Chapter 15

Ordinary Differential Equations

Mathematical models in many different fields.

Systems of differential equations form the basis of mathematical models in a wide range of fields – from engineering and physical sciences to finance and biological sciences. Differential equations are relations between unknown functions and their derivatives. Computing numerical solutions to differential equations is one of the most important tasks in technical computing, and one of the strengths of MATLAB.

If you have studied calculus, you have learned a kind of mechanical process for differentiating functions represented by formulas involving powers, trig functions, and the like. You know that the derivative of x^3 is $3x^2$ and you may remember that the derivative of $\tan x$ is $1 + \tan^2 x$. That kind of differentiation is important and useful, but not our primary focus here. We are interested in situations where the functions are not known and cannot be represented by simple formulas. We will compute numerical approximations to the values of a function at enough points to print a table or plot a graph.

Imagine you are traveling on a mountain road. Your altitude varies as you travel. The altitude can be regarded as a function of time, or as a function of longitude and latitude, or as a function of the distance you have traveled. Let's consider the latter. Let x denote the distance traveled and $y = y(x)$ denote the altitude. If you happen to be carrying an altimeter with you, or you have a deluxe GPS system, you can collect enough values to plot a graph of altitude versus distance, like the first plot in figure 15.1.

Suppose you see a sign saying that you are on a 6% uphill grade. For some

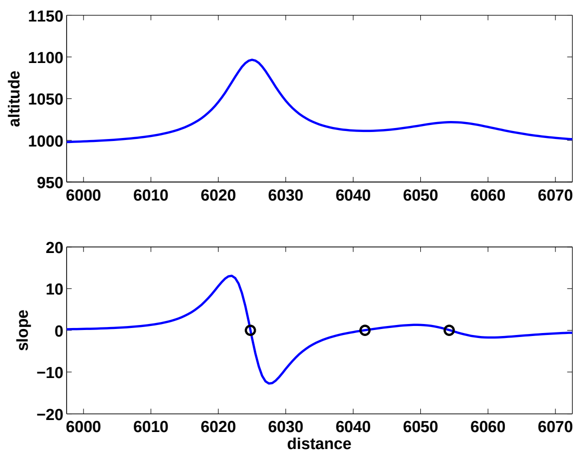

Figure 15.1. *Altitude along a mountain road, and derivative of that altitude. The derivative is zero at the local maxima and minima of the altitude.*

value of x near the sign, and for $h = 100$, you will have

$$\frac{y(x+h) - y(x)}{h} = .06$$

The quotient on the left is the *slope* of the road between x and $x + h$.

Now imagine that you had signs every few meters telling you the grade at those points. These signs would provide approximate values of the rate of change of altitude with respect to distance traveled, This is the derivative dy/dx. You could plot a graph of dy/dx, like the second plot in figure 15.1, even though you do not have closed-form formulas for either the altitude or its derivative. This is how MATLAB solves differential equations. Note that the derivative is positive where the altitude is increasing, negative where it is decreasing, zero at the local maxima and minima, and near zero on the flat stretches.

Here is a simple example illustrating the numerical solution of a system of differential equations. Figure 15.2 is a screen shot from Spacewar, the world's first video game. Spacewar was written by Steve "Slug" Russell and some of his buddies at MIT in 1962. It ran on the PDP-1, Digital Equipment Corporation's first computer. Two space ships, controlled by players using switches on the PDP-1 console, shoot space torpedoes at each other.

The space ships and the torpedoes orbit around a central star. Russell's program needed to compute circular and elliptical orbits, like the path of the torpedo in the screen shot. At the time, there was no MATLAB. Programs were written in terms of individual machine instructions. Floating-point arithmetic was so slow that it was desirable to avoid evaluation of trig functions in the orbit calculations. The orbit-generating program looked something like this.

```
x = 0
y = 32768
```

