

Solving ODE in MATLAB

P. Howard

Fall 2007

Contents

1	Finding Explicit Solutions	1
1.1	First Order Equations	2
1.2	Second and Higher Order Equations	3
1.3	Systems	3
2	Finding Numerical Solutions	4
2.1	First-Order Equations with Inline Functions	5
2.2	First Order Equations with M-files	7
2.3	Systems of ODE	7
2.4	Passing Parameters	10
2.5	Second Order Equations	10
3	Laplace Transforms	10
4	Boundary Value Problems	11
5	Event Location	12
6	Numerical Methods	15
6.1	Euler's Method	15
6.2	Higher order Taylor Methods	18
7	Advanced ODE Solvers	20
7.1	Stiff ODE	21

1 Finding Explicit Solutions

MATLAB has an extensive library of functions for solving ordinary differential equations. In these notes, we will only consider the most rudimentary.

1.1 First Order Equations

Though MATLAB is primarily a numerics package, it can certainly solve straightforward differential equations symbolically.¹ Suppose, for example, that we want to solve the first order differential equation

$$y'(x) = xy. \quad (1.1)$$

We can use MATLAB's built-in `dsolve()`. The input and output for solving this problem in MATLAB is given below.

```
>>y = dsolve('Dy = y*x','x')
y = C1*exp(1/2*x^2)
```

Notice in particular that MATLAB uses capital D to indicate the derivative and requires that the entire equation appear in single quotes. MATLAB takes t to be the independent variable by default, so here x must be explicitly specified as the independent variable. Alternatively, if you are going to use the same equation a number of times, you might choose to define it as a variable, say, `eqn1`.

```
>>eqn1 = 'Dy = y*x'
eqn1 =
Dy = y*x
>>y = dsolve(eqn1,'x')
y = C1*exp(1/2*x^2)
```

To solve an initial value problem, say, equation (1.1) with $y(1) = 1$, use

```
>>y = dsolve(eqn1,'y(1)=1','x')
y =
1/exp(1/2)*exp(1/2*x^2)
```

or

```
>>inits = 'y(1)=1';
>>y = dsolve(eqn1,inits,'x')
y =
1/exp(1/2)*exp(1/2*x^2)
```

Now that we've solved the ODE, suppose we want to plot the solution to get a rough idea of its behavior. We run immediately into two minor difficulties: (1) our expression for $y(x)$ isn't suited for array operations (`.*`, `./`, `.^`), and (2) y , as MATLAB returns it, is actually a symbol (a *symbolic object*). The first of these obstacles is straightforward to fix, using `vectorize()`. For the second, we employ the useful command `eval()`, which evaluates or executes text strings that constitute valid MATLAB commands. Hence, we can use

¹Actually, whenever you do symbolic manipulations in MATLAB what you're really doing is calling Maple.

```
>>x = linspace(0,1,20);
>>z = eval(vectorize(y));
>>plot(x,z)
```

You may notice a subtle point here, that *eval()* evaluates strings (*character arrays*), and *y*, as we have defined it, is a symbolic object. However, *vectorize* converts symbolic objects into strings.

1.2 Second and Higher Order Equations

Suppose we want to solve and plot the solution to the second order equation

$$y''(x) + 8y'(x) + 2y(x) = \cos(x); \quad y(0) = 0, y'(0) = 1. \quad (1.2)$$

The following (more or less self-explanatory) MATLAB code suffices:

```
>>eqn2 = 'D2y + 8*Dy + 2*y = cos(x)';
>>inits2 = 'y(0)=0, Dy(0)=1';
>>y=dsolve(eqn2,inits2,'x')
y =
1/65*cos(x)+8/65*sin(x)+(-1/130+53/1820*14^(1/2))*exp((-4+14^(1/2))*x)
-1/1820*(53+14^(1/2))*14^(1/2)*exp(-(4+14^(1/2))*x)
>>z = eval(vectorize(y));
>>plot(x,z)
```

1.3 Systems

Suppose we want to solve and plot solutions to the system of three ordinary differential equations

$$\begin{aligned} x'(t) &= x(t) + 2y(t) - z(t) \\ y'(t) &= x(t) + z(t) \\ z'(t) &= 4x(t) - 4y(t) + 5z(t). \end{aligned} \quad (1.3)$$

First, to find a general solution, we proceed as in Section 4.1.1, except with each equation now braced in its own pair of (single) quotation marks:

```
>>[x,y,z]=dsolve('Dx=x+2*y-z','Dy=x+z','Dz=4*x-4*y+5*z')
x =
2*C1*exp(2*t)-2*C1*exp(t)-C2*exp(3*t)+2*C2*exp(2*t)-1/2*C3*exp(3*t)+1/2*C3*exp(t)
y =
2*C1*exp(t)-C1*exp(2*t)+C2*exp(3*t)-C2*exp(2*t)+1/2*C3*exp(3*t)-1/2*C3*exp(t)
z =
-4*C1*exp(2*t)+4*C1*exp(t)+4*C2*exp(3*t)-4*C2*exp(2*t)-C3*exp(t)+2*C3*exp(3*t)
```

(If you use MATLAB to check your work, keep in mind that its choice of constants $C1$, $C2$, and $C3$ probably won't correspond with your own. For example, you might have $C = -2C1 + 1/2C3$, so that the coefficients of $\exp(t)$ in the expression for x are combined. Fortunately, there is no such ambiguity when initial values are assigned.) Notice that since no independent variable was specified, MATLAB used its default, t . For an example in which the independent variable is specified, see Section 4.1.1. To solve an initial value problem, we simply define a set of initial values and add them at the end of our *dsolve()* command. Suppose we have $x(0) = 1$, $y(0) = 2$, and $z(0) = 3$. We have, then,

```
>>inits='x(0)=1,y(0)=2,z(0)=3';
>>[x,y,z]=dsolve('Dx=x+2*y-z','Dy=x+z','Dz=4*x-4*y+5*z',inits)
x =
6*exp(2*t)-5/2*exp(t)-5/2*exp(3*t)
y =
5/2*exp(t)-3*exp(2*t)+5/2*exp(3*t)
z =
-12*exp(2*t)+5*exp(t)+10*exp(3*t)
```

Finally, plotting this solution can be accomplished as in Section 4.1.1.

```
>>t=linspace(0,.5,25);
>>xx=eval(vectorize(x));
>>yy=eval(vectorize(y));
>>zz=eval(vectorize(z));
>>plot(t, xx, t, yy, t, zz)
```

The figure resulting from these commands is included as Figure 1.1.

