Numerical Methods for ODE in MATLAB

MATLAB has a number of tools for numerically solving ordinary differential equations. We will focus on one of its most rudimentary solvers, *ode45*, which implements a version of the Runge–Kutta 4th order algorithm. (This is essentially the Taylor method of order 4, though implemented in an extremely clever way that avoids partial derivatives.) For a complete list of MATLAB's solvers, type *helpdesk* and then search for *nonlinear numerical methods.*

First Order Equations

Example 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 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

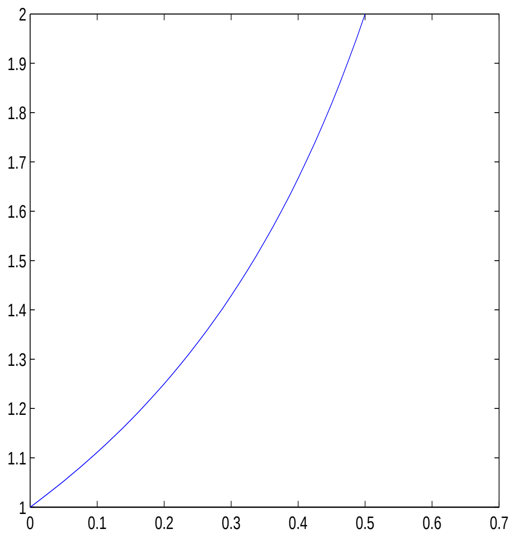

Figure 1: Plot of the solution to $y' = xy^2 + y$, with $y(0) = 1$.

```
>>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
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, respecively *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 RelTol = .001 and 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 multipled 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. △

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. Solve the Lotka–Volterra predator–prey system

$$\begin{aligned}\frac{dy_1}{dt} &= ay_1 - by_1y_2; \quad y_1(0) = y_1^0\\ \frac{dy_2}{dt} &= -ry_2 + cy_1y_2; \quad y_2(0) = y_2^0,\end{aligned}$$

with $a = .5471$, $b = .0281$, $r = .8439$, $c = .0266$, and $y_1^0 = 30$, $y_2^0 = 4$. (These values correspond with data collected by the Hudson Bay Company between 1900 and 1920.)

In this case, we begin by writing the right hand side of this equation as the MATLAB M-file *lv.m:*

```
function yprime = lv(t,y)
%LV: Contains Lotka-Volterra equations
a = .5471;b = .0281;c = .0266;r = .8439;
yprime = [a*y(1)-b*y(1)*y(2);-r*y(2)+c*y(1)*y(2)];
```

We can now solve the equations over a 20 year period as follows:

```
>>[t,y]=ode45(@lv,[0 20],[30;4])
```

The output from this command consists of a column of times and a matrix of populations. The first column of the matrix corresponds with y_1 (prey in this case) and the second corresponds with y_2 (predators). Suppose we would like to plot the prey population as a function of time. In general, $y(n, m)$ corresponds with the entry of y at the intersection of row n and column m, so we can refer to the entire first column of y with $y(:, 1)$, where the colon means all entries. In this way, Figure 2 can be created with the command

```
>>plot(t,y(:,1))
```

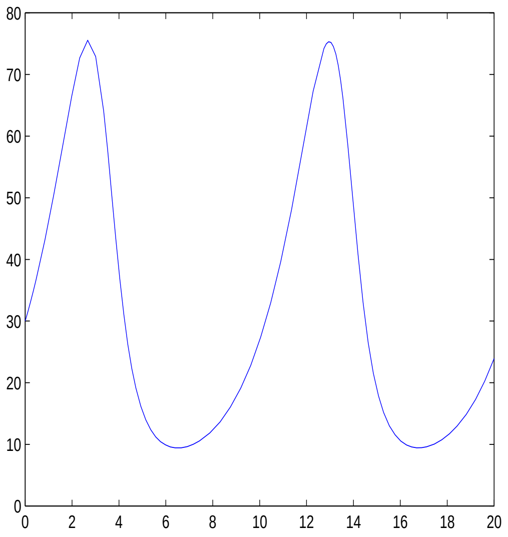

Figure 2: Prey populations as a function of time.

Similarly, we can plot the predator population along with the prey population with

