Contents

6. Solving ode’s in MATLAB ................................................................. 1
   Single first order differential equations ........................................ 1
   Systems of first order equations .................................................. 3
   Second order differential equations ............................................. 6
   The Lorenz system ...................................................................... 7
   Computational problems and some solutions .................................. 9
   Exercises. ................................................................................. 14
6. Solving ode’s in MATLAB

This is one of the most important chapters in this manual. Here we will describe how to use MATLAB’s own solvers to find approximate solutions to almost any system of differential equations. The methods described herein are used regularly by engineers and scientists, and are available in any version of MATLAB. In addition to being very powerful, they are easy to use, as the reader will discover.

As this Manual is going to print, a new suite of ode solvers is being developed for MATLAB. The first thing the reader should do is to find out whether the new suite is available or not. To do this enter `help funfun` at the command line. MATLAB will print out a list of functions that act on other functions. Included in the list there will be some that solve differential equations. The list will certainly contain `ode23` and `ode45`. If, in addition, the list contains `ode113`, `stf15`, or `stf23`, then you have the new suite of solvers installed, otherwise you do not.

In either case there will be a solver named `ode45`. This is the one we recommend for use with this Manual. Although the old and the new versions have the same name, the usage syntax is slightly different, and the two solvers operate slightly differently. We will carefully point out the difference in the syntax, but we will not explore the internal workings of the routines.

Although we will almost exclusively use `ode45`, it is important to realize that the calling syntax is the same for the other solvers. I.e., in the old version `ode23` and `ode45` use the same syntax, and in the new version all five of the solvers use the same syntax. We will spend a paragraph discussing the use of `stf15` at the end of the chapter.

**Single first order differential equations**

The basic syntax for using `ode45` is

\[
[t,x] = \text{ode45}('yprime',t0,tf,y0); \quad \% \text{Old ode45}
\]

or

\[
[t,x] = \text{ode45}('yprime',t0,tf,y0); \quad \% \text{New ode45}
\]

Here we are solving the equation \( y' = yprime(t,y) \), with the initial condition \( y(t_0) = y_0 \), on the interval \( t_0 \leq t \leq t_f \). The description of `yprime` must be in a derivative m-file with the name `yprime.m`. The format for derivative m-files is described in Chapter 5; they need the same format for both versions of `ode45` as they need for `eul`, `rk2`, or `rk4`.

Notice that the only difference in the syntax is that in the new version, the initial and final times \( t_0 \) and \( t_f \) must be put into a vector \([t_0,t_f]\).
Both versions of ode45 use variable step Runge-Kutta procedures. 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 the \( k^{th} \) step should be less than a predetermined amount. In the old version of ode45 this means that

\[
|\text{estimated error}| \leq \max(1, |y_k|) \times \text{tolerance}, \tag{1}
\]

where \( y_k \) is the calculated solution at step \( k \). The default value of tolerance is \( 10^{-6} \). In the new version of ode45 the estimated error is required to satisfy a similar, but more complicated inequality, which involves two tolerance parameters. We will discuss this later in this chapter. For the uses in this Manual the default values of the tolerance parameters will provide sufficient accuracy.

The output of ode45 (both versions) consists of the column vector \( t \), and the matrix \( x \). \( t \) is a list of the values of the variable \( t \) at which the approximate solution has been calculated. In the case of a single, first order equation, \( x \) is also a column vector. Each entry is the approximate solution at the corresponding value of \( t \).

As an example, consider the equation \( \dot{x} = \cos(t)/(2x - 2) \), with initial condition \( x(0) = 3 \), on the interval \([0, 2\pi]\). To solve this with ode45 we enter:

\[
[t,x]=\text{ode45}('test',0,2*\pi,3); \quad \% \text{old version}
\]

or

\[
[t,x]=\text{ode45}('test',[0,2*\pi],3); \quad \% \text{new version}
\]

where \text{test.m} is the m-file which contains the definition of the equation, i.e.,

\[
\begin{align*}
\text{function} & \quad \text{xpr} = \text{test}(t,x) \\
\text{xpr} & = \cos(t)/(2*(x-1));
\end{align*}
\]

To see the answer, we can enter \( t \) at the MATLAB prompt followed by \( x \). We discover that both \( t \) and \( x \) are column vectors with the same number of entries. If you are using the new version of MATLAB, these vectors will probably be longer than can be displayed on a screen. In this case enter \( t(1:20) \) and \( x(1:20) \), which will display the first twenty entries of \( t \) and \( x \).

