and Euler's method. It is clear that the midpoint method is far superior in this case.

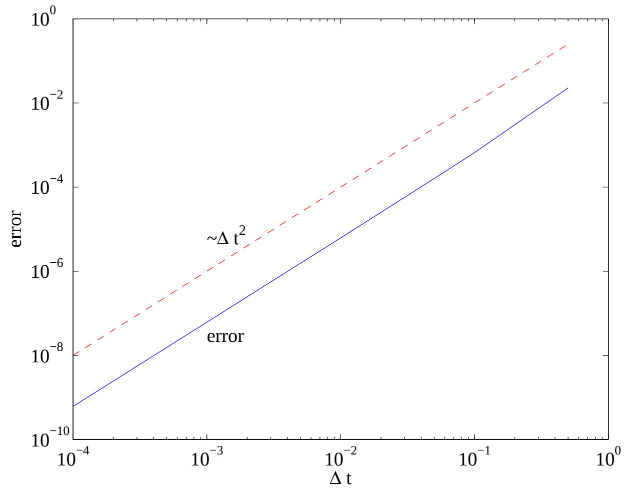

Fig. 1.10 Scaling of the error of the midpoint method as applied to the model problem $dy/dt = -y$. We fi nd that the midpoint algorithm scales as Δt^2.

Finally, we close the section on the midpoint method by evaluating the error in the midpoint method. We create a derivatives function called `derivs1st` that represents the first order equation that was the focus of the first few sections of this chapter.

```
function dy = derivs1st(time,y)
dy =  -y;
```

We can now use either the midpoint solver or the Euler solver to evaluate this equation. To test the error we solve the equation until $t = 1$ with the midpoint solver and compute the difference with the exact solution at that time. We then repeat this test at several different time step sizes. The result is shown in Figure 1.10 and the generating program is given by Program 1.9. The figure shows on the plot the slope of the line Δt^2, showing that the midpoint method scales as Δt^2. This scaling means that if we halve the time step then we bring the error down by a factor of 4.

Program 1.9: Matlab script to compute the error in the midpoint solver on the equation $dy/dt = -y$. The error is defined as the difference to the exact solution. The program runs the midpoint routine for several different time step sizes. The error is plotted on a log-log scale.

```
dt = [ 0.0001 0.0005 0.001 0.005 0.01 0.05 0.1 0.5];
t_final = 1;

for j = 1:length(dt)
   [t,y] = midpointsolver(1,dt(j),t_final,@derivs1st);
   X(j,2) = abs(y(end,1) - exp(-1));
   X(j,1) = dt(j);
end
loglog(X(:,1),X(:,2))
hold on
loglog(X(:,1),X(:,1).^2,'r--')
```

1.4 RUNGE-KUTTA METHOD

There are many different schemes for solving ODEs numerically. We have introduced two simple schemes to introduce some basic basic concepts and provide you some examples for programming. Many of the more advanced techniques are more complex to derive, analyze, or program but all schemes are based on the ideas we have introduced. 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, in a sense takes, four steps, shooting across one quarter of the interval, estimating the derivative, then shooting to the midpoint, and so on. The precise manner in which the method propagates across a time step is done in the optimal way for the four steps. We will not provide a formal derivation of the Runge-Kutta algorithm, instead we will present the method and implement it.

The general system of ODEs can be written as,

$$\frac{dy}{dt} = f(y,t). \tag{1.47}$$

The Runge-Kutta method is defined as:

$$k1 = \Delta t f(t^N, y^N) \tag{1.48}$$

$$k2 = \Delta t f(t^N + \Delta t/2, y^N + k1/2) \tag{1.49}$$

$$k3 = \Delta t f(t^N + \Delta t/2, y^N + k2/2) \tag{1.50}$$

$$k4 = \Delta t f(t^N + \Delta t, y^N + k3) \tag{1.51}$$

$$y^{N+1} = y^N + \frac{k1}{6} + \frac{k2}{3} + \frac{k3}{3} + \frac{k4}{6} \tag{1.52}$$

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. The programming of this method will follow the format used already for the midpoint and Euler methods and is provided in the program below.

