Ordinary Differential Equations in MATLAB

P. Howard

Fall 2003

Contents

1 Solving Ordinary Differential Equations in MATLAB **1**
1.1 Finding Explicit Solutions 1
1.1.1 First Order Equations 1
1.1.2 Second and Higher Order Equations 2
1.1.3 Systems 3
1.2 Finding Numerical Solutions 4
1.2.1 First Order Equations 4
1.2.2 Second Order Equations 5
1.2.3 Solving Systems of ODE 6
1.3 Plotting Direction Fields 7
1.4 Plotting Direction Fields: A Second Example 9
1.5 Plotting Phase Diagrams 9
1.6 Phase Diagrams: A second example 11
1.7 Laplace Transforms 12

2 Advanced ODE Topics **12**
2.1 Parameters in ODE 12
2.2 Boundary Value Problems 13
2.3 Event Location 14

1 Solving Ordinary Differential Equations in MATLAB

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

1.1 Finding Explicit Solutions

1.1.1 First Order Equations

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

$$y'(x) = xy. \tag{1.1}$$

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

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

Suppose we would like to evaluate our solution $y(x)$ at the point $x = 2$. We need only define x as 2 in the workspace and then evaluate y:

```
>>x=2;
>>eval(y)
```

On the other hand, we may want to find the value of x for which y is equal to 7. We want to use *solve()*, so our arguments should be strings. MATLAB's command for concatenating strings is *strcat()*. First, we convert y into a string with the command *char()*, and then we attach the string *'=7'*. We have,

```
>>solve(strcat(char(y),'=7'))
```

1.1.2 Second and Higher Order Equations

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

$$y''(x) + 8y'(x) + 2y(x) = \cos(x); \qquad y(0) = 0,\ y'(0) = 1. \tag{1.2}$$

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)
```

1.1.3 Systems

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

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

First, to find a general solution, we proceed as in Section 1.1.1, 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 1.1.1. 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 6.1.2.

```
>>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.

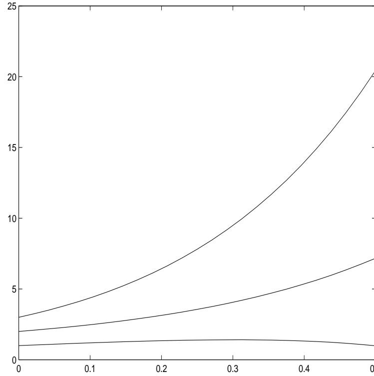

Figure 1.1: Solutions to equation (1.3).

1.2 Finding Numerical Solutions

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

1.2.1 First Order Equations

Suppose we want to numerically solve the first order ordinary differential equation $y'(x) = xy^2 + y$. First, we write an M-file, *firstode.m*, defining the function *yprime* as the derivative of y.[1]

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

Notice that all *firstode.m* does is take values x and y and return the value at the point (x, y) for the derivative $y'(x)$. A script for solving the ODE and plotting its solutions now takes the following form:

```
>>xspan = [0,.5];
>>y0 = 1;
>>[x,y]=ode23('firstode',xspan,y0);
>>x
x =
      0
 0.0500
 0.1000
 0.1500
 0.2000
 0.2500
 0.3000
 0.3500
 0.4000
 0.4500
 0.5000
```

[1] Actually, for an equation this simple, we don't have to work as hard as we're going to work here, but I'm giving you an idea of things to come.

```
>>y
y =
   1.0000
   1.0526
   1.1111
   1.1765
   1.2500
   1.3333
   1.4286
   1.5384
   1.6666
   1.8181
   1.9999
>>plot(x,y)
```

Notice that *xspan* is the domain of x for which we're asking MATLAB to solve the equation, and $y0 = 1$ means we're taking the initial value $y(0) = 1$. MATLAB solves the equation at discrete points and places the domain and range in vectors x and y. These are then easily manipulated, for example to plot the solution with *plot(x,y)*. Finally, observe that it is not the differential equation itself that goes in the function *ode23*, but rather the derivatives of the differential equation, which MATLAB assumes to be a first order system.