A better way to see the relationship between \( t \) and \( x \) is to print them to the screen side by side. This can be accomplished by realizing that the command \([t,x]\) will have MATLAB put the two column vectors into one matrix. Entering \([t,x]\) (or \([t(1:20), \ x(1:20)]\)) at the MATLAB prompt will print the two side by side. Try it.
Figure 6.1. A solution using ode45.

Plotting the solution is quite easy. The following will do that and make the graph look pretty.

\begin{verbatim}
>> plot(t, x)
>> title('The solution to $x' = \frac{\cos(t)}{(2x-2)}$, with $x(0) = 3.$')
>> xlabel('t')
>> ylabel('x')
>> grid
\end{verbatim}

The result (for the old version of ode45) is shown in Figure 6.1.

**Systems of first order equations**

Actually, systems are no harder to handle using ode45 than are single equations. As an example, we will solve the system

\[
\begin{align*}
x_1' &= x_2 - x_1^2 \\
x_2' &= -x_1 - 2x_1x_2
\end{align*}
\]

numerically on the interval $[0, 10]$, with initial conditions $x_1(0) = 0$ and $x_2(0) = 1$. It is first necessary to describe the system in a derivative m-file. We will use the vector $x$ in MATLAB to
denote the solution. The first component, which in MATLAB is denoted by $x(1)$, will correspond to $x_1$. Similarly $x(2)$ will correspond to $x_2$. For the derivatives we will use the vector $\mathbf{x}'$, with the first component corresponding to $x_1'$, and the second to $x_2'$. Then our system of differential equations can be entered into an m-file as follows:

```matlab
function xpr=test2(t,x);
    xpr(1)=x(2)-x(1)^2;
    xpr(2)=-x(1)-2*x(1)*x(2);
```

![Figure 6.2. Solution to a system using ode45.](image)

Now, to solve this system we use almost the same command as before:

```matlab
[t,x]=ode45('test2',0,10,[0,1]); % Old version.
```

or

```matlab
[t,x]=ode45('test2',[0,10],[0,1]); % New version.
```

The only difference is that for the initial value we must have a vector $[0,1]$, which is the same as $(x_1(0),x_2(0))$. 
The output vector \( t \) is similar to what it was in the earlier case. It is a vector containing the list of the \( t \) values where the approximate solutions were computed. Print it to your screen by entering \( t \) at the MATLAB prompt to see what it looks like. It will be quite long. You might want to check the size of \( t \) by entering \( \text{size}(t) \). To see the first twenty entries, enter \( t(1:20) \).

This time \( x \) is really a matrix instead of a column vector. To see the first twenty rows, enter \( x(1:20,:) \) at the MATLAB prompt. Each row of this matrix is a vector containing the approximate solution at the corresponding value of \( t \). The first component of this row vector is the approximate value of \( x_1 \), and the second is that of \( x_2 \).

To plot both components of the solution as functions of \( t \), it is only necessary to enter \( \text{plot}(t,x) \). The results are shown in Figure 6.2. If only the first component of the solution is wanted, enter \( \text{plot}(t,x(:,1)) \). The colon in the notation \( x(:,1) \) indicates that we want all rows, and the 1 that we want the first column. Similarly, if only the second component is wanted, enter \( \text{plot}(t,x(:,2)) \). This is an example of the very sophisticated subscripting options available in MATLAB.

It is also possible to plot the components against each other with the command

\[
\text{plot}(x(:,1),x(:,2))
\]

The result is called a phase plane plot, and it is shown in Figure 6.3.

