Physical Modeling in MATLAB

Version 4.0

Allen B. Downey

Green Tea Press

Needham, Massachusetts

Physical Modeling in MATLAB

Green Tea Press
9 Washburn Ave
Needham MA 02492

This book was typeset by the author using pdflatex, among other free, open-source programs. The LaTeX source for this book is available from https://greenteapress.com/matlab.

Contents

Preface

Modeling and simulation are powerful tools for explaining the world, making predictions, designing things that work, and making them work better. Learning to use these tools can be difficult; this book is my attempt to make the experience as enjoyable and productive as possible.

By reading this book—and working on the exercises—you will learn some programming, some modeling, and some simulation. With basic programming skills, you can create models for a wide range of physical systems. My goal is to help you develop these skills in a way you can apply immediately to real-world problems.

This book presents the entire modeling process, including model selection, analysis, simulation, and validation. I explain this process in Chapter 1, and there are examples throughout the book.

0.1 Who This Book Is For

To make this book accessible to the widest possible audience, I've tried to minimize the prerequisites.

This book is intended for people who have never programmed before. I start from the beginning, define new terms when they are introduced, and present only the features you need, when you need them.

I assume that you know trigonometry and some calculus, but not very much. If you understand that a derivative represents a rate of change, that's enough. You will learn about differential equations and some linear algebra, but I will explain what you need to know as we go along.

I will assume you know basic physics, in particular the concepts of force, acceleration, velocity, and position. If you know Newton's second law of motion in the form $F = ma$, that's enough.

0.2 Overview

Here's what you will find in this book:

Chapter 1: Modeling and Simulation Presents the modeling framework we'll use in this book, introduces the MATLAB and Octave programming languages, and helps you debug some of the errors you are likely to make while you are getting started

Chapter 2: Scripts Introduces scripts, which are files that contain MATLAB/Octave code. It also presents variables, values, and the assignment statement

Chapter 3: Loops Presents the `for` loop, sequences, series, plotting, and a way of writing programs called incremental development

Chapter 4: Vectors Introduces vectors, which provide a way to store a sequence of values. And it presents common vector operators including reduce and apply

Chapter 5: Functions Discusses name collisions and an important tool for avoiding them: functions. It also explains input variables and function calls

Chapter 6: Conditionals Presents conditional statements, which check for conditions and determine the behavior of programs. And it introduces a program development process called encapsulation and generalization

Chapter 7: Zero-Finding Introduces `fzero`, which is a MATLAB function that finds the zeros, or roots, of nonlinear equations. It also presents some tips that might help you with debugging

Chapter 8: Functions of Vectors Combines two topics from previous chapters: vectors and functions. It presents functions that take vectors as input variables and return them as output variables. And it introduces logical vectors, which contain a sequence of true and false values.

Chapter 9: Ordinary Differential Equations Introduces the most important idea in the book, differential equations, and two ways to solve them, Euler's method and a MATLAB function called `ode45`

Chapter 10: Systems of ODEs Uses a system of differential equations to simulate the interactions of predator and prey species and presents several ways to plot the results

Chapter 11: Second-Order Systems Describes Newtonian motion using a second-order differential equation and uses `ode45` to simulate falling objects with and without air resistance

Chapter 12: Two Dimensions Extends the methods from the previous chapter to simulate projectiles like baseballs. It introduces spatial vectors as a way to represent quantities with two and three dimensions

Chapter 13: Optimization Introduces `fminsearch`, which is a MATLAB function that searches for the maximum or minimum of a function

Chapter 14: Springs and Things Adds new forces to the toolkit, including spring forces and universal gravitation. It uses them to simulate the orbit of the Earth around the Sun

Chapter 15: Under the Hood Reviews some of the MATLAB functions we've used—`fzero`, `ode45`, and `fminsearch`—and explains more about how they work

I hope you enjoy the book and find it valuable.

0.3 Installing Software

This book is based on MATLAB, a programming language originally developed at the University of New Mexico and now produced by MathWorks, Inc.

MATLAB is a high-level language with features that make it well-suited for modeling and simulation, and it comes with a program development environment that makes it well-suited for beginners.

However, one challenge for beginners is that MATLAB uses vectors and matrices for almost everything, which can make it hard to get started. The organization of this book is meant to help: we start with simple numerical computations, adding vectors in Chapter 4 and matrices in Chapter 10.

Another drawback of MATLAB is that it is "proprietary"; that is, it belongs to MathWorks, and you can only use it with a license, which can be expensive.

Fortunately, the GNU Project has developed a free, open-source alternative called Octave (see `https://www.gnu.org/software/octave`).

Most programs written in MATLAB can run in Octave without modification, and the other way around. All programs in this book have been tested with Octave, so if you don't have access to MATLAB, you should be able to work with Octave. The biggest difference you are likely to see is in the error messages.

To install and run MATLAB, see `https://greenteapress.com/matlab/matlab`.

The first time you run it, a start window should appear to guide you through some configuration.

To install Octave, I recommend that you use Anaconda, which is a package management system that makes it easy to work with Octave and supporting software.

Anaconda installs everything at the user level, so you can install it without admin or root permissions. Follow the instructions for your operating system at https://greenteapress.com/matlab/anaconda.

Once you have Anaconda, you can install Octave by launching the Jupyter Prompt (on Windows) or a Terminal (on Mac OS or Linux), typing the following, and pressing enter:

```
conda install -c conda-forge octave
```

Then you can launch it by typing:

```
octave
```

and pressing enter.

0.4 Working with the Code

The code for each chapter in this book is in a ZIP file you can download from https://greenteapress.com/matlab/zip. Once you have the ZIP file, you can unzip it on the command line by running

```
unzip PhysicalModelingInMatlab.zip
```

In Windows you can right-click on the ZIP file and select **Extract All**.

The result should be a folder called `PhysicalModelingInMatlab` that contains subfolders that contain files containing MATLAB code. They are plain text files, so you can read them with any application that reads text, but most often you will read them with MATLAB.

I'll provide more information about working with these files when we get to them, but that should be enough to get you started.

Contributors

If you discover an error in this book or the supporting code, or you have suggestions for improving them, please send them to *downey@greenteapress.com*. Or, if you are a GitHub user, you can open an issue or a pull request at https://github.com/AllenDowney/PhysicalModelingInMATLAB.

Special thanks to my collaborators at No Starch Press for their work on this book: Alex Freed, Katrina Taylor, Kelly Kearney, Barbara Yien, Bill Pollock, Richard Hutchinson, and Lisa Devoto Farrell.

Other people who have found errors and helped improve this book include Michael Lintz, Kaelyn Stadtmueller, Roydan Ongie, Keerthik Omanakuttan, Pietro Peterlongo, Li Tao, Steven Zhang, Elena Oleynikova, Kelsey Breseman, Philip Loh, Harold Jaffe, Vidie Pong, Nik Martelaro, Arjun Plakkat, Zhen Gang Xiao, Zavier Patrick Aguila, Michael Cline, Denny Chen, Matt Wiens, and Craig Scratchley.

Chapter 1

Modeling and Simulation

This book is about modeling and simulating physical systems. Before we can build any models, it'll help to have a high-level understanding of what a model is. We'll also need to familiarize ourselves with the tools we use to build them. In this chapter, we'll look at the modeling process and introduce MATLAB, the programming language we'll use to represent models and run simulations. At the end of the chapter you'll find exercises you can use to test your knowledge.

1.1 Modeling

When I say "modeling," I'm talking about something like Figure 1.1. In the lower-left corner of the figure is the *system*, something in the real world we're interested in. Often, it's something complicated, so we have to decide which details can be left out; removing details is called *abstraction*.

The result of abstraction is a *model*, shown in the upper left; a model is a description of the system that includes only the features we think are essential. A model can be represented in the form of diagrams and equations, which can be used for mathematical analysis. It can also be implemented in the form of a computer program, which can run simulations.

The result of analysis and simulation might be a prediction about what the system will do, an explanation of why it behaves the way it does, or a specific design engineered to satisfy a requirement or optimize performance.

We can validate predictions and test designs by taking measurements from the real world and comparing the data we get with the results from analysis and simulation.

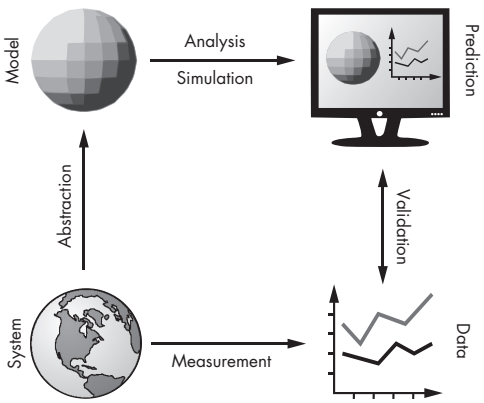

Figure 1.1: The modeling process

For any physical system there are many possible models, each one including and excluding different features or including different levels of detail. The goal of the modeling process is to find the model best suited to its purpose (prediction, explanation, or design).

Sometimes the best model is the most detailed. If we include more features, the model is more realistic, and we expect its predictions to be more accurate. But often a simpler model is better. If we include only the essential features and leave out the rest, we get models that are easier to work with, and the explanations they provide can be clearer and more compelling.

As an example, suppose someone asked you why the orbit of the Earth is nearly elliptical. If you model the Earth and Sun as point masses (ignoring their actual size), compute the gravitational force between them using Newton's law of universal gravitation, and compute the resulting orbit using Newton's laws of motion, you can show that the result is an ellipse.

Of course, the actual orbit of Earth is not a perfect ellipse, because of the gravitational forces of the Moon, Jupiter, and other objects in the solar system, and because Newton's laws of motion are only approximately true (they don't take into account relativistic effects).

But adding these features to the model would not improve the explanation; more detail would only be a distraction from the fundamental cause. However, if the goal is to predict the position of the Earth with great precision, including more details might be necessary.

Choosing the best model depends on what the model is for. It is usually a good idea to start with a simple model, even if it's likely to be too simple, and test whether it's good enough for its purpose. Then you can add features gradually, starting with the ones you expect to be most essential. This process is called *iterative modeling*.

Comparing the results of successive models provides a form of *internal validation* so you can catch conceptual, mathematical, and software errors. And by adding and removing features,

you can tell which ones have the biggest effect on the results, and which can be ignored. Comparing results with data from the real world provides *external validation*, which is generally the strongest test.

Figure 1.1 shows that models can be used for both analysis and simulation; in this book we will do some analysis, but the focus is on simulation. And the tool we will use to build simulations is MATLAB. So let's get started.

1.2 A Glorified Calculator

MATLAB is a programming language with features that support modeling and simulation. It has a lot of features, so it can seem overwhelming, but at heart MATLAB is a glorified calculator. When you start MATLAB you should see a window entitled MATLAB that contains smaller windows entitled Current Folder, Command Window, and Workspace. In Octave, Current Folder is called File Browser.