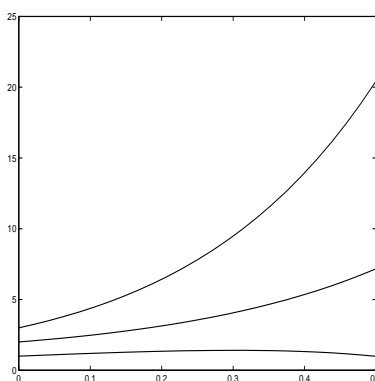


Figure 1.1: Solutions to equation (1.3).

2 Finding Numerical Solutions

MATLAB has a number of tools for numerically solving ordinary differential equations. We will focus on the main two, the built-in functions *ode23* and *ode45*, which implement versions of Runge–Kutta 2nd/3rd-order and Runge–Kutta 4th/5th-order, respectively.

2.1 First-Order Equations with Inline Functions

Example 2.1. Numerically approximate the solution of the first order differential equation

$$\frac{dy}{dx} = xy^2 + y; \quad y(0) = 1,$$

on the interval $x \in [0, .5]$.

For any differential equation in the form $y' = f(x, y)$, we begin by defining the function $f(x, y)$. For single equations, we can define $f(x, y)$ as an inline function. Here,

```
>>f=inline('x*y^2+y')
f =
Inline function:
f(x,y) = x*y^2+y
```

The basic usage for MATLAB's solver *ode45* is

```
ode45(function,domain,initial condition).
```

That is, we use

```
>>[x,y]=ode45(f,[0 .5],1)
```

and MATLAB returns two column vectors, the first with values of x and the second with values of y . (The MATLAB output is fairly long, so I've omitted it here.) Since x and y are vectors with corresponding components, we can plot the values with

```
>>plot(x,y)
```

which creates Figure 2.1.

Choosing the partition. In approximating this solution, the algorithm *ode45* has selected a certain partition of the interval $[0, .5]$, and MATLAB has returned a value of y at each point in this partition. It is often the case in practice that we would like to specify the partition of values on which MATLAB returns an approximation. For example, we might only want to approximate $y(.1)$, $y(.2)$, ..., $y(.5)$. We can specify this by entering the vector of values $[0, .1, .2, .3, .4, .5]$ as the domain in *ode45*. That is, we use

```
>>xvalues=0:.1:.5
xvalues =
0 0.1000 0.2000 0.3000 0.4000 0.5000
>>[x,y]=ode45(f,xvalues,1)
x =
0
0.1000
0.2000
0.3000
0.4000
```

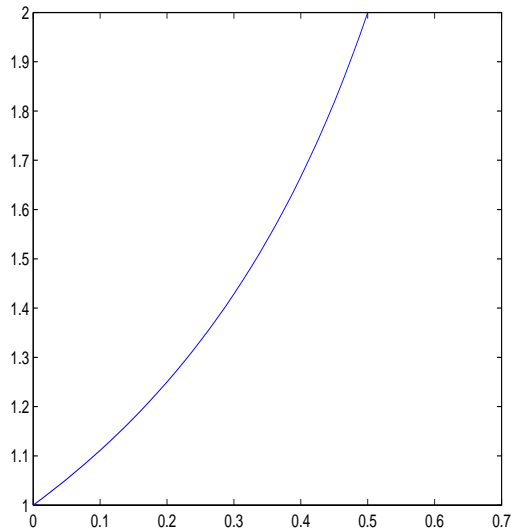


Figure 2.1: Plot of the solution to $y' = xy^2 + y$, with $y(0) = 1$.

```
0.5000
y =
1.0000
1.1111
1.2500
1.4286
1.6667
2.0000
```

It is important to point out here that MATLAB continues to use roughly the same partition of values that it originally chose; the only thing that has changed is the values at which it is printing a solution. In this way, no accuracy is lost.

Options. Several options are available for MATLAB's *ode45* solver, giving the user limited control over the algorithm. Two important options are relative and absolute tolerance, respectively *RelTol* and *AbsTol* in MATLAB. At each step of the *ode45* algorithm, an error is approximated for that step. If y_k is the approximation of $y(x_k)$ at step k , and e_k is the approximate error at this step, then MATLAB chooses its partition to insure

$$e_k \leq \max(\text{RelTol} * y_k, \text{AbsTol}),$$

where the default values are $\text{RelTol} = .001$ and $\text{AbsTol} = .000001$. As an example for when we might want to change these values, observe that if y_k becomes large, then the error e_k will be allowed to grow quite large. In this case, we increase the value of RelTol . For the equation $y' = xy^2 + y$, with $y(0) = 1$, the values of y get quite large as x nears 1. In fact, with the default error tolerances, we find that the command

```
>>[x,y]=ode45(f,[0,1],1);
```

leads to an error message, caused by the fact that the values of y are getting too large as x nears 1. (Note at the top of the column vector for y that it is multiplied by 10^{14} .) In order to fix this problem, we choose a smaller value for RelTol.

```
>>options=odeset('RelTol',1e-10);
>>[x,y]=ode45(f,[0,1],1,options);
>>max(y)
ans =
2.425060345544448e+07
```

In addition to employing the option command, I've computed the maximum value of $y(x)$ to show that it is indeed quite large, though not as large as suggested by the previous calculation. \triangle

2.2 First Order Equations with M-files

Alternatively, we can solve the same ODE as in Example 2.1 by first defining $f(x, y)$ as an M-file *firstode.m*.

```
function yprime = firstode(x,y);
% FIRSTODE: Computes yprime = x*y^2+y
yprime = x*y^2 + y;
```

In this case, we only require one change in the *ode45* command: we must use a pointer @ to indicate the M-file. That is, we use the following commands.

```
>>xspan = [0,.5];
>>y0 = 1;
>>[x,y]=ode23(@firstode,xspan,y0);
>>x
```

2.3 Systems of ODE

Solving a system of ODE in MATLAB is quite similar to solving a single equation, though since a system of equations cannot be defined as an inline function we must define it as an M-file.

