8. Solving ODEs in MATLAB

This is one of the most important chapters in this manual. Here we will describe how to use MATLAB's built-in numerical solvers to find approximate solutions to almost any system of differential equations. At this point readers might be thinking, "I've got `dfield6` and `pplane6` and I'm all set. I don't need to know anything further about numerical solvers." Unfortunately, this would be far from the truth. For example, how would we handle a system modeling a driven, damped oscillator such as

$$\begin{aligned} x_1' &= x_2, \\ x_2' &= -2x_1 - 3x_2 + \cos t. \end{aligned} \tag{8.1}$$

System (8.1) is **not autonomous**! `Pplane6` can only handle autonomous systems. Furthermore, how would we handle a system such as

$$\begin{aligned} x_1' &= x_1 + 2x_2, \\ x_2' &= -x_3 - x_4, \\ x_3' &= x_1 + x_3 + x_4, \\ x_4' &= -2x_2 - x_4. \end{aligned} \tag{8.2}$$

The right-hand sides of system (8.2) do not explicitly involve the independent variable t, so the system is autonomous. However, system (8.2) is **not planar**! `Pplane6` can only handle autonomous systems of two first order differential equations (planar systems).

Finally, in addition to handling systems of differential equations that are neither planar nor autonomous, we have to develop some sense of trust in the output of numerical solvers. How accurate are they?

MATLAB's ODE Suite

A new suite of ODE solvers was introduced with version 5 of MATLAB. The suite now contains the seven solvers `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, and `ode23tb`. We will spend most of our time discussing the general purpose solver `ode45`. We will also briefly discuss `ode15s`, an excellent solver for stiff systems. Although we will not discuss the other solvers, it is important to realize that the calling syntax is the same for each solver in MATLAB's suite. In the next chapter, we will discuss the MATLAB function `odesolve`, which puts a graphical user interface around the MATLAB solvers.

In addition to being very powerful, MATLAB's numerical solvers are easy to use, as you will soon discover. The methods described herein are used regularly by engineers and scientists, and are available in any version of MATLAB. The student who learns the techniques described here will find them useful in many later circumstances.

The `ode45` solver uses a variable step Runge-Kutta procedure. Six derivative evaluations are used to calculate an approximation of order five, and then another of order four. These are compared to come up with an estimate of the error being made at the current step. It is required that this estimated error at any step should be less than a predetermined amount. This predetermined amount can be changed using two tolerance parameters, and a very high degree of accuracy can be required of the solver. We will discuss this later in this chapter. For most of the uses in this Manual the default values of the tolerance parameters will provide sufficient accuracy, and we will first discuss the use of the solver with its default settings.

Single First Order Differential Equations

We are looking at an initial value problem of the form $x' = f(t, x)$, with $x(t_0) = x_0$. The calling syntax for using `ode45` to find an approximate solution is `ode45(odefcn,tspan,x0)`, where `odefcn` calls for a functional evaluation of $f(t, x)$, `tspan=[t0,tfinal]` is a vector containing the initial and final times, and `x0` is the x-value of the initial condition.

Example 1. *Use* `ode45` *to plot the solution of the initial value problem*

$$x' = \frac{\cos t}{2x - 2}, \quad x(0) = 3, \tag{8.3}$$

on the interval $[0, 2\pi]$.

Note that equation (8.3) is in normal form $x' = f(t, x)$, where $f(t, x) = \cos t/(2x - 2)$. From the data we have `tspan = [0,2*pi]` and `x0 = 3`. We need to encode the `odefcn`. Open your editor and create an ODE function M-file with the contents

```
function xprime = ch8examp1(t,x)
xprime = cos(t)/(2*x - 2);
```

Save the file as `ch8examp1.m`. Notice that $f(0, 2) = \cos 0/(2 \cdot 2 - 2) = 1/2$ and

```
>> ch8examp1(0,2)
ans =
   0.5000
```

so the M-file gives the correct answer. If your output does not match ours, check your function code for errors. If you still have difficulty, revisit Chapter 4, especially the discussion of the MATLAB path.

You can get a fast plot of the solution by executing `ode45(@ch8examp1,[0,2*pi],3)`.[1] Notice that we did not specify any outputs. This causes `ode45` to plot the solution. Usually you will want the output in order to analyze it and treat it in your own way. The following code should produce an image similar to that shown in Figure 8.1.

```
>> [t,x] = ode45(@ch8examp1,[0,2*pi],3);
>> plot(t,x)
>> title('The solution of x''=cos(t)/(2x - 2), with x(0) = 3.')
>> xlabel('t'), ylabel('x'), grid
```

The output of the `ode45` command consists of a column vector `t`, containing the t-values at which the solution was computed, and a column vector `x` containing the computed x-values.[2]

[1] If you are using a version of MATLAB prior to version 6.0, the calling sequence should be `ode45('ch8examp1',[0,2*pi],3)`. We discussed this issue in Chapter 4. There are other differences in the syntax to be used with MATLAB's ODE solvers between version 6 and prior versions. We will emphasize the version 6 usage. For systems without parameters, users of older versions of MATLAB need only replace the function handles by the function names between single quotes. As further differences arise, we will explain them, but if you are using an older version, you should execute `help ode45` to be sure of the correct syntax.

[2] To see the output from the `ode45` routine execute `[t,x]`.

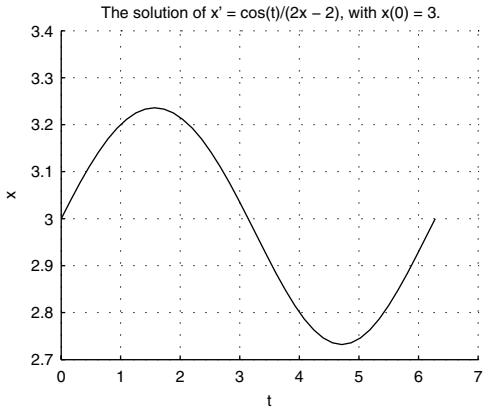

Figure 8.1. The solution of the initial value problem in Example 1.

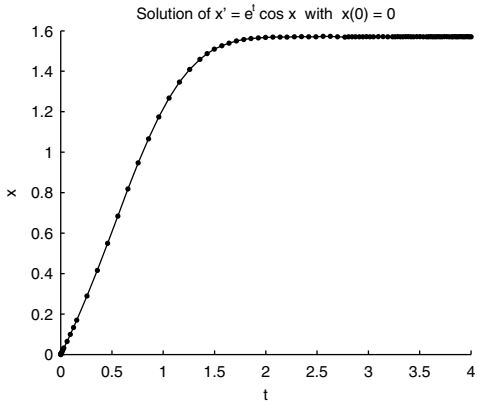

Figure 8.2. The solution of the initial value problem in Example 2.

Using inline functions. If you do not want to save your work, the easiest way to encode the needed `odefcn` is to create the inline function

```
>> f = inline('cos(t)/(2*x - 2)','t','x')
f =
     Inline function:
     f(t,x) = cos(t)/(2*x - 2)
```

We notice that $f(0, 2) = 1/2$, and

```
>> f(0,2)
ans =
    0.5000
```

so it seems to be working correctly. The command `ode45(f,[0,2*pi],3)` [3] will plot the solution in a figure window, with the computed points plotted as circles. Try it yourself. If you want to save the computed data, execute `[t,x] = ode45(f,[0,2*pi],3);`.

Function M-File Drivers and Subfunctions[4]

In Chapter 4 we explained how to use function M-files to ease the process of doing computations and creating reproducible graphics. This technique is especially useful when using ODE solvers, as the next example will illustrate.

Example 2. *Use* `ode45` *to plot the solution of the initial value problem*

$$\frac{dx}{dt} = e^t \cos x, \quad x(0) = 0, \tag{8.4}$$

[3] Notice that we use just `f` when we are using an inline function. Function handles and single quotes are not needed.

[4] Subfunctions were introduced in version 6 of MATLAB and are not available in earlier versions.

on the interval [0, 4]. *Plot the computed points in a distinctive marker style as well as the curve connecting them.*

We will put all of the commands into a function M-file, with the `odefcn` as a subfunction. We will also plot the computed points with dots to make them visible. Create the file

```
function ch8examp2
close all
[t,x] = ode45(@dfile,[0,4],0);
plot(t,x,'.-')
xlabel('t'), ylabel('x')
title('Solution of x'' = e^t cos x  with  x(0) = 0')

function xprime = dfile(t,x)
xprime = exp(t)*cos(x);
```

and save it as `ch8examp2.m`. Executing `ch8examp2` will create Figure 8.2. Notice in Figure 8.2 that the computed points start out close to each other and then spread out for a while before becoming really close to each other as t gets larger than 3. This illustrates the fact that `ode45` is a variable step solver.

The file `ch8examp2` is a function M-file which serves as a master routine or driver, with the ODE function defined as a subfunction. Subfunctions are available only within the function where they are defined. Notice that `ch8examp2` cannot be a script M-file, since subfunctions are allowed only in function M-files.

There are several advantages to using subfunctions inside master functions in this way. Once you have written the master file for a particular purpose, it is easy to reproduce the result. It puts everything involved with a problem in one file. This file can be given a suggestive name, as we did here, to make it easy to interpret later. With all of the commands in one file, it is relatively easy to make changes and correct errors. Furthermore, once you have written one such file, additional ones might well be simple modifications of the first. Frequently, all that is needed is to change the values of the various properties. This applies to the ODE subfunction as well. There is really no need to give this subfunction a distinctive name. Call it `dfile`, or just `f`. Since the subfunction is available only within the master function it does not matter if it has the same name in every such file. Finally, notice that it is not really more work to produce the file than to to execute the original commands.