1.2.1 The Interpreter

The Command Window runs the *interpreter*, which allows you to enter *commands*; once entered, the interpreter executes the command and prints the result. Initially, the Command Window contains a welcome message with information about the version of the software you're running, followed by a *prompt*, which looks like this:

```
>>
```

The `>>` symbol prompts you to enter a command. The simplest kind of command is a mathematical *expression*, like `2 + 1`. If you type an expression and then press enter (or return), MATLAB *evaluates* the expression and prints the result.

```
>> 2 + 1
ans = 3
```

Just to be clear: in this example, MATLAB displayed `>>`; I typed `2 + 1` and then hit enter, and MATLAB displayed `ans = 3`.

In this expression, the plus sign is an *operator* and the numbers `2` and `1` are *operands*. An expression can contain any number of operators and operands. You don't have to put spaces between them; some people do and some people don't. Here's an example with no spaces:

```
>> 1+2+3+4+5+6+7+8+9
ans = 45
```

Speaking of spaces, you might have noticed that MATLAB puts a blank line between `ans =` and the result. In my examples I'll leave it out to save room.

The other arithmetic operators are pretty much what you would expect. Subtraction is denoted by a minus sign (`-`), multiplication is designated by an asterisk (`*`), division is denoted by a forward slash (`/`).

```
>> 2*3 - 4/5
ans = 5.2000
```

Another common operator is exponentiation, which uses the `^` symbol, sometimes called "caret" or "hat." So, 2 raised to the 16th power is

```
>> 2^16
ans = 65536
```

The order of operations is what you would expect from basic algebra: exponentiation happens before multiplication and division, and multiplication and division happen before addition and subtraction. If you want to override the order of operations, you can use parentheses.

```
>> 2 * (3-4) / 5
ans = -0.4000
```

When I added the parentheses, I removed some spaces to make the grouping of operands clearer to a human reader. This is the first of many style guidelines I will recommend for making your programs easier to read. Style doesn't change what the program does; the MATLAB interpreter doesn't check for style. But human readers do, and the most important human who will read your code is you.

And that brings us to the First Theorem of Debugging:

> Readable code is debuggable code.

It's worth spending time to make your code pretty; it will save you time debugging!

1.2.2 Math Functions

MATLAB knows how to compute pretty much every math function you've heard of. For example, it knows all the trigonometric functions—here's how you use them:

```
>> sin(1)
ans = 0.8415
```

This command is an example of a *function call*. The name of the function is `sin`, which is the usual abbreviation for the trigonometric sine. The value in parentheses is called the *argument*.

The trig functions `sin`, `cos`, and `tan`—among many others—work in radians, so the argument in the example is interpreted as 1 radian. MATLAB also provides trig functions that work in degrees: `sind`, `cosd`, and `tand`.

Some functions take more than one argument, in which case the arguments are separated by commas. For example, `atan2` computes the inverse tangent, which is the angle in radians between the positive x-axis and the point with the given x- and y-coordinates.

```
>> atan2(1,1)
ans = 0.7854
```

If that bit of trigonometry isn't familiar to you, don't worry about it. It's just an example of a function with multiple arguments.

MATLAB also provides exponential functions, like `exp`, which computes e raised to the given power. So `exp(1)` is just e:

```
>> exp(1)
ans = 2.7183
```

The inverse of `exp` is `log`, which computes the logarithm base e:

```
>> log(exp(3))
ans = 3
```

This example also demonstrates that function calls can be *nested*; that is, you can use the result from one function as an argument for another.

More generally, you can use a function call as an operand in an expression.

```
>> sqrt(sin(0.5)^2 + cos(0.5)^2)
ans = 1
```

As you probably guessed, `sqrt` computes the square root.

There are lots of other math functions, but this isn't meant to be a reference manual. To learn about other functions, you should read the documentation.

1.3 Variables

Of course, MATLAB is good for more than just evaluating expressions. One of the features that makes MATLAB more powerful than a calculator is the ability to give a name to a value. A named value is called a *variable*.

MATLAB comes with a few predefined variables. For example, the name `pi` refers to the mathematical quantity π, which is approximately this:

```
>> pi
ans = 3.1416
```

And if you do anything with complex numbers, you might find it convenient that both `i` and `j` are predefined as the square root of -1.

You can use a variable name anywhere you can use a number—for example, as an operand in an expression,

```
>> pi * 3^2
ans = 28.2743
```

or as an argument to a function:

```
>> sin(pi/2)
ans = 1
```

Whenever you evaluate an expression, MATLAB assigns the result to a variable named `ans`. You can use `ans` in a subsequent calculation as shorthand for "the value of the previous expression."

```
>> 3^2 + 4^2
ans = 25

>> sqrt(ans)
ans = 5
```

But keep in mind that the value of `ans` changes every time you evaluate an expression.

1.3.1 Assignment Statements

You can create your own variables, and give them values, with an *assignment statement*. The assignment operator is the equals sign (=), used like so:

```
>> x = 6 * 7
x = 42
```

This example creates a new variable named `x` and assigns it the value of the expression `6 * 7`. MATLAB responds with the variable name and the computed value.

There are a few rules when assigning variables a value. In every assignment statement, the left side has to be a legal variable name. The right side can be any expression, including function calls. Almost any sequence of lower- and uppercase letters is a legal variable name. Some punctuation is also legal, but the underscore (`_`) is the only commonly used non-letter. Numbers are fine, but not at the beginning. Spaces are not allowed. Variable names are *case sensitive*, so `x` and `X` are different variables.

Let's look at some examples of assignment statements.

```
>> fibonacci0 = 1;

>> LENGTH = 10;

>> first_name = 'bob'
first_name = 'bob'
```

The first two examples demonstrate the use of the semicolon, which suppresses the output from a command. In this case MATLAB creates the variables and assigns them values but displays nothing.

The third example demonstrates that not everything in MATLAB is a number. A sequence of characters in single quotes is a *string*.

Although `i`, `j`, and `pi` are predefined, you are free to reassign them. It's common to use `i` and `j` for other purposes, but it's rare to assign a different value to `pi`.

1.3.2 Variables in the Workspace

When you create a new variable, it appears in the Workspace window and is added to the *workspace*, which is a set of variables and their values.

The `who` command prints the names of the variables in the workspace:

```
>> x = 5;
>> y = 7;
>> z = 9;
>> who

Your variables are:

x  y  z
```

The `clear` command removes specified variables from the workspace:

```
>> clear x
>> who

Your variables are:

y z
```

But be careful: if you don't specify any variables, `clear` removes them all.

To display the value of a variable, you can use the `disp` function:

```
>> disp(z)
     9
```

but it's easier to just type the variable name:

```
>> z
z = 9
```

Now that you've seen how to use them, let's take a step back and think about why we'd use variables.

1.3.3 Why Variables?

There are a number of reasons to use variables. A big one is to avoid recomputing a value you use repeatedly. For example, if your computation uses e frequently, you might want to compute it once and save the result.

```
>> e = exp(1)
e = 2.7183
```

Variables also make the connection between the code and the underlying mathematics more apparent. If you're computing the area of a circle, you might want to use a variable named `r`:

```
>> r = 3
r = 3

>> area = pi * r^2
area = 28.2743
```

That way, your code resembles the familiar formula $a = \pi r^2$.

You might also use a variable to break a long computation into a sequence of steps. Suppose you're evaluating a big, hairy expression like this:

```
p = ((x - theta) * sqrt(2 * pi) * sigma)^-1 * ...
exp(-1/2 * (log(x - theta) - zeta)^2 / sigma^2)
```

You can use an ellipsis to break the expression into multiple lines. Just enter `...` at the end of the first line and continue on to the next. But often it's better to break the computation into a sequence of steps and assign intermediate results to variables:

```
shiftx = x - theta
denom = shiftx * sqrt(2 * pi) * sigma
temp = (log(shiftx) - zeta) / sigma
exponent = -1/2 * temp^2
p = exp(exponent) / denom
```

The names of the intermediate variables explain their role in the computation: `shiftx` is the value of `x` shifted by `theta`, it should be no surprise that `exponent` is the argument of `exp`, and `denom` ends up in the denominator. Choosing informative names makes the code easier to read and understand, which makes it easier to debug.

1.4 Errors

Every error is a learning opportunity. Whenever you learn a new feature, you should try to make as many errors as possible, as soon as possible. When you make deliberate errors, you see what the error messages are. Later, when you make accidental errors, you'll know what the messages mean.

Let's look at some common errors. A big one for beginners is leaving out the `*` for multiplication, as in this example:

```
area = pi r^2
```

This code should produce the following error message:

```
 area = pi r^2
           |
Error: Invalid expression. Check for missing multiplication
operator, missing or unbalanced delimiters, or other syntax
error. To construct matrices, use brackets instead of parentheses.
```

The message indicates that the expression is invalid and suggests several things that might be wrong. In this case, one of its guesses is right: we're missing a multiplication operator.

Another common error is to leave out the parentheses around the arguments of a function. For example, in math notation it's common to write something like $\sin \pi$, but in MATLAB if you write

```
sin pi
```

you should get the following error message:

```
Undefined function 'sin' for input arguments of type 'char'.
```

The problem is that when you leave out the parentheses, MATLAB treats the argument as a string of characters (which have type `'char'`). In this case the error message is helpful, but in other cases the results can be baffling. For example, if you call `abs`, which computes absolute values, and forget the parentheses, you get a surprising result:

```
>> abs pi
ans =  112   105
```

I won't explain this result; for now, I'll just suggest that you should *always* put parentheses around arguments.

Here's another common error. If you were translating the mathematical expression

$$\frac{1}{2\sqrt{\pi}}$$

into MATLAB, you might be tempted to write this:

```
1 / 2 * sqrt(pi)
```

But that would be wrong because of the order of operations. Division and multiplication are evaluated from left to right, so this expression would multiply `1/2` by `sqrt(pi)`.

To keep `sqrt(pi)` in the denominator, you could use parentheses,

```
1 / (2 * sqrt(pi))
```

or make the division explicit,

```
1 / 2 / sqrt(pi)
```

The last two examples bring us to the Second Theorem of Debugging:

> The only thing worse than getting an error message is *not* getting an error message.

Beginning programmers often hate error messages and do everything they can to make the messages go away. Experienced programmers know that error messages are your friend. They can be hard to understand, and even misleading, but it's worth the effort to understand them.

1.5 Documentation

MATLAB comes with two forms of documentation, `help` and `doc`. The `help` command works in the Command Window; just enter `help` followed by the name of a command.

```
>> help sin
```

You should see output like this:

```
 sin    Sine of argument in radians.
    sin(X) is the sine of the elements of X.

    See also asin, sind, sinpi.
```

Some documentation uses vocabulary we haven't covered yet. For example, `the elements of X` might not make sense until we get to vectors and matrices a few chapters from now.

The `doc` pages are usually better. If you enter `doc sin`, a browser window appears with more detailed information about the function, including examples of how to use it. The examples often use vectors and matrices, so they may not make sense yet, but you can get a preview of what's coming.

1.6 Chapter Review

This chapter provided an overview of the modeling process, including abstraction, analysis and simulation, measurement, and validation.

It also introduced MATLAB, the programming language we'll use to write simulations. So far, we've seen variables and values, arithmetic operations, and mathematical functions.

Here are a few terms from this chapter you might want to remember.