![Figure 6.3. The phase plane plot of the solution.](image)

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 \( t \). MATLAB does
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

$$y'' = f(t, y, y'),$$

we set $x_1 = y$, and $x_2 = y'$. Then $\mathbf{x} = (x_1, x_2)$ is a solution to the first order system

$$\begin{align*}
x_1' &= x_2 \\
x_2' &= f(t, x_1, x_2).
\end{align*}$$

More importantly, if $\mathbf{x} = (x_1, x_2)$ is a solution of the system in (3), 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 (2).

As a concrete example, consider the nonlinear equation $y'' + y y' + y = 0$, with initial conditions $y(0) = 0$, and $y'(0) = 1$. The corresponding system is

$$\begin{align*}
x_1' &= x_2 \\
x_2' &= -x_1x_2 - x_1.
\end{align*}$$

Thus, we must create the m-file

```
function xpr=test3(t,x);
    xpr(1)=x(2);
    xpr(2)=-x(1)*x(2)-x(1);
```

and to solve the initial value problem on the interval $[0, 10]$, we enter

```
>> [t,x]=ode45('test3',0,10,[0,1]);  \% Old version.
```

or

```
>> [t,x]=ode45('test3',[0,10],[0,1]);  \% New version.
```

To plot the solution $y$, we simply remember that $y = x_1$. Thus, $\mathbf{x}(\cdot, 1)$ represents the solution, and we enter `plot(t,x(:,1))`. 

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 as $y$, and $z$. 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 as $x$ and $y$. 

2. The corresponding system is

$$\begin{align*}
x_1' &= x_2 \\
x_2' &= f(t, x_1, x_2).
\end{align*}$$

2. For example, enter `plot3(t,x(:,1),x(:,2))` to see the solution with $t$ along the $z$-axis, and the two components of the solution as $x$ and $y$. 

2. 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 as $x$ and $y$. 

2. To plot the solution $y$, we simply remember that $y = x_1$. Thus, $\mathbf{x}(\cdot, 1)$ represents the solution, and we enter `plot(t,x(:,1))`. 

2. 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 as $y$, and $z$. 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 as $x$ and $y$. 

2. 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 as $x$ and $y$. 

2. To plot the solution $y$, we simply remember that $y = x_1$. Thus, $\mathbf{x}(\cdot, 1)$ represents the solution, and we enter `plot(t,x(:,1))`. 

2. 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 as $y$, and $z$. 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 as $x$ and $y$. 

2. 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 as $x$ and $y$. 

2. To plot the solution $y$, we simply remember that $y = x_1$. Thus, $\mathbf{x}(\cdot, 1)$ represents the solution, and we enter `plot(t,x(:,1))`.
The function \texttt{ode45} is set up to solve systems of first order differential equations. We have just seen how a second order equation can be replaced by an equivalent first order system, and then solved. This method is completely general. Any system of ordinary differential equations can be replaced by an equivalent first order system. It is only necessary to add new variables for derivatives of the original variables, just as we did in this example. As a result, \texttt{ode45} has the capability of solving almost any system of ordinary differential equations.

### The Lorenz system

There is no fixed upper bound on the number of equations in a system of first order differential equations which can be solved by MATLAB. As an example we will solve the Lorenz system. This is a system of three equations which was 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

\[
\begin{align*}
    x' &= -ax + ay \\
    y' &= rx - y - xz \\
    z' &= -bz + xy
\end{align*}
\]  

where \(a\), \(b\), and \(r\) are positive constants.

![A solution to the Lorenz system](image)

**Figure 6.4.** A solution to the Lorenz system.
In MATLAB we will use the vector \( \mathbf{u} \), where \( \mathbf{u}(1) \) corresponds to \( x \), \( \mathbf{u}(2) \) to \( y \), and \( \mathbf{u}(3) \) to \( z \). The vector \( \mathbf{upr} \) will be used for the corresponding derivatives. Then the derivative m-file \( \text{lор.m} \) could look like:

```matlab
function upr = lor(t,u)

global AA BB RR

upr(1) = -AA*u(1) + AA*u(2);
upr(2) = RR*u(1) - u(2) -u(1)*u(3);
upr(3) = -BB*u(3) + u(1)*u(2);
```

(It would be natural to name this file \( \text{lorenz.m} \), but there is already an m-file with that name. To see what it does, enter \( \text{lorenz} \) at the MATLAB prompt. What you will see is a solution to the Lorenz system (2).)