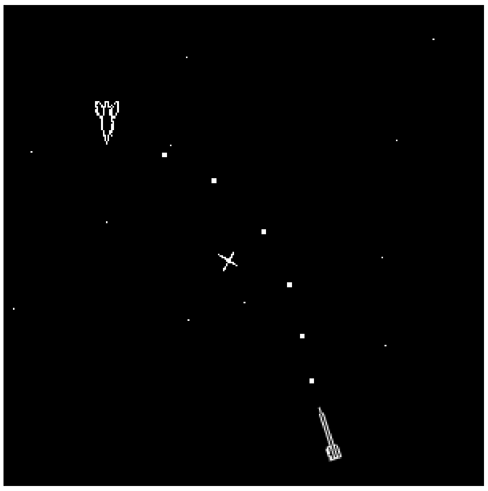

Figure 15.2. *Spacewar, the world's first video game. The gravitational pull of the central star causes the torpedo to move in an elliptical orbit.*

```
L: plot x y
   load y
   shift right 2
   add x
   store in x
   change sign
   shift right 2
   add y
   store in y
   go to L
```

What does this program do? There are no trig functions, no square roots, no multiplications or divisions. Everything is done with shifts and additions. The initial value of y, which is 2^{15}, serves as an overall scale factor. All the arithmetic involves a single integer register. The "shift right 2" command takes the contents of this register, divides it by $2^2 = 4$, and discards any remainder.

If Spacewar orbit generator were written today in MATLAB, it would look something the following. We are no longer limited to integer values, so we have changed the scale factor from 2^{15} to 1.

```
x = 0;
y = 1;
h = 1/4;
n = 2*pi/h;
plot(x,y,'.')
for k = 1:n
```

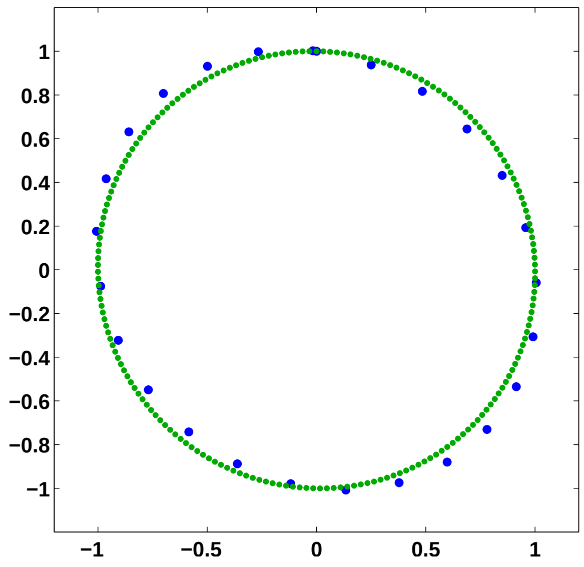

Figure 15.3. *The 25 blue points are generated by the Spacewar orbit generator with a step size of 1/4. The 201 green points are generated with a step size of 1/32.*

```
        x = x + h*y;
        y = y - h*x;
        plot(x,y,'.')
    end
```

The output produced by this program with `h = 1/4` and `n = 25` is shown by the blue dots in figure 15.3. The blue orbit is actually an ellipse that deviates from an exact circle by about 7%. The output produced with `h = 1/32` and `n = 201` is shown by the green dots. The green orbit is another ellipse that deviates from an exact circle by less than 1%.

Think of `x` and `y` as functions of time, t. We are computing $x(t)$ and $y(t)$ at discrete values of t, incremented by the step size h. The values of x and y at time $t+h$ are computed from the values at time t by

$$\begin{aligned} x(t+h) &= x(t) + hy(t) \\ y(t+h) &= y(t) - hx(t+h) \end{aligned}$$

This can be rewritten as

$$\begin{aligned} \frac{x(t+h) - x(t)}{h} &= y(t) \\ \frac{y(t+h) - y(t)}{h} &= -x(t+h) \end{aligned}$$

You have probably noticed that the right hand side of this pair of equations involves two different values of the time variable, t and $t + h$. That fact turns out to be important, but let's ignore it for now.

Look at the left hand sides of the last pair of equations. The quotients are approximations to the derivatives of $x(t)$ and $y(t)$. We are looking for two functions with the property that the derivative of the first function is equal to the second and the derivative of the second function is equal to the negative of the first.