The *interpreter* is the program that reads and executes MATLAB or Octave code. It prints a *prompt* to indicate that it's waiting for you to type a *command*, which is a line of code executed by the interpreter.

An *operator* is a symbol, like `*` or `+`, that represents a mathematical operation. An *operand* is a number or variable that appears in an expression along with operators. An *expression* is a sequence of operands and operators that specifies a mathematical computation and yields a value.

A *function* is a named computation; for example, `log10` is the name of a function that computes logarithms in base 10. A *function call* is a command that causes a function to execute and compute a result. An *argument* is an expression that appears in a function call to specify the value the function operates on.

A *variable* is a named value. An *assignment statement* is a command that creates a new variable (if necessary) and gives it a value. A *workspace* is a set of variables and their values.

Finally, a *string* is a value that consists of a sequence of characters (as opposed to a number).

In the next chapter, you'll start writing longer programs and learn about floating-point numbers.

1.7 Exercises

Before you go on, you might want to work on the following exercises.

Exercise 1.1. You might have heard that a penny dropped from the top of the Empire State Building would be going so fast when it hit the pavement that it would be embedded in the concrete or that if it hit a person it would break their skull.

We can test this myth by making and analyzing a model. To get started, we'll assume that the effect of air resistance is small. This will turn out to be a bad assumption, but bear with me.

If air resistance is negligible, the primary force acting on the penny is gravity, which causes the penny to accelerate downward.

If the initial velocity is 0, the velocity after t seconds is at, and the distance the penny has dropped is

$$h = at^2/2$$

Using algebra, we can solve for t:

$$t = \sqrt{2h/a}$$

Plugging in the acceleration of gravity, $a = 9.8\,\mathrm{m/s^2}$, and the height of the Empire State Building, $h = 381\,\mathrm{m}$, we get $t = 8.8\,\mathrm{s}$. Then, computing $v = at$ we get a velocity on impact of $86\,\mathrm{m/s}$, which is about 190 miles per hour. That sounds like it could hurt.

Use MATLAB to perform these computations, and check that you get the same result.

Exercise 1.2. The result in the previous exercise is not accurate because it ignores air resistance. In reality, once the penny gets to about $18\,\mathrm{m/s}$, the upward force of air resistance equals the downward force of gravity, so the penny stops accelerating. After that, it doesn't matter how far the penny falls; it hits the sidewalk at about $18\,\mathrm{m/s}$, much less than $86\,\mathrm{m/s}$.

As an exercise, compute the time it takes for the penny to reach the sidewalk if we assume that it accelerates with constant acceleration $a = 9.8\,\mathrm{m/s^2}$ until it reaches terminal velocity, then falls with constant velocity until it hits the sidewalk.

The result you get is not exact, but it's a pretty good approximation.

Chapter 2

Scripts

So far we've typed all of our programs "at the prompt." If you're only writing a few lines, this isn't so bad. But what if you're writing a hundred? Retyping each line of code every time you want to change or test your program will be time-consuming and tedious. Luckily, you don't have to. In this chapter, we'll look at a way to run many lines at once: scripts.

2.1 Your First Script

A *script* is a file that contains MATLAB code. When you run a script, MATLAB executes the commands in it, one after another, exactly as if you had typed them at the prompt. Scripts are also sometimes called *M-files* because they use the extension *.m*, short for MATLAB.

You can create scripts with any text editor or word processor, but the simplest way is to click the **New Script** button in the upper-left corner of the MATLAB interface, which opens a text editor designed for MATLAB.

To try it out, create a new script and enter the following code:

```
x = 5
```

Then press the **Save** button. A dialog window should appear where you can choose the filename and the folder where your script will go. Change the name to *myscript.m* and save it into any folder you like.

Now click the green **Run** button. You might get a message that says the script is not found in the current folder. If so, click the button that says Change Folder and it should run.

You can also run a script by typing its name in the Command Window and pressing enter. For example, if you enter `myscript`, MATLAB should execute your script and display the result:

```
>> myscript
x = 5
```

There are a few things to keep in mind when using scripts. First, you should not include the extension *.m* when you run a script. If you do, you'll get an error message like this:

```
>> myscript.m
Undefined variable "myscript" or class "myscript.m".
```

Second, when you name a new script, try to choose something meaningful and memorable. Don't choose a name that's already in use; if you do, you'll replace one of MATLAB's functions with your own (at least temporarily). You might not notice right away, but you might get some confusing behavior later.

Also, the name of the script cannot contain spaces. If you create a file named *my script.m*, MATLAB will complain when you try to run it:

```
>> my script
Undefined function or variable 'my'.
```

It can be hard to remember which folder a script is in. To keep things simple, for now, I suggest putting all of your scripts in one folder.

2.2 Why Scripts?

There are a few good reasons to use a script. When you're writing more than a couple of lines of code, it might take a few tries to get everything right. Putting your code in a script makes it easier to edit than typing it at the prompt. Likewise, if you're running a script repeatedly, it's much faster to type the name of the script than to retype the code! And you might be able to reuse a script from one project to the next, saving you considerable time across projects.

But the great power of scripts comes with great responsibility: you have to make sure that the code you are running is the code you think you are running. Whenever you start a new script, start with something simple, like `x = 5`, that produces a visible effect. Then run your script and confirm that you get what you expect. When you type the name of a script, MATLAB searches for the script in a *search path*, which is a sequence of folders. If it doesn't find the script in the first folder, it searches the second, and so on. If you have scripts with the same name in different folders, you could be looking at one version and running another.

If the code you are running is not the code you are looking at, you'll find debugging a frustrating exercise! So it's no surprise that this is the Third Theorem of Debugging:

> Be sure that the code you are running is the code you think you are running.

Now that you've seen how to write a script, let's use one to do something a little more complicated.

2.3 The Fibonacci Sequence

The *Fibonacci sequence*, denoted F, is a sequence of numbers where each number is the sum of the previous two. It's defined by the equations $F_1 = 1$, $F_2 = 1$, and, for $i > 2$, $F_i = F_{i-1} + F_{i-2}$. The following expression computes the nth Fibonacci number:

$$F_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right]$$

We can translate this expression into MATLAB, like this:

```
s5 = sqrt(5);
t1 = (1 + s5) / 2;
t2 = (1 - s5) / 2;
diff = t1^n - t2^n;
ans = diff / s5
```

I use temporary variables like `t1` and `t2` to make the code readable and the order of operations explicit. The first four lines have a semicolon at the end, so they don't display anything. The last line assigns the result to `ans`.

If we save this script in a file named *fibonacci1.m*, we can run it like this:

```
>> n = 10
>> fibonacci1
ans = 55.0000
```

Before calling this script, you have to assign a value to `n`. If `n` is not defined, you get an error:

```
>> clear n
>> fibonacci1
Undefined function or variable 'n'.

Error in fibonacci1 (line 9)
diff = t1^n - t2^n;
```

This script only works if there is a variable named `n` in the workspace; otherwise, you should get an error. MATLAB will tell you what line of the script the error is in and display the line.

Error messages can be helpful, but beware! In this example, the message says the error is in `fibonacci`, but the actual problem is that we have not assigned a value to `n`. And that brings us to the Fourth Theorem of Debugging:

> Error messages tell you where the problem was discovered, not where it was caused.

Often you have to work backwards to find the line of code (or missing line) that caused the problem.

2.4 Floating-Point Numbers

In the previous section, the result we computed was `55.0000`. Since the Fibonacci numbers are integers, you might have been surprised to see the zeros after the decimal point.

They are there because MATLAB performs calculations using floating-point numbers. With *floating-point numbers*, integers can be represented exactly, but most fractions cannot.

For example, if you compute the fraction 2/3, the result is only approximate—the correct answer has an infinite number of 6s:

```
>> 2/3
ans = 0.6666
```

It's not as bad as this example makes it seem: MATLAB uses more digits than it shows by default. You can use the `format` command to change the output format:

```
>> format long
>> 2/3
ans = 0.666666666666667
```

In this example, the first 14 digits are correct; the last one has been rounded off.

Large and small numbers are displayed in *scientific notation*. For example, if we use the built-in function `factorial` to compute 100!, we get the following result:

```
>> factorial(100)
ans = 9.332621544394410e+157
```

The `e` in this notation is *not* the transcendental number known as e; it's just an abbreviation for "exponent." So this means that 100! is approximately 9.33×10^{157}. The exact solution is a 158-digit integer, but with double-precision floating-point numbers, we only know the first 16 digits.

You can enter numbers using the same notation.

```
>> speed_of_light = 3.0e8
speed_of_light = 300000000
```

Although the floating-point format can represent very large and small numbers, there are limits. The predefined variables `realmax` and `realmin` contain the largest and smallest numbers MATLAB can handle.

```
>> realmax
ans = 1.797693134862316e+308

>> realmin
ans = 2.225073858507201e-308
```

If the result of a computation is too big, MATLAB "rounds up" to infinity.

```
>> factorial(170)
ans = 7.257415615307994e+306

>> factorial(171)
ans = Inf
```

Division by zero also returns `Inf`.

```
>> 1/0
ans = Inf
```

For operations that are undefined, MATLAB returns `NaN`, which stands for "not a number."

```
>> 0/0
ans = NaN
```

2.5 Comments

Short, simple programs are easy to read, but as they get bigger and more complex, it gets harder to figure out what they do and how. That's what comments are for.

A *comment* is a line of text added to a program to explain how it works. It has no effect on the execution of the program; it is there for human readers. Good comments make programs more readable; bad comments are useless at best and misleading at worst.

To write a comment, you use the percent symbol (%) followed by the text of the comment.

```
>> speed_of_light = 3.0e8     % meters per second
speed_of_light = 300000000
```

The comment runs from the percent symbol to the end of the line. In this case it specifies the units of the value. In an ideal world, MATLAB would keep track of units and propagate them through the computation, but for now that burden falls on the programmer.

Avoid comments that are redundant with the code:

```
>> x = 5        % assign the value 5 to x
```

Good comments provide additional information that's not in the code, like units in the example above, or the meaning of a variable:

```
>> p = 0         % position from the origin in meters
>> v = 100       % velocity in meters / second
>> a = -9.8      % acceleration of gravity in meters / second^2
```

If you use longer variable names, you might not need explanatory comments, but there's a trade-off: longer variable names are clearer, but longer code can become harder to read. Also, if you're translating from math that uses short variable names, it can be useful to make your program consistent with your math.

2.6 Documentation

Every script should provide *documentation*, which is a comment that explains what the script does and what its requirements are.

For example, I might put something like this at the beginning of *fibonacci1.m*:

```
% Computes a numerical approximation of the nth Fibonacci number.
% Precondition: you must assign a value to n before running this script.
% Postcondition: the result is stored in ans.
```

A *precondition* is something that must be true when the script starts in order for it to work correctly. A *postcondition* is something that will be true when the script completes.

If there is a comment at the beginning of a script, MATLAB assumes it's the documentation for the script. So if you type `help fibonacci1`, you get the contents of the comment (without the percent signs).

```
>> help fibonacci1
  Computes a numerical approximation of the nth Fibonacci number.
  Precondition: you must assign a value to n before running this script.
  Postcondition: the result is stored in ans.
```

That way, scripts that you write behave just like predefined scripts. You can even use the `doc` command to see your comment in the Help Window.