The new feature here is the use of global variables. The symbols \( AA \), \( BB \), and \( RR \) correspond to the parameters \( a \), \( b \), and \( r \) in the system in (2). The line \( \text{global AA BB RR} \) declares them to be global variables. This means that, if we enter the same line in the Command Window, and then assign \( AA \), \( BB \), and \( RR \) values in the Command Window, these values will also be assigned to them whenever the derivative m-file \( \text{lor.m} \) is used. Hence we can change the values of the parameters in the Command Window, and we do not have to rewrite the m-file every time we want to make a change. For parameters in a differential system, this can save a lot of time.

![The Lorenz attractor.](image)

**Figure 6.5.** The Lorenz attractor.
We will use $a = 10$, $b = 8/3$, and $r = 28$ at the start. Then to solve the Lorenz system over the interval $[0, 7]$, with initial values $x(0) = 1$, $y(0) = 2$, and $z(0) = 3$, we proceed as follows:

```matlab
>> global AA BB RR
>> AA = 10; BB=8/3; RR=28;
>> [t,u] = ode45('lor', [0,7], [1,2,3]);
```

We can plot the solution with the command `plot(t,u)`. After some labeling we get what is shown in Figure 6.4.

In Figure 6.4, there appears to be transient behavior in the interval $0 \leq t \leq 1$. Let’s compute the solution over a longer period, say $0 \leq t \leq 20$. If your computer is very fast, use the upper bound of 100 instead of 20. We will then plot the part corresponding to $t > 1$ in three dimensions. This is easily accomplished:

```matlab
>> [t,u] = ode45('lor', [0,100], [1,2,3]); % New version.
>> N= find(t>1);
>> v=u(N,:);
>> plot3(v(:,1),v(:,2),v(:,3))
```

Some explanation is necessary. The command `N= find(t>1);` produces a list of the indices of those elements of the vector `t` where $t > 1$. Then `v=u(N,:);` produces a matrix containing the rows of `u` with indices in `N`; i.e., `v` consists of only those solution points corresponding to $t > 1$.

After a little grooming we get Figure 6.5, which is a fine picture 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. Redo the above computation, and plot with other choices of initial conditions, and compare with Figure 6.5.

**Computational problems and some solutions**

Both versions of the routine `ode45` are extremely flexible, fast, and accurate. However, the computational solution of all differential equations cannot be achieved with just one solution method. Here we will discuss some of the problems that can arise. We will also indicate how to solve some of these problems.

**How accurate is `ode45` really?**

This is a very good question. The description earlier of how the old version of the routine works (see (1)) could lead one to believe that an estimate of the sort

$$|\text{error}| \leq \max(1, |y|) \times \text{tolerance}$$
is valid. It would be wonderful if this were true, and for many examples it is true. Still there are examples that show different results, even over intervals of relatively short length. It is a good idea to introduce a safety factor, so a more reliable estimate of the error would be something like

\[ |\text{error}| \leq 100 \times \max(1, |y|) \times \text{tolerance}. \] (5)

The user can use this estimate to decide which tolerance should be used to achieve a desired upper bound on error.

To change the tolerance in the old version of \texttt{ode45}, it is only necessary to add it as an extra parameter in the calling syntax. For example,

\[
[t,x] = \texttt{ode45('test',0,30*pi,3,1e-8)};
\]

would set the tolerance to \(10^{-8}\).

The new version of \texttt{ode45} takes a more sophisticated approach to error checking. If \(y^k\) 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 error vector, which has a component for every component of \(y\), and it is required that each component of the error vector satisfy one of the following two inequalities:

\[
|\text{estimated error}^k_j| \leq \max(|y^k_j|, |y^{k-1}_j|) \times \text{rtol}
\]

\[
|\text{estimated error}^k_j| \leq \text{atol}_j
\]

Here rtol is called the relative tolerance, and atol is the absolute tolerance. Notice that the absolute tolerance is a vector quantity, with a component for each equation in the system being solved.