Example 2.2. Solve the system of Lorenz equations,²

$$\begin{aligned}\frac{dx}{dt} &= -\sigma x + \sigma y \\ \frac{dy}{dt} &= \rho x - y - xz \\ \frac{dz}{dt} &= -\beta z + xy,\end{aligned}\tag{2.1}$$

²The Lorenz equations have some properties of equations arising in atmospheric. Solutions of the Lorenz equations have long served as an example for chaotic behavior.

where for the purposes of this example, we will take $\sigma = 10$, $\beta = 8/3$, and $\rho = 28$, as well as $x(0) = -8$, $y(0) = 8$, and $z(0) = 27$. The MATLAB M-file containing the Lorenz equations appears below.

```
function xprime = lorenz(t,x);
%LORENZ: Computes the derivatives involved in solving the
%Lorenz equations.
sig=10;
beta=8/3;
rho=28;
xprime=[-sig*x(1) + sig*x(2); rho*x(1) - x(2) - x(1)*x(3); -beta*x(3) + x(1)*x(2)];
```

Observe that x is stored as $x(1)$, y is stored as $x(2)$, and z as stored as $x(3)$. Additionally, $xprime$ is a **column vector**, as is evident from the semicolon following the first appearance of $x(2)$. If in the Command Window, we type

```
>>x0=[-8 8 27];
>>tspan=[0,20];
>>[t,x]=ode45(@lorenz,tspan,x0)
```

Though not given here, the output for this last command consists of a column of times followed by a matrix with three columns, the first of which corresponds with values of x at the associated times, and similarly for the second and third columns for y and z . The matrix has been denoted x in the statement calling *ode45*, and in general any coordinate of the matrix can be specified as $x(m,n)$, where m denotes the row and n denotes the column. What we will be most interested in is referring to the columns of x , which correspond with values of the components of the system. Along these lines, we can denote *all rows* or *all columns* by a colon $:$. For example, $x(:,1)$ refers to all rows in the first column of the matrix x ; that is, it refers to all values of our original x component. Using this information, we can easily plot the Lorenz strange attractor, which is a plot of z versus x :

```
>>plot(x(:,1),x(:,3))
```

See Figure 2.2.

Of course, we can also plot each component of the solution as a function of t , and one useful way to do this is to stack the results. We can create Figure 2.3 with the following MATLAB code.

```
>>subplot(3,1,1)
>>plot(t,x(:,1))
>>subplot(3,1,2)
>>plot(t,x(:,2))
>>subplot(3,1,3)
>>plot(t,x(:,3))
```

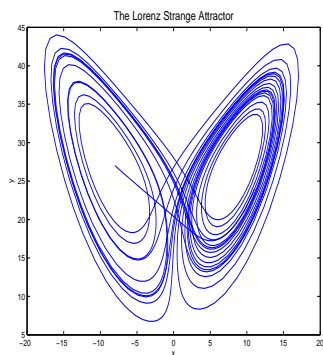



Figure 2.2: The Lorenz Strange Attractor

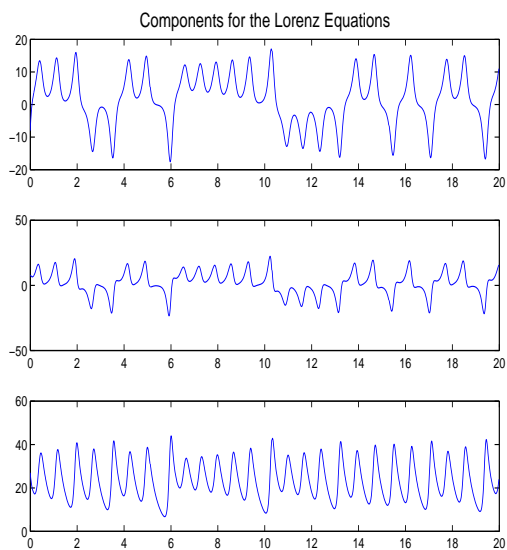


Figure 2.3: Plot of coordinates for the Lorenz equations as a function of t .

2.4 Passing Parameters

In analyzing system of differential equations, we often want to experiment with different parameter values. For example, in studying the Lorenz equations we might want to consider the behavior as a function of the values of σ , β , and ρ . Of course, one way to change this is to manually re-open the M-file *lorenz.m* each time we want to try new values, but not only is a slow way to do it, it's unwieldy to automate. What we can do instead is pass parameter values directly to our M-file through the *ode45* call statement. In order to see how this works, we first alter *lorenz.m* into *lorenz1.m*, the latter of which accepts a vector of parameters that we denote p .

```
function xprime = lorenz1(t,x,p);
%LORENZ: Computes the derivatives involved in solving the
%Lorenz equations.
sig=p(1); beta=p(2); rho=p(3);
xprime=[-sig*x(1) + sig*x(2); rho*x(1) - x(2) - x(1)*x(3); -beta*x(3) + x(1)*x(2)];
```

We can now send parameter values with *ode45*.

```
>>p=[10 8/3 28];
>>[t,x]=ode45(@lorenz1,tspan,x0,[],p);
```

2.5 Second Order Equations

The first step in solving a second (or higher) order ordinary differential equation in MATLAB is to write the equation as a first order system. As an example, let's return to equation (1.2) from Subsection 1.2. Taking $y_1(x) = y(x)$ and $y_2(x) = y'(x)$, we have the system

$$\begin{aligned}y_1'(x) &= y_2(x) \\ y_2'(x) &= -8y_2(x) - 2y_1(x) + \cos(x).\end{aligned}$$

We can now proceed as in Section 2.3.