2.7 Assignment and Equality

For beginning programmers, a common source of confusion is assignment and the use of the equals sign.

In mathematics, the equals sign means that the two sides of the equation have the same value. In MATLAB, an assignment statement *looks* like a mathematical equality, but it's not.

One difference is that the sides of an assignment statement are not interchangeable. The right side can be any legal expression, but the left side has to be a variable, which is called the *target* of the assignment. So this is legal:

```
>> y = 1;
>> x = y + 1
x = 2
```

But this is not:

```
>> y + 1 = x
 y + 1 = x
       |
Error: Incorrect use of '=' operator.
To assign a value to a variable, use '='.
To compare values for equality, use '==='.
```

In this case the error message is not very helpful. The problem here is that the expression on the left side is not a valid target for an assignment.

Another difference between assignment and equality is that a mathematical equality is true (or false) for all eternity; an assignment statement is temporary. When you assign `x = y + 1`, you get the *current* value of `y`. If `y` changes later, `x` does not get updated.

A third difference is that a mathematical equality is a statement that may or may not be true. In mathematics, $y = y + 1$ is a statement that happens to be false for all values of y. In MATLAB, `y = y + 1` is a sensible and useful assignment statement. It reads the current value of `y`, adds 1, and replaces the old value with the new value.

```
>> y = 1;
>> y = y + 1
y = 2
```

When you read MATLAB code, you might find it helpful to pronounce the equals sign as "gets" rather than "equals." So `x = y + 1` is pronounced "`x` gets the value of `y` plus one."

2.8 Chapter Review

This chapter presented scripts and suggested reasons to use them. We computed elements of a Fibonacci sequence, but because we used floating-point numbers, the results were sometimes only approximate. And we saw how to add comments to a program to document what it does and explain how it works.

Here are some terms from this chapter you might want to remember.

An *M-file* is a file that contains a *script*, which is a sequence of MATLAB/Octave commands. The *search path* is the list of folders where the interpreter looks for M-files.

A *precondition* is something that must be true when the script starts in order for it to work correctly; a *postcondition* is something that will be true when the script completes.

The *target* of an assignment statement is the variable on the left side.

Floating-point is a way to represent and store numbers in a computer. *Scientific notation* is a format for typing and displaying large and small numbers; for example, `3.0e8` represents 3.0×10^8 or 300,000,000.

A *comment* is part of a program that provides additional information about the program, but does not affect its execution.

In the next chapter, you'll learn how to write programs that perform repetitive tasks using loops.

2.9 Exercises

Before you go on, you might want to work on the following exercises.

Exercise 2.1. To test your understanding of assignment statements, write a few lines of code that swap the values of `x` and `y`. Put your code in a script called *swap.m* and test it.

If it works correctly, you should be able to run it like this:

```
>> x = 1, y = 2
x = 1
y = 2

>> swap

>> x, y
x = 2
y = 1
```

Exercise 2.2. Imagine that you are the operator of a bike-share system with two locations: Boston and Cambridge.

You observe that every day 5 percent of the bikes in Boston are dropped off in Cambridge, and 3 percent of the bikes in Cambridge get dropped off in Boston. At the beginning of the month, there are 100 bikes at each location.

Write a script called *bike_update.m* that updates the number of bikes in each location from one day to the next. The precondition is that the variables `b` and `c` contain the number of bikes in each location at the beginning of the day. The postcondition is that `b` and `c` have been modified to reflect the net movement of bikes.

To test your program, initialize `b` and `c` at the prompt and then execute the script. The script should display the updated values of `b` and `c`, but not any intermediate variables.

Remember that bikes are countable things, so `b` and `c` should always be integer values. You might want to use the `round` function to compute the number of bikes that move each day.

If you execute your script repeatedly, you can simulate the passage of time from day to day (you can repeat a command by pressing the up arrow and then enter).

What happens to the bikes? Do they all end up in one place? Does the system reach an equilibrium, does it oscillate, or does it do something else?

In the next chapter, we will see how to execute your script automatically and how to plot the values of `b` and `c` over time.

Chapter 3

Loops

The programs we have seen so far are *straight-line code*; that is, they execute one instruction after another from top to bottom. This chapter introduces one of the most important programming-language features, the `for` loop, which allows simple programs to perform complex, repetitive tasks. This chapter also introduces the mathematical concepts of sequence and series, and a process for writing programs, incremental development.

We'll start by reviewing the exercise from the previous chapter; if you didn't do it, you might want to take a look before you go on.

3.1 Updating Variables

In Section 2.2, I asked you to write a program that models a bike-share system with bikes moving between two stations. Each day 5 percent of the bikes in Boston are dropped off in Cambridge, and 3 percent of the bikes in Cambridge get dropped off in Boston.

To update the state of the system, you might have been tempted to write something like

```
b = b - 0.05*b + 0.03*c
c = c + 0.05*b - 0.03*c
```

But that would be wrong, so very wrong. Why? The problem is that the first line changes the value of `b`, so when the second line runs, it gets the old value of `c` and the new value of `b`. As a result, the change in `b` is not always the same as the change in `c`, which violates the Principle of Conservation of Bikes!

One solution is to use temporary variables like `b_new` and `c_new`:

```
b_new = b - 0.05*b + 0.03*c
c_new = c + 0.05*b - 0.03*c
b = b_new
c = c_new
```

This has the effect of updating the variables *simultaneously*; that is, it reads both old values before writing either new value.

The following is an alternative solution that has the added advantage of simplifying the computation:

```
b_to_c = 0.05*b - 0.03*c
b = b - b_to_c
c = c + b_to_c
```

It's easy to look at this code and confirm that it obeys Conservation of Bikes. Even if the value of `b_to_c` is wrong, at least the total number of bikes is right. And that brings us to the Fifth Theorem of Debugging:

> The best way to avoid a bug is to make it impossible.

In this case, removing redundancy also eliminates the opportunity for a bug.

3.2 Bug Taxonomy

The more you understand bugs, the better you will be at debugging. There are four kinds of bugs:

Syntax error You have written a command that cannot execute because it violates one of the language's syntax rules. For example, in MATLAB, you can't have two operands in a row without an operator, so `pi r^2` contains a syntax error. When the interpreter finds a syntax error, it prints an error message and stops running your program.

Runtime error Your program starts running, but something goes wrong along the way. For example, if you try to access a variable that doesn't exist, that's a runtime error. When the interpreter detects the problem, it prints an error message and stops.

Logical error Your program runs without generating any error messages, but it doesn't do the right thing. The problem in the previous section, where we changed the value of `b` before reading the old value, is a logical error.

Numerical error Most computations in MATLAB are only approximately right. Most of the time the errors are small enough that we don't care, but in some cases the round-off errors are a problem.

Syntax errors are usually the easiest to deal with. Sometimes the error messages are confusing, but MATLAB can usually tell you where the error is, at least roughly.

Runtime errors are harder because, as I mentioned before, MATLAB can tell you where it detected the problem, but not what caused it.

Logical errors are hard because MATLAB can't help at all. From MATLAB's point of view there's nothing wrong with the program; only you know what the program is supposed to do, so only you can check it.

Numerical errors can be tricky because it's not clear whether the problem is your fault. For most simple computations, MATLAB produces the floating-point value that is closest to the exact solution, which means that the first 15 significant digits should be correct.

But some computations are ill-conditioned, which means that even if your program is correct, the round-off errors accumulate and the number of correct digits can be smaller. Sometimes MATLAB can warn you that this is happening, but not always! *Precision* (the number of digits in the answer) does not imply *accuracy* (the number of digits that are right).

3.3 Absolute and Relative Error

There are two ways of thinking about numerical errors. The first is *absolute error*, or the difference between the correct value and the approximation. We often write the magnitude of the error, ignoring its sign, when it doesn't matter whether the approximation is too high or too low.

The second way to think about numerical errors is *relative error*, where the error is expressed as a fraction (or percentage) of the exact value.

For example, we might want to estimate 9! using the formula $\sqrt{18\pi}(9/e)^9$. The exact answer is $9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 362,880$. The approximation is $359,536.87$. So the absolute error is $3,343.13$.

At first glance, that sounds like a lot—we're off by three thousand—but we should consider the size of the thing we are estimating. For example, \$3,000 matters a lot if we're talking about an annual salary, but not at all if we're talking about the national debt.

A natural way to handle this problem is to use relative error. In this case, we would divide the error by $362,880$, yielding 0.00921, which is just less than 1 percent. For many purposes, being off by 1 percent is good enough.

3.4 for Loops

Let's return to the bike-share example. In the previous chapter, we wrote a *bike_update.m* script to simulate a day in the life of a bike-share system. To simulate an entire month, we'd have to run the script 30 times. We could enter the same command 30 times, but it's simpler to use a *loop*, which is a set of statements that executes repeatedly.

To create a loop, we can use the `for` statement, like this:

```
for i=1:30
    bike_update
end
```

The first line includes what looks like an assignment statement, and it *is* like an assignment statement, except that it runs more than once. The first time, it creates the variable `i` and assigns it the value 1. The second time, `i` gets the value 2, and so on, up to and including 30.

The colon operator (`:`) specifies a *range* of integers. You can create a range at the prompt:

```
>> 1:5
ans =  1     2     3     4     5
```

The variable you use in the `for` statement is called the *loop variable*. It's common to use the names `i`, `j`, and `k` as loop variables.

The statements inside the loop are called the *body*. By convention, they are indented to show that they're inside the loop, but the indentation doesn't affect the execution of the program. The `end` statement marks the end of the loop.

To see a loop in action you can run one that displays the loop variable:

```
>> for i=1:5
    i
end

i = 1
i = 2
i = 3
i = 4
i = 5
```

As this example shows, you *can* run a `for` loop from the command line, but it's more common to put it in a script.

Exercise 3.1. Create a script named *bike_loop.m* that uses a `for` loop to run *bike_update.m* 30 times. Before you run it, you have to assign values to `b` and `c`. For this exercise, start with the values `b = 100` and `c = 100`.

If everything goes smoothly, your script will display a long stream of numbers on the screen. It's probably too long to fit, and even if it did fit, it would be hard to interpret. A graph would be much better!

3.5 Plotting

If the output of your program is a long stream of numbers, it can be hard to see what is happening. Plotting the results can make things clearer.

The `plot` function is a versatile tool for plotting two-dimensional graphs. Unfortunately, it's so versatile that it can be hard to use (and hard to read the documentation). We'll start simple and work our way up.

To plot a single point, type

```
>> plot(1, 2, 'o')
```

A Figure Window should appear with a graph and a single blue circle at x position 1 and y position 2.

The letter in single quotes is a *style string* that specifies how the point should be plotted; `o` indicates a circle. Other shapes include `+`, `*`, `x`, `s` (for a square), `d` (for a diamond), and `^` (for a triangle).

You can also specify the color by starting the style string with a color code:

```
>> plot(1, 2, 'ro')
```

Here, `r` stands for red; the other colors include `g` for green, `b` for blue, `c` for cyan, `m` for magenta, `y` for yellow, and `k` for black.