Systems of First Order Equations

Actually, systems are no harder to handle using `ode45` than are single equations. Consider the following system of n first order differential equations:

$$\begin{aligned} x_1' &= f_1(t, x_1, x_2, \ldots, x_n), \\ x_2' &= f_2(t, x_1, x_2, \ldots, x_n), \\ &\vdots \\ x_n' &= f_n(t, x_1, x_2, \ldots, x_n). \end{aligned} \tag{8.5}$$

System (8.5) can be written in the vector form

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}' = \begin{bmatrix} x_1' \\ x_2' \\ \vdots \\ x_n' \end{bmatrix} = \begin{bmatrix} f_1(t, [x_1, x_2, \ldots, x_n]^T) \\ f_2(t, [x_1, x_2, \ldots, x_n]^T) \\ \vdots \\ f_n(t, [x_1, x_2, \ldots, x_n]^T) \end{bmatrix} \tag{8.6}$$

If we define the vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$, system (8.6) becomes

$$\mathbf{x}' = \begin{bmatrix} f_1(t, \mathbf{x}) \\ f_2(t, \mathbf{x}) \\ \vdots \\ f_n(t, \mathbf{x}) \end{bmatrix}. \tag{8.7}$$

Finally, if we define $\mathbf{F}(t, \mathbf{x}) = [f_1(t, \mathbf{x}), f_2(t, \mathbf{x}), \ldots, f_n(t, \mathbf{x})]^T$, then system (8.7) can be written as

$$\mathbf{x}' = \mathbf{F}(t, \mathbf{x}), \tag{8.8}$$

which, most importantly, has a form identical to the single first order differential equation, $x' = f(t, x)$, used in Example 1. Consequently, if `function xprime=F(t,x)` is the first line of a function ODE file, it is extremely important that you understand that `x` is a vector[5] with entries `x(1)`, `x(2)`,..., `x(n)`. Confused? Perhaps a few examples will clear the waters.

Example 3. *Use* `ode45` *to solve the initial value problem*

$$\begin{aligned} x_1' &= x_2 - x_1^2, \\ x_2' &= -x_1 - 2x_1x_2, \end{aligned} \tag{8.9}$$

on the interval $[0, 10]$*, with initial conditions* $x_1(0) = 0$ *and* $x_2(0) = 1$.

We can write system (8.9) as the vector equation

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}' = \begin{bmatrix} x_2 - x_1^2 \\ -x_1 - 2x_1x_2 \end{bmatrix}. \tag{8.10}$$

If we set $\mathbf{F}(t, [x_1, x_2]^T) = \left[x_2 - x_1^2, -x_1 - 2x_1x_2\right]^T$ and $\mathbf{x} = [x_1, x_2]^T$, then system (8.10) takes the form $\mathbf{x}' = \mathbf{F}(t, \mathbf{x})$. The key thing to realize is the fact that $\mathbf{x} = [x_1, x_2]^T$ is a vector with two components, x_1 and x_2. Similarly, $\mathbf{x}' = [x_1', x_2']^T$ is also a vector with two components, x_1' and x_2'.

Open your editor and create the following ODE file[6].

```
function xprime = F(t,x)
xprime = zeros(2,1);    %The output must be a column vector
xprime(1) = x(2) - x(1)^2;
xprime(2) = -x(1) - 2*x(1)*x(2);
```

[5] In MATLAB, if `x=[2;4;6;8]`, then `x(1)=2`, `x(2)=4`, `x(3)=6`, and `x(4)=8`.

[6] Even though this system is autonomous, the ODE suite does not permit you to write the first line of the ODE file without the `t` variable as in `function xprime=F(x)`. If you did, you would receive lots of error messages.

Save the file as `F.m`. The line `xprime = zeros(2,1)` *initializes* `xprime`, creating a *column* vector with two rows and 1 column. Each of its entries is a zero. Note that the next two lines of the ODE file duplicate exactly what you see in system (8.9).

Even though we write $\mathbf{x} = [x_1, x_2]^T$ in the narrative, it is extremely important that you understand that $\mathbf{x}$ is a *column* vector; that is,

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}.$$

Therefore, the variable `x` in the ODE file is also a *column* vector.

As always, it is an excellent idea to test that your function ODE file is working properly before continuing. Because $\mathbf{F}(t, [x_1, x_2]^T) = [x_2 - x_1^2, -x_1 - 2x_1x_2]^T$, we have that $\mathbf{F}(2, [3, 4]^T) = [4 - 3^2, -3 - 2(3)(4)]^T = [-5, -27]^T$. Test this result at the MATLAB prompt.

```
>> F(2,[3;4])
ans  =
    -5
   -27
```

Notice that the function `F(t,x)` expects two inputs, a scalar `t` and a column vector `x`. We used `t = 2` and `x = [3;4]`. Because this system is autonomous and independent of t, the command `F(5,[3;4])` should also produce `[-5;-27]`. Try it!

Before we call `ode45`, we must establish the initial condition. Recall that the initial conditions for system (8.9) were given as $x_1(0) = 0$ and $x_2(0) = 1$. Consequently, our initial condition vector will be

$$\mathbf{x}(0) = \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Of course, this is a column vector, so we use `x0 = [0;1]` or `x0 = [0, 1]'`. Then

```
>> [t,x] = ode45(@F,[0,10],[0;1]);
```

computes the solution.[7]

Typing `whos` at the MATLAB prompt yields[8]

```
>> whos
  Name      Size                    Bytes  Class

  t        77x1                       616  double array
  x        77x2                      1232  double array
```

Note that `t` and `x` both have 77 rows. Each row in the solution array `x` corresponds to a time in the same row of the column vector `t`. The matrix `x` has two columns. The first column of `x` contains the solution

[7] If you are working with version 5 of MATLAB, execute `[t,x]=ode45('F',[0,10],[0;1])` instead.

[8] You may have more variables in your workspace than the number shown here. In addition the number of rows in `t` and `x` may differ from system to system.

for x_1, while the second column of `x` contains the solution x_2. Typing `[t,x]` at the MATLAB prompt and viewing the resulting output will help make this connection[9].

Given a column vector `v` and a matrix `A` which have the same number of rows, the MATLAB command `plot(v,A)` will produce a plot of each column of the matrix `A` versus the vector `v`.[10] Consequently, the command `plot(t,x)` should produce a plot of each column of the matrix `x` versus the vector `t`. The following commands were used to produce the plots of x_1 versus t and x_2 versus t in Figure 8.3.

```
>> plot(t,x)
>> title('x_1'' = x_2 - x_1^2 and x_2'' = -x_1 - 2x_1x_2')
>> xlabel('t'), ylabel('x_1 and x_2')
>> legend('x_1','x_2'), grid
```

Since we are limited to black and white in this Manual, the different colors of the curves are not available, so we changed the curve for x_2 to a dotted linestyle. To do this we clicked the mouse button near the x_2 curve and selected **Edit→Current Object Properties ...**. We then made the change of line style in the Editor.

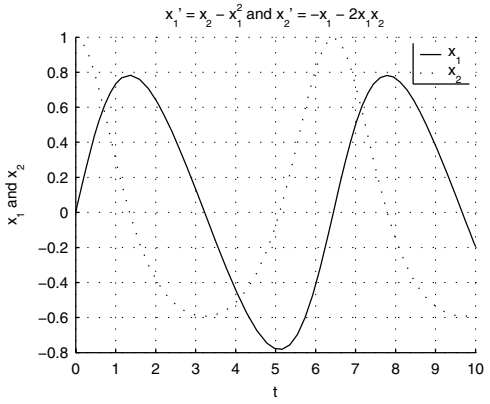

Figure 8.3. `plot(t,x)` plots both components of the solution.

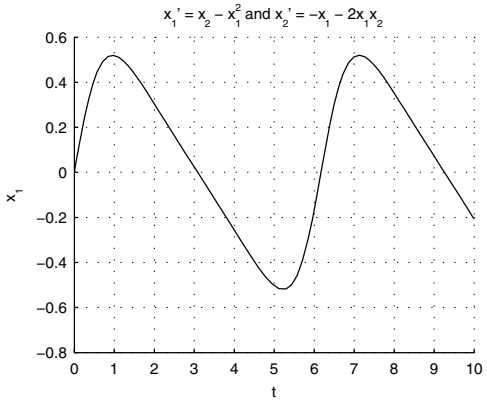

Figure 8.4. `plot(t,x(:,1))` plots the first component of the solution.

If only the first component of the solution is wanted, enter `plot(t,x(:,1))`. The colon in the notation `x(:,1)` indicates[11] that we want all rows, and the 1 indicates that we want the first column. The result is shown in Figure 8.4. Similarly, if only the second component is wanted, enter `plot(t,x(:,2))`. This is an example of the very sophisticated indexing options available in MATLAB. It is also possible to plot the components of the solution against each other with the commands

```
>> plot(x(:,1),x(:,2))
>> title('x_1'' = x_2-x_1^2 and x_2'' = -x_1-2x_1x_2')
>> xlabel('x_1'), ylabel('x_2'), grid
```

[9] For example, note that the time values in the vector `t` run from 0 through 10, as they should.

[10] Execute `help plot` to see a complete description of what the `plot` command is capable of.

[11] Some MATLAB users pronounce the notation `x(:,1)` as "x, all rows, first column."