3 Laplace Transforms

One of the most useful tools in mathematics is the Laplace transform. MATLAB has built-in routines for computing both Laplace transforms and inverse Laplace transforms. For example, to compute the Laplace transform of $f(t) = t^2$, type simply

```
>>syms t;
>>laplace(t^2)
```

In order to invert, say, $F(s) = \frac{1}{1+s}$, type

```
>>syms s;
>>ilaplace(1/(1+s))
```

Solving Second Order D.E

Matlab has two functions, ode23 and ode45, which are capable of numerically solving differential equations. Both of them use a similar numerical formula, Runge-Kutta, but to a different order of approximation.

The syntax for actually solving a differential equation with these functions is:

```
[T,Y] = ode45('yprime',t0,tF,y0);
```

'yprime' is the name of a function that you write that describes your system of differential equations. t0 and tF are the initial and final times that you want a solution for, and y0 is a vector of the initial values of the variables in your system of equations. You get back a vector of times, T, and a matrix Y that has the values of each variable in your system of equations over the times in the time vector. Each column of Y is a different variable.

The hardest part of this is actually defining your differential equation so that matlab understands it. Matlab has a very specific way to define a differential equation, as a function that takes one vector of variables in the differential equation, plus a time vector, as an argument and returns the derivative of that vector. The only way that matlab keeps track of which variable is which inside the vector is the order you choose to use the variables in. You define your differential equations based on that ordering of variables in the vector, you define your initial conditions in the same order, and the columns of your answer are also in that order.

If you follow a careful system to write your differential equation function each time you need to solve a differential equation, it's not too difficult. I'll do an example in this system to explain how it works. In my example, x and y are functions of t.

$$\begin{aligned}\ddot{x} + \dot{y} - 2\dot{x} &= 3 \\ \dot{y} &= 3x + 5\end{aligned}$$

The system of differential equations we're trying to solve is

The first thing to notice is that this is not a first order differential equation, because it has an \ddot{x} in it. To create a function that returns a second derivative, one of the variables you give it has to be the first derivative. So the variables the function needs to start with are x , \dot{x} , and y . The function needs to tell matlab how to get from those variables to \dot{x} , \ddot{x} , and \dot{y} .

The first step is to write the first line of the function. Since x and y are functions of t, and we eventually want the differential equation to give us x and y, we'll define the function with

```
function dxy = diffxy(t, xy)
```

This function is called diffxy. That means it should be in a file called diffxy.m

. It takes a vector xy (which has all three of the variables we said we had to give matlab to define our differential equation in it) and a time vector, and gives us the derivative of the three things in the xy vector.

The next step is to assign some variables from the xy vector that this function is given. This is when we decide what order the three things in the xy vector will be in. We could just keep calling them xy(1), xy(2), and xy(3), but then we might forget which one of them was

supposed to represent x , which one was supposed to represent y , etc. So the first thing we do is rename some variables:

```
x = xy(1);
xdot = xy(2);
y = xy(3);
```

Now we can use x , \dot{x} , and y to define the three things that this function has to return--- \dot{x} , \ddot{x} , and \dot{y} . This is where we actually refer to the differential equation and implement it.

The first definition is trivial. We need to express \dot{x} (the derivative of x) in terms of the three variables we're given. \dot{x} is already one of the three variables we're given, so we can use it to start with:

```
xdot = xdot;
```

Then \ddot{x} comes from the first equation:

```
xdoubledot = 3 - ydot + 2*xdot;
```

Whoops, we don't have a \dot{y} yet. It's not one of the three variables we're given in xy . But the second equation tells us what \dot{y} is equal to in terms of things we are given in xy . So what we really want to say is

```
ydot = 3*x + 2*y + 5;
```

```
xdoubledot = 3 - ydot + 2*xdot;
```

The last thing we want to define is \dot{y} , which is the derivative of y . We just defined it above in order to get \ddot{x} , so we're done.

So dxy , the thing we return, has to contain \dot{x} , \ddot{x} , and \dot{y} in order. The final thing we have to do is combine those values into the vector dxy :

```
dxy = [xdot; xdoubledot; ydot];
```

Then we have a finished function that takes t and xy , and returns dxy , which is the derivative of all the elements in xy . You can follow this approach for any number of differential equations.

The final program, `diffxy.m`, looks like

```
function dxy = diffxy(t, xy)
%
%split xy into variables in our equations
%
x = xy(1);
xdot = xy(2);
y = xy(3);
%
% define the derivatives of these variables from equations
%
xdot = xdot;
ydot = 3*x + 2*y + 5;
xdoubledot = 3 - ydot + 2*xdot;
%
%return the derivatives in dxy in the right order
%
dxy = [xdot; xdoubledot; ydot]
To get a solution, we can type
[T, XY] ode45('diffxy',0,10,[0 1 0])
```

which uses times from 0 to 10 and assumes that the initial value of x is 0, \dot{x} is 1, and \dot{y} is 0, and gives us a time vector and a three column vector that contains values of x , \dot{x} , and y over that time vector.