Program 1.10: General Runge-Kutta solver that has the same usage as the the Euler and midpoint solvers created earlier. This program should be created and placed in a file called `rksolver.m`.

```
%% Runge-Kutta Solver
function [t,data] = rksolver(y,dt,t_final,derivs_Handle)
time = 0;
Nsteps = round(t_final/dt)  %% number of steps to take.
t = zeros(Nsteps,1);
data = zeros(Nsteps,length(y));

t(1) = time;     %% store intial condition
data(1,:)  = y';
for i =1:Nsteps
    k1 = dt*feval(derivs_Handle,time     ,y     );
    k2 = dt*feval(derivs_Handle,time+dt/2,y+k1/2);
```

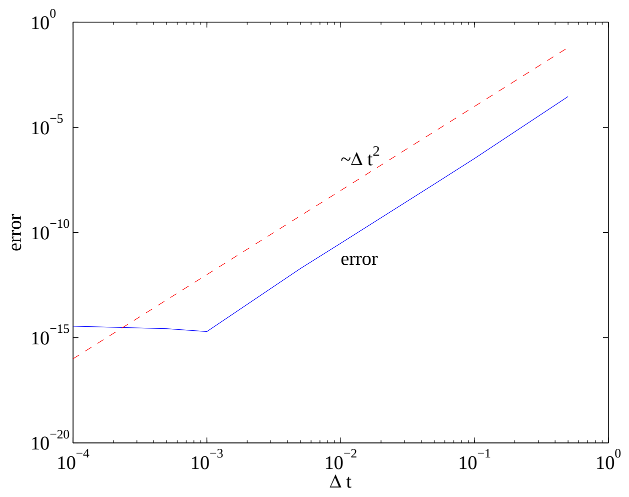

Fig. 1.11 The error between the Runge-Kutta method and exact solution as a function of time step size. One the plot we also display a function that scales as Δt^4. We see that this fi ts the slop of the data quite well, therefore error in the Runge-Kutta approximation scales as Δt^4. Once the error reaches 10^{-15} then the error is dominated by the resolution of double precision numbers.

```
        k3 = dt*feval(derivs_Handle,time+dt/2,y+k2/2);
        k4 = dt*feval(derivs_Handle,time+dt  ,y+k3  );
        y = y + k1/6 + k2/3 + k3/3 + k4/6;
        time = time+dt;
        t(i+1) = time;
        data(i+1,:)  = y';
    end
```

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 $t = 1$. We have conducted this same test with the midpoint and Euler solvers. In Figure 1.11 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 Δt^4 on the same graph to show that the error of the Runge-Kutta method scales as Δt^4. This is quite good - if we halve the time step size we reduce the error by 16 times. To generate this figure only requires minor modification to Program 1.9. The minimum error of 10^{-15} is due to the accuracy of representing real numbers with a finite number of digits.

1.5 BUILT-IN MATLAB SOLVERS

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 use the Runge-Kutta method as the basic workhorse. 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 and debugging sophisticated techniques that work really well, why should we replicate all their work? We turn to these routines at this time.

Just as you have developed three solvers that have the same functionality, MATLAB has employed several different algorithms for solving differential equations. The most common of the MATLAB solvers is `ode45`; 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. With these algorithms, the user specifies the amount of acceptable error and the algorithm adjusts the time step size to maintain this value constant. Therefore, with adaptive algorithms you cannot generate a plot of error vs. time step size. The algorithm does have the flexibility to force a fixed 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 this error constant.

An example of the usage of MATLAB's `ode45` command is illustrated in the commands below. We show the same example of the mass on the spring as in previous sections. Below we can see that the usage of MATLAB the `ode45` command is quite similar to the methods we developed in previous sections. The command `odeset` allows the user to control all the options for the solver including method, maximum time step size, acceptable error tolerances, and output format options. The reader should consult the MATLAB help functions to discover the different options with the `odeset` command.

```
options = odeset('AbsTol',1e-9);            %% set solver options
[t,y]   = ode45(@derivs_spring,[0 30],1,options);   %% solve equations
plot(t,y);  %% plot position
```

Notice that the assumption of the `ode45` command is that the function that supplies the derivatives has the form `dy_dt = derivativefunction(t,y)`. We have assumed the same format for the derivative functions throughout this chapter. You may use the same derivative functions with the routines that you have written as well as the MATLAB solvers. Besides `ode45`, MATLAB has several other solvers that are designed for different types of equations. There are also a variety of plotting and display functions that accompany the differential equation solvers. The functionality of MATLAB is well documented on their web-page and we leave it to the student to explore the different functions.

1.6 CHECKING THE SOLUTION

One of the common 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. 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 1.6. 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 with the result.

One simple test of a numerical method is to change the time step and see what happens. If you have implemented the method correctly the answer should converge as the time step is decreased. If you know the order of your approximation then you know how fast this convergence 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 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.

Finally, we note that most of the examples that we cover in this class are easily tackled with with the tools presented in this chapter. This does not mean that there are not systems where the details of the numerical method can contaminate the results. However, the algorithms included with MATLAB are very robust and work well in many applications.