1.2.2 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 Section 1.1.2. 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 now record the derivatives of this system as a function file. We have

```
function yprime = secondode(x,y);
%SECONDODE: Computes the derivatives of y_1 and y_2,
%as a colum vector
yprime = [y(2); -8*y(2)-2*y(1)+cos(x)];
```

Observe that y_1 is stored as y(1) and y_2 is stored as y(2), each of which are column vectors. Additionally, *yprime* is a column vector, as is evident from the semicolon following the first appearance of $y(2)$. The MATLAB input and output for solving this ODE is given below.

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

>>[x,y]

ans =

        0    1.0000         0
   0.0001    1.0000   -0.0001
   0.0005    1.0000   -0.0005
   0.0025    1.0000   -0.0025
```

```
0.0124   0.9999   -0.0118
0.0296   0.9996   -0.0263
0.0531   0.9988   -0.0433
0.0827   0.9972   -0.0605
0.1185   0.9948   -0.0765
0.1613   0.9912   -0.0904
0.2113   0.9864   -0.1016
0.2613   0.9811   -0.1092
0.3113   0.9755   -0.1143
0.3613   0.9697   -0.1179
0.4113   0.9637   -0.1205
0.4613   0.9576   -0.1227
0.5000   0.9529   -0.1241
```

In the final expression above, the first column tabulates x values, while the second and third columns tabulate y_1 and y_2 (y(1) and (y(2))), or y and y'. Recall from Section 5 on matrices that for the matrix $y(m, n)$, m refers to the row and n refers to the column. Here, y is a matrix with two columns (y_1 and y_2) and 17 rows, one for each value of x. To get, for instance, the 4th entry of the vector y(1), type y(4,1)—4$^{\text{th}}$ row, 1$^{\text{st}}$ column. To refer to the entirety of y_1, use $y(:, 1)$, which MATLAB reads as *every* row, first column. Thus to plot y_1 versus y_2 we use *plot(y(:,1),y(:,2))*.

1.2.3 Solving Systems of ODE

Actually, we've already solved a system of ODE; the first thing we did in the previous example was convert our second order ODE into a first order system. As a second example, let's consider the famous Lorenz equations, which have some properties of equations arising in atmospherics, and whose solution has long served as an example for chaotic behavior. We have

$$\frac{dx}{dt} = -\sigma x + \sigma y \tag{1.4}$$

$$\frac{dy}{dt} = -y - xz \tag{1.5}$$

$$\frac{dz}{dt} = -bz + xy \quad -br, \tag{1.6}$$

where for the purposes of this example, we will take $\sigma = 10$, $b = 8/3$, and $r = 28$, 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;
b=8/3;
r=28;
xprime=[-sig*x(1) + sig*x(2); -x(2) - x(1)*x(3); -b*x(3) + x(1)*x(2) - b*r];
```

If in the Command Window, we type

```
>>tspan=[0,50];
>>[t,x]=ode45('lorenz',tspan,x0);
>>plot(x(:,1),x(:,3))
```

the famous "Lorenz strange attractor" is sketched (see Figure 1.2.)

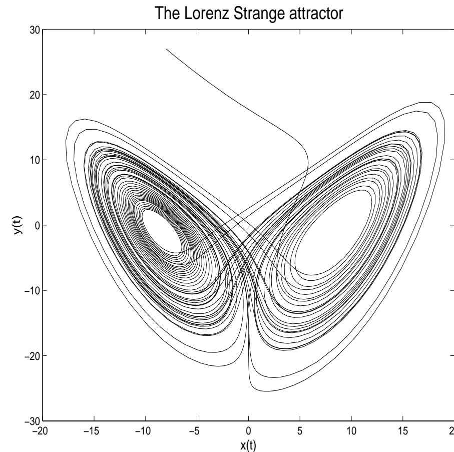

Figure 1.2: The Lorenz Strange Attractor

1.3 Plotting Direction Fields

A useful ODE technique involves plotting the *direction field* of solutions to the ODE of interest. Briefly, consider the general ODE

$$\frac{dy}{dx} = f(x, y).$$

If $f(x, y)$ is complicated, it may be impossible to find a solution $y(x)$ explicitly. At each point (x, y), however, it is certainly not difficult to find the slope of $y(x)$: by the definition of derivative, the slope is simply $f(x, y)$. To plot a direction field, we simply go to each point (x, y) in the x–y plane (actually, a discrete grid of points) and draw in the slope of the tangent line at this point. We will require a few new MATLAB commands to do this, so let's go through them one at a time. The first is *meshgrid().* The command

```
>>[x,y]=meshgrid(a:k:b, c:j:d)
```