In effect, the Spacewar orbit generator is using a simple numerical method involving a step size h to compute an approximate solution to the system of differential equations

$$\begin{aligned}\dot{x} &= y\\ \dot{y} &= -x\end{aligned}$$

The dot over x and y denotes differentiation with respect to t.

$$\dot{x} = \frac{dx}{dt}$$

The initial values of x and y provide the initial conditions

$$\begin{aligned}x(0) &= 0\\ y(0) &= 1\end{aligned}$$

The exact solution to the system is

$$\begin{aligned}x(t) &= \sin t\\ y(t) &= \cos t\end{aligned}$$

To see why, recall the trig identities

$$\begin{aligned}\sin(t+h) &= \sin t\cos h + \cos t \sin h\\ \cos(t+h) &= \cos t \cos h - \sin t \sin h\end{aligned}$$

For small values of h,

$$\begin{aligned}\sin h &\approx h,\\ \cos h &\approx 1\end{aligned}$$

Consequently

$$\begin{aligned}\frac{\sin(t+h) - \sin t}{h} &\approx \cos t,\\ \frac{\cos(t+h) - \cos t}{h} &\approx -\sin t,\end{aligned}$$

If you plot $x(t)$ and $y(t)$ as functions of t, you get the familiar plots of sine and cosine. But if you make a *phase plane* plot, that is $y(t)$ versus $x(t)$, you get a circle of radius 1.

It turns out that the solution computed by the Spacewar orbit generator with a fixed step size h is an ellipse, not an exact circle. Decreasing h and taking more steps generates a better approximation to a circle. Actually, the fact that $x(t+h)$ is used instead of $x(t)$ in the second half of the step means that the method is not quite as simple as it might seem. This subtle change is responsible for the fact that the method generates ellipses instead of spirals. One of the exercises asks you to verify this fact experimentally.

Mathematical models involving systems of ordinary differential equations have one *independent* variable and one or more *dependent* variables. The independent variable is usually time and is denoted by t. In this book, we will assemble all the dependent variables into a single vector y. This is sometimes referred to as the *state* of the system. The state can include quantities like position, velocity, temperature, concentration, and price.

In MATLAB a system of odes takes the form

$$\dot{y} = F(t, y)$$

The function F always takes two arguments, the scalar independent variable, t, and the vector of dependent variables, y. A program that evaluates $F(t, y)$ hould compute the derivatives of all the state variables and return them in another vector.

In our circle generating example, the state is simply the coordinates of the point. This requires a change of notation. We have been using $x(t)$ and $y(t)$ to denote position, now we are going to use $y_1(t)$ and $y_2(t)$. The function F defines the velocity.

$$\begin{aligned} \dot{y}(t) &= \begin{pmatrix} \dot{y}_1(t) \\ \dot{y}_2(t) \end{pmatrix} \\ &= \begin{pmatrix} y_2(t) \\ -y_1(t) \end{pmatrix} \end{aligned}$$

MATLAB has several functions that compute numerical approximations to solutions of systems of ordinary differential equations. The suite of ode solvers includes `ode23`, `ode45`, `ode113`, `ode23s`, `ode15s`, `ode23t`, and `ode23tb`. The digits in the names refer to the *order* of the underlying algorithms. The order is related to the complexity and accuracy of the method. All of the functions automatically determine the step size required to obtain a prescribed accuracy. Higher order methods require more work per step, but can take larger steps. For example `ode23` compares a second order method with a third order method to estimate the step size, while `ode45` compares a fourth order method with a fifth order method.

The letter "`s`" in the name of some of the ode functions indicates a *stiff* solver. These methods solve a matrix equation at each step, so they do more work per step than the nonstiff methods. But they can take much larger steps for problems where numerical stability limits the step size, so they can be more efficient overall.

You can use `ode23` for most of the exercises in this book, but if you are interested in the seeing how the other methods behave, please experiment.

All of the functions in the ode suite take at least three input arguments.