1.7 EXAMPLES

In this final section we will introduce some systems that have larger equation sets and exhibit some non-linear behaviors. These examples are meant to provide further guidance to the student on implementing numerical methods for a variety of problems. This section is also meant to look at systems that have interesting and complex behavior that is not tractable via pure mathematical analysis. However, these results coupled with more advanced analysis techniques can uncover even more unusual system behavior.

1.7.1 Lorentz Attractor

Lorentz proposed a system of differential equations as a simple model of atmospheric convection and hoped to use his equations to aid in weather prediction. The details of the derivation of the model are beyond the scope of this course, so we will have to take the equations for granted. Since the resulting equations were very complex, Lorentz solved his equations numerically. Computers were slow at this time and one day, rather than re-run a particular calculation from time=0, he used the data written out by the program from a previous day at an intermediate time. He noticed that he when he solved his equations, he got completely different answers if he started the solution from the beginning or stopped halfway and restarted. Lorentz tracked the difference down to the fact that when he typed in the restart conditions, he only was using the first few significant digits. He soon discovered that the system was very sensitive to the initial condition. For only a small change in initial condition the solution to the equations significantly diverged over time. Further, he also noticed that while the variables plotted as a function of time seemed random, the variables plotted against each other showed regular and interesting patterns. We will explore his system using the numerical solvers that we have developed.

The system of equations that Lorentz developed with were

$$\frac{dx}{dt} = 10(y - x) \tag{1.53}$$

$$\frac{dy}{dt} = x(27 - z) - y \tag{1.54}$$

$$\frac{dz}{dt} = xy - \frac{8}{3}z \tag{1.55}$$

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, 27, and 8/3 represent properties of the atmospheric system. The constants are not universal and the system will behave differently for different constants. For the purposed of this section we will take these numbers as a given. In order to solve this system of equations, we need only implement the derivatives into a function file and then we may use the `ode45` command or the solvers that we have written to generate the solution.

Program 1.11: Function to compute the derivatives of the Lorentz equations.

```
function dy = lorentz(time,y)
dy = zeros(3,1);
X = 1;
Y = 2;
Z = 3;
dy(X) = 10*(y(Y)-y(X));
dy(Y) = y(X)*(27-y(Z)) -y(Y);
dy(Z) = y(X)*y(Y) - 8/3*y(Z);
```

Some interesting results are shown in Figure 1.12. In this figure we demonstrate several interesting features of the Lorentz equations. The first concept is that of sensitivity to initial conditions. The first image shows the time history of two initial conditions that differ by only 1%. The two systems evolve identically for some time then diverge. Such a basic result is one reason why detailed weather prediction is difficult more than a few days out. Regardless of the quality of the model, simply not knowing the precise condition to start the model (i.e. today's weather) means that details cannot be predicted far in the future. The next plot shows the evolution using different solvers, midpoint and Runge-Kutta. The same reason the equations are sensitive to the initial condition makes them sensitive to the details of the numerical method. Just like the small differences in the initial condition caused a very

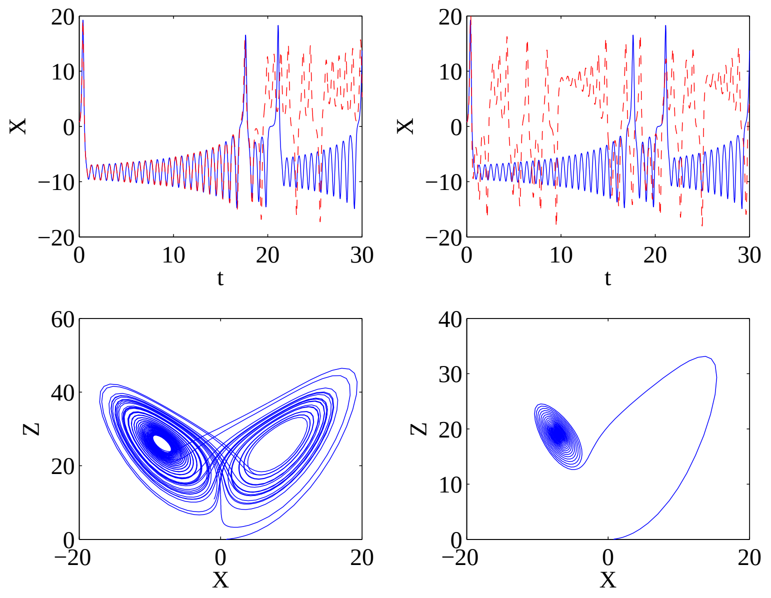

Fig. 1.12 Various results generated from the Lorenz equations. The figures go left to right, then down and are as follows: i) shows the time history of the variable x for the initial condition [1,0,0] (solid line) and [1,0.01,0.01] (dashed line); we see the sensitivity to the initial condition; ii) shows the time history of x for the same initial condition but for the Runge-Kutta (solid) and midpoint (dashed) solvers; iii) shows the plot of x vs. z, even though the variables have a random looking time history the solution predictably lies somewhere on the "butterfly"; iv) the same plot as iii, only the parameter 27 was changed to 20 in the equation for dy/dt.