creates a set of points (x, y), where x lies between a and b, incremented by k, and y lies between c and d, incremented by j. For example, *[x,y]=meshgrid(1:.5:2,0:1:2)* creates the set of nine points: $(1, 0)$, $(1, 1)$, $(1, 2)$, $(1.5, 0)$, $(1.5, 1)$, $(1.5, 2)$, $(2, 0)$, $(2, 1)$, $(2, 2)$. The second new command we will need is *quiver().* Aptly named, the command *quiver(a,b,x,y)* begins at the point (a, b) and plots an arrow in the direction of the vector $\overrightarrow{v} = (x, y)$. For example, *quiver(0,0,1,1)* begins at the point $(0, 0)$ and draws an arrow inclined 45° from the horizontal. Recalling that the vector $\overrightarrow{v} = (x, y)$ has slope $\frac{y}{x}$, we make the useful observation that the command *quiver(x ,y ,dx, dy)* begins at the point (x, y) and draws a vector with slope $\frac{dy}{dx}$. Fortunately, the last two commands we'll need are significantly easier to master. The command *size(A)* simply returns the dimensions of the matrix A, while *ones(m,n)* creates a matrix with m rows and n columns with a 1 in each entry. Combining these, we find that the direction field for $y'(x) = x + \sin(y)$ (given in Figure 1.3) can be created by the following commands.

```
>>[x,y]=meshgrid(-3:.3:3,-2:.3:2);
>>dy = x + sin(y);
>>dx=ones(size(dy));
>>quiver(x,y,dx,dy)
```

The expression *dx=ones(size(dy))* may become more clear if you think of our ODE as

$$\frac{dy}{dx} = \frac{f(x, y)}{1}.$$

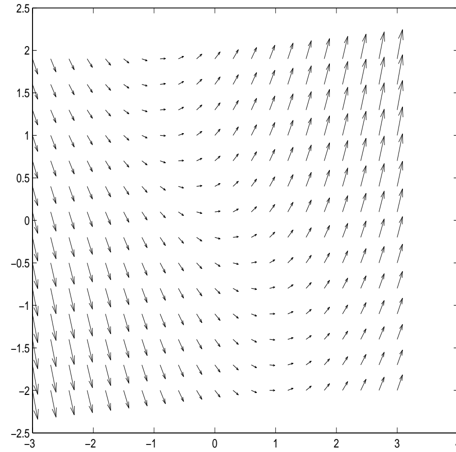

Figure 1.3: Direction field for $y'(x) = x + \sin(y)$.

In this case, $dy = f(x, y) = x + \sin(y)$ and $dx = 1$.

Looking at Figure 1.3, we observe that each arrow has a different length. Since we're only really concerned with direction here, we might make all vectors unit length. For this, we use

```
>>[x,y]=meshgrid(-3:.3:3,-2:.3:2);
>>dy = x+sin(y);
>>dx = ones(size(dy));
>>dyu = dy./sqrt(dx.^2+dy.^2);
>>dxu = dx./sqrt(dx.^2+dy.^2);
>>quiver(x,y,dxu,dyu)
```

which produces Figure 1.4.

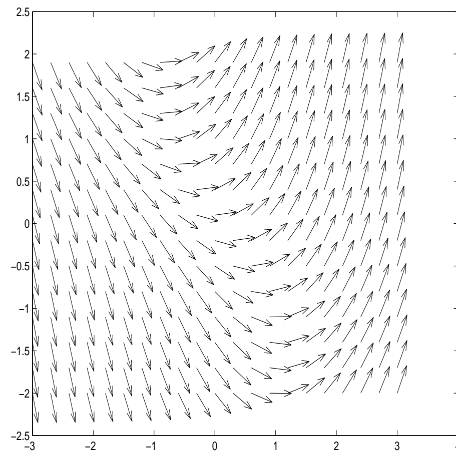

Figure 1.4: Normalized direction field for $y'(x) = x + \sin(y)$.

1.4 Plotting Direction Fields: A Second Example

As a second example of a direction field, consider the ODE

$$\frac{dy}{dx} = x^2 - y.$$

Proceeding almost exactly as above, we have

```
>>[x,y]=meshgrid(-3:.5:3,-3:.5:3);
>>dy=x.^2-y;
>>dx=ones(size(dy));
>>dyu=dy./sqrt(dy.^2+dx.^2);
>>dxu=dx./sqrt(dy.^2+dx.^2);
>>quiver(x,y,dxu,dyu)
```

We obtain Figure 1.5.