Just as before, the user can use these inequalities to determine the values of rtol and atol which are required to achieve the desired accuracy. The default values are rtol = \(10^{-3}\), and each component of atol = \(10^{-6}\).

To change the tolerances, it is necessary to use an options vector, which we will denote by \texttt{opt}. There are a large number of possible options for \texttt{ode45} and the other solvers in the new suite. To change any of them from the default values, we use the command \texttt{intopt}. For example, in the solution to the Lorenz system, we might want to use rtol = 1e-4, and atol = \([1e-5, 1e-6, 1e-6]\). To accomplish this we enter

\[
\texttt{opt} = \texttt{intopt('rtol',1e-4,'atol',[1e-5, 1e-6, 1e-6])};
\]

\[
[t,u] = \texttt{ode45('lor',[0,20],[1,2,3],opt)};
\]

If we only wanted to change rtol, the first line should be

\[
\texttt{opt} = \texttt{intopt('rtol',1e-4)};
\]

In other words, we only specify those options which need to be changed from their default values.
The user has to be especially careful in using the estimate (5) for the old version of ode45 with systems of equations. Different components of the solution vector might have markedly different sizes, and one or more of the components might be lost in the estimate. The term |y| will be dominated by the largest components of the vector y, and the term |error| will likewise be dominated by the largest components of the vector error. The error made in calculating the smallest component might be very large relative to the size of the smallest component, and still be small in comparison to the errors in the larger components. This possibility is removed by the more sophisticated error estimation in the new version of ode45.

There is still a way in which excessive error can sneak up on the user. Consider the situation where it is the difference of two components of the solution which is really important. In that case each component has to be calculated to greater accuracy to ensure that the difference is sufficiently accurate.

The considerations to this point were for solutions over “relatively short” intervals of the independent variable. When solutions are required over long intervals, we have a whole new ballgame. No solver can give reliable results over long intervals for all equations. For example, many equations have regions where solutions are extremely sensitive to initial conditions. This means that two solutions with initial conditions which are very close can eventually move very far apart. From a computational point of view, it must be recognized that small errors are being made all the time. Even if the error is very small, in a region of extreme sensitivity to initial conditions, the error might be enough to move to a solution which is diverging from the one sought. This is in the nature of differential equations, and cannot be anticipated by a solver without the intervention of the user.

When computing over a relatively large interval, no solution should be accepted uncritically. Compare the solution to what is expected. If nothing else, compute the solution again with a decreased tolerance, and compare the two solutions. For the new version of ode45, there is another option that is sometimes useful when computing over large intervals. This is hmax, the maximum allowed integration step. The default value is $(tf-t0)/10$ where t0 is the initial time, and tf is the final time. Reducing the value of hmax can sometimes assist the computation.

**Behavior near singularities**

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

1. ode45 can integrate right through the singularity, not even realizing it is there. In this case the accuracy of the result is highly doubtful, especially in that range beyond the singularity. This phenomenon will happen, for example, with the initial value problem $x' = x/(t - 1)$, with initial value $x(0) = 1$ on the interval $[0, 2]$.
2. The old version of ode45 can find the singularity and report it with a comment like

   **Singularity likely at t = 0.972656**

What this comment really means is that an extremely small step size was required to achieve the error limit. This is usually a sign of a singularity but not always. The new version is
more circumspect. It’s comment is

**Step failure at 1.129671e+01 with a minimum step size of 1.003350e-13**

Either of these messages probably indicate the presence of a singularity in the solution. An example of this is $x' = x/(1 - t)$, with initial value $x(0) = 1$ on the interval $[0, 1]$. One very nice thing about ode45 is that if this happens, the output is still available and can be analyzed or used as the user desires.

3. **ode45** can choose smaller and smaller step sizes upon the approach to a singularity 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.

**Kinky plots**

This is a problem with the old version of ode45. It can sometimes complete its computation in relatively few steps. This happens when ode45 decides to use large step sizes, perhaps as large as the maximum allowed step size, which is $h_{\text{max}} = (t_f - t_0)/16$. Conceivably a computation could be completed in as few as 16 steps. Together with the initial point, this gives us only 17 data points. Plotting a curve using 17 points in MATLAB usually results in a figure that looks smooth, but sometimes the result is a little kinky. As an example, the graph in Figure 6.1 is not as smooth as might be desired. The easiest solution for this problem is to use a smaller tolerance, which will force the use of a smaller step size, and will provide more data points.