different evolution, errors in the numerical method can cause too numerical solutions to be very different. The next plot shows the plot of x versus z. Even though the time history will follow a somewhat random and unpredictable pattern, predictably any current state will always fall somewhere on the butterfly. One can think that this plot shows that even though the detailed state of the system is unknown, there is some predictability and pattern to the behavior. Finally, we change the parameter 27 to 20 in the equation for dy/dt and see that the system behaves quite differently. The system is not only sensitive to the initial condition but is sensitive to the parameters in the equation. When the parameter is 27 the system oscillates around the "butterfly" forever. When the parameter is 20 the system is drawn to a steady state. This system has been much discussed in the literature and many detailed results and analysis exist therein. These results were meant to give the student a taste of complex behavior and interesting systems that can be easily approached using numerical methods.

1.7.2 Forced Pendulum

A simple pendulum can be an extremely rich non-linear system. Imagine a heavy mass, m on the end of rigid and light rod of length L. The other end of the rod is connected to a small motor which supply a torque, τ. We drive the motor in a sinusoidal fashion and we are able to control the torque and frequency, ω. Gravity, g, acts downward and the angle of the pendulum, θ, is considered zero when at rest. Friction in the motor and bearings supplies a torque proportional to the angular velocity with a coefficient, β.

Applying Newton's laws we obtained the force balance as

$$mL\frac{d^2\theta}{dt^2} = -mg\sin(\theta) - \beta\frac{d\theta}{dt} + \tau\sin(\omega t) \tag{1.56}$$

which can be rearranged to read

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\sin(\theta) - \frac{\beta}{mL}\frac{d\theta}{dt} + \frac{\tau}{mL}\sin(\omega t). \tag{1.57}$$

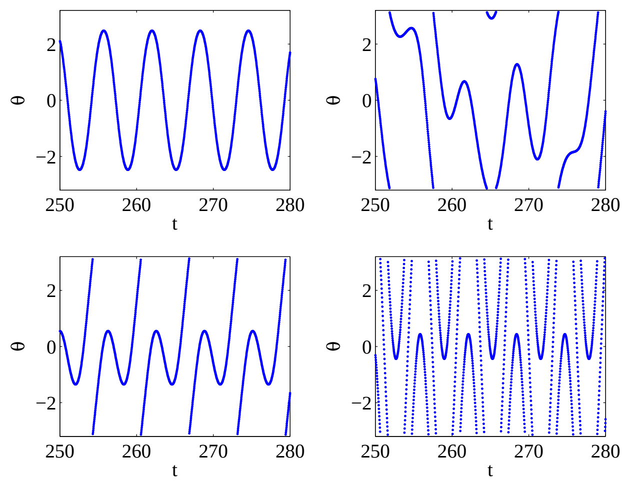

Fig. 1.13 Evolution of θ with time for different forcing amplitudes. Going left to right then down, the amplitude is 1.6, 1.61, 2, and 7. We see in the fi rst two images that the pendulum is swinging at a steady state. A little more energy and the pendulum swings over the top.

A further simplification arises if we redefine a scaled time that has no units and is scaled by the natural frequency. The new time is defined as

$$\hat{t} = t\sqrt{g/L}. \tag{1.58}$$

Making this substitution yields

$$\frac{d^2\theta}{d\hat{t}^2} = -\sin(\theta) - \frac{\beta}{m\sqrt{gL}}\frac{d\theta}{d\hat{t}} + \frac{\tau}{mg}\sin(\omega\hat{t}\sqrt{g/L}). \tag{1.59}$$

We choose to force the pendulum at its natural frequency (much like a child on a swing would like to do) and the damping parameter is chosen to be 0.2. We can now study a variety of interesting system behaviors.

Program 1.11: Function to compute the derivatives of the pendulum equations.

```
function dy = pend(time,y)
dy = zeros(2,1);
dy(1) =  y(2);                                %%% dtheta/dt = omega
dy(2) = -0.2*y(2) - sin(y(1)) + 2*sin(time);   %%% d omega/dt = -v theta
```

In Figure 1.13 we show the non-linear evolution of the pendulum system. In the four images, we show the evolution of θ for different forcing amplitudes. In the first two images we find that a small increase in the amount of forcing energy takes the pendulum from a steady oscillation between ± 140 degrees to a more random oscillation that swings over the top. The non-linear behavior as the pendulum swings at high amplitudes allows this sudden transition rather than an steadily increasing steady oscillation amplitude to $\pm 180^o$.