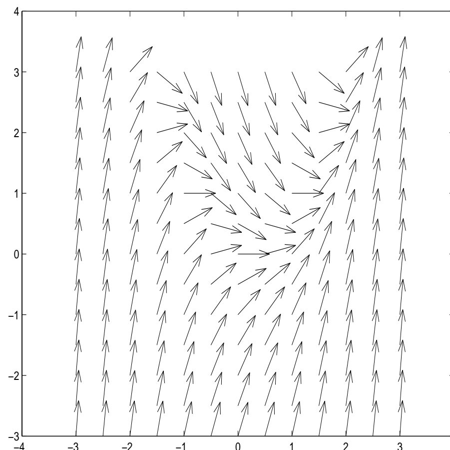

Figure 1.5: Normalized direction field for $y'(x) = x^2 - y$.

1.5 Plotting Phase Diagrams

A topic closely related to direction fields is *phase diagrams*, a critical tool in the study of systems of two differential equations. A first order system of ordinary differential equations is called autonomous if it can be written in the form

$$\begin{aligned} \tfrac{dx}{dt} &= f(x,y) \\ \tfrac{dy}{dt} &= g(x,y). \end{aligned}$$

That is, if the independent variable—t, here—does not explicitly appear on the right-hand side. Such systems are often too difficult to analyze exactly; however, we can obtain a great deal of information from the following clever trick. Dividing $\frac{dy}{dt}$ by $\frac{dx}{dt}$, we have (forgoing even so much as a nod toward rigor)

$$\frac{dy}{dx} = \frac{\frac{dy}{dt}}{\frac{dx}{dt}} = \frac{g(x,y)}{f(x,y)},$$

or the *phase equation*

$$\frac{dy}{dx} = \frac{g(x,y)}{f(x,y)}.$$

For reasons I'll point out below, we refer to plots y versus x as phase diagrams.

The standard introductory example to phase diagrams involves the equations of a pendulum. If $x(t)$ represents the angle with which the pendulum is displaced from the vertical at time t, and $y(t)$ represents its angular velocity (that is, $y(t) = x'(t)$), then Newton's second law asserts that

$$\begin{aligned} \frac{dx}{dt} &= y \\ \frac{dy}{dt} &= -\frac{g}{l} \sin(x), \end{aligned}$$

where $g = 9.81\text{m/s}^2$ is approximately acceleration due to gravity at the earth's surface, and l is the length of the pendulum (which we will take to be 1). (See Problem 7 in Section 4.10 of [NSS] for a discussion of the derivation of this equation.) Our phase plane equation becomes

$$\frac{dy}{dx} = -\frac{\frac{g}{l}\sin(x)}{y}, \tag{1.7}$$

from which we observe immediately that $y = 0$ may cause some trouble.

Following our remarks from Section 6.3, we can have MATLAB sketch in a direction field for (1.7). The following commands suffice (though, for reasons discussed below, the resulting figure is disappointing):

```
>>[x,y]=meshgrid(-5:.4:5,-10:.51:10);
>>dy = -9.81*sin(x)./y;
>>dx = ones(size(dy));
>>dxu=dx./sqrt(dx.^2+dy.^2);
>>dyu=dy./sqrt(dx.^2+dy.^2);
>>quiver(x,y,dxu,dyu)
```

Notice that I've chosen my grid for y so that y is never 0. Clearly, except at points x for which $\sin(x)$ is 0, the slope for $y = 0$ is infinite, corresponding with a vertical line. Unfortunately, this simple analysis doesn't yield quite as much information as we would like. The main problem is that our arrows all point toward the right, as though the "independent" variable x were marching steadily forward (as truly independent variables do). But what we really want here is to know what direction the solutions are moving as t moves foward. For this, we will have to go back to our original system of differential equations.