![Figure 6.6. The effect of the refine option.](image)
With the new version of ode45 there is almost never a problem with kinky plots. The new version can still use large step sizes, and it can take as few as 10 steps to compute a solution. However, the routine interpolates between the solution at the step points. The interpolation process uses the differential equation to compute the interpolating points, so they remain almost as accurate as the solution at the step points.

The number of interpolating points is controlled by the option refine. The default value of refine is 4. This should be sufficient to insure that most solution curves are smooth. If it should happen that a solution curve is not smooth, the solution can be recomputed with a larger value of refine. As an example consider the first equation in this chapter, \( x' = \cos(t)/(2t - 2) \). This time we solve it over the interval \([0, 10\pi]\). The solution is periodic with period \(2\pi\), so we get five periods. One would expect the graph of such a function to be kinky, unless a large number of points were plotted. Indeed, the default value of refine does lead to a somewhat kinky graph of the solution. So, let’s set refine = 8. The following list of commands

```plaintext
>> [t,x] = ode45('test',[0,10*pi],3,intopt('refine',8));
>> l=length(t)-1)/8;    % The number of steps.
>> tp=t(8*[0:1]+1);    % The values of t at the step points.
>> xp=x(8*[0:1]+1);    % The values of x at the step points.
>> plot(t,x,tp,xp,'o',tp,xp,'--')
```

results in Figure 6.6. In this figure, the solid curve is the plot of the entire solution, and the circles represent the solution at the step points. Consequently, the dashed curve is what we would get without the refine option.

**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 in this manual, but we will say a word about them in case the reader should have to deal with one.

An example of a stiff system is the van der Pol equation

\[
x' = y \\
y' = -x + \mu(1 - x^2)y
\]

when the parameter \( \mu \) is very large, say \( \mu = 1000 \).

Runge-Kutta solution methods do not work well with stiff equations. The fast varying component requires a Runge-Kutta algorithm to use very small steps — so small that it takes a prohibitively long time to compute the solution. The new suite of ode solvers contains two routines, stff15 and stff23 which are designed to solve stiff equations. stff15 is the first of these to try.

A very attractive feature of the new suite of solvers is that the calling syntax is the same for all of them. Thus the syntax for using stff15 is the same as we have been using with ode45. For example,

```plaintext
[t,x] = stff15('vdpol',tspan,init);
```
will compute the solution of the van der Pol system, if the system is described in `vdpol.m`. `tspan = [t0, tf]` is the interval of computation, and `init = [x0, y0]` is the initial conditions. These routines also take an options vector as an additional parameter, and the makeup of that is again the same as it is for `ode45`. In particular the same options are available. There is one change — the default value of `refine` is 1.

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 `stf15`.

**What about solving in the negative direction in t?**

This is simply not allowed by the old version of `ode45`. Furthermore, there does not seem to be an easy way around it. If it is necessary to solve in the negative direction, rewrite the differential equation by replacing `t` with `-t`, and then solve as usual. Of course it will then be necessary to change the sign of the vector `t`.

The new version can solve in the negative direction without a hitch.