4 Boundary Value Problems

For various reasons of arguable merit most introductory courses on ordinary differential equations focus primarily on initial value problems (IVP's). Another class of ODE's that often arise in applications are boundary value problems (BVP's). Consider, for example, the differential equation

$$\begin{aligned}y'' - 3y' + 2y &= 0 \\ y(0) &= 0 \\ y(1) &= 10,\end{aligned}$$

where our conditions $y(0) = 0$ and $y(1) = 10$ are specified on the boundary of the interval of interest $x \in [0, 1]$. (Though our solution will typically extend beyond this interval, the most common scenario in boundary value problems is the case in which we are only interested in values of the independent variable between the specified endpoints.) The first step in solving this type of equation is to write it as a first order system with $y_1 = y$ and $y_2 = y'$, for which we have

$$\begin{aligned}y_1' &= y_2 \\ y_2' &= -2y_1 + 3y_2.\end{aligned}$$

We record this system in the M-file *bvpexample.m*.

```
function yprime = bvpexample(t,y)
%BVPEXAMPLE: Differential equation for boundary value
%problem example.
yprime=[y(2); -2*y(1)+3*y(2)];
```

Next, we write the boundary conditions as the M-file *bc.m*, which records boundary *residues*.

```
function res=bc(y0,y1)
%BC: Evaluates the residue of the boundary condition
res=[y0(1);y1(1)-10];
```

By *residue*, we mean the left-hand side of the boundary condition once it has been set to 0. In this case, the second boundary condition is $y(1) = 10$, so its residue is $y(1) - 10$, which is recorded in the second component of the vector that *bc.m* returns. The variables *y0* and *y1* represent the solution at $x = 0$ and at $x = 1$ respectively, while the 1 in parentheses indicates the first component of the vector. In the event that the second boundary condition was $y'(1) = 10$, we would replace $y1(1) - 10$ with $y1(2) - 10$.

We are now in a position to begin solving the boundary value problem. In the following code, we first specify a grid of x values for MATLAB to solve on and an initial guess for the vector that would be given for an initial value problem $[y(0), y'(0)]$. (Of course, $y(0)$ is known, but $y'(0)$ must be a guess. Loosely speaking, MATLAB will solve a family of initial value problems, searching for one for which the boundary conditions are met.) We solve the boundary value problem with MATLAB's built-in solver *bvp4c*.

```

>>sol=bvpinit(linspace(0,1,25),[0 1]);
>>sol=bvp4c(@bvpexample,@bc,sol);
>>sol.x
ans =
Columns 1 through 9
0 0.0417 0.0833 0.1250 0.1667 0.2083 0.2500 0.2917 0.3333
Columns 10 through 18
0.3750 0.4167 0.4583 0.5000 0.5417 0.5833 0.6250 0.6667 0.7083
Columns 19 through 25
0.7500 0.7917 0.8333 0.8750 0.9167 0.9583 1.0000
>>sol.y
ans =
Columns 1 through 9
0 0.0950 0.2022 0.3230 0.4587 0.6108 0.7808 0.9706 1.1821
2.1410 2.4220 2.7315 3.0721 3.4467 3.8584 4.3106 4.8072 5.3521
Columns 10 through 18
1.4173 1.6787 1.9686 2.2899 2.6455 3.0386 3.4728 3.9521 4.4805
5.9497 6.6050 7.3230 8.1096 8.9710 9.9138 10.9455 12.0742 13.3084
Columns 19 through 25
5.0627 5.7037 6.4090 7.1845 8.0367 8.9726 9.9999
14.6578 16.1327 17.7443 19.5049 21.4277 23.5274 25.8196

```

We observe that in this case MATLAB returns the solution as a structure whose first component *sol.x* simply contains the *x* values we specified. The second component of the structure *sol* is *sol.y*, which is a matrix containing as its first row values of $y(x)$ at the *x* grid points we specified, and as its second row the corresponding values of $y'(x)$.

5 Event Location

Typically, the ODE solvers in MATLAB terminate after solving the ODE over a specified domain of the independent variable (the range we have referred to above as *xspan* or *tspan*). In applications, however, we often would like to stop the solution at a particular value of the dependent variable (for example, when an object fired from the ground reaches its maximum height or when a population crosses some threshold value). As an example, suppose we would like to determine the period of the pendulum from Example 3.1. Since we do not know the appropriate time interval (in fact, that's what we're trying to determine), we would like to specify that MATLAB solve the equation until the pendulum swings through some specified fraction of its complete cycle and to give the time this took. In our case, we will record the time it takes the pendulum to reach the bottom of its arc, and multiply this by 4 to arrive at the pendulum's period. (In this way, the event is independent of the pendulum's initial conditions.) Our pendulum equation

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l} \sin \theta$$

is stored in *pendode.m* with $l = 1$ (see Example 3.1). In addition to this file, we write an *events* file *pendevent.m* that specifies the event we are looking for.

```
function [lookfor stop direction]=pendevent(t,x)
%PENDEVENT: MATLAB function M-file that contains the event
%that our pendulum reaches its center point from the right
lookfor = x(1) ; %Searches for this expression set to 0
stop = 1; %Stop when event is located
direction = -1; %Specify direction of motion at event
```

In *pendevent.m*, the line *lookfor=x(1)* specifies that MATLAB should look for the event $x(1) = 0$ (that is, $x(t) = 0$). (If we wanted to look for the event $x(t) = 1$, we would use *lookfor=x(1)-1*.) The line *stop=1* instructs MATLAB to stop solving when the event is located, and the command *direction=-1* instructs MATLAB to only accept events for which $x(2)$ (that is, x') is negative (if the pendulum starts to the right of center, it will be moving in the negative direction the first time it reaches the center point).