As usual, we will take up and right to be our positive directions and left and down to be our negative directions. To determine the direction in which $x(t)$ is moving as time increases all we have to study is its change with respect to time, or its derivative. Glancing at our equations, we see that $\frac{dx}{dt} = y$, so that if $y > 0$, then $x(t)$ must be increasing, or moving to the right, and if $y < 0$, then $x(t)$ must be decreasing, or moving to the left. If x is increasing, we don't need to change anything (recall that our problem was that we took x to be increasing everywhere). If it's decreasing, that is, if $y < 0$, then we must change the sign of both dx and dy, to turn the vector around. The upshot of all this is that we cannot lump the expression for dx into the one for dy. Which is to say, dx can no longer be considered identically one. [2]The following MATLAB code suffices to give Figure 1.6.

```
>>[x,y]=meshgrid(-5:.4:5,-10:.51:10);
>>dy=-9.81*sin(x).*sign(y).^2;
>>dx = y.*sign(x).^2;
>>dxu=dx./sqrt(dx.^2+dy.^2);
>>dyu=dy./sqrt(dx.^2+dy.^2);
>>quiver(x,y,dxu,dyu)
```

The only new command here is *sign()*, which returns a 1 for each matrix entry that is positive, a -1 for each matrix entry that is negative, and a 0 for each matrix entry that is 0. In order to insure that the dx and

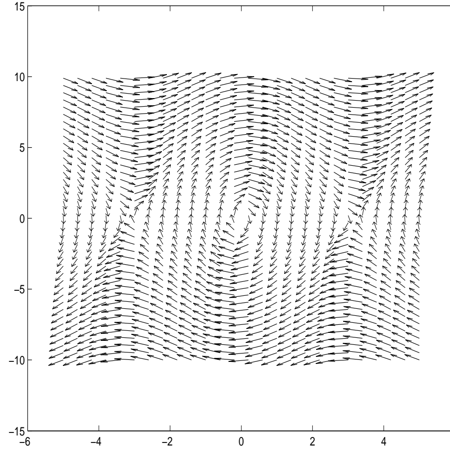

Figure 1.6: Phase Diagram for Pendulum Equations.

dy are both computed at each grid point (x, y), both x and y must appear in each expression, dy and dx. Hence, the occurence of *sign(x).^2* and *sign(y).^2*, which are both identically 1.

Observe that we can recover quite a bit of information from Figure 1.6. For example, notice that the arrows form circles around the point $(0, 0)$. Each circle corresponds with a steady swing of the pendulum, the larger the circle, the greater the swing. We call the point $(0, 0)$ a stable equilibirum point. Alternatively, notice that the equilibrium point $(\pi, 0)$ and $(-\pi, 0)$ are both unstable. (What do they correspond with physically?)

A final word: the expression *phase diagram* arose because the angle x is also referred to as the phase of the pendulum.

1.6 Phase Diagrams: A second example

As a second example, let's consider the phase diagram arising in the case of the Lotka–Volterra predator–prey model

$$\begin{aligned} x'(t) &= \quad ax(t) - bx(t)y(t) \\ y'(t) &= \quad -ry(t) + cx(t)y(t), \end{aligned}$$

where $x(t)$ typically presents the prey (the typical example is a rabbit) and $y(t)$ represents the predator (a wildcat, for example). For this system, the phase plane equation becomes

$$\frac{dy}{dx} = \frac{-ry + cxy}{ax - bxy},$$

which is too difficult to solve exactly—at least by routine methods. For the purposes of this example, we will take $a = r = 2$, $b = c = 1$. First notice that our grid should avoid the points $x = 0$ and $y = 2$. The following MATLAB script suffices to produce Figure 1.7.

```
>>[x,y]=meshgrid(.1:.2:4,.1:.2:4);
>>dy=-2*y+x.*y;
>>dx=2*x-x.*y;
>>dyu=dy./sqrt(dy.^2+dx.^2);
>>dxu=dx./sqrt(dy.^2+dx.^2);
>>quiver(x,y,dxu,dyu)
```

[2] Recall that the vector oppositely directly from (a, b) is $(-a, -b)$: both components must be multiplied by negative one.

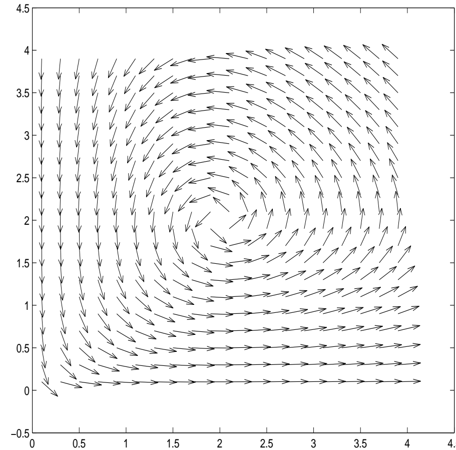

Figure 1.7: Phase Diagram for Pendulum Equations.