**Exercises.**

1. Use `ode45` to solve the following initial value problems numerically over the indicated intervals. Superimpose the solutions with different initial values on the same time plot.
   a) \[ x' = x^2 - t, \quad x(0) = 0, 0.5, -0.5, 0 \leq t \leq 4. \]
   b) \[ x' = \cos(t)/(2 + \cos(t)), \quad x(0) = 0, 1, \pi/2, 4, -\pi/2, -2, 0 \leq t \leq 15. \]
2. Use `ode45` to compute a solution to the initial value problem \( \dot{x} = x/(1 + x), \quad x(0) = 1 \) on the interval \([0, 3]\). Plot the solution. This equation cannot be solved explicitly, but since it is separable an implicit solution can be found. Check how closely the computed solution satisfies the implicit equation.
3. Using `ode45` calculate the solutions to the differential equation \( y'' - (y')^2 - y^2 = 0 \) with each of the following initial conditions. In each case make a time plot and a phase plane plot of the results.
   a) \( y(0) = e, \quad y'(0) = -e. \)
   b) \( y(0) = 1, \quad y'(0) = -1. \)
   c) \( y(0) = 1, \quad y'(0) = 0. \)
   d) \( y(0) = 1, \quad y'(0) = 3. \)
   e) Find the general solution to the equation \( yy'' - (y')^2 - y^2 = 0. \) (Hint: Calculate \((y'/y)'.\))
4. The equation \( x'' = ax' - bx^3 - kx \) was devised by Lord Raleigh to model the motion of the reed in a clarinet. With \( a = 5, \quad b = 4, \quad k = 5 \), 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 \( 2 \). 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.
5. Consider the system of equations

\[
\begin{align*}
    x_1' &= x_2 \\
    x_2' &= -x_1
\end{align*}
\]

with initial conditions \( x_1(0) = 1, x_2(0) = 0 \). Find the exact solution. Then solve the system numerically over the interval \([0, 4\pi]\) using both \texttt{eul} with step size \( h = 0.01 \) and \texttt{ode45} with the default tolerance. Plot the phase plane for both solutions. This will give you a different picture of the errors made in the two methods.

6. A predator-prey population model.

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 would depend 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

\[
x' = ax - bxy.
\]

(6)

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, i.e., there are constants \( c \) and \( d \) such that

\[
y' = -cy + dxy.
\]

(7)

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 (3) and (4) 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 (3) and (4) must be changed to

\[
\begin{align*}
    x' &= ax - bxy - ex \\
    y' &= -cy + dxy - ey
\end{align*}
\]

(8)

where \( e \) is another constant.

b) To see the effect of indiscriminate reduction, compute the solutions to the system in (5) 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.

7. **Harmonic motion.**

The equation for the motion of a spring is

\[ my'' + cy' + ky = F(t), \]

where \( m \) is the mass, \( c \) is the damping constant, and \( k \) is the spring constant. \( F(t) \) represents the external force. For the following exercises, assume that \( m = 1 \text{kg} \), and that the spring constant \( k = 16 \text{N/m} \). We will be concerned with unforced oscillations, so we are assuming that \( F(t) = 0 \). In each of the following cases compute the solution with initial conditions \( x(0) = 1 \), and \( \dot{x}(0) = 0 \) over the interval \([0, 20]\). Prepare both a time plot and a phase plane plot.

a) (No damping) \( c = 0 \).

b) (Under damping) \( c = 2 \).

c) (Critical damping) \( c = 8 \).

d) (Over damping) \( c = 10 \).

8. **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 \( m = 1 \text{kg} \), that the spring constant \( k = 16 \text{N/m} \), and that the damping constant is \( c = 1 \text{kg/sec} \). The external force is assumed to be of the form \( F(t) = A \cos(\omega t) \), measured in Newtons, where \( \omega \) is the frequency of the driving force. The natural frequency of the spring is \( \omega_0 = \sqrt{k/m} = 4 \text{ rad/sec} \).

a) Let \( l = 0 \) and \( A = 10 \). Compute the solution with initial conditions \( y(0) = 1 \), and \( \dot{y}(0) = 0 \) on the interval \([0, 20]\), with \( \omega = 3.5 \text{ rad/sec} \). 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 \( A = 10 \), compute the amplitude of the steady state solution as follows. 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 \( \mathbf{Y} \) is the vector of \( y \)-values, and \( \mathbf{T} \) is the vector of corresponding values of \( t \). At the MATLAB prompt type

\[ \text{max(abs(Y.*(T>15)))} \]

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

Why is this true? The expression \( \text{(T>15)} \) yields a vector the same size as \( \mathbf{T} \), and an element of this vector will be 1 if the corresponding element of \( \mathbf{T} \) is larger than 15 and 0 otherwise. Thus \( \mathbf{Y} \cdot \text{(T>15)} \) is a vector the same size as \( \mathbf{T} \) or \( \mathbf{Y} \) with all of the values corresponding to \( t \leq 15 \) set to 0, and the other values of \( \mathbf{Y} \) 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

\[
[t,y]=\text{ode45}([\text{duff'}],0,20,[1,0]);
y = y(:,1);
amplitude = \max(\text{abs}(y.*(t>15)))
\]