```
>>plot(t,y(:,1),t,y(:,2),'–')
```

where the predator population has been dashed (see Figure 3).

Finally, we can plot an integral curve in the phase plane with

```
>>plot(y(:,1),y(:,2))
```

which creates Figure 4. In this case, the integral curve is closed, and we see that the populations are cyclic. (Such an integral curve is called a *cycle*.)

Passing parameters. In analyzing systems of differential equations, we often want to experiment with parameter values. For example, we might want to solve the Lotka–Volterra system with different values of a, b, c, and r. In fact, we would often like a computer program

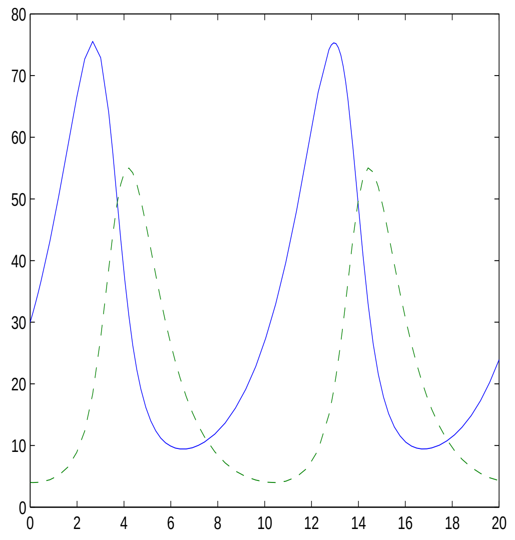

Figure 3: Plot of predator and prey populations for the Lotka–Volterra model.

to try different values of these parameters (typically while searching for a set that correspond with certain behavior), and so we require a method that doesn't involve manually altering our file ODE file every time we want to change parameter values. We can accomplish this by passing the parameters through the function statement. In order to see how this works, we will alter *lv.m* so that the values a, b, c, and r are taken as input. We have

```
function yprime = lvparams(t,y,params)
%LVPARAMS: Contains Lotka-Volterra equations
a=params(1); b=params(2); c=params(3); r=params(4);
yprime = [a*y(1)-b*y(1)*y(2);-r*y(2)+c*y(1)*y(2)];
```

We can now send the parameters with *ode45*.

```
>>params = [.5471 .0281 .0266 .8439];
>>[t,y]=ode45(@lvparams,[0 20],[30;4],[],params);
```

MATLAB requires a statement in the position for options, so I have simply placed an empty set here, designated by two square brackets pushed together. △

Higher Order Equations

In order to solve a higher order equation numerically in MATLAB we proceed by first writing the equation as a first order system and then proceeding as in the previous section.

Example 3. Solve the second order equation

$$y'' + x^2y' - y = \sin x; \quad y(0) = 1, y'(0) = 3$$

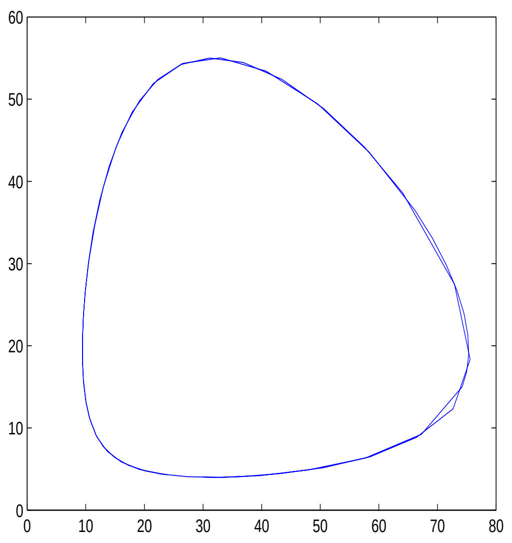

Figure 4: Integral curve for the Lotka–Volterra equation.

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

In order to write this equation as a first order system, we set $y_1(x) = y(x)$ and $y_2(x) = y'(x)$. In this way,

$$\begin{aligned} y_1' &= y_2; \quad y_1(0) = 1 \\ y_2' &= y_1 - x^2 y_2 + \sin x; \quad y_2(0) = 3, \end{aligned}$$

which is stored in the MATLAB M-file *secondode.m*.