1.7 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))
```

2 Advanced ODE Topics

In Section 1 we covered the basic ODE topics for a generic introductory course. Here, we will discuss some topics that will be of particular importance for the projects we'll consider in M442.

2.1 Parameters in ODE

It is often the case that we have some parameter in our differential equation that needs to be determined during the course of our analysis. For example, in the ballistics project, you will need to consider the coefficient of air resistance, b, for an object projected through the air. For the case of linear air resistance, this can be accomplished in a straightforward manner from an exact solution $y(t)$ and appropriate experimental data. For the case of nonlinear air resistance, however, an exact solution is cumbersome, and we are best served by carrying out the analysis directly from the ODE. As an example, let's study how this approach could be carried out in the case of linear air resistance.

First, let's recall what we're after. We are given that when dropped from a height $y(0) = 4.06$ m, the projectile takes $t = .95$s to hit the ground. Therefore, we want to find the value of b so that if we solve the initial value problem

$$\frac{d^2y}{dt^2} = -g - b\frac{dy}{dt}, \qquad y(0) = 4.06, \qquad y'(0) = 0, \tag{2.1}$$

we obtain $y(.95) = 0$. In what follows we will consider $y(.95)$ as a function of b and find its zeros. As usual, we begin by writing the derivatives of our function as an M-file:

```
function ydot=linsys(t,y);
%LINSYS: ODE for linear air resistance
g=9.81;
global b;
ydot=[y(2);-g-b*y(2)];
```

The only new command here is *global b*, which will allow the value of the parameter b to move from one M-file to the next, and to the MATLAB command line as well. Notice that for script M-files, all variables are global by definition—running a script M-file is equivalent to typing the commands one after the other into the command line. Not so for function M-files. We now define a second function *linheight.m*:

```
function height=linheight(bb);
%LINHEIGHT: Solves ODE for linear air resistance
global b;
b = bb;
tspan = [0 .95];
y0 = [4.06 0];
[t,z]=ode23('linsys', tspan, y0);
height = z(length(t)); %Extracts z at t = .95 (final value)
```

With these two M-files thus defined, we find b simply by typing

```
>>fzero('linheight',1)
```

Though this is the most complicated combination of files we've seen, the only new command in *length(V)*, which gives the number of elements in the vector V. In this case, *length(t)* is the index of the last component of t, so that *z(length(t))* is the value of z at $t = .95$.

2.2 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). Again, let's look to the model of linear air resistance for an example. In the Ballistics project, we are told that when fired straight up from a height of $y(0) = .39$m, the darts take 2.13 seconds to hit the ground—which is to say, $y(2.13) = 0$. Consequently, we have the boundary value problem

$$\frac{d^2y}{dt^2} = -g - b\frac{dy}{dt}, \qquad y(0) = .39, \qquad y(2.13) = 0, \tag{2.2}$$

where we think of the points $t = 0$ and $t = 2.13$ as the endpoints, or boundaries, of the domain we are interested in. We could solve this particular equation by obtaining a general solution for the ODE and using $y(0)$ and $y(2.13)$ to evaluate the constants of integration (which is what you should have done for the linear air resistance part of the project), but the point here is to study how we could solve (2.2) with MATLAB. The first step in this analysis is again to write (2.2) as a first order system, and to write the right hand side of this system as a function file. Fortunately, this information has already been stored as *linsys.m* from Section 2.1 (you need only alter it by inserting the value you found in Section 2.1 for b). Next we write the boundary conditions as a very simple M-file:

```
function res=bc(ya,yb)
%BC: Evaluates the residue of the boundary condition
res=[ya(1)-.39;yb(1)];
```

It should be clear that by *residue* we mean the left hand side of the boundary condition after it has been set to zero. We now solve the BVP (2.2) with the commands

```
>>sol=bvpinit([0 .25 .5 .75 1 1.25 1.5 1.75 2 2.13],[1,0]);
>>sol=bvp4c(@linsys,@bc,sol);
>>sol.y

ans =

  Columns 1 through 7

   0.3900    2.8036    4.4619    5.4160    5.7133    5.3985    4.5127
  11.2370    8.1086    5.1918    2.4722   -0.0636   -2.4279   -4.6324