Now by changing the value of \( \omega \) in `duff.m` you can quickly compute how the amplitude changes with \( \omega \). Do this for eight evenly spaced values of \( \omega \) between 3 and 5. Use `plot` to make a graph of amplitude vs. frequency. For approximately what value of \( \omega \) 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 \( \omega \), accurate to 0.2 rad/sec, for which the amplitude reaches its maximum.

d) For the hard spring in c), set \( A = 40 \). You are to redo c), but with two different choices of initial conditions, and for eight evenly spaced values between 5 and 7. The initial conditions are \( y(0) = y'(0) = 0 \), and \( y(0) = 6, y'(0) = 0 \). 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).

9. 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 6.4, and phase space plots, such as Figure 6.5. You might use plots of \( z \) versus \( x \), etc. You might also use the techniques of the following exercise.

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.

10. Viewing the Lorenz attractor in three dimensions, as we do in Figure 6.5, is less than completely satisfactory. We really need to view the set from several different viewpoints in order to see what it really looks like. This is easily possible in MATLAB. The command `view` allows us to change the viewpoint of a three dimensional plot. The syntax is `view(azimuth, elevation)`, where elevation is the angle of the viewpoint above the \((x, y)\) plane, and azimuth is the angle of the viewpoint in the \((x, y)\) plane, measured counterclockwise from the \(x\)-axis. Both angles are measured in degrees. Entering `view(az, el)` causes the plot to be redone from the new viewpoint.

a) Look at a Lorenz attractor from a variety of viewpoints, and print the two you think are most revealing.

This process can be carried further using MATLAB’s movie making capability. We will describe how to have MATLAB display a movie of the attractor rotating about the \(x\)-axis. The key is to create the following function m-file, and save it as `rotmovie.m`.

\[
\text{function } M = \text{rotmovie}(u)\\
M = \text{moviein}(18);
\]
11. 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 gravitiational 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 \( 2\pi \) units; i.e., in our units, a year is \( 2\pi \) 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 \(((1 - 2\mu)/2, \sqrt{3}/2)\). If \((x, y)\) is the position of the spacecraft, then the distances to \( M_1 \) and \( M_2 \) are

\[ r_1^2 = (x + \mu)^2 + y^2, \]
\[ r_2^2 = (x - 1 + \mu)^2 + y^2. \]
Finally, the equations of motion in this moving frame are

\[
\begin{align*}
x'' - 2y' - x &= -(1 - \mu)(x + \mu)/r_1^3 - \mu(x - 1 + \mu)/r_2^3, \\
y'' + 2x' - y &= -(1 - \mu)y/r_1^3 - \mu y/r_2^3.
\end{align*}
\tag{9}
\]

a) Find a system of four first order equations which is equivalent to (9).

b) If the two bodies are the earth and the moon, \( \mu = 0.0122 \). The Lagrange points are stable for \( 0 < \mu < 1/2 - \sqrt{69/18} \approx 0.03852 \), and in particular for the earth/moon system. Write \( x = (1 - 2\mu)/2 + \xi \), and \( y = \sqrt{3}/2 + \eta \), so \((\xi, \eta)\) is the position of the spacecraft relative to the Lagrange point. Starting with initial conditions which are less than 1/2 unit away from the Lagrange point, compute the solution. For each solution that you compute, make a plot of \( \eta \) vs \( \xi \) 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 \( \mu \) is in the stable range, are the Lagrange points asymptotically stable, or just stable?

e) Find all five equilibrium points for the system you found in a). 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 equilibrium points on the \( x \)-axis are always unstable. This is a hard algebraic problem.

g) Show that the Lagrange points are stable for \( 0 < \mu < 1/2 - \sqrt{69/18} \). Decide whether or not these points are asymptotically stable. This is a very difficult algebraic problem.