When you use `plot` this way, it can only plot one point at a time. If you run `plot` again, it clears the figure before making the new plot. The `hold` command lets you override that behavior: `hold on` tells MATLAB not to clear the figure when it makes a new plot; `hold off` returns to the default behavior.

Try this:

```
>> clf
>> hold on
>> plot(1, 1, 'ro')
>> plot(2, 2, 'go')
>> plot(3, 3, 'bo')
>> hold off
```

The `clf` command clears the figure before we start plotting.

If you run the code above, you should see a figure with three circles. MATLAB scales the plot automatically so that the axes run from the lowest values in the plot to the highest.

Exercise 3.2. Modify *bike_loop.m* so that it clears the figure before running the loop. Then, each time through the loop, it should plot the value of `b` versus the value of `i` with a red circle.

Once you get that working, modify it so it plots the values of `c` with blue diamonds.

3.6 Sequences

Now that we have the ability to write loops, we can use them to explore sequences and series, which are useful for describing and analyzing systems that change over time.

In mathematics, a *sequence* is a set of numbers that corresponds to the positive integers. The numbers in the sequence are called *elements*. In math notation, the elements are denoted with subscripts, so the first element of the series A is A_1, followed by A_2, and so on.

A `for` loop is a natural way to compute the elements of a sequence. As an example, in a geometric sequence, each element is a constant multiple of the previous element. As a more specific example, let's look at the sequence with $A_1 = 1$ and the relationship $A_{i+1} = A_i/2$, for all i. In other words, each element is half as big as the one before it.

The following loop computes the first 10 elements of A:

```
a = 1
for i=2:10
    a = a/2
end
```

The first line initializes the variable `a` with the first element of the sequence, A_1. Each time through the loop, we find the next value of `a` by dividing the previous value by 2, and assign the result back to `a`. At the end, `a` contains the 10th element.

The other elements are displayed on the screen, but they are not saved in a variable. Later, we'll see how to save the elements of a sequence in a vector.

This loop computes the sequence *recurrently*, which means that each element depends on the previous one. For this sequence, it's also possible to compute the ith element *directly*, as a function of i, without using the previous element.

In math notation, $A_i = A_1 r^{i-1}$, where r is the ratio of successive elements. In the previous example, $A_{i+1} = A_i/2$, so $r = 1/2$.

Exercise 3.3. Write a script named *sequence.m* that uses a loop to compute elements of A directly.

3.7 Series

In mathematics, a *series* is the sum of the elements of a sequence. It's a terrible name, because in common English, "sequence" and "series" mean pretty much the same thing, but in math, a sequence is a set of numbers, and a series is an expression (a sum) that has a single value. In math notation, a series is often written using the summation symbol $\sum$.

For example, the sum of the first 10 elements of A is

$$\sum_{i=1}^{10} A_i$$

A `for` loop is a natural way to compute the value of this series:

Listing 3.1: A program that calculates a simple series

```
A1 = 1;
total = 0;
for i=1:10
    a = A1 * (1/2)^(i-1);
    total = total + a;
end
ans = total
```

Let's walk through what's happening here. `A1` is the first element of the sequence, so we assign it to be `1`; we also create `total`, which will store the cumulative sum. Each time through the loop, we set `a` to the ith element and add `a` to the `total`. At the end, outside the loop, we store `total` as `ans`.

The way we're using `total` is called an *accumulator*, that is, a variable that accumulates the result a little bit at a time.

Exercise 3.4. This example computes the terms of the series directly. As an exercise, write a script named *series.m* that computes the same sum by computing the elements recurrently. You will have to be careful about where you start and stop the loop.

3.8 Generalization

As written, the previous example always adds up the first 10 elements of the sequence, but we might be curious to know what happens to `total` as we increase the number of terms in the series. If you've studied geometric series, you might know that this series converges on 2; that is, as the number of terms goes to infinity, the sum approaches 2 asymptotically.

To see if that's true for our program, we can replace the constant `10` in Listing 3.1 with a variable named `n`:

Listing 3.2: Updating our code from Listing 3.1 to have a variable number of terms

```
A1 = 1;
total = 0;
for i=1:n
    a = A1 * (1/2)^(i-1);
    total = total + a;
end
ans = total
```

The code in Listing 3.2 can now compute any number of terms, with the precondition that you have to set `n` before you execute the code. I put this code in a file named *series.m*, then ran it with different values of `n`:

```
>> n=10; series
total = 1.99804687500000

>> n=20; series
total = 1.99999809265137

>> n=30; series
total = 1.99999999813735

>> n=40; series
total = 1.99999999999818
```

It sure looks like it's converging on 2.

Replacing a constant with a variable is called *generalization*. Instead of computing a fixed, specific number of terms, the new script is more general; it can compute any number of terms. This is an important idea we'll come back to when we talk about functions.

3.9 Incremental Development

As you start writing longer programs, you might find yourself spending more time debugging. The more code you write before you start debugging, the harder it is to find the problem.

Incremental development is a way of programming that tries to minimize the pain of debugging by developing and testing in small steps. The fundamental steps are:

1. Always start with a working program. If you have an example from a book, or a program you wrote that is similar to what you are working on, start with that. Otherwise, start with something you *know* is correct, like `x = 5`. Run the program and confirm that you are running the program you think you are running. This step is important, because in most environments there are little things that can trip you up when you start a new project. Get them out of the way so you can focus on programming.
2. Make one small, testable change at a time. A *testable* change is one that displays something on the screen (or has some other effect) that you can check. Ideally, you should know what the correct answer is or be able to check it by performing another computation.
3. Run the program and see if the change worked. If so, go back to step 2. If not, you'll have to do some debugging, but if the change you made was small, it shouldn't take long to find the problem.

With incremental development, your code is more likely to work the first time, and if it doesn't, the problem is more likely to be obvious. And that brings us to the Sixth Theorem of Debugging:

> The best kind of debugging is the kind you don't have to do.

In practice, there are two problems with incremental development. First, sometimes you have to write extra code to generate visible output that you can check. This extra code is called *scaffolding* because you use it to build the program and then remove it when you are done.

But the time you save on debugging is almost always worth the time you invest in scaffolding.

The second problem is that when you're getting started, you might not know how to choose steps that get from `x = 5` to the program you're trying to write. We'll look at an extended example in Chapter 6.

If you find yourself writing more than a few lines of code before you start testing and you're spending a lot of time debugging, you should try incremental development.

3.10 Chapter Review

In this chapter, we used a loop to perform a repetitive computation—updating a model 30 times—and to compute sequences and series. Also, we used the `plot` function to visualize the results.

Here are some terms from this chapter you might want to remember.

Absolute error is the difference between an approximation and an exact answer. *Relative error* is the same difference expressed as a fraction or percentage of the exact answer.

A *loop* is part of a program that runs repeatedly. A *loop variable* is a variable that gets assigned a different value each time through the loop. A *range* is a sequence of values assigned to the loop variable, often specified with the colon operator—for example, `1:5`. The *body* of a loop is the set of statements inside the loop that runs repeatedly. An *accumulator* is a variable that is used to accumulate a result a little bit at a time.

In mathematics, a *sequence* is a set of numbers that correspond to the positive integers. The numbers that make up the sequence are called *elements*. A *series* is the sum of a sequence of elements. Sometimes we compute the elements of a sequence *recurrently*, which means that each new element depends on previous elements. Sometimes we can compute the elements *directly*, without using previous elements.

Generalization is a way to make a program more versatile, for example, by replacing a specific value with a variable that can have any value. *Incremental development* is a way of programming by making a series of small, testable changes. *Scaffolding* is code you write to help you program or debug but that is not part of the finished program.

In the next chapter, we'll use a vector to store the results from a loop.

3.11 Exercises

Before you go on, you might want to work on the following exercises.

Exercise 3.5. Years ago I was in a fudge shop and saw a sign that said "Buy one pound of fudge, get another quarter pound free." That's simple enough.

But if I ran the fudge shop, I would offer a special deal to anyone who could solve the following problem:

> If you buy a pound of fudge, we'll give you another quarter pound free. And then we'll give you a quarter of a quarter pound, or one-sixteenth. And then we'll give you a quarter of that, and so on. How much fudge would you get in total?

Write a script called *fudge.m* that solves this problem. Hint: start with *series.m* and generalize it by replacing the ratio `1/2` with a variable, `r`.

Exercise 3.6. We have already seen the Fibonacci sequence, F, which is defined recurrently as

$$\text{for } i \geq 3, \quad F_i = F_{i-1} + F_{i-2}$$

In order to get started, you have to specify the first two elements, but once you have those, you can compute the rest. The most common Fibonacci sequence starts with $F_1 = 1$ and $F_2 = 1$.

Write a script called *fibonacci2.m* that uses a `for` loop to compute the first 10 elements of this Fibonacci sequence. As a postcondition, your script should assign the 10th element to `ans`.

Now generalize your script so that it computes the nth element for any value of `n`, with the precondition that you have to set `n` before you run the script. To keep things simple for now, you can assume that `n` is greater than 0.

Hint: you'll have to use two variables to keep track of the previous two elements of the sequence. You might want to call them `prev1` and `prev2`. Initially, `prev1` $= F_1$ and `prev2` $= F_2$. At the end of the loop, you'll have to update `prev1` and `prev2`; think carefully about the order of the updates!

Chapter 4

Vectors

In the previous chapter we used a loop to compute the elements of a sequence, but we were only able to store the last element. In this chapter, we'll use a vector to store all of the elements. We'll also learn how to select elements from a vector, and how to perform vector arithmetic and common vector operations like reduce and apply.

4.1 Creating Vectors

A *vector* in MATLAB is a sequence of numbers. There are several ways to create vectors; one of the most common is to put a sequence of numbers in square brackets:

```
>> [1 2 3]
ans = 1     2     3
```

Another way to create a vector is the colon operator, which we have already used to create a range of values in a `for` loop.

```
>> 1:3
ans = 1     2     3
```

In general, anything you can do with a number, you can also do with a vector. For example, you can assign a vector value to a variable:

```
>> X = [1 2 3]
X = 1     2     3
```

Note that variables that contain vectors are often capital letters. That's just a convention; MATLAB doesn't require it, but it's a useful way to remember which variables are vectors.

4.2 Vector Arithmetic

As with numbers, you can do arithmetic with vectors. If you add a number to a vector, MATLAB increments each element of the vector:

```
>> Y = X + 5
Y = 6     7     8
```

The result is a new vector; the original value of `X` has not changed.

If you add two vectors, MATLAB adds the corresponding elements of each vector and creates a new vector that contains the sums:

```
>> Z = X + Y
Z = 7     9    11
```

But adding vectors only works if the operands are the same size. Otherwise you get an error:

```
>> W = [1 2]
W = 1     2

>> X + W
Matrix dimensions must agree.
```

This error message might be confusing, because we think of `X` and `W` as vectors, not matrices. But in MATLAB, a vector is a kind of matrix. We'll come back to matrices later, but in the meantime, you might see the term in error messages and documentation.