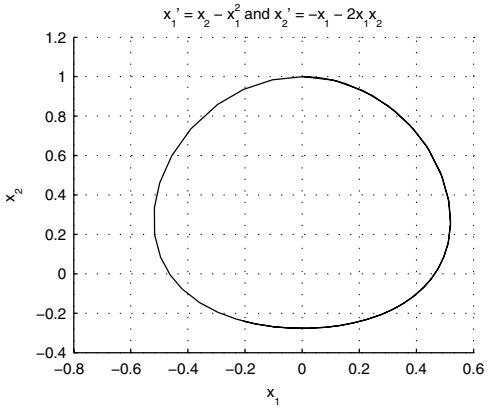

Figure 8.5. Phase plane plot of the solution.

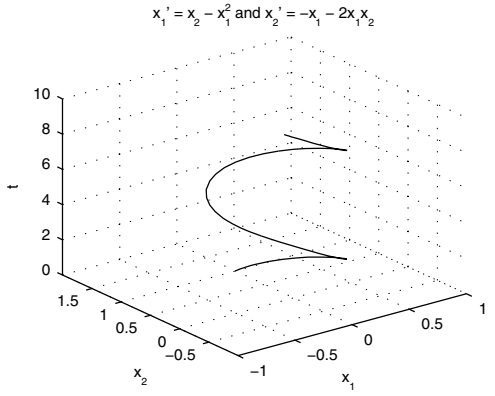

Figure 8.6. 3D plot of the solution.

The result is called a *phase plane* plot, and it is shown in Figure 8.5.

Another way to present the solution to the system graphically is in a three dimensional plot, where both components of the solution are plotted as separate variables against the independent variable t. MATLAB does this using the command `plot3`. For example, enter `plot3(t,x(:,1),x(:,2))` to see the plot with t along the x-axis, and the two components of the solution along the y-axis and z-axis, respectively. Alternatively, enter `plot3(x(:,1),x(:,2),t)` to see the solution with t along the z-axis, and the two components of the solution along the x-axis and y-axis, respectively. The result of this command is shown in Figure 8.6. The three dimensional graph can be rotated by first selecting the rotation icon in the toolbar, and then clicking and dragging on the figure.

Solving systems with eul, rk2, and rk4. These solvers, introduced in Chapter 5, can also be used to solve systems. The syntax for doing so is not too different from that used with `ode45`. For example, to solve the system in (8.9) with `eul`, we use the command

```
>> [t,x] = eul(@F,[0,10],[0;1],h);
```

where `h` is the chosen step size. To use a different solver it is only necessary to replace `eul` with the `rk2` or `rk4`. Thus, the only difference between using one of these solvers and `ode45` is that it is necessary to add the step size as an additional parameter.

Second Order Differential Equations

To solve a single second order differential equation it is necessary to replace it with the equivalent first order system. For the equation

$$y'' = f(t, y, y'), \tag{8.11}$$

we set $x_1 = y$, and $x_2 = y'$. Then $\mathbf{x} = [x_1, x_2]^T$ is a solution to the first order system

$$\begin{aligned} x_1' &= x_2, \\ x_2' &= f(t, x_1, x_2). \end{aligned} \tag{8.12}$$

Conversely, if $\mathbf{x} = [x_1, x_2]^T$ is a solution of the system in (8.12), we set $y = x_1$. Then we have $y' = x_1' = x_2$, and $y'' = x_2' = f(t, x_1, x_2) = f(t, y, y')$. Hence y is a solution of the equation in (8.11).

Example 4. *Plot the solution of the initial value problem*

$$y'' + yy' + y = 0, \quad y(0) = 0,\ y'(0) = 1, \tag{8.13}$$

on the interval [0, 10].

First, solve the equation in (8.13) for y''.

$$y'' = -yy' - y \tag{8.14}$$

Introduce new variables for y and y'.

$$x_1 = y \quad \text{and} \quad x_2 = y' \tag{8.15}$$

Then we have by (8.14) and (8.15),

$$x_1{}' = y' = x_2, \quad \text{and} \quad x_2{}' = y'' = -yy' - y = -x_1x_2 - x_1,$$

or, more simply,

$$\begin{aligned} x_1' &= x_2, \\ x_2' &= -x_1x_2 - x_1. \end{aligned} \tag{8.16}$$

If we let

$$\mathbf{F}(t, [x_1, x_2]^T) = \begin{bmatrix} x_2 \\ -x_1x_2 - x_1 \end{bmatrix} \tag{8.17}$$

and $\mathbf{x} = [x_1, x_2]^T$, then $\mathbf{x}' = [x_1{}', x_2{}']^T$ and system (8.16) takes the form $\mathbf{x}' = \mathbf{F}(t, \mathbf{x})$.

Let's address this example by putting everything into one master file. This file could be

```
function ch8examp4
[t,x] = ode45(@dfile,[0,10],[0;1]);
plot(t,x(:,1))
title('y'''' + yy'' + y = 0, y(0) = 0, y''(0) = 1')
xlabel('t'), ylabel('y'), grid

function xprime = dfile(t,x)
xprime = zeros(2,1);
xprime(1) = x(2);
xprime(2) = -x(1)*x(2) - x(1);
```

Save the file as `ch8examp4.m`. When we execute `ch8examp4` we get Figure 8.7.

Let's examine the file `ch8examp4`. First notice that the subfunction `dfile` is the ODE file, and implements the function **F** defined in (8.17). According to (8.13), the initial condition is

$$\mathbf{x}(0) = \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} y(0) \\ y'(0) \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Hence the second line in `ch8examp4` calls `ode45` on the equation defined in the subfunction with the proper initial conditions. It is important to note that the original question called for a solution of $y'' + yy' + y = 0$. Recall from (8.15) that $y = x_1$. Consequently, the plot command in `ch8examp4` will plot y versus t, as shown in Figure 8.7. If we replace the `plot` command with `plot(x(:,1),x(:,2))` we will get a phase-plane plot like that in Figure 8.8.

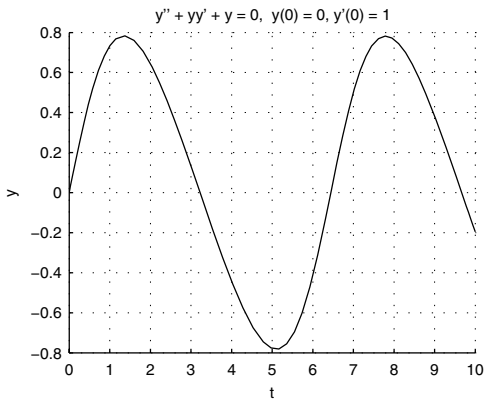

Figure 8.7. The solution to the initial value problem (8.13).

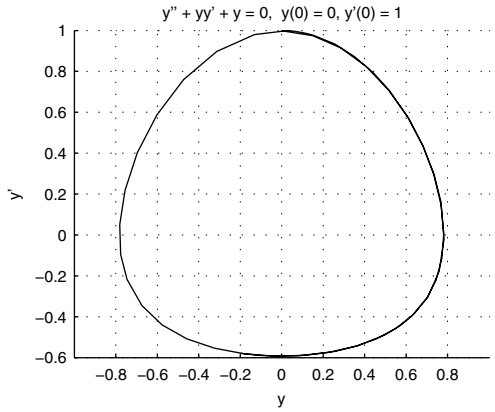

Figure 8.8. Phase-plane plot of the solution.

The Lorenz System and Passing Parameters

The solvers in MATLAB can solve first order systems containing as many equations as you like. As an example we will solve the Lorenz system. This is a system of three equations published in 1963 by the meteorologist and mathematician E. N. Lorenz. It represents a simplified model for atmospheric turbulence beneath a thunderhead. The equations are

$$\begin{aligned} x' &= -ax + ay, \\ y' &= rx - y - xz, \\ z' &= -bz + xy, \end{aligned} \tag{8.18}$$

where a, b, and r are positive constants.

We could proceed with the variables x, y, and z, but to make things a bit simpler, set $u_1 = x$, $u_2 = y$, and $u_3 = z$. With these substitutions, system (8.18) takes the following vector form.

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}' = \begin{bmatrix} -au_1 + au_2 \\ ru_1 - u_2 - u_1u_3 \\ -bu_3 + u_1u_2 \end{bmatrix} \tag{8.19}$$

If we let $\mathbf{F}(t, [u_1, u_2, u_3]^T) = [-au_1 + au_2, ru_1 - u_2 - u_1u_3, -bu_3 + u_1u_2]^T$ and $\mathbf{u} = [u_1, u_2, u_3]^T$, then $\mathbf{u}' = [u_1', u_2', u_3']^T$ and system (8.19) takes the form $\mathbf{u}' = \mathbf{F}(t, \mathbf{u})$, prompting one to write the ODE file as follows.

```
function uprime = F(t,u)
uprime = zeros(3,1);
uprime(1) = -a*u(1) + a*u(2);
uprime(2) = r*u(1) - u(2) - u(1)*u(3);
uprime(3) = -b*u(3) + u(1)*u(2);
```

However, there is a big problem with this ODE file. MATLAB executes its functions, including those defined in function M-files, in separate workspaces disjoint from the command window workspace and

from the workspaces of other functions. The variables defined in one workspace are not available in another unless they are passed to it explicitly. The impact of this is that we will have to pass the parameters a, b, and r from the command window to the function ODE file.

One way to do this is to rewrite the ODE file as[12]

```
function uprime = lor1(t,u,a,b,r)
uprime = zeros(3,1);
uprime(1) = -a*u(1) + a*u(2);
uprime(2) = r*u(1) - u(2) - u(1)*u(3);
uprime(3) = -b*u(3) + u(1)*u(2);
```

We've decided to name our function `lor1` instead of `F`, so we must save the file as `lor1.m`[13].