- `F`, the function defining the differential equations,
- `tspan`, the vector specifying the integration interval,

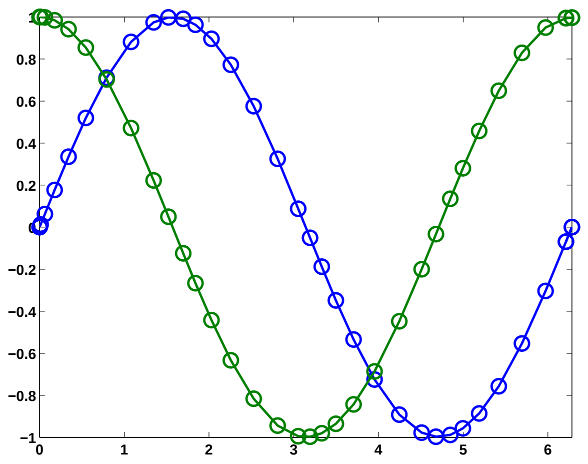

Figure 15.4. *Graphs of sine and cosine generated by ode23.*

- `y0`, the vector of initial conditions.

There are several ways to write the function describing the differential equation. Anticipating more complicated functions, we can create a MATLAB program for our circle generator that extracts the two dependent variables from the state vector. Save this in a file named `mycircle.m`.

```
function ydot = mycircle(t,y)
ydot = [y(2); -y(1)];
```

Notice that this function has two input arguments, `t` and `y`, even though the output in this example does not depend upon `t`.

With this function definition stored in `mycircle.m`, the following code calls `ode23` to compute the solution over the interval $0 \le t \le 2\pi$, starting with $x(0) = 0$ and $y(0) = 1$.

```
tspan = [0 2*pi];
y0 = [0; 1];
ode23(@mycircle,tspan,y0)
```

With no output arguments, the ode solvers automatically plot the solutions. Figure 15.4 is the result for our example. The small circles in the plot are not equally spaced. They show the points chosen by the step size algorithm.

To produce the phase plot plot shown in figure 15.5, capture the output and plot it yourself.

```
tspan = [0 2*pi];
y0 = [0; 1];
[t,y] = ode23(@mycircle,tspan,y0)
plot(y(:,1),y(:,2)'-o')
```

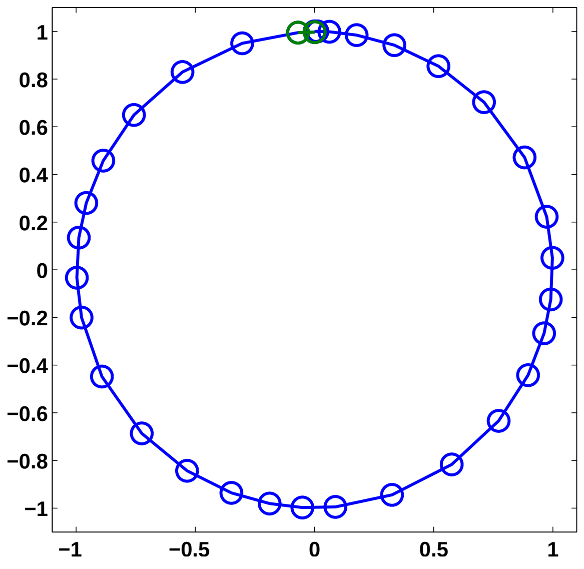

Figure 15.5. *Graph of a circle generated by ode23.*

```
axis([-1.1 1.1 -1.1 1.1])
axis square
```

The circle generator example is so simple that we can bypass the creation of the function file `mycircle.m` and write the function in one line.

```
acircle = @(t,y) [y(2); -y(1)]
```

The expression created on the right by the "`@`" symbol is known as an *anonymous function* because it does not have a name until it is assigned to `acircle`. Since the "`@`" sign is included in the definition of `acircle`, you don't need it when you call an ode solver.

Once `acircle` has been defined, the statement

```
ode23(acircle,tspan,y0)
```

automatically produces figure 15.4. And, the statement

```
[t,y] = ode23(acircle,tspan,y0)
```

captures the output so you can process it yourself.