```
function yprime = secondode(x,y);
%SECONDODE: Computes the derivatives of y_1 and y_2,
%as a colum vector
yprime = [y(2); y(1)-x^2*y(2)+sin(x)];
```

We can now proceed with the following MATLAB code, which produces Figure ...

```
>>[x,y]=ode45(@secondode,[0,5],[1;3]);
>>plot(x,y(:,1))
```

Assignments

1. Numerically approximate the solution of the first order differential equation

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

on the interval $x \in [0, 1]$. Turn in a plot of your approximation.

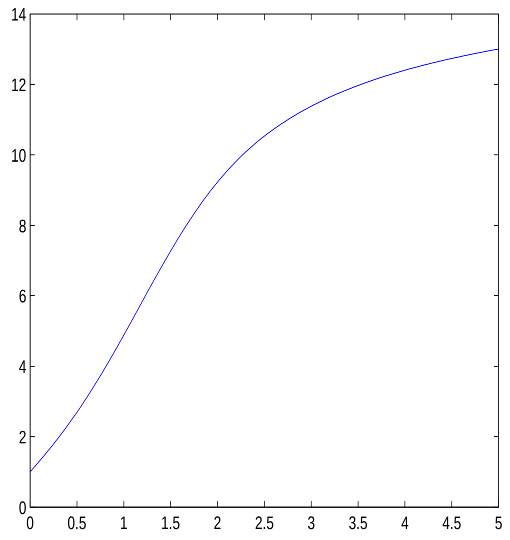

Figure 5: Plot of the solution to $y'' + x^2 y' - y = \sin x; \quad y(0) = 1, y'(0) = 3.$

2. Repeat Problem 1, proceeding so that MATLAB only returns values for $x_0 = 0$, $x_1 = .1$, $x_2 = .2$, ..., $x_{10} = 1$. Turn in a list of values $y(x_1)$, $y(x_2)$ etc.

3. The logistic equation

$$\frac{dp}{dt} = rp(1 - \frac{p}{K}); \quad p(0) = p_0,$$

is solved by

$$p_{\text{exact}}(t) = \frac{p_0 K}{p_0 + (K - p_0)e^{-rt}}.$$

For parameter values $r = .0215$, $K = 446.1825$, and for initial value $p_0 = 7.7498$ (consistent with the U. S. population in millions), solve the logistic equation numerically for $t \in [0, 200]$, and compute the error

$$E = \sum_{t=1}^{200} (p(t) - p_{\text{exact}}(t))^2.$$

4. Repeat Problem 3 with RelTol $= 10^{-10}$.

5. Numerically approximate the solution of the SIR epidemic model

$$\begin{aligned} \frac{dy_1}{dt} &= -\,by_1y_2; \quad y_1(0) = y_1^0 \\ \frac{dy_2}{dt} &= by_1y_2 - ay_2; \quad y_2(0) = y_2^0, \end{aligned}$$

with parameter values $a = .4362$ and $b = .0022$ and initial values $y_1^0 = 762$ and $y_2^0 = 1$. Turn in a plot of susceptibles and infectives as a function of time for $t \in [0, 20]$. Also, plot y_2 as a function of y_1, and discuss how this plot corresponds with our stability analysis for this equation. Check that the maximum of this last plot is at $\frac{a}{b}$, and give values for S_f and S_R as defined in class.