Another way to represent the complex, non-linear behavior of this system is to ask the question, in the high forcing case does the system first over-rotate going clockwise or counter-clockwise? We span the space of initial conditions in both angle and angular velocity and plot a black pixel for clockwise and and white pixel for counter-clockwise. The result of this exercise is shown in Figure 1.13. We observe very complex structure and find that the system is quite sensitive to the initial condition. For such a simple system it is very difficult to make this precise prediction in a real physical system.

Problems

1.1 The numerical derivative is only an approximation, in this problem we will explore the error in that approximation. When the true solution and the numerical solution are known, one definition of the error is the relative

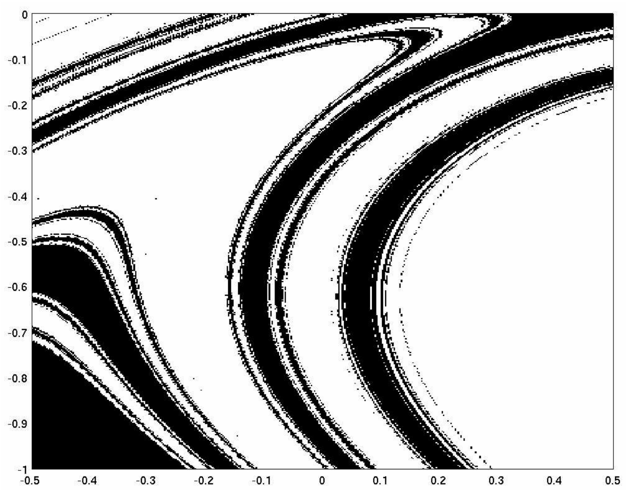

Fig. 1.14 Map of the direction that the pendulum fi rst rotates past $\theta = \pi$. The black pixels are for clockwise rotation and the white pixels for counter-clockwise. The x and y axis are the initial condition space, θ and ω respectively. The pendulum has a forcing amplitude of 3. The interesting features is that one can zoom in even further and fi nd fi ner structure and features of this plot.

difference between the true and approximate derivative:

$$\text{error} = \frac{y_{exact} - y_{numerical}}{y_{exact}}$$

Modify Program 1 to compute the error between the true and numerical derivative and plot the error as a function of time. Change the number of time samples from 20 to 10, and rerun your error program. What happened to the error? Try changing the number of samples to 40. Predict what will happen if you change the number of samples to 200. Re-run your program and see if you are correct.

1.2 Modify program 1 to plot the numerical derivative and analytical derivative of the following functions on the interval $0 < t < 5$. Compare the analytical and the numerical result by plotting both functions on the same plot: $y = t^3 - t$, $y = e^{-5t}$, $y = log(t)$.

1.3 Implement Euler's method (Program 2) for the equation $dy/dy = -y$ with $y(0) = 1$. Verify Figure 1.3.

1.4 Adjust the size of Δt to observe that the exact and numerical solution become closer to each other as you make the time step smaller and take more time steps. Define the error as the absolute difference between the analytical and numerical solutions at $t = 2$. Using the programs created above, compute the error at $t = 2$. Check the error for $\Delta t = 0.5$, $\Delta t = 0.25$, and $\Delta t = 0.125$. What happens to the error when you change the time step size by a factor of 2?

1.5 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 $0 < x < \pi$.

1.6 Consider the differential equation

$$\frac{dy}{dt} = sin(y)$$

with the initial condition $y = 1$. Using a time step of $\delta t = 0.1$ what is numerical value of the error after one time step with Euler's method.

1.7 Implement Programs 1.4-1.6 and make sure that they can all provide the same results as presented in this chapter. Work with each program to understand each step. Modify the program as you wish to fit your own programming style.

1.8 Implement the midpoint solver as shown in Program 1.9 and verify that your program is working correctly. Reproduce Figure 1.9.

1.9 Damping due to friction provides a force proportional to velocity by some constant, β. Add friction to the differential equations for the mass spring system. Plot the solution for $\beta = [0.01, 0.1, 10]$ on the same graph. Solve using the Midpoint method and Euler's method.

1.10 Implement the Runge-Kutta solver as shown in Program 1.9 and verify that it is working correctly by solving the basic first order system and the mass-spring system. Reproduce Figure 1.9 showing the comparison of Euler's method, the midpoint method, and Runge-Kutta.