We can now solve the ODE up until the time our pendulum reaches the center point with the following commands issued in the Command Window:

```
>>options=odeset('Events',@pendevent);
>>x0=[pi/4 0];
>>[t, x, te, xe, ie]=ode45(@pendode, [0, 10], x0, options);
>>te
te =
0.5215
>>xe
xe =
-0.0000 -2.3981
```

Here, x_0 is a vector of initial data, for which we have chosen that the pendulum begin with angle $\pi/4$ and with no initial velocity. The command *ode45()* returns a vector of times t , a matrix of dependent variables x , the time at which the event occurred, te , and the values of x when the event occurred, xe . In the event that a vector of events is specified, index vector ie describes which event has occurred in each instance. Here, only a single event has been specified, so $ie=1$. In this case, we see that the event occurred at time $t = .5215$, and consequently the period is $P = 2.086$ (within numerical errors). Though the exact period of the pendulum is difficult to analyze numerically, it is not difficult to show through the small angle approximation $\sin \theta \cong \theta$ that for θ small the period of the pendulum is approximately $P = 2\pi\sqrt{\frac{l}{g}}$, which in our case gives $P = 2.001$. (While the small angle approximation gives a period independent of θ , the period of a pendulum does depend on θ .)

In order to better understand this index ie , let's verify that the time is the same for each quarter swing. That is, let's record the times at which $\theta = 0$ and additionally the times at which $\theta' = 0$ and look at the times between them. In this case, *pendevent.m* is replaced by *pendevent1.m*.


```

function [lookfor stop direction]=pendevent1(t,x)
%PENDEVENT1: MATLAB function M-file that contains the event
%that our pendulum returns to its original position pi/4
lookfor = [x(1);x(2)]; %Searches for this expression set to 0
stop = [0;0]; %Do not stop when event is located
direction = [0;0]; %Either direction accepted

```

In this case, we are looking for two different events, and so the variables in *pendevent1.m* are vectors with two components, each corresponding with an event. In this case, we do not stop after either event, and we do not specify a direction. In the Command Window, we have the following.

```

>>options=odeset('Events',@pendevent1);
>>x0=[pi/4 0];
>>[t, x, te, xe, ie]=ode45(@pendode,[0 2],x0,options);
>>te
te =
0.0000
0.5216
1.0431
1.5646
>>xe
xe =
0.7854 -0.0000
-0.0000 -2.3972
-0.7853 0.0000
0.0000 2.3970
>>ie
ie =
2
1
2
1

```

We see that over a time interval $[0, 2]$ the event times are approximately 0, .5216, 1.0431, and 1.5646. Looking at the matrix *xe*, for which the first value in each row is an angular position and the second is an angular velocity, we see that the first event corresponds with the starting position, the second event corresponds with the pendulum's hanging straight down, the third event corresponds with the pendulum's having swung entirely to the opposite side, and the fourth event corresponds with the pendulum's hanging straight down on its return trip. It's now clear how *ie* works: it is 2 when the second event we specified occurs and 1 when the first event we specified occurs.

6 Numerical Methods

Though we can solve ODE on MATLAB without any knowledge of the numerical methods it employs, it's often useful to understand the basic underlying principles. In this section we will use Taylor's Theorem to derive methods for approximating the solution to a differential equation.

6.1 Euler's Method

Consider the general first order differential equation

$$\frac{dy}{dx} = f(x, y); \quad y(x_0) = y_0, \quad (6.1)$$

and suppose we would like to solve this equation on the interval of x -values $[x_0, x_n]$. Our goal will be to approximate the value of the solution $y(x)$ at each of the x values in a partition $P = [x_0, x_1, x_2, \dots, x_n]$. Since $y(x_0)$ is given, the first value we need to estimate is $y(x_1)$. By Taylor's Theorem, we can write

$$y(x_1) = y(x_0) + y'(x_0)(x_1 - x_0) + \frac{y'(c)}{2}(x_1 - x_0)^2,$$

where $c \in (x_0, x_1)$. Observing from our equation that $y'(x_0) = f(x_0, y(x_0))$, we have

$$y(x_1) = y(x_0) + f(x_0, y(x_0))(x_1 - x_0) + \frac{y'(c)}{2}(x_1 - x_0)^2.$$

If our partition P has small subintervals, then $x_1 - x_0$ will be small, and we can regard the smaller quantity $\frac{y'(c)}{2}(x_1 - x_0)^2$ as an error term. That is, we have

$$y(x_1) \approx y(x_0) + f(x_0, y(x_0))(x_1 - x_0). \quad (6.2)$$

We can now compute $y(x_2)$ in a similar manner by using Taylor's Theorem to write

$$y(x_2) = y(x_1) + y'(x_1)(x_2 - x_1) + \frac{y'(c)}{2}(x_2 - x_1)^2.$$

Again, we have from our equation that $y'(x_1) = f(x_1, y(x_1))$, and so

$$y(x_2) = y(x_1) + f(x_1, y(x_1))(x_2 - x_1) + \frac{y'(c)}{2}(x_2 - x_1)^2.$$

If we drop the term $\frac{y'(c)}{2}(x_2 - x_1)^2$ as an error, then we have

$$y(x_2) \approx y(x_1) + f(x_1, y(x_1))(x_2 - x_1),$$

where the value $y(x_1)$ required here can be approximate by the value from (6.2). More generally, for any $k = 1, 2, \dots, n - 1$ we can approximate $y(x_{k+1})$ from the relation

$$y(x_{k+1}) \approx y(x_k) + f(x_k, y(x_k))(x_{k+1} - x_k),$$

where $y(x_k)$ will be known from the previous calculation. As with methods of numerical integration, it is customary in practice to take our partition to consist of subintervals of equal width,

$$(x_{k+1} - x_k) = \Delta x = \frac{x_n - x_0}{n}.$$

(In the study of numerical methods for differential equations, this quantity is often denoted h .) In this case, we have the general relationship

$$y(x_{k+1}) \approx y(x_k) + f(x_k, y(x_k))\Delta x.$$

If we let the values y_0, y_1, \dots, y_n denote our approximations for y at the points x_0, x_1, \dots, x_n (that is, $y_0 = y(x_0)$, $y_1 \approx y(x_1)$, etc.), then we can approximate $y(x)$ on the partition P by iteratively computing