If you divide two vectors, you might be surprised by the result:

```
>> X / Y
ans = 0.2953
```

MATLAB is performing an operation called right division, which is not what we expected. To divide the elements of `X` by the elements of `Y`, you have to use `./`, which is element-wise division:

```
>> X ./ Y
ans = 0.1667    0.2857    0.3750
```

Multiplication has the same problem. If you use `*`, MATLAB does matrix multiplication. With these two vectors, matrix multiplication is not defined, so you get an error:

```
>> X * Y
Error using  *
Incorrect dimensions for matrix multiplication.
Check that the number of columns in the first matrix
matches the number of rows in the second matrix.
To perform element-wise multiplication, use '.*'.
```

In this case, the error message is pretty helpful. As it suggests, you can use `.*` to perform element-wise multiplication:

```
>> X .* Y
ans = 6    14    24
```

As an exercise, see what happens if you use the exponentiation operator (`^`) with a vector.

4.3 Selecting Elements

You can select an element from a vector with parentheses:

```
>> Y = [6 7 8 9]
Y = 6    7     8     9

>> Y(1)
ans = 6

>> Y(4)
ans = 9
```

This means that the first element of `Y` is `6` and the fourth element is `9`. The number in parentheses is called the *index* because it indicates which element of the vector you want.

The index can be a variable name or a mathematical expression:

```
>> i = 1;
>> Y(i)
ans = 6
>> Y(i+1)
ans = 7
```

We can use a loop to display the elements of `Y`:

```
for i=1:4
     Y(i)
end
```

Each time through the loop we use a different value of `i` as an index into `Y`.

In the previous example we had to know the number of elements in `Y`. We can make it more general by using the `length` function, which returns the number of elements in a vector:

```
for i=1:length(Y)
     Y(i)
end
```

This version works for a vector of any length.

4.4 Indexing Errors

An index can be any kind of expression, but the value of the expression has to be a positive integer, and it has to be less than or equal to the length of the vector. If it's zero or negative, you'll get an error:

```
>> Y(0)
Array indices must be positive integers or logical values.
```

If it's not an integer, you get an error:

```
>> Y(1.5)
Array indices must be positive integers or logical values.
```

If the index is too big, you also get an error:

```
>> Y(5)
Index exceeds the number of array elements (4).
```

The error messages use the word *array* rather than *matrix*, but they mean the same thing, at least for now.

4.5 Vectors and Sequences

Vectors and sequences go together nicely. For example, another way to evaluate the Fibonacci sequence from Chapter 2 is to store successive values in a vector. Remember that the definition of the Fibonacci sequence is $F_1 = 1$, $F_2 = 1$, and $F_i = F_{i-1} + F_{i-2}$ for $i > 2$.

Listing 4.1 shows how we can compute the elements of this sequence and store them in a vector, using a capital letter for the vector `F` and lower-case letters for the integers `i` and `n`.

Listing 4.1: Calculating the Fibonacci sequence using a vector

```
F(1) = 1
F(2) = 1
for i=3:n
    F(i) = F(i-1) + F(i-2)
end
```

If you had any trouble with , you have to appreciate the simplicity of this script. The MATLAB syntax is similar to the math notation, which makes it easier to check for correctness.

If you only want the nth Fibonacci number, storing the whole sequence wastes some space. But if wasting space makes your code easier to write and debug, that's probably okay.

4.6 Plotting Vectors

If you call `plot` with a vector as an argument, MATLAB plots the indices on the x-axis and the elements on the y-axis. To plot the Fibonacci numbers we computed in Listing 4.1, we'd use

```
plot(F)
```

Figure 4.1 shows the result.

This way of looking at a vector is often useful for debugging, especially if it is big enough that displaying the elements on the screen is unwieldy.

4.7 Common Vector Operations

We've covered some of the basic features of vectors. Let's now look at some common patterns we use to work with data stored in vectors.

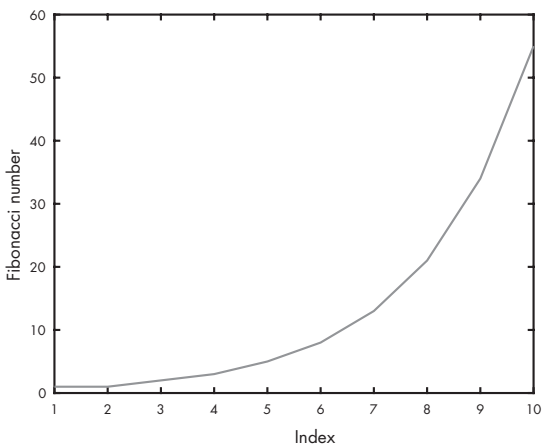

Figure 4.1: The first 10 elements of the Fibonacci sequence

4.7.1 Reduce

We frequently use loops to run through the elements of a vector and add them up, multiply them together, compute the sum of their squares, and so on. This kind of operation is called *reduce*, because it reduces a vector with multiple elements down to a single number.

For example, the loop in Listing 4.2 adds up the elements of a vector named `X` (which we assume has been defined).

Listing 4.2: Reducing a vector to a single scalar value (the sum)

```
total = 0
for i=1:length(X)
    total = total + X(i)
end
ans = total
```

The use of `total` as an accumulator is similar to what we saw in Chapter 3. Again, we use the `length` function to find the upper bound of the range, so this loop will work regardless of the length of `X`. Each time through the loop, we add in the `i`th element of `X`, so at the end of the loop `total` contains the sum of the elements.

MATLAB provides functions that perform some reduce operations. For example, the `sum` function computes the sum of the elements in a vector, and `prod` computes the product.

4.7.2 Apply

Another common use of a loop is to run through the elements of a vector, perform some operation on the elements, and create a new vector with the results. This operation is called *apply*, because you apply the operation to each element in the vector.

For example, the loop in Listing 4.3 creates a vector `Y` that contains the squares of the elements of `X` (assuming, again, that `X` is already defined).

Listing 4.3: Making a new vector Y by squaring the elements in X

```
for i=1:length(X)
    Y(i) = X(i)^2
end
```

Many apply operations can be done with element-wise operators. The following statement is more concise than the loop in Listing 4.3.

```
Y = X .^ 2
```

It also runs faster!

4.8 Chapter Review

In this chapter, we used a vector to store the elements of a sequence. We learned how to select elements from a vector and perform vector arithmetic. We performed reduce and apply operations using `for` loops, MATLAB functions, and element-wise operations.

Here are some terms from this chapter you might want to remember.

A *vector* is a sequence of values, which is a kind of *matrix*, also called an *array* in some MATLAB documentation.

An *index* is an integer value used to indicate one of the elements in a vector or matrix (also called a *subscript* in some MATLAB documentation).

An operation is *element-wise* if it acts on the individual elements of a vector or matrix (unlike some linear algebra operations).

You can *apply* an operation to all elements of a vector, and you can *reduce* a vector to a single value, for example by computing the sum of the elements.

In the next chapter, we'll meet the most important idea in computer programming: functions!

4.9 Exercises

Before you go on, you might want to work on the following exercises.

Exercise 4.1. Write a loop that computes the first n elements of the geometric sequence $A_{i+1} = A_i/2$ with $A_1 = 1$. Notice that math notation puts A_{i+1} on the left side of the equality. When you translate to MATLAB, you might want to rewrite it with A_i on the left side.

Exercise 4.2. Write an expression that computes the square root of the sum of the squares of the elements of a vector, without using a loop.

Exercise 4.3. The ratio of consecutive Fibonacci numbers, F_{n+1}/F_n, converges to a constant value as n increases. Write a script that computes a vector with the first n elements of a Fibonacci sequence (assuming that the variable `n` is defined) and then computes a new vector that contains the ratios of consecutive Fibonacci numbers. Plot this vector to see if it seems to converge. What value does it converge on?

Exercise 4.4. The following set of equations is based on a famous example of a chaotic system, the Lorenz attractor (see https://greenteapress.com/matlab/lorenz):

$$\begin{aligned} x_{i+1} &= x_i + \sigma\,(y_i - x_i)\,dt \\ y_{i+1} &= y_i + [x_i(r - z_i) - y_i]\,dt \\ z_{i+1} &= z_i + (x_i y_i - b z_i)\,dt \end{aligned}$$

1. Write a script that computes the first 10 elements of the sequences X, Y, and Z and stores them in vectors named `X`, `Y`, and `Z`.

   Use the initial values $X_1 = 1$, $Y_1 = 2$, and $Z_1 = 3$ with the values $\sigma = 10$, $b = 8/3$, $r = 28$, and $dt = 0.01$.

2. Read the documentation for `plot3` and `comet3`, and plot the results in three dimensions.

3. Once the code is working, use semicolons to suppress the output and then run the program with sequence lengths of 100, 1,000, and 10,000.

4. Run the program again with different starting conditions. What effect does it have on the result?

5. Run the program with different values for σ, b, and r, and see if you can get a sense of how each variable affects the system.

Exercise 4.5. The logistic map (see https://greenteapress.com/matlab/logistic) is described by the following equation:

$$X_{i+1} = rX_i(1 - X_i)$$

where X_i is a number between 0 and 1, and r is a positive number.

1. Write a script named *logmap.m* that computes the first 50 elements of X with `r` = 3.9 and `X1` = 0.5, where `r` is the parameter of the logistic map and `X1` is the initial value.
2. Plot the results for a range of values of r from 2.4 to 4.0. How does the behavior of the system change as you vary r?

Chapter 5

Functions

This chapter introduces the most important idea in computer programming: functions! To explain why functions are so important, I'll start by explaining one of the problems they solve: name collisions.

5.1 Name Collisions

All scripts run in the same workspace, so if one script changes the value of a variable, all other scripts see the change. With a small number of simple scripts, that's not a problem, but eventually the interactions between scripts become unmanageable.

For example, the following script computes the sum of the first `n` terms in a geometric sequence, but it also has the *side effect* of assigning values to `A1`, `total`, `i`, and `a`.

```
A1 = 1;
total = 0;
for i=1:10
    a = A1 * 0.5^(i-1);
    total = total + a;
end
ans = total
```

If you were using any of those variable names before calling this script, you might be surprised to find, after running the script, that their values had changed. If you have two scripts that

use the same variable names, you might find that they work separately and then break when you try to combine them. This kind of interaction is called a *name collision*.

As the number of scripts you write increases, and they get longer and more complex, name collisions become more of a problem. Avoiding this problem is one of several motivations for functions.

5.2 Defining Functions

A *function* is like a script, except that each function has its own workspace, so any variables defined inside a function only exist while the function is running and don't interfere with variables in other workspaces, even if they have the same name. Function inputs and outputs are defined carefully to avoid unexpected interactions.

To define a new function, you create an M-file with the name you want and put a function definition in it. For example, to create a function named `myfunc`, create an M-file named *myfunc.m* and put the following definition into it (Listing 5.1):

Listing 5.1: A function definition

```
function res = myfunc(x)
    s = sin(x)
    c = cos(x)
    res = abs(s) + abs(c)
end
```

The first non-comment word of the file has to be `function`, because that's how MATLAB tells the difference between a script and a function file.

A function definition is a compound statement. The first line is called the *signature* of the function; it defines the inputs and outputs of the function. In Listing 5.1 the *input variable* is named `x`. When this function is called, the argument provided by the user will be assigned to `x`.