We will use $a = 10$, $b = 8/3$, $r = 28$, and the initial condition

$$\mathbf{u}(0) = \begin{bmatrix} u_1(0) \\ u_2(0) \\ u_3(0) \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}.$$

The commands[14]

```
>> [t,u] = ode45(@lor1,[0,7],[1;2;3],[],10,8/3,28);
>> plot(t,u)
>> title('A solution to the Lorenz system')
>> xlabel('t'), ylabel('x, y, and z')
>> legend('x','y','z'), grid
```

should produce an image similar to that in Figure 8.9. [15] Notice the use of `[]` as an entry in the `ode45` command. This is just a place holder. This place in the order of entries is reserved for options. We will explain these later.

Global Variables. There is another way of passing variables to function M-files that you might find more to your liking. This is the use of *global variables.* The only hard thing about the use of global variables is to remember that they must be declared in the both the command window and in the function M-file. Here is an example.

---

[12] In version 5 of MATLAB the first line of this file must read
`function uprime = lor1(t,u,flag,a,b,r)`. Notice the input `flag`.

[13] It would be natural to name this file `lorenz.m`, but there is already an M-file in the MATLAB directory tree with that name. If you enter `lorenz` at the MATLAB prompt you will see a solution to the Lorenz system (8.18) displayed in a very attractive manner.

[14] In version 5 of MATLAB you must invoke the solver in the first line using the syntax `[t,u]=ode45('lor1', [0,7],[1;2;3],[],10,8/3,28);`.

[15] The linestyles have been changed in Figure 8.9 to compensate for the unavailability of color.

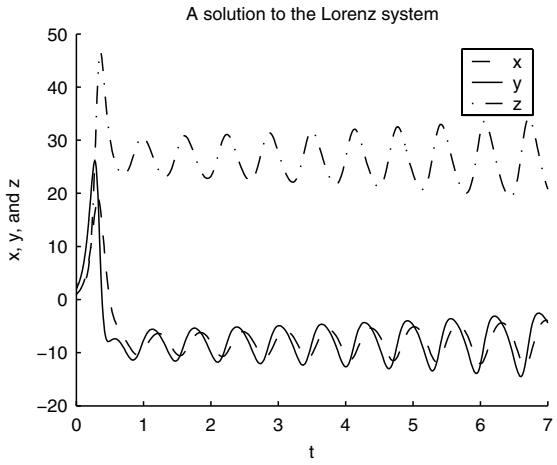

Figure 8.9. A solution to the Lorenz system.

First change the function ODE file to

```
function uprime = lor2(t,u)
global A B R
uprime = zeros(3,1);
uprime(1) = -A*u(1) + A*u(2);
uprime(2) = R*u(1) - u(2) - u(1)*u(3);
uprime(3) = -B*u(3) + u(1)*u(2);
```

Save this second version as `lor2.m`. The second line declares the parameters to be global variables. Notice that we are using uppercase names for global variables. This is good practice, but is not required.

Next, we declare the global variables in the command window and initialize them

```
>> global A B R
>> A = 10; B = 8/3; R = 28;
```

The global variables `A`, `B`, and `R` are now available in both the command window and in the workspace of the ODE file. You can find the numerical solution of the Lorenz system with the command[16]

```
>> [t,u] = ode45(@lor2,[0,7],[1;2;3]);
```

Eliminating Transient Behavior.

In many mechanical and electrical systems we experience *transient behavior*. This involves behavior that is present when the system starts, but dies out quickly in time, leaving only a fairly regular *steady-state* behavior. In Figure 8.9 there appears to be transient behavior until about $t = 1$, and then things seem

[16] In version 5 of MATLAB use `[t,u] = ode45('lor2',[0,7],[1;2;3])`.

to settle down. To examine the situation more closely, let's compute the solution over a longer period, say $0 \le t \le 100$, and then plot the part corresponding to $t > 10$ in three dimensions. We would also like to allow for randomly chosen initial values in order to see if the steady-state behavior is somehow independent of the initial conditions.

This is easily accomplished. The hardest part is choosing the initial conditions, and we can use the MATLAB command `rand` for this. If you use `help rand` you will see that `rand` produces random numbers in the range [0, 1]. We want each component of our initial conditions to be randomly chosen from the interval $[-50, 50]$. It is necessary to scale the output from `rand` to this interval. The command `u0 = 100*(rand(3,1) - 0.5)` will do the job.

Since we will want to repeat our experiment several times, we will put everything, including the ODE file, into a master function M-file named `loren.m`. We will use global variables initialized in both the master file and the subfunction. Including formatting commands, our file is

```
function loren

  global A B R
  A = 10; B = 8/3; R = 28;
  u0 = 100*(rand(3,1) - 0.5);
  [t,u] = ode45(@lor2,[0,100],u0);
  N = find(t>10);
  v = u(N,:);
  plot3(v(:,1),v(:,2),v(:,3))
  title('The Lorenz attractor')
  xlabel('x'), ylabel('y'), zlabel('z')
  grid, shg

function uprime = lor2(t,u)
  global A B R
  uprime = zeros(3,1);
  uprime(1) = -A*u(1) + A*u(2);
  uprime(2) = R*u(1) - u(2) - u(1)*u(3);
  uprime(3) = -B*u(3) + u(1)*u(2);
```

Further explanation is necessary. First, the command `N = find(t>10);` produces a list of the indices of those elements of `t` which satisfy $t > 10$. Then `v = u(N,:);` produces a matrix containing the rows of `u` with indices in `N`. The result is that `v` consists of only those solution points corresponding to $t > 10$, and the resulting plot represents the long time behaviour of the solution after the transient effects are gone.

Now, every time we execute `loren` we get a result very like Figure 8.10. The object in this figure is an approximation of the *Lorenz attractor*. The Lorenz system, with this particular choice of parameters, has the property that any solution, no matter what its initial point, is attracted to this rather complicated, butterfly-shaped, set. We will return to this in the exercises. There we will also examine the Lorenz system for different values of the parameters.

Click on the rotate icon in the toolbar, and click and drag the mouse in the axes in Figure 8.10 to

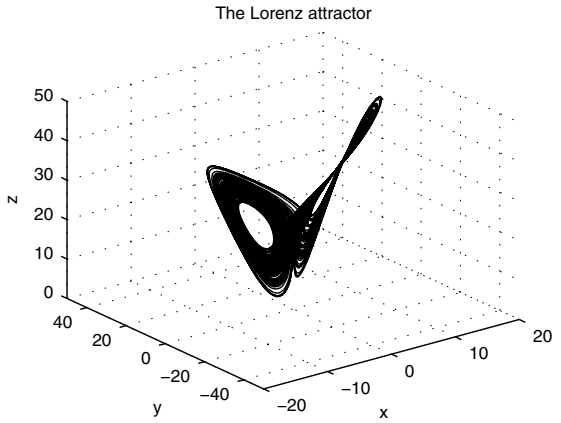

Figure 8.10. The Lorenz attractor.

rotate the axes to a different view of the Lorenz attractor. If the animation is unbearably slow, go back and repeat the commands for the construction of the Lorenz attractor, only use a smaller time interval, say [0, 20].

For an animated plot, add the line `figure(1)` after the first `function` statement, and the line

```
comet3(v(:,1),v(:,2),v(:,3))
```

immediately before the `plot` command.

Improving Accuracy

It has been mentioned before that `ode45` chooses its own step size in order to achieve a predetermined level of accuracy. Although the default level is sufficient for many problems, sometimes it will be necessary to improve that "predetermined level of accuracy."

The accuracy is controlled by the user by defining two optional inputs to `ode45`, and the other solvers. These are the *relative tolerance*, RelTol, and the *absolute tolerance*, AbsTol. If $\mathbf{y}^k$ is the computed solution at step k, then each component of the solution is required to satisfy its own error restriction. This means that we consider an estimated error vector, which has a component for every component of $\mathbf{y}^k$, and it is required that for each j, the jth component of the error vector satisfy

$$|\text{estimated error}_j^k| \leq \text{tolerance}_j, \tag{8.20}$$

where

$$\text{tolerance}_j = \max(|y_j^k| \times \text{RelTol}, \text{AbsTol}_j). \tag{8.21}$$

A little algebra discloses that

$$\text{tolerance}_j = \begin{cases} \text{AbsTol}_j, & \text{if } |y_j^k| \leq \text{AbsTol}_j/\text{RelTol}, \\ |y_j^k| \times \text{RelTol}, & \text{otherwise.} \end{cases} \tag{8.22}$$

Thus the relative tolerance controls the tolerance unless $|y_j^k|$ is relatively small, where "relatively small" means $|y_j^k| \leq \text{AbsTol}_j/\text{RelTol}$.

Notice that the relative tolerance is a number, but the absolute tolerance is a vector quantity, with a component for each equation in the system being solved. This allows the user to set the tolerance differently for different components of the solution. The default values for `RelTol` is 10^{-3}, and `AbsTol` is a vector, each component of which is 10^{-6}. It is also possible to use a single number for the absolute tolerance, if the same tolerance will work for each component.

Choosing the Tolerances. The philosophy behind the use of the two tolerances is that the relative tolerance should bound the estimated error by a certain fraction of the size of the computed quantity y_j^k for most values of that quantity. The absolute tolerance should come into play only for values of y_j^k which are small in comparison to the range of values that y_j^k assumes. This is to prevent the routine from trying too hard if the computed quantity turns out to be very small at some steps. For example, with the defaults of $\text{RelTol} = 10^{-3}$ and $\text{AbsTol} = 10^{-6}$, we have $\text{tolerance}_j = \text{AbsTol}_j$ only when $|y_j^k| \leq 10^{-3}$.