$$y_{k+1} = y_k + f(x_k, y_k)\Delta x. \quad (6.3)$$

Example 6.1. Use Euler's method (6.3) with $n = 10$ to solve the differential equation

$$\frac{dy}{dx} = \sin(xy); \quad y(0) = \pi,$$

on the interval $[0, 1]$. We will carry out the first few iterations in detail, and then we will write a MATLAB M-file to carry it out in its entirety. First, the initial value $y(0) = \pi$ gives us the values $x_0 = 0$ and $y_0 = \pi$. If our partition is composed of subintervals of equal width, then $x_1 = \Delta x = \frac{1}{10} = .1$, and according to (6.3)

$$y_1 = y_0 + \sin(x_0 y_0)\Delta x = \pi + \sin(0).1 = \pi.$$

We now have the point $(x_1, y_1) = (.1, \pi)$, and we can use this and 6.3 to compute

$$y_2 = y_1 + \sin(x_1 y_1)\Delta x = \pi + \sin(.1\pi)(.1) = 3.1725.$$

We now have $(x_2, y_2) = (.2, 3.1725)$, and we can use this to compute

$$y_3 = y_2 + \sin(x_2 y_2)\Delta x = 3.1725 + \sin(.2(3.1725))(.1) = 3.2318.$$

More generally, we can use the M-file *euler.m*.

```
function [xvalues, yvalues] = euler(f,x0,xn,y0,n)
%EULER: MATLAB function M-file that solve the
%ODE y'=f, y(x0)=y0 on [x0,xn] using a partition
%with n equally spaced subintervals
dx = (xn-x0)/n;
x(1) = x0;
y(1) = y0;
for k=1:n
x(k+1)=x(k) + dx;
y(k+1)= y(k) + f(x(k),y(k))*dx;
end
xvalues = x';
yvalues = y';
```

We can implement this file with the following code, which creates Figure 6.1.

```
>>f=inline('sin(x*y)')
f =
Inline function:
f(x,y) = sin(x*y)
>>[x,y]=euler(f,0,1,pi,10)
x =
0
0.1000
0.2000
0.3000
0.4000
0.5000
0.6000
0.7000
0.8000
0.9000
1.0000
y =
3.1416
3.1416
3.1725
3.2318
3.3142
3.4112
3.5103
3.5963
3.6548
3.6764
3.6598
>>plot(x,y)
>>[x,y]=euler(f,0,1,pi,100);
>>plot(x,y)
```

For comparison, the exact values to four decimal places are given below.

```
x =
0
0.1000
0.2000
0.3000
0.4000
0.5000
0.6000
```

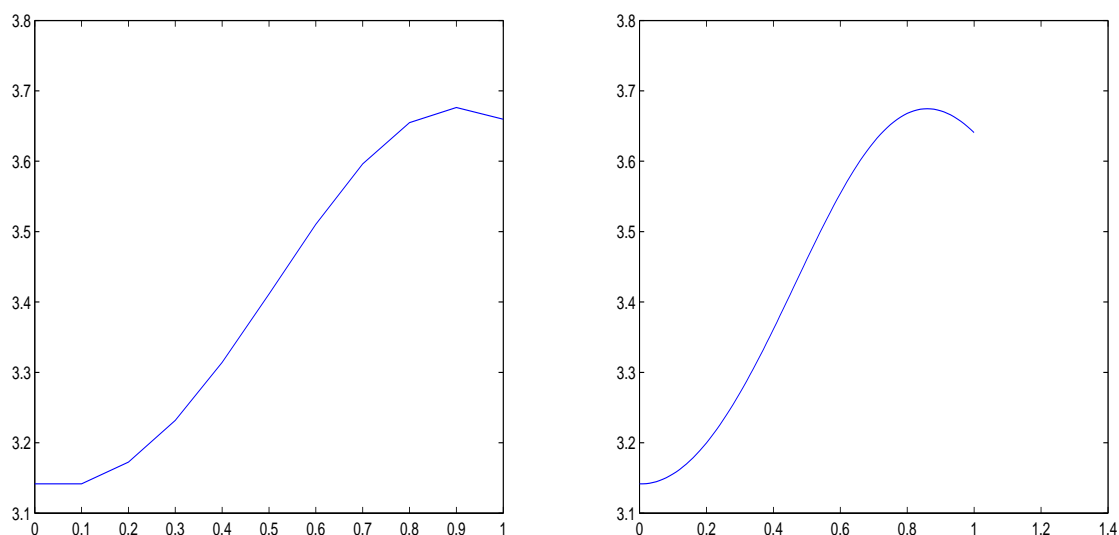


Figure 6.1: Euler approximations to the solution of $y' = \sin(x, y)$ with $\Delta x = .1$ (left) and $\Delta x = .01$ (right).

0.7000
0.8000
0.9000
1.0000
y =
3.1416
3.1572
3.2029
3.2750
3.3663
3.4656
3.5585
3.6299
3.6688
3.6708
3.6383

△

6.2 Higher order Taylor Methods

The basic idea of Euler's method can be improved in a straightforward manner by using higher order Taylor polynomials. In deriving Euler's method, we approximated $y(x)$ with a first order polynomial. More generally, if we use a Taylor polynomial of order n we obtain

the *Taylor method of order n* . In order to see how this is done, we will derive the Taylor method of order 2. (Euler's method is the Taylor method of order 1.)