The *output variable* is named `res`, which is short for *result*. You can call the output variable whatever you want, but as a convention, I like to call it `res`. Usually the last thing a function does is assign a value to the output variable.

Once you've defined a new function, you call it the same way you call built-in MATLAB functions. If you call the function as a statement, MATLAB puts the result into `ans`:

```
>> myfunc(1)

s = 0.84147098480790

c = 0.54030230586814

res = 1.38177329067604

ans = 1.38177329067604
```

But it's more common (and better style) to assign the result to a variable:

```
>> y = myfunc(1)

s = 0.84147098480790

c = 0.54030230586814

res = 1.38177329067604

y = 1.38177329067604
```

While you're debugging a new function, you might want to display intermediate results like this, but once it's working, you'll want to add semicolons to make it a *silent function*. A silent function computes a result but doesn't display anything (except sometimes warning messages). Most built-in functions are silent.

Each function has its own workspace, which is created when the function starts and destroyed when the function ends. If you try to access (read or write) the variables defined inside a function, you will find that they don't exist.

```
>> clear
>> y = myfunc(1);
>> who
Your variables are: y

>> s
Undefined function or variable 's'.
```

The only value from the function that you can access is the result, which in this case is assigned to `y`.

If you have variables named `s` or `c` in your workspace before you call `myfunc`, they will still be there when the function completes.

```
>> s = 1;
>> c = 1;
>> y = myfunc(1);
>> s, c

s = 1
c = 1
```

So inside a function you can use whatever variable names you want without worrying about collisions.

5.3 Function Documentation

At the beginning of every function file, you should include a comment that explains what the function does:

```
% res = myfunc(x)
% Compute the Manhattan distance from the origin to the
% point on the unit circle with angle (x) in radians.

function res = myfunc(x)
% this is not part of documentation given by help function

    s = sin(x);
    c = cos(x);
    res = abs(s) + abs(c);
end
```

When you ask for `help`, MATLAB prints the comment you provide.

```
>> help myfunc
  res = myfunc(x)
  Compute the Manhattan distance from the origin to the
  point on the unit circle with angle (x) in radians.
```

There are lots of conventions about what should be included in these comments. Among other things, it's a good idea to include the following:

Signature The signature of the function, which includes the name of the function, the input variable(s), and the output variable(s).

Description A clear, concise, abstract description of what the function does. An *abstract* description is one that leaves out the details of *how* the function works and includes only information that someone using the function needs to know. You can put additional comments inside the function that explain the details.

Variables An explanation of what the input variables mean; for example, in this case it is important to note that `x` is considered to be an angle in radians.

Conditions Any preconditions and postconditions.

5.4 Naming Functions

There are a few "gotchas" that come up when you start defining functions. The first is that the "real" name of your function is determined by the file name, *not* by the name you put in the function signature. As a matter of style, you should make sure that they are always the same, but if you make a mistake, or if you change the name of a function, it's easy to get confused.

In the spirit of making errors on purpose, edit *myfunc.m* and change the name of the function from `myfunc` to `something_else`, like this:

```
function res = something_else (x)
    s = sin(x);
    c = cos(x);
    res = abs(s) + abs(c);
end
```

Now call the function from the Command Window, like this:

```
>> y = myfunc(1)
y = 1.3818
```

The function is still called `myfunc`, because that's the name of the file. If you try to call it like this:

```
>> y = something_else(1)
Undefined function or variable 'something_else'.
```

It doesn't work. The name of the file is what matters; the name of the function is ignored.

The second "gotcha" is that the name of the file can't have spaces. For example, if you rename the file to *my func.m* and try to run it, you get:

```
>> y = my func(1)
 y = my func(1)
        |
Error: Unexpected MATLAB expression.
```

This fails because MATLAB thinks `my` and `func` are two different variable names.

The third "gotcha" is that your function names can collide with built-in MATLAB functions. For example, if you create an M-file named *sum.m* and then call `sum`, MATLAB might call your new function, not the built-in version! Which one actually gets called depends on the order of the directories in the search path and (in some cases) on the arguments. As an example, put the following code in a file named *sum.m*:

```
function res = sum(x)
   res = 7;
end
```

And then try this:

```
>> sum(1:3)

ans = 6

>> sum

ans = 7
```

In the first case MATLAB used the built-in function; in the second case it ran your function! This kind of interaction can be very confusing. Before you create a new function, check to see if there is already a MATLAB function with the same name. If there is, choose another name!

5.5 Multiple Input Variables

Functions can take more than one input variable. For example, the following function in Listing 5.2 takes two input variables, `a` and `b`:

Listing 5.2: A function that computes the sum of squares of two numbers

```
function res = sum_squares(a, b)
    res = a^2 + b^2;
end
```

This function computes the sum of squares of two numbers, `a` and `b`.

If we call it from the Command Window with arguments 3 and 4, we can confirm that the sum of their squares is 25.

```
>> ss = sum_squares(3, 4)
ss = 25
```

The arguments you provide are assigned to the input variables in order, so in this case 3 is assigned to `a` and 4 is assigned to `b`. MATLAB checks that you provide the right number of arguments; if you provide too few, you get

```
>> ss = sum_squares(3)
Not enough input arguments.

Error in sum_squares (line 4)
    res = a^2 + b^2;
```

This error message might be confusing, because it suggests that the problem is in `sum_squares` rather than in the function call. Keep that in mind when you're debugging.

If you provide too many arguments, you get

```
ss = sum_squares(3, 4, 5)
Error using sum_squares
Too many input arguments.
```

That's a better error message, because it's clear that the problem isn't in the function, it's in the way we're using the function.

5.6 Chapter Review

Now that we know about functions, and all the ways they can go wrong, let's put them to good use. In the next chapter we'll develop a program that uses several functions to search for Pythagorean triples (and I'll explain what those are).

Here are a few terms in this chapter you might want to remember.

A *function* is a named sequence of statements stored in an M-file. A function can have one or more *input variables*, which get their values when the function is called, and *output variables*, which return a value from the function to the caller.

The first line of a function definition is its *signature*, which specifies the name of the function, the input variables, and the output variables.

A *silent function* doesn't display anything, generate a figure, or have any other effect other than returning output values.

5.7 Exercise

Before you go on, you might want to work on the following exercise.

Exercise 5.1. Write a function called `hypotenuse` that takes two parameters, `a` and `b`, that represent the lengths of two sides of a right triangle. It should assign to `res` the length of the third side of the triangle, given by the formula

$$c = \sqrt{a^2 + b^2}$$

Chapter 6

Conditionals

In this chapter, we'll use functions and a new feature—conditional statements—to search for Pythagorean triples. A Pythagorean triple is a set of integers, like 3, 4, and 5, that are the lengths of the sides of a right triangle. Mathematically, it's a set of integers a, b, and c such that $a^2 + b^2 = c^2$. This example will also demonstrate the *incremental development* process we talked about in Chapter 3.

6.1 Relational Operators

Suppose we have three variables, `a`, `b`, and `c`, and we want to check whether they form a Pythagorean triple. We can use the equality operator (`==`) to compare two values:

```
>> a = 3;
>> b = 4;
>> c = 5;
>> a^2 + b^2 == c^2

ans = logical 1
```

The result is a *logical* value, which means it's either `1`, which means "true," or `0`, which means "false." Here's an example where the result is false:

```
>> c = 6;
>> a^2 + b^2 == c^2
ans = logical 0
```

It's a common error to use the assignment operator (=) instead of the equality operator (==). If you do, you get an error:

```
>> a^2 + b^2 = c^2
 a^2 + b^2 = c^2
           |
Error: Incorrect use of '=' operator.
To assign a value to a variable, use '='.
To compare values for equality, use '=='.
```

The equality operator is one of several *relational operators*, so called because they test relations between values. For example, `x < 10` is true (`1`) if the value of `x` is less than `10` or false (`0`) if otherwise. And `x > 0` is true if `x` is greater than `0`.

The other relational operators are `<=` for "less or equal," `>=` for "greater or equal," and `~=` for "not equal."

6.2 if Statement

Now suppose that when we find a Pythagorean triple we want to display a message. The `if` statement allows you to check for certain conditions and execute statements if the conditions are met. For example:

```
if a^2 + b^2 == c^2
    disp("Yes, that is a Pythagorean triple.")
end
```

The syntax is similar to a `for` loop. The first line specifies the condition we're interested in. If the condition is true, MATLAB executes the *body* of the statement, which is the indented sequence of statements between the `if` and the `end`.

MATLAB doesn't require you to indent the body of an `if` statement, but it makes your code more readable, so you should do it.

If the condition is not satisfied, the statements in the body are not executed.

Sometimes there are alternative statements to execute when the condition is false. In that case, you can extend the `if` statement with an `else` clause.

The complete version of the previous example might look like this:

```
if a^2 + b^2 == c^2
    disp("Yes, that is a Pythagorean triple.")
else
    disp("No, that is not a Pythagorean triple.")
end
```

Statements like `if` and `for` that contain other statements are called *compound* statements. All compound statements finish with `end`.

6.3 Incremental Development

Now that we have relational operators and `if` statements, let's start writing the program.

Here are the steps we will follow to develop the program incrementally:

1. Write a script named *find_triples.m* and start with a loop that enumerates values of `a` and displays them.
2. Write a second loop that enumerates values of `b` and a third loop that enumerates values of `c`.
3. Use the if statement from the previous section to check whether `a`, `b`, and `c` form a Pythagorean triple.
4. Display the values that pass the test.
5. Transform the script into a function and make it take an input variable that specifies the range to search.

Along the way, we'll optimize the program to eliminate unnecessary work.

6.4 Logical Functions

The first step is to create a *logical function*, which is a function that returns a logical value. The following function takes three input variables, `a`, `b`, and `c`, and returns true (`1`) if they form a Pythagorean triple and false (`0`) otherwise.

```
function res = is_pythagorean(a, b, c)
    if a^2 + b^2 == c^2
        res = 1;
    else
        res = 0;
    end
end
```

We can use this function like so:

```
>> is_pythagorean(3, 4, 5)
ans = 1
```

But we can write the same function more concisely, like this:

```
function res = is_pythagorean(a, b, c)
    res = a^2 + b^2 == c^2;
end
```

The result of the equality operator is a logical value, which we can assign directly to `res`.

Put this function in a file called *is_pythagorean.m*, so we can use it as part of our program.

6.5 Nested Loops

The next step is to write loops that enumerate different values of `a`, `b`, and `c`. Create a new file called *find_triples.m* where we'll develop the rest of the program.

We'll start with a loop for `a`:

```
for a=1:3
    a
end
```

It might seem silly to start with such a simple program, but this is an essential element of incremental development: start simple and test as you go.

The output is as expected.

```
1
2
3
```

Now we'll add a second loop for `b`. It might be tempting to write something like this:

```
for a=1:3
    disp(a)
end
for b=1:4
    disp(b)
end
```

But that loops through the values of `a` and then loops through the values of `b`, and that's not what we want.