  Columns 8 through 10

   3.0947    1.1803         0
  -6.6878   -8.6043   -9.5491
```

Observe that *sol.y* contains values of y and y' at each t-value specified in the first brackets of *bvpinit*. One value you might find interesting above is 11.2370, from the first column. This is the initial velocity, v, with which the projectile is launched, assuming linear damping.

2.3 Event Location

A MATLAB tool that will save us a lot of work in the nonlinear part of the ballistics project is *event location.* Typically, the ODE solver in MATLAB terminates after solving the ODE over a specified domain of the independent variable (xspan=[0,.5] above, where the independent variable is x). In applications, however, we often would like to stop the solution at a particular value of the dependent variable (for example, when a pendulum reaches the peak of its arc, or when a population crosses some threshhold value). For example, in the ballistics project, we are interested in finding the value of $x(t)$ (the distance) at the time t for which $y(t) = 0$—when the dart strikes the ground. In order to give you the idea of how event location works, let's work through the case you did by hand: linear damping. Here, you have the system of equations,

$$y''(t) = -g - by'(t); \qquad y(0) = .18; \quad y'(0) = v\sin\theta \tag{2.3}$$

$$x''(t) = -bx'(t); \qquad x(0) = 0; \quad x'(0) = v\cos\theta \tag{2.4}$$

where now $y(0) = .18$ indicates that the object has been fired from a platform .18 meters off the ground, θ is the angle of inclination with which the object was fired, and v is the initial velocity with which the object was launched. Suppose our goal is to find the distance the object travels prior to hitting the ground. As usual, the first step consists in writing (2.4) as a first order system. We can do this with the substitutions $z_1 = y$, $z_2 = x$, $z_3 = y'$, and $z_4 = x'$, which gives

$$\begin{aligned} z_1' &= z_3 \\ z_2' &= z_4 \\ z_3' &= -g - bz_3 \\ z_4' &= -bz_4 \end{aligned}$$

The event we will be looking for is the object's hitting the ground, or $y(t) = 0$. Notice that so long as $y(0) \neq 0$, the first time for which $y(t) = 0$ will correspond to the object's hitting the ground. In order to be a bit more general, however, we will also specify the direction we would like the object to be going—down.

As usual, we begin by writing the right-hand side of our system as a MATLAB function M-file, in this case *linproj.m*.

```
function zprime = linproj(t,z);
%PROJ: ODE for projectile motion with linear
%air resistance.
g=9.81; %Specify approximate gravitational constant
b=.28; %Representative value
zprime=[z(3);z(4);-g-b*z(3);-b*z(4)];
```

Additionally, we now define an *events* function that specifies what event we want MATLAB to look for and what MATLAB should do when it finds the event.

```
function [lookfor,stop,direction] = linevent(t,z);
%LINEVENT: Contains the event we are looking for.
%In this case, z(1) = 0 (hitting ground).
lookfor = z(1); %Sets this to 0
stop = 1; %Stop when event is located
direction = -1; %Specify downward direction
```

In *linevent.m*, the line *lookfor=z(1)* specifies that MATLAB should look for the event $z(1) = 0$ (that is, $y(t) = 0$). (If we wanted to look for the event $y(t) = 1$, we would use *lookfor=z(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 y(2) (that is, y') is negative.

We can now solve the ODE up until the time our projectile strikes the ground with the following commands issued in the Command Window:

```
>>options=odeset('Events',@linevent)
>>z0=[.18;0;5;10];
>>[t,z,te,ze,ie]=ode45(@linode,[0 10],z0,options)
```

Here, $z0$ is a vector of initial data (though the velocities don't represent any particular angle). The command *oder45()* returns a vector of times t, a matrix of dependent variables z, the time at which the event occurred, te, the values of z when the event occurred, ze, and finally the index when the event occurred, ie.

Index

boundary value problems, 13

direction fields, 7
dsolve, 1

eval, 2
event location, 14

global, 13

inverse laplace, 12

laplace, 12
Laplace transforms, 12
length(), 13

meshgrid, 7

ode, 4
ones(), 7

parameters in ODE, 12
phase diagrams, 9

quiver, 7

sign, 10
size(), 7

vectorize(), 2