Many additional options for the ode solvers can be set via the function `odeset`. For example

```
opts = odeset('outputfcn',@odephas2)
ode23(acircle,tspan,y0,opts)
axis square
axis([-1.1 1.1 -1.1 1.1])
```

will also produce figure 15.5.

Use the command

```
doc ode23
```

to see more details about the MATLAB suite of ode solvers. Consult the ODE chapter in our companion book, *Numerical Computing with MATLAB*, for more of the mathematical background of the ode algorithms, and for `ode23tx`, a textbook version of `ode23`.

Here is a very simple example that illustrates how the functions in the ode suite work. We call it "`ode1`" because it uses only one elementary first order algorithm, known as Euler's method. The function does not employ two different algorithms to estimate the error and determine the step size. The step size `h` is obtained by dividing the integration interval into 200 equal sized pieces. This would appear to be appropriate if we just want to plot the solution on a computer screen with a typical resolution, but we have no idea of the actual accuracy of the result.

```
function [t,y] = ode1(F,tspan,y0)
% ODE1  World's simplest ODE solver.
%   ODE1(F,[t0,tfinal],y0) uses Euler's method to solve
%      dy/dt = F(t,y)
%   with y(t0) = y0 on the interval t0 <= t <= tfinal.

t0 = tspan(1);
tfinal = tspan(end);
h = (tfinal - t0)/200;
y = y0;
for t = t0:h:tfinal
   ydot = F(t,y);
   y = y + h*ydot;
end
```

However, even with 200 steps this elementary first order method does not have satisfactory accuracy. The output from

```
[t,y]] = ode1(acircle,tspan,y0)
```

is

```
t =
   6.283185307179587
y =
   0.032392920185564
   1.103746317465277
```

We can see that the final value of `t` is `2*pi`, but the final value of `y` has missed returning to its starting value by more than 10 percent. Many more smaller steps would be required to get graphical accuracy.

Recap

```
%% Ordinary Differential Equations Chapter Recap
% This is an executable program that illustrates the statements
% introduced in the Ordinary Differential Equations Chapter
% of "Experiments in MATLAB".
% You can access it with
%
%    odes_recap
%    edit odes_recap
%    publish odes_recap
%
% Related EXM programs
%
%    ode1

%% Spacewar Orbit Generator.
    x = 0;
    y = 1;
    h = 1/4;
    n = 2*pi/h;
    plot(x,y,'.')
    hold on
    for k = 1:n
       x = x + h*y;
       y = y - h*x;
       plot(x,y,'.')
    end
    hold off
    axis square
    axis([-1.1 1.1 -1.1 1.1])

%% An Anonymous Function.
    acircle = @(t,y) [y(2); -y(1)];

%% ODE23 Automatic Plotting.
    figure
    tspan = [0 2*pi];
    y0 = [0; 1];
    ode23(acircle,tspan,y0)

%% Phase Plot.
    figure
    tspan = [0 2*pi];
    y0 = [0; 1];
    [t,y] = ode23(acircle,tspan,y0)
    plot(y(:,1),y(:,2),'-o')
    axis square
    axis([-1.1 1.1 -1.1 1.1])

%% ODE23 Automatic Phase Plot.
    opts = odeset('outputfcn',@odephas2)
    ode23(acircle,tspan,y0,opts)
    axis square
    axis([-1.1 1.1 -1.1 1.1])

%% ODE1 implements Euler's method.
% ODE1 illustrates the structure of the MATLAB ODE solvers,
% but it is low order and employs a coarse step size.
% So, even though the exact solution is periodic, the final value
% returned by ODE1 misses the initial value by a substantial amount.

    type ode1
    [t,y] = ode1(acircle,tspan,y0)
    err = y - y0
```

Exercises

15.1 *Walking to class.* You leave home (or your dorm room) at the usual time in the morning and walk toward your first class. About half way to class, you realize that you have forgotten your homework. You run back home, get your homework, run to class, and arrive at your usual time. Sketch a rough graph by hand showing your distance from home as a function of time. Make a second sketch of your velocity as a function of time. You do not have to assume that your walking and running velocities are constant, or that your reversals of direction are instantaneous.