Let's talk about choosing the relative tolerance first. This is the easier case since we are simply talking about the number of significant digits we want in the answer. For example, the default relative tolerance is 10^{-3}, or 0.1%. In an ideal world this will ensure that the first three digits of the computed answer will be correct. If we want five significant digits we should choose 10^{-5}, or 0.001%.

It would be nice if things were as simple as these considerations make them seem. Unfortunately the inequalities in (8.20), (8.21), and (8.22) at best control the errors made at an individual step. These errors propagate, and errors made at an early step can propagate to something much larger. In fact, the errors can propagate exponentially. The upshot is that over long time periods computational solutions can develop serious inaccuracies. It is usually a good idea is to introduce a safety factor into your choice of tolerances. For example if you really want three significant digits, it might be good to choose the relative tolerance smaller than 10^{-3}, say 10^{-5}. Even then it is imperative that you check your answer with what you expect and reduce the tolerance even further if it seems to be called for. (See Example 5.)

To choose the absolute tolerance vector, remember that the philosophy is that the tolerance should be equal to the absolute tolerance only when the computed value y_j^k is small relative to the range of values of that component. Suppose that we expect that $|y_j^k| \leq M_j$ at all computed points. Then we might demand that AbsTol dominate the tolerance only when $|y_j^k| \leq 10^{-3} \times M_j$. From (8.22) we see that we want $\text{AbsTol}_j/\text{RelTol} = 10^{-3} \times M_j$, so we set

$$\text{AbsTol}_j = 10^{-3} \times M_j \times \text{RelTol}. \tag{8.23}$$

Admittedly, it is usually impossible to know M_j before the solution is computed. However, one gains information about this as one studies the solution. Of course the factor of 10^{-3} can be replaced by a smaller or larger number if the situation at hand merits it.

Let's use these considerations to decide what tolerances to use for the Lorenz equations. Since we are only going to plot the solution, two place accuracy along the entire solution should be sufficient. Thus, the default relative tolerance of 10^{-3} would seem to be sufficient, but if we use a safety factor of 10^{-2} we get a relative tolerance of 10^{-5}. If we were to compute over longer periods of time, we might want to choose a smaller value, but without examining the computed results a relative tolerance smaller than 10^{-5} does not seem justified.

To choose the absolute tolerance vector, we notice from Figures 8.9 and 8.10 that the normal ranges of the components are $|x| \leq 20$, $|y| \leq 30$, and $|z| \leq 40$. Hence according to (8.23), we should choose

$$\begin{aligned}
\text{AbsTol}_1 &\leq 10^{-3} \times 20 \times 10^{-3} = 2 \times 10^{-5}, \\
\text{AbsTol}_2 &\leq 10^{-3} \times 30 \times 10^{-3} = 3 \times 10^{-5}, \\
\text{AbsTol}_3 &\leq 10^{-3} \times 40 \times 10^{-3} = 4 \times 10^{-5}.
\end{aligned}$$

We see that the default absolute tolerance of 10^{-6} for each seems to be adequate.

In our approach to choosing tolerances everything depends on choosing the relative tolerance. From our earlier discussion this seems simple enough. However, there are situations in which more accuracy may be required than appears on the surface. Suppose, for example, that it is the difference of two components of the solution which is really important. To be more specific, suppose we compute a solution $\mathbf{y}$, and what we are really interested in is $y_1 - y_2$, the difference of the first two components of $\mathbf{y}$. Suppose in addition that y_1 and y_2 are very close to each other. Suppose in fact that we know they will always agree in the first 6 digits. If we set the relative tolerance to 10^{-7}, then in an ideal world we will get 7 significant digits for y_1 and y_2. For example we might get $y_1 = 1234569 * * * * *$ and $y_2 = 1234563 * * * * *$, where the asterisks indicate digits that are not known accurately. Then $y_1 - y_2 = 6 * * * * *$, so we only know 1 significant digit in the quantity we are really interested in. Consequently to get more significant digits in $y_1 - y_2$ we have to choose a smaller relative tolerance. To get 3 significant digits we need 10^{-9}, or with a suitable safety factor, 10^{-11}.

Using the Tolerances in the Solvers. Having chosen our tolerances, how do we use them in `ode45` and the other MATLAB solvers? These solvers have many options, many more than we will describe in this book. All of the options are input into the solver using an options vector. Suppose, for example, that we wanted to solve the Lorenz system with relative tolerance 10^{-5} and absolute tolerance vector $[10^{-8}, 10^{-7}, 10^{-7}]$. We build the options vector with the command `odeset`, and then enter it as a parameter to `ode45`. If we use the first version of the Lorenz ODE file, `lor1.m`, then the needed commands are

```
>> options  =  odeset('RelTol',1e-5,'AbsTol',[1e-8 1e-7 1e-7]);
>> [t,u] = ode45(@lor1,[0,100],[1;2;3],options,10,8/3,28);
```

If the run takes inordinately long, try a shorter time span, say `[0,20]` instead of `[0,100]`. Notice that the syntax used with `ode45` is exactly the same as was used earlier for the Lorenz system, except that we have inserted the parameter `options` between the initial conditions and the parameters sent to the routine.

If you wish to use global variables, then use the second version of the Lorenz ODE file, `lor2`, with

```
>> options  =  odeset('RelTol',1e-5,'AbsTol',[1e-8 1e-7 1e-7]);
>> [t,u] = ode45(@lor2,[0,100],[1;2;3],options);
```

Again, this assumes that you have declared the global variables and initialized them.

An experimental approach to choosing the relative tolerance. Perhaps the best way to choose the relative tolerance is illustrated by the next example.

Example 5. *The initial value problem*[17]

$$y'' + 0.1y' + \sin y = \cos t \quad \text{with} \quad y(0) = 0 \quad \text{and} \quad y'(0) = 2$$

models a forced damped pendulum. Use `ode45` *to solve the problem over the interval* $[0, 150]$.

The homogeneous problem has stable equilibria where y is an integer multiple of 2π and $y' = 0$. The relatively large initial velocity, and the driving force can lead to erratic behavior. Small computing errors can lead to incorrect solutions, as we will see. The problem is to choose an appropriate relative tolerance for `ode45`.

This time we will take an experimental approach to the problem. We will solve the problem with a tolerance `reltol`, and then solve it again with tolerance `reltol/10`. We will compare the two solutions graphically, and only when they are sufficiently close will we be satisfied. The function M-file `borrelli.m` will do this for us.

```
function borrelli(reltol)
  close all
  tspan = [0,150];
  velocity = 2; position = 0;
  color = 'r';
  hold on
  for kk = 1:2
    opts = odeset('RelTol',reltol);
    [t,y]=ode45(@dfile,tspan,[position;velocity],opts);
    plot(t,y(:,1),'-','color',color)
    reltol = reltol/10;
    color = 'k';
  end
  hold off
  xlabel('t'), ylabel('y')
  title('y'''' + 0.1y'' + sin(y) = cos(t), y(0)=0, y''(0)=2')
  set(gca,'YTick',-2*pi:2*pi:6*pi,...
          'YTickLabel',{'-2pi','0','2pi','4pi','6pi'},...
          'YGrid','on')

function yprime=dfile(t,y)
  yprime=zeros(2,1);
  yprime(1) = y(2);
  yprime(2) = -sin(y(1)) - 0.1*y(2) + cos(t);
```

The file needs some explanation. The command `close all` closes all figures, and allows the routine to start with a clean slate each time it is used. Use this command with care, since it may close a figure

[17] For more about this example, see Robert Borrelli, Courtney Coleman *Computers, Lies, and the Fishing Season*, College Math Journal Volume 25, Number 5 (1994) and John Hubbard, *The Forced Damped Pendulum: Chaos, Complication, and Control* The American Mathematical Monthly (October 1999)

that you want to use later. The `for` loop computes and plots the solution twice. The first time it uses the input `reltol` and plots the curve in red. The second time through it uses `reltol/10` and plots the curve in black. We will be able to easily compare the graphs and we will consider the input `reltol` to be successful if very little of the red curve shows through the black curve. Finally, the `set` command sets the ticks on the y-axis to be multiples of π.

We first execute `borrelli(1e-3)`. Remember that 10^{-3} is the default relative tolerance for `ode45`. The result is shown in Figure 8.11. The solutions agree pretty well up to about $t = 25$ and then diverge dramatically. Clearly the default relative tolerance is not sufficiently small. When we execute `borrelli(1e-4)` we get Figure 8.12. This time the curves overlap, so using a relative tolerance of 10^{-4} will do an adequate job, although 10^{-5} would be a more conservative choice.

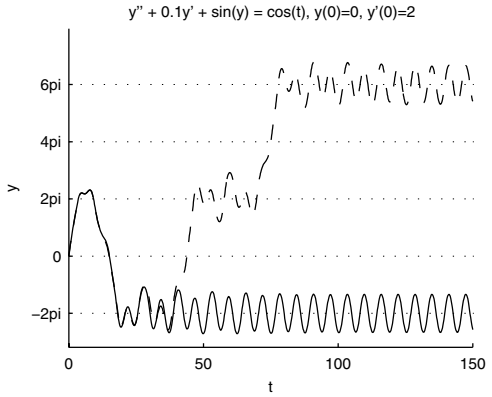

Figure 8.11. The solutions with relative tolerances 10^{-3} (solid) and 10^{-4} (dashed).