Again letting $P = [x_0, x_1, \dots, x_n]$ denote a partition of the interval $[x_0, x_n]$ on which we would like to solve (6.1), our starting point for the Taylor method of order 2 is to write down the Taylor polynomial of order 2 (with remainder) for $y(x_{k+1})$ about the point x_k . That is, according to Taylor's theorem,

$$y(x_{k+1}) = y(x_k) + y'(x_k)(x_{k+1} - x_k) + \frac{y''(x_k)}{2}(x_{k+1} - x_k)^2 + \frac{y'''(c)}{3!}(x_{k+1} - x_k)^3,$$

where $c \in (x_k, x_{k+1})$. As with Euler's method, we drop off the error term (which is now smaller), and our approximation is

$$y(x_{k+1}) \approx y(x_k) + y'(x_k)(x_{k+1} - x_k) + \frac{y''(x_k)}{2}(x_{k+1} - x_k)^2.$$

We already know from our derivation of Euler's method that $y'(x_k)$ can be replaced with $f(x_k, y(x_k))$. In addition to this, we now need an expression for $y''(x_k)$. We can obtain this by differentiating the original equation $y'(x) = f(x, y(x))$. That is,

$$y''(x) = \frac{d}{dx}y'(x) = \frac{d}{dx}f(x, y(x)) = \frac{\partial f}{\partial x}(x, y(x)) + \frac{\partial f}{\partial y}(x, y(x))\frac{dy}{dx},$$

where the last equality follows from a generalization of the chain rule to functions of two variables (see Section 10.5.1 of the course text). From this last expression, we see that

$$y''(x_k) = \frac{\partial f}{\partial x}(x_k, y(x_k)) + \frac{\partial f}{\partial y}(x_k, y(x_k))y'(x_k) = \frac{\partial f}{\partial x}(x_k, y(x_k)) + \frac{\partial f}{\partial y}(x_k, y(x_k))f(x_k, y(x_k)).$$

Replacing $y''(x_k)$ with the right-hand side of this last expression, and replacing $y'(x_k)$ with $f(x_k, y(x_k))$, we conclude

$$\begin{aligned} y(x_{k+1}) &\approx y(x_k) + f(x_k, y(x_k))(x_{k+1} - x_k) \\ &\quad + \left[\frac{\partial f}{\partial x}(x_k, y(x_k)) + \frac{\partial f}{\partial y}(x_k, y(x_k))f(x_k, y(x_k)) \right] \frac{(x_{k+1} - x_k)^2}{2}. \end{aligned}$$

If we take subintervals of equal width $\Delta x = (x_{k+1} - x_k)$, this becomes

$$\begin{aligned} y(x_{k+1}) &\approx y(x_k) + f(x_k, y(x_k))\Delta x \\ &\quad + \left[\frac{\partial f}{\partial x}(x_k, y(x_k)) + \frac{\partial f}{\partial y}(x_k, y(x_k))f(x_k, y(x_k)) \right] \frac{\Delta x^2}{2}. \end{aligned}$$

Example 6.2. Use the Taylor method of order 2 with $n = 10$ to solve the differential equation

$$\frac{dy}{dx} = \sin(xy); \quad y(0) = \pi,$$

on the interval $[0, 1]$.

We will carry out the first few iterations by hand and leave the rest as an exercise. To begin, we observe that

$$\begin{aligned}f(x, y) &= \sin(xy) \\ \frac{\partial f}{\partial x}(x, y) &= y \cos(xy) \\ \frac{\partial f}{\partial y}(x, y) &= x \cos(xy).\end{aligned}$$

If we let y_k denote an approximation for $y(x_k)$, the Taylor method of order 2 becomes

$$\begin{aligned}y_{k+1} &= y_k + \sin(x_k y_k)(.1) \\ &\quad + \left[y_k \cos(x_k y_k) + x_k \cos(x_k y_k) \sin(x_k y_k) \right] \frac{(.1)^2}{2}.\end{aligned}$$

Beginning with the point $(x_0, y_0) = (0, \pi)$, we compute

$$y_1 = \pi + \pi(.005) = 3.1573,$$

which is closer to the correct value of 3.1572 than was the approximation of Euler's method. We can now use the point $(x_1, y_1) = (.1, 3.1573)$ to compute

$$\begin{aligned}y_2 &= 3.1573 + \sin(.1 \cdot 3.1573)(.1) \\ &\quad + \left[3.1573 \cos(.1 \cdot 3.1573) + .1 \cos(.1 \cdot 3.1573) \sin(.1 \cdot 3.1573) \right] \frac{(.1)^2}{2} \\ &= 3.2035,\end{aligned}$$

which again is closer to the correct value than was the y_2 approximation for Euler's method. \triangle

7 Advanced ODE Solvers

In addition to the ODE solvers *ode23* and *ode45*, which are both based on the Runge–Kutta scheme, MATLAB has several additional solvers, listed below along with MATLAB's help-file suggestions regarding when to use them.

- Multipstep solvers
 - *ode113*. If using stringent error tolerances or solving a computationally intensive ODE file.
- Stiff problems (see discussion below)
 - *ode15s*. If *ode45* is slow because the problem is stiff.
 - *ode23s*. If using crude error tolerances to solve stiff systems and the mass matrix is constant.
 - *ode23t*. If the problem is only moderately stiff and you need a solution without numerical damping.
 - *ode23tb*. If using crude error tolerances to solve stiff systems.

7.1 Stiff ODE

By a *stiff* ODE we mean an ODE for which numerical errors compound dramatically over time. For example, consider the ODE

$$y' = -100y + 100t + 1; \quad y(0) = 1.$$

Since the dependent variable, y , in the equation is multiplied by 100, small errors in our approximation will tend to become magnified. In general, we must take considerably smaller steps in time to solve stiff ODE, and this can lengthen the time to solution dramatically. Often, solutions can be computed more efficiently using one of the solvers designed for stiff problems.

Index

boundary value problems, 11

dsolve, 2

eval, 2

event location, 12

inverse laplace, 10

laplace, 10

Laplace transforms, 10

ode, 4

ode113(), 20

ode15s(), 20

ode23s(), 20

ode23t(), 20

ode23tb(), 20

stiff ODE, 21

vectorize(), 2