15.2 *Divided differences.* Create your own graphic like our figure 15.1. Make up your own data, `x` and `y`, for distance and altitude. You can use

```
    subplot(2,1,1)
```

and

```
    subplot(2,1,2)
```

to place two plots in one figure window. The statement

```
    d = diff(y)./diff(x)
```

computes the *divided difference* approximation to the derivative for use in the second subplot. The length of the vector `d` is one less than the length of `x` and `y`, so you can add one more value at the end with

```
d(end+1) = d(end)
```

For more information about `diff` and `subplot`, use

```
help diff
help subplot
```

15.3 *Orbit generator.* Here is a complete MATLAB program for the orbit generator, including appropriate setting of the graphics parameters. Investigate the behavior of this program for various values of the step size `h`.

```
axis(1.2*[-1 1 -1 1])
axis square
box on
hold on
x = 0;
y = 1;
h = ...
n = 2*pi/h;
plot(x,y,'.')
for k = 1:n
   x = x + h*y;
   y = y - h*x;
   plot(x,y,'.')
end
```

15.4 *Modified orbit generator.* Here is a MATLAB program that makes a simpler approximation for the orbit generator. What does it do? Investigate the behavior for various values of the step size `h`.

```
axis(1.5*[-1 1 -1 1])
axis square
box on
hold on
x = 0;
y = 1;
h = 1/32;
n = 6*pi/h;
plot(x,y,'.')
for k = 1:n
   savex = x;
   x = x + h*y
   y = y - h*savex;
   plot(x,y,'.')
end
```

15.5 *Linear system* Write the system of differential equations

$$\begin{aligned} \dot{y_1} &= y_2 \\ \dot{y_2} &= -y_1 \end{aligned}$$

in matrix-vector form,

$$\dot{y} = Ay$$

where y is a vector-valued function of time,

$$y(t) = \begin{pmatrix} y_1(t) \\ y_2(t) \end{pmatrix}$$

and A is a constant 2-by-2 matrix. Use our `ode1` as well as `ode23` to experiment with the numerical solution of the system in this form.

15.6 *Example from ode23.* The first example in the documentation for `ode23` is

$$\begin{aligned} \dot{y_1} &= y_2\, y_3 \\ \dot{y_2} &= -y_1\, y_3 \\ \dot{y_3} &= -0.51\, y_1\, y_2 \end{aligned}$$

with initial conditions

$$\begin{aligned} y_1(0) &= 0 \\ y_2(0) &= 1 \\ y_3(0) &= 1 \end{aligned}$$

Compute the solution to this system on the interval $0 \le t \le 12$. Reproduce the graph included in the documentation provided by the command

```
doc ode23
```

15.7 *A cubic system.* Make a phase plane plot of the solution to the ode system

$$\begin{aligned} \dot{y_1} &= y_2^3 \\ \dot{y_2} &= -y_1^3 \end{aligned}$$

with initial conditions

$$\begin{aligned} y_1(0) = 0 \\ y_2(0) = 1 \end{aligned}$$

on the interval

$$0 \le t \le 7.4163$$

What is special about the final value, $t = 7.4163$?

15.8 *A quintic system.* Make a phase plane plot of the solution to the ode system

$$\begin{aligned} \dot{y_1} &= y_2^5 \\ \dot{y_2} &= -y_1^5 \end{aligned}$$

with initial conditions

$$\begin{aligned} &y_1(0) = 0 \\ &y_2(0) = 1 \end{aligned}$$

on an interval

$$0 \le t \le T$$

where T is the value between 7 and 8 determined by the periodicity condition

$$\begin{aligned} y_1(T) &= 0 \\ y_2(T) &= 1 \end{aligned}$$

15.9 *A quadratic system.* What happens to solutions of

$$\begin{aligned} \dot{y_1} &= y_2^2 \\ \dot{y_2} &= -y_1^2 \end{aligned}$$

Why do solutions of

$$\begin{aligned} \dot{y_1} &= y_2^p \\ \dot{y_2} &= -y_1^p \end{aligned}$$

have such different behavior if p is odd or even?