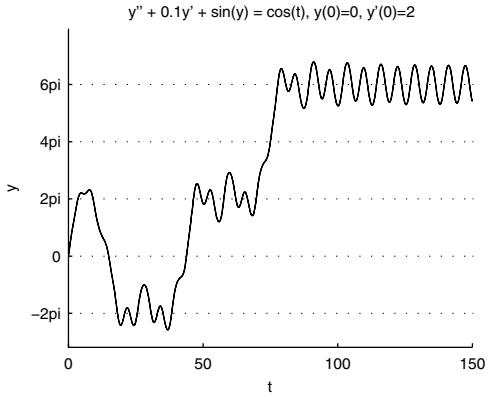

Figure 8.12. The solutions with relative tolerances 10^{-4} (solid) and 10^{-5} (dashed) overlap.

Example 5 illustrates a procedure that should be followed whenever you compute the solution to a system of differential equations over a relatively long interval. Compare the results with a chosen relative tolerance and with that divided by 10. Reduce the relative tolerance until the two solutions agree. Only then can you have confidence in your solution.

Behavior Near Discontinuities

Any numerical method will run into difficulties where the equation and/or the solution to the equation has a point of discontinuity, and `ode45` is no exception. There are at least three possible outcomes when `ode45` meets a discontinuity.

1. `ode45` can integrate right through a discontinuity in the equation or the solution, not even realizing it is there. In this case the accuracy of the result is highly doubtful, especially in that range beyond the discontinuity. This phenomenon will happen, for example, with the initial value problem $x' = (x + 3t^2)/t$, with initial value $x(-1) = 1$ on the interval $[-1, 1]$. Another example is $x' = t/(x + 1)$, with $x(-2) = 0$ on the interval $[-2, -3/2]$.
2. `ode45` can find the discontinuity and report it with a warning. This is usually the sign that there is

a discontinuity in the solution. An example of this is $x' = x^2$, with $x(0) = 1$ on the interval $[0, 2]$. The solution is $x(t) = 1/(1-t)$, which has a singularity at $t = 1$. When asked to solve this initial value problem, `ode45` responds with

```
Warning: Failure at t=9.999694e-001.  Unable to meet integration
  tolerances without reducing the step size below the smallest
  value allowed (3.552605e-015) at time t.
```

One very nice thing about `ode45` is that if this happens, the output up to the point where the routine stops is made available. For example, if you execute `[t,w]  = ode45('gizmo',[0,3.4],[0; 1; 2]);`, and the computation stops because a step size is called for which is smaller than the minimum allowed, then the variables `t` and `w` will contain the results of the computation up to that point.

3. `ode45` can choose smaller and smaller step sizes upon the approach to a discontinuity and go on calculating for a very long time — hours in some cases. For cases like this, it is important for the user to know how to stop a calculation in MATLAB. On most computers the combination of the control key and C depressed simultaneously will do the trick. An example is $x' = t/(x+1)$, with $x(-10) = 2$ on the interval $[-10, -5]$. Many stiff equations and systems will also require an inordinate solution time using `ode45`.

Stiff Equations

Solutions to differential equations often have components which are varying at different rates. If these rates differ by a couple of orders of magnitude, the equations are called stiff. Stiff equations will not arise very often in this manual, but we will say a word about them in case you should have to deal with one. We will also put some examples in the exercises.

The equation $x' = e^t \cos x$ from Example 2 is stiff over relatively large time intervals, simply because the factor e^t gets very large. To see what this does to `ode45`, execute

```
>> f = inline('exp(t)*cos(x)','t','x');
>> ode45(f,[0,10],0)
```

Since we have not required any output from `ode45`, it will plot the solution as it computes it. You will notice that the speed of the plot slows up dramatically as t increases. When you are tired of waiting for the solution to end, click on the **Stop** button on the lower left of the figure window. This very slow computation is the typical response of `ode45` to a stiff equation or system. The fast reaction rate, measured in this case by the factor e^t, requires `ode45` to take extremely small steps, and therefore a long time to complete the solution.

The MATLAB suite of ode solvers contains four routines, `ode15s`, `ode23s`, `ode23t`, and `ode23tb`, which are designed to solve stiff equations. `ode15s` is the first of these to try. To see what it does in this case, execute

```
>> ode15s(f,[0,10],0)
```

Since it is designed specifically for stiff systems, it completes its work in seconds. Notice that the calling syntax for `ode15s` is identical to that of `ode45`. This is a feature of all of the solvers in the ODE suite.

Another example of a stiff system is the van der Pol system

$$x' = y,$$
$$y' = -x + \mu(1 - x^2)y,$$

when the parameter μ is very large, say $\mu = 1000$.

But how do we tell if a system is stiff? It is often obvious from the physical situation being modeled that there are components of the solution which vary at rates that are significantly different, and therefore the system is stiff. Sometimes the presence of a large parameter in the equations is a tip off that the system is stiff. However, there is no general rule that allows us to recognize stiff systems. A good operational rule is to try to solve using `ode45`. If that fails, or is very slow, try `ode15s`.

Other Possibilities

The MATLAB ODE Suite consists of a very powerful group of solvers. We have only touched the surface of the options available to control them, and the variations of the ways in which they can be used. To mention just a few, it is possible to modify the output of the solver so that the solution is automatically plotted in the way the user desires. It is also possible to get the solvers to detect events. For example the 3-body problem of the sun, earth, and moon can be modeled by a system of 12 ODEs, and the solver can be set up to automatically detect the times of eclipses of the sun and moon.

A very nice description of the ODE Suite appears in *Using MATLAB*, which is part of the MATLAB documentation. You should be able to find it by entering `helpdesk` at the MATLAB prompt. After the MATLAB Help Desk opens, click on Using MATLAB, then on Mathematics, and finally on Differential Equations.

A more technical paper dealing the the ODE Suite is also available from the Online Manuals list. It is written by Larry Shampine and Mark Reichelt, who built the Suite.

Exercises

In Exercises 1 – 3 we will deal with the differential equation $y' = -2y + 2\cos t \sin 2t$. In preparation create the function M-file

```
function yprime = steady(t,y)
yprime = -2*y+2*cos(t).*sin(2*t);
```

and save the file as `steady.m`.

1. MATLAB's solvers are able to handle multiple initial conditionsfor single first order equations. To do so we fake `ode45` into assuming that we are solving a system consisting of the same equation repeated for each initial condition. For this purpose it is only necessary to make the function ODE file array smart, as we have for `steady.m`.
   a) To solve the equation $y' = -2y + 2\cos t \sin 2t$, for initial conditions $y(0) = -5$, $y(0) = -4, \ldots,$ $y(0) = 5$, enter the code `[t,y] = ode45(@steady,[0,30],-5:5);`.
   b) Graph the solutions with `plot(t,y)` and note that each solution approaches a periodic steady-state solution.
   c) Repeat parts a) and b) with time spans [0, 10] and $[0, 2\pi]$ to get a closer look at the convergence to the steady-state solution.

2. Another nice feature of MATLAB's solvers is the ability to output solutions at specified times. This is useful when you want to compare two functions at specified time points or place the output from multiple calls to ode45 in a matrix, where it is required that the rows have equal length. Enter `[t,y] = ode45(@steady,0:.25:3,1);`, then enter `[t,y]`. This should clarify what is going on.
3. MATLAB's solvers can automatically plot solutions *as they are computed*. All you need to do is leave off the output variables when calling the solver. Try `ode45('steady',[0,30],-5:5)`.

In Exercises 4 – 7, plot the solutions to all of the initial value problems on one figure. Choose a solution interval starting at $t = 0$ that well illustrates any steady-state phenomena that you see. Exercise 1 might be helpful.

4. $y' + 4y = 2\cos t + \sin 4t$, with $y(0) = -5, -4, \ldots, 5$.
5. $y' + 4y = t^2$, with $y(0) = -9, -8, -7, -1, 1, 7, 8, 9$.
6. $x' = \cos t - x^3$, with $x(0) = -3, -2, \ldots, 3$.
7. $x' + x + x^3 = \cos^2 t$, with $x(0) = -3, -2, \ldots, 3$.
8. Separate variables and find an explicit solution of $x' = t/(1 + x)$, $x(0) = 1$.
   a) Use `ode45` to find the numerical solution of the initial value problem on the interval [0, 3]. Store the solution in the variables `t` and `x_ode45`.
   b) Use the explicit solution to find exact solution values at each point in the vector `t` obtained in part a). Store the results in the variable `x_exact`. Compare the solutions graphically with `plot(t, x_exact, t, x_ode45,'r.')`. Obtain a printout with a legend.
   c) For a finer graphical image of the error plot `x_exact - x_ode45` versus `t` in a new figure. Pay special attention to the dimensions on the y-axis.

In Exercises 9 – 12, write a function ODE file for the system. Solve the initial value problem using `ode45`. Provide plots of x_1 versus t, x_2 versus t, and x_2 versus x_1 on the time interval provided. See Exercises 1 – 6 in Chapter 7 for a nice method to arrange these plots.

9. ${x_1}' = x_2$ and ${x_2}' = (1 - {x_1}^2)x_2 - x_1$, with $x_1(0) = 0$ and $x_2(0) = 4$ on [0, 10].
10. ${x_1}' = x_2$ and ${x_2}' = -25x_1 + 2\sin 4t$, with $x_1(0) = 0$ and $x_2(0) = 2$ on $[0, 2\pi]$.
11. ${x_1}' = (x_2 + x_1/5)(1 - x_1^2)$ and ${x_2}' = -x_1(1 - x_2^2)$, with $x_1(0) = 0.8$ and $x_2(0) = 0$ on [0, 30].
12. ${x_1}' = x_2$ and ${x_2}' = -x_1 + x_2(1 - 3x_1^2 - 3x_2^2)$, with $x_1(0) = 0.2$ and $x_2(0) = 0.2$ on [0, 20].
13. The built-in plotting routines of MATLAB's solvers were illustrated in Exercise 3. They work equally well with systems. For example, create

```
function yprime = heart(t,y)
yprime = zeros(2,1);
yprime(1) = y(2);
yprime(2) = -16*y(1)+4*sin(2*t);
```

    and save as `heart.m`.
    a) Enter `ode45(@heart,[0,2*pi],[0;2])` and note that MATLAB dynamically plots both y_1 and y_2 versus t.
    b) To change the output routine, enter `options = odeset('OutputFcn','odephas2')`, followed by `ode45(@heart,[0,2*pi],[0;2],options)`. Note that MATLAB dynamically plots y_2 versus y_1 in the phase plane.
14. Use the technique of Example 4 to change the initial value problem $yy'' - (y')^2 - y^2 = 0,\ y(0) = 1,\ y'(0) = -1$ to a system with initial conditions. Use `ode45` to create a plot of y versus t.
15. An unforced spring-mass system with no damping can be modeled by the equation $y'' + ky = 0$, where k is the spring constant. However, suppose that the spring loses elasticity with time. A possible model for an "aging" spring-mass system is $y'' + 2e^{-0.12t}y = 0$. Suppose that the spring is stretched two units from equilibrium and released from rest. Use the technique of Example 4 to plot a solution of y versus t on the time interval [0, 100].

16. The equation $x'' = ax' - b(x')^3 - kx$ was devised by Lord Rayleigh to model the motion of the reed in a clarinet. With $a = 5$, $b = 4$, and $k = 5$, solve this equation numerically with initial conditions $x(0) = A$, and $x'(0) = 0$ over the interval $[0, 10]$ for the three choices $A = 0.5$, 1, and 2. Use MATLAB's `hold` command to prepare both time plots and phase plane plots containing the solutions to all three initial value problems superimposed. Describe the relationship that you see between the three solutions.

17. Consider the forced, damped oscillator $y'' + 0.05y' + 36y = \sin 6.3t$, with initial conditions $y(0) = 5$, $y'(0) = 0$. Use the technique of Example 4 to transform the equation to system of first order ODEs, write and save the odefile as `osc.m`, and solve the system with the command `[t,x] = ode45(@osc,[0,100],[5;0]);`.

    a) The command `plot3(x(:,1),x(:,2),t)` provides a three dimensional plot with time on the vertical axis.

    b) The command `close all, comet3(x(:,1),x(:,2),t)` provides an animated view of the result in part a).

18. Consider the forced, undamped oscillator modeled by the equation $y'' + {\omega_0}^2 y = 2\cos\omega t$. The parameter ω_0 is the *natural frequency* of the unforced system $y'' + {\omega_0}^2 y = 0$. Write an ODE function M-file for this equation with natural frequency $\omega_0 = 2$, and the driving frequency ω passed as a parameter.

    a) If the driving frequency nearly matches the natural frequency, then a phenomenon called *beats* occurs. As an example, solve the equation using `ode45` with $y(0) = 0$ and $y'(0) = 0$ over the interval $[0, 60\pi]$ with $\omega = 1.9$.

    b) If the driving frequency matches the natural frequency, then a phenomenon called *resonance* occurs. As an example, solve the equation using `ode45` with $y(0) = 0$ and $y'(0) = 0$ over the interval $[0, 60\pi]$ with $\omega = 2$.

    c) Use the `plot3` and `comet3` commands, as described in Exercise 17, to create three dimensional plots of the output in parts a) and b).

19. **Harmonic motion.** An unforced, damped oscillator is modeled by the equation

$$my'' + cy' + ky = 0,$$

    where m is the mass, c is the damping constant, and k is the spring constant. Write the equation as a system of first order ODEs and create a function ODE file that allows the passing of parameters m, c, and k. In each of the following cases compute the solution with initial conditions $y(0) = 1$, and $y'(0) = 0$ over the interval $[0, 20]$. Prepare both a plot of y versus t and a phase plane plot of y' versus y.

    a) (No damping) $m = 1$, $c = 0$, and $k = 16$.

    b) (Under damping) $m = 1$, $c = 2$, and $k = 16$.

    c) (Critical damping) $m = 1$, $c = 8$, and $k = 16$.

    d) (Over damping) $m = 1$, $c = 10$, and $k = 16$.

20. The system

$$\varepsilon \frac{dx}{dt} = x(1-x) - \frac{(x-q)}{(q+x)} fz,$$
$$\frac{dz}{dt} = x - z,$$

    models a chemical reaction called an *oregonator*. Suppose that $\varepsilon = 10^{-2}$ and $q = 9 \times 10^{-5}$. Put the system into normal form and write an ODE function M-file for the system that passes f as a parameter. The idea is to vary the parameter f and note its affect on the solution of the oregonator model. We will use the initial conditions $x(0) = 0.2$ and $y(0) = 0.2$ and the solution interval $[0, 50]$.

    a) Use `ode45` to solve the system with $f = 1/4$. This should provide no difficulties.

    b) Use `ode45` to solve the system with $f = 1$. This should set off all kinds of warning messages.

    c) Try to improve the accuracy with `options = odeset('RelTol',1e-6)` and using `options` as the options parameter in `ode45`. You should find that this slows computation to a crawl as the system is very stiff.

d) The secret is to use `ode15s` instead of `ode45`. Then you can note the oscillations in the reaction.

21. Consider the three-dimensional oregonator model

$$\begin{aligned}\frac{dx}{dt} &= \frac{qy - xy + x(1-x)}{\varepsilon},\\ \frac{dy}{dt} &= \frac{-qy - xy + fz}{\varepsilon'},\\ \frac{dz}{dt} &= x - z.\end{aligned}$$

Set $\varepsilon = 10^{-2}$, $\varepsilon' = 2.5 \times 10^{-5}$, and $q = 9 \times 10^{-5}$. Set up an ODE file that will allow the passing of the parameter f. Slowly increase the parameter f from a low of 0 to a high of 1 until you clearly see the system oscillating on the interval [0, 50] with initial conditions $x(0) = 0.2$, $y(0) = 0.2$, and $z(0) = 0.2$. You might want to read the comments in Exercise 20. Also, you'll want to experiment with `plot`, `zoom`, and `semilogy` to best determine how to highlight the oscillation in all three components.

22. The system

$$\begin{aligned}m_1 x'' &= -k_1 x + k_2(y - x),\\ m_2 y'' &= -k_2(y - x),\end{aligned}$$

models a *coupled* oscillator. Imagine a spring (with spring constant k_1) attached to a hook in the ceiling. Mass m_1 is attached to the spring, and a second spring (with spring constant k_2) is attached to the bottom of mass m_1. If a second mass, m_2, is attached to the second spring, you have a coupled oscillator, where x and y represent the displacements of masses m_1 and m_2 from their respective equilibrium positions.

If you set $x_1 = x$, $x_2 = x'$, $x_3 = y$, and $x_4 = y'$, you can show that

$$\begin{aligned}x_1{}' &= x_2,\\ x_2{}' &= -\frac{k_1}{m_1}x_1 + \frac{k_2}{m_1}(x_3 - x_1),\\ x_3{}' &= x_4,\\ x_4{}' &= -\frac{k_2}{m_2}(x_3 - x_1).\end{aligned}$$

Assume $k_1 = k_2 = 2$ and $m_1 = m_2 = 1$. Create an ODE file and name it `couple.m`. Suppose that the first mass is displaced upward two units, the second downward two units, and both masses are released from rest. Plot the position of each mass versus time.

23. In the 1920's, the Italian mathematician Umberto Volterra proposed the following mathematical model of a predator-prey situation to explain why, during the first World War, a larger percentage of the catch of Italian fishermen consisted of sharks and other fish eating fish than was true both before and after the war. Let $x(t)$ denote the population of the prey, and let $y(t)$ denote the population of the predators.

In the absence of the predators, the prey population would have a birth rate greater than its death rate, and consequently would grow according to the exponential model of population growth, i.e. the growth rate of the population would be proportional to the population itself. The presence of the predator population has the effect of reducing the growth rate, and this reduction depends on the number of encounters between individuals of the two species. Since it is reasonable to assume that the number of such encounters is proportional to the number of individuals of each population, the reduction in the growth rate is also proportional to the product of the two populations, i.e., there are constants a and b such that

$$x' = a\,x - b\,x\,y. \tag{8.24}$$

Since the predator population depends on the prey population for its food supply it is natural to assume that in the absence of the prey population, the predator population would actually decrease, i.e. the growth rate

would be negative. Furthermore the (negative) growth rate is proportional to the population. The presence of the prey population would provide a source of food, so it would increase the growth rate of the predator species. By the same reasoning used for the prey species, this increase would be proportional to the product of the two populations. Thus, there are constants c and d such that

$$y' = -c\,y + d\,x\,y. \tag{8.25}$$

a) A typical example would be with the constants given by $a = 0.4$, $b = 0.01$, $c = 0.3$, and $d = 0.005$. Start with initial conditions $x_1(0) = 50$ and $x_2(0) = 30$, and compute the solution to (8.22) and (8.23) over the interval [0, 100]. Prepare both a time plot and a phase plane plot.

After Volterra had obtained his model of the predator-prey populations, he improved it to include the effect of "fishing," or more generally of a removal of individuals of the two populations which does not discriminate between the two species. The effect would be a reduction in the growth rate for each of the populations by an amount which is proportional to the individual populations. Furthermore, if the removal is truly indiscriminate, the proportionality constant will be the same in each case. Thus, the model in equations (8.22) and (8.23) must be changed to

$$\begin{aligned} x' &= a\,x - b\,x\,y - e\,x, \\ y' &= -\,c\,y + d\,x\,y - e\,y, \end{aligned} \tag{8.26}$$

where e is another constant.

b) To see the effect of indiscriminate reduction, compute the solutions to the system in (8.24) when $e = 0$, 0.01, 0.02, 0.03, and 0.04, and the other constants are the same as they were in part a). Plot the five solutions on the same phase plane, and label them properly.

c) Can you use the plot you constructed in part b) to explain why the fishermen caught more sharks during World War I? You can assume that because of the war they did less fishing.

Student Projects

The following problems are quite involved, with multiple parts, and thus are good candidates for individual or group student projects.

1. **The Non-linear Spring and Duffing's Equation.**

A more accurate description of the motion of a spring is given by *Duffing's equation*

$$my'' + cy' + ky + ly^3 = F(t).$$

Here m is the mass , c is the damping constant, k is the spring constant, and l is an additional constant which reflects the "strength" of the spring. Hard springs satisfy $l > 0$, and soft springs satisfy $l < 0$. As usual, $F(t)$ represents the external force.

Duffing's equation cannot be solved analytically, but we can obtain approximate solutions numerically in order to examine the effect of the additional term ly^3 on the solutions to the equation. For the following exercises, assume that $m = 1$ kg, that the spring constant $k = 16$ N/m, and that the damping constant is $c = 1$ kg/sec. The external force is assumed to be of the form $F(t) = A\cos(\omega t)$, measured in Newtons, where ω is the frequency of the driving force. The natural frequency of the spring is $\omega_0 = \sqrt{k/m} = 4$ rad/sec.

a) Let $l = 0$ and $A = 10$. Compute the solution with initial conditions $y(0) = 1$, and $y'(0) = 0$ on the interval [0, 20], with $\omega = 3.5$ rad/sec. Print out a graph of this solution. Notice that the steady-state part of the solution dominates the transient part when t is large.

b) With $l = 0$ and $A = 10$, we want to compute the amplitude of the steady-state solution. The amplitude is the maximum of the values of $|y(t)|$. Because we want the amplitude of the steady-state oscillation, we only want to allow large values of t, say $t > 15$. This will allow the transient part of the solution to decay. To compute this number in MATLAB, do the following. Suppose that `Y` is the vector of y-values, and `T` is the vector of corresponding values of t. At the MATLAB prompt type

```
max(abs(Y.*(T>15)))
```

Your answer will be a good approximation to the amplitude of the steady-state solution.

Why is this true? The expression `(T>15)` yields a vector the same size as `T`, and an element of this vector will be 1 if the corresponding element of `T` is larger than 15 and 0 otherwise. Thus `Y.*(T>15)` is a vector the same size as `T` or `Y` with all of the values corresponding to $t \leq 15$ set to 0, and the other values of `Y` left unchanged. Hence, the maximum of the absolute values of this vector is just what we want.

Set up a script M-file that will allow you to do the above process repeatedly. For example, if the derivative M-file for Duffing's equation is called `duff.m`, the script M-file could be

```
[t,y] = ode45(@duff'0,20,[1,0]);
y  =  y(:,1);
amplitude  =  max(abs(y.*(t>15)))
```

Now by changing the value of ω in `duff.m` you can quickly compute how the amplitude changes with ω. Do this for eight evenly spaced values of ω between 3 and 5. Use `plot` to make a graph of amplitude vs. frequency. For approximately what value of ω is the amplitude the largest?

c) Set $l = 1$ (the case of a hard spring) and $A = 10$ in Duffing's equation and repeat b). Find the value of ω, accurate to 0.2 rad/sec, for which the amplitude reaches its maximum.

d) For the hard spring in c), set $A = 40$. With $y(0) = y'(0) = 0$, and for eight (or more) evenly spaced values of ω between 5 and 7 compute the magnitude of the steady-state oscillation. Plot the graph of amplitude vs. frequency. Repeat the computation with initial conditions $y(0) = 6$, $y'(0) = 0$, and plot the two graphs of amplitude vs. frequency on the same figure. (The phenomenon you will observe is called Duffing's hysteresis.)

e) Set $l = -1$ and $A = 10$ (the case of a soft spring) in Duffing's equation and repeat c).

2. **The Lorenz System.**

The purpose of this exercise is to explore the complexities displayed by the Lorenz system as the parameters are varied. We will keep $a = 10$, and $b = 8/3$, and vary r. For each value of r, examine the behavior of the solution with different initial conditions, and make conjectures about the limiting behavior of the solutions as $t \to \infty$. The solutions should be plotted in a variety of ways in order to get the information. These include time plots, such as Figure 8.9, and phase space plots, such as Figure 8.10. In the case of phase space plots, use the `rotate3d` command to capture a view of the system that provides the best visual information. You might also use other plots, such as z versus x, etc.

Examine the Lorenz system for a couple of values of r in each of the following intervals. Describe the limiting behavior of the solutions.

a) $0 < r < 1$.

b) $1 < r < 470/19$. There are two distinct cases.

c) $470/19 < r < 130$. The case done in the text. This is a region of chaotic behavior of the solutions.

d) $150 < r < 165$. Things settle down somewhat.

e) $215 < r < 280$. Things settle down even more.

3. **Motion Near the Lagrange Points.**

Consider two large spherical masses of mass $M_1 > M_2$. In the absence of other forces, these bodies will move in elliptical orbits about their common center of mass (if it helps, think of these as the earth and the moon). We will consider the motion of a third body (perhaps a spacecraft), with mass which is negligible in comparison to M_1 and M_2, under the gravitational influence of the two larger bodies. It turns out that there are five equilibrium points for the motion of the small body relative to the two larger bodies. Three of these were found by Euler, and are on the line connecting the two large bodies. The other two were found by Lagrange and are called the Lagrange points. Each of these forms an equilateral triangle in the plane of motion with the positions of the two large bodies. We are interested in the motion of the spacecraft when it starts near a Lagrange point.

In order to simplify the analysis, we will make some assumptions, and choose our units carefully. First we will assume that the two large bodies move in circles, and therefore maintain a constant distance from each other. We will take the origin of our coordinate system at the center of mass, and we will choose rotating coordinates, so that the x-axis always contains the two large bodies. Next we choose the distance between the large bodies to be the unit of distance, and the sum of the the two masses to be the unit of mass. Thus $M_1 + M_2 = 1$. Finally we choose the unit of time so that a complete orbit takes 2π units; i.e., in our units, a year is 2π units. This last choice is equivalent to taking the gravitational constant equal to 1.

With all of these choices, the fundamental parameter is the relative mass of the smaller of the two bodies

$$\mu = \frac{M_2}{M_1 + M_2} = M_2.$$

Then the location of M_1 is $(-\mu, 0)$, and the position of M_2 is $(1 - \mu, 0)$. The position of the Lagrange point is $(x_0, y_0) = ((1 - 2\mu)/2, \sqrt{3}/2)$. If (x, y) is the position of the spacecraft, then the distances to M_1 and M_2 are r_1 and r_2 where

$$r_1^2 = (x + \mu)^2 + y^2,$$
$$r_2^2 = (x - 1 + \mu)^2 + y^2.$$

Finally, Newton's equations of motion in this moving frame are

$$\begin{aligned} x'' - 2y' - x &= -(1 - \mu)(x + \mu)/r_1^3 - \mu(x - 1 + \mu)/r_2^3, \\ y'' + 2x' - y &= -(1 - \mu)y/r_1^3 - \mu y/r_2^3. \end{aligned} \tag{8.27}$$

a) Find a system of four first order equations which is equivalent to system (8.25).

b) The Lagrange points are stable for $0 < \mu < 1/2 - \sqrt{69}/18) \approx 0.03852$, and in particular for the earth/moon system, for which $\mu = 0.0122$. Starting with initial conditions which are less than $1/100$ unit away from the Lagrange point, compute the solution. Write $x = (1 - 2\mu)/2 + \xi$, and $y = \sqrt{3}/2 + \eta$, so that $(\xi, \eta) = (x, y) - (x_0, y_0)$ is the position of the spacecraft relative to the Lagrange point. For each solution that you compute, make a plot of η vs. ξ to see the motion relative to the Lagrange point, and a plot of y vs. x, which also includes the positions of M_1 and M_2 to get a more global view of the motion.

c) Examine the range of stability by computing and plotting orbits for $\mu = 0.037$ and $\mu = 0.04$.

d) What is your opinion? Assuming μ is in the stable range, are the Lagrange points just stable, or are they asymptotically stable?

e) Show that there are five equilibrium points for the system you found in a). Show that there are three on the x-axis (the Euler points), and two Lagrange points as described above. This is an algebraic problem of medium difficulty. It is not a computer problem unless you can figure out how to get the Symbolic Toolbox to find the answer.

f) Show that the Euler points are always unstable. This is a hard algebraic problem.

g) Show that the Lagrange points are stable for $0 < \mu < 1/2 - \sqrt{69}/18)$ (hard). Decide whether or not these points are asymptotically stable (extremely difficult).