Instead, we want to consider every possible pair of values, like this:

```
for a=1:3
    for b=1:4
        disp([a,b])
    end
end
```

Now one loop is inside the other. The inner loop gets executed three times, once for each value of `a`, so here's what the output looks like (I've adjusted the spacing to make the structure clear):

```
>> find_triples
     1     1
     1     2
     1     3
     1     4
     2     1
     2     2
     2     3
     2     4
     3     1
     3     2
     3     3
     3     4
```

The left column shows the values of `a` and the right column shows the values of `b`.

The next step is to search for values of `c` that might make a Pythagorean triple. The largest possible value for `c` is `a + b`, because otherwise we couldn't form a triangle (see https://greenteapress.com/matlab/triangle).

```
for a=1:3
    for b=1:4
        for c=1:a+b
            disp([a,b,c])
        end
    end
end
```

After each small change, run the program again and check the output.

6.6 Putting It Together

Now instead of displaying all of the triples, we'll add an `if` statement and display only Pythagorean triples:

```
for a=1:3
    for b=1:4
        for c=1:a+b
            if is_pythagorean(a, b, c)
                disp([a,b,c])
            end
        end
    end
end
```

The result is just one triple:

```
>> find_triples
     3     4     5
```

You might notice that we're wasting some effort here. After checking the case when `a` is 1 and `b` is 2, there's no point in checking the case when `a` is 2 and `b` is 1. We can save the extra work by adjusting the range of `b`:

```
for b=a:4
```

We can save even more work by adjusting the range of `c`:

```
for c=b:a+b
```

Here's the final version:

```
for a=1:3
    for b=a:4
        for c=b:a+b
            if is_pythagorean(a, b, c)
                disp([a,b,c])
            end
        end
    end
end
```

6.7 Encapsulation and Generalization

As a script, this program has the side effect of assigning values to `a`, `b`, and `c`, which would be bad if any of those names were in use. By wrapping the code in a function, we can avoid name collisions; this process is called *encapsulation* because it isolates this program from the workspace.

The first draft of the function takes no input variables:

```
function res = find_triples()
    for a=1:3
        for b=a:4
            for c=b:a+b
                if is_pythagorean(a, b, c)
                    disp([a,b,c])
                end
            end
        end
    end
end
```

The empty parentheses in the signature are not necessary, but they make it apparent that there are no input variables. Similarly, it's a good idea when calling the new function to use parentheses as a reminder that it's a function, not a script:

```
>> find_triples()
```

The output variable isn't necessary, either; it never gets assigned a value. But I put it there as a matter of habit and so my function signatures all have the same structure.

The next step is to generalize this function by adding input variables. The natural generalization is to replace the constant values `3` and `4` with a variable so we can search an arbitrarily large range of values.

```
function res = find_triples(n)
    for a=1:n
        for b=a:n
            for c=b:a+b
                if is_pythagorean(a, b, c)
                    disp([a,b,c])
                end
            end
        end
    end
end
```

Here are the results for the range from 1 to 15:

```
>> find_triples(15)
     3     4     5
     5    12    13
     6     8    10
     8    15    17
     9    12    15
```

The triples $5, 12, 13$ and $8, 15, 17$ are new, but the others are just multiples of the $3, 4, 5$ triangle.

6.8 Adding a continue Statement

As a final improvement, let's modify the function so it only displays the "lowest" of each Pythagorean triple, and not the multiples.

The simplest way to eliminate the multiples is to check whether `a` and `b` share a common factor. If they do, dividing both by the common factor yields a smaller, similar triangle that has already been checked.

MATLAB provides a `gcd` function that computes the greatest common divisor of two numbers. If `gcd(a,b)` is greater than 1, `a` and `b` share a common factor and we can use the `continue` statement to skip to the next pair. Listing 6.1 contains the final version of this function:

Listing 6.1: Our final Pythagorean triples function

```
function res = find_triples(n)
    for a=1:n
        for b=a:n
```

```
            for c=b:a+b
                if gcd(a,b) > 1
                        continue
                end
                if is_pythagorean(a, b, c)
                    disp([a,b,c])
                end
            end
        end
    end
end
```

The `continue` statement causes the program to end the current iteration immediately, jump to the top of the loop, and "continue" with the next iteration.

In this case, since there are three loops, it might not be obvious which loop to jump to, but the rule is to jump to the innermost loop (which is what we want).

Here are the results with `n = 40`:

```
>> find_triples(40)
     3     4     5
     5    12    13
     7    24    25
     8    15    17
     9    40    41
    12    35    37
    20    21    29
```

6.9 How Functions Work

Let's review the sequence of steps that occur when you call a function:

1. Before the function starts running, MATLAB creates a new workspace for it.
2. MATLAB evaluates each of the arguments and assigns the resulting values, in order, to the input variables (which live in the *new* workspace).
3. The body of the code executes. Somewhere in the body a value gets assigned to the output variable.

4. The function's workspace is destroyed; the only thing that remains is the value of the output variable and any side effects the function had (like displaying values).
5. The program resumes from where it left off. The value of the function call is the value of the output variable.

When you're reading a program and you come to a function call, there are two ways to interpret it. You can think about the mechanism I just described, and follow the execution of the program into the function and back, or you can assume that the function works correctly, and go on to the next statement after the function call.

When you use a built-in function, it's natural to assume that it works, in part because you don't usually have access to the code in the body of the function. But when you start writing your own functions, you might find yourself following the "flow of execution." This can be useful while you are learning, but as you gain experience, you should get more comfortable with the idea of writing a function, testing it to make sure it works, and then forgetting about the details of how it works.

Forgetting about details is called *abstraction*; in the context of functions, abstraction means forgetting about *how* a function works and just assuming (after appropriate testing) that it works. For many people, it takes some time to get comfortable with functions. If you are one of them, you might be tempted to avoid functions, and sometimes you can get by without them.

But experienced programmers use functions extensively, for several good reasons. First, each function has its own workspace, so using functions helps avoid name collisions. Functions also lend themselves to incremental development: you can debug the body of the function first (as a script), then encapsulate it as a function, and then generalize it by adding input variables.

Also, functions allow you to divide a large problem into small pieces, work on the pieces one at a time, and then assemble a complete solution.

Once you have a function working, you can forget about the details of how it works and concentrate on what it does. This process of abstraction is an important tool for managing the complexity of large programs.

6.10 Chapter Review

In this chapter, we encountered relational operators and `if` statements, and we used them to develop a program that searches for Pythagorean triples. We wrote a *logical function*, which is a function that returns a logical value (`1` for "true" or `0` for "false").

We also saw an example of *incremental development*, or developing programs gradually, adding just a few lines of code at a time and testing as you go. If you develop programs this way, you will have fewer bugs and you will find them more quickly.

This chapter defined two new terms: *encapsulation* is the process of wrapping part of a program in a function in order to limit interactions (including name collisions) between the function and the rest of the program; *abstraction* is the process of ignoring the details of how a function works in order to focus on a simpler model of what the function does.

The next chapter introduces a new tool, called `fzero`, that we'll use to solve nonlinear equations.

6.11 Exercise

Before you go on, you might want to work on the following exercise.

Exercise 6.1. There is an interesting connection between Fibonacci numbers and Pythagorean triples. If F is a Fibonacci sequence, then

$$\left(F_i F_{i+3},\, 2F_{i+1}F_{i+2},\, F_{i+1}^2 + F_{i+2}^2\right)$$

is a Pythagorean triple, for all $i \geq 1$.

Write a function named `fib_triple` that takes `n` as an input variable, computes the first `n` Fibonacci numbers, stores them in a vector, and checks whether this formula produces Pythagorean triples for numbers in the sequence.

Chapter 7

Zero-Finding

In this chapter we'll use the MATLAB function `fzero` to solve nonlinear equations. Nonlinear equations are useful for modeling physical systems; for example, in one of the exercises at the end of this chapter, you can use `fzero` to find the equilibrium point of an object floating on water. Using `fzero` is also an opportunity to learn about *function handles*, which we'll need for the following chapters.

7.1 Solving Nonlinear Equations

What does it mean to "solve" an equation? That may seem like an obvious question, but let's take a minute to think about it, starting with a simple example.

Suppose we want to know the value of a variable, x, but all we know about it is the relationship $x^2 = a$. If you've taken algebra, you probably know how to solve this equation: you take the square root of both sides and get $x = \pm\sqrt{a}$. Then, with the satisfaction of a job well done, you move on to the next problem.

But what have you really done? The relationship you derived is equivalent to the relationship you started with—they contain the same information about x—so why is the second one preferable to the first?

There are two reasons. One is that the relationship is now *explicit* in x: because x is all alone on the left side, we can treat the right side as a recipe for computing x, assuming that we know the value of a.

The other reason is that the recipe is written in terms of operations we know how to perform. Assuming that we know how to compute square roots, we can compute the value of x for any value of a.

When people talk about solving an equation, what they usually mean is something like "finding an equivalent relationship that is explicit in one of the variables." In the context of this book, that's what we'll call an *analytic solution*, to distinguish it from a *numerical solution*, which is what we are going to do next.

To demonstrate a numerical solution, consider the equation $x^2 - 2x = 3$. You could solve this analytically, either by factoring it or by using the quadratic formula, and you would discover that there are two solutions, $x = 3$ and $x = -1$. Alternatively, you could solve it numerically by rewriting it as $x = \pm\sqrt{2x+3}$.

This equation is not explicit, since x appears on both sides, so it's not clear that this move did any good at all. But suppose we had reason to expect a solution near 4. We could start with $x = 4$ as an *initial value* and then use the equation $x = \sqrt{2x+3}$ to compute successive approximations of the solution. (To understand why this works, see https://greenteapress.com/matlab/fixed.)

Here's what happens:

```
>> x = 4;
>> x = sqrt(2*x+3)
x = 3.3166

>> x = sqrt(2*x+3)
x = 3.1037

>> x = sqrt(2*x+3)
x = 3.0344

>> x = sqrt(2*x+3)
x = 3.0114

>> x = sqrt(2*x+3)
x = 3.0038
```

After each iteration, `x` is closer to the correct answer, and after five iterations the relative error is about 0.1 percent, which is good enough for most purposes.

Techniques that generate numerical solutions are called *numerical methods*. The nice thing about the method we just used is that it's simple. But it doesn't always work, and it's not often used in practice. We'll see a better alternative in the next section.

7.1.1 Zero-Finding

The MATLAB function `fzero` that uses numerical methods to search for solutions to nonlinear equations. In order to use it, we have to rewrite the equation as an *error function*, like this:

$$f(x) = x^2 - 2x - 3$$

The value of the error function is 0 if x is a solution and nonzero if it is not. This function is useful because we can use values of $f(x)$, evaluated at various values of x, to infer the location of the solutions. And that's what `fzero` does. Values of x where $f(x) = 0$ are called zeros of the function or *roots*.

To use `fzero` you have to define a MATLAB function that computes the error function, like this:

```
function res = error_func(x)
    res = x^2 - 2*x -3;
end
```

You can call `error_func` from the Command Window and confirm that 3 and -1 are zeros:

```
>> error_func(3)
ans = 0

>> error_func(-1)
ans = 0
```

But let's pretend that we don't know where the roots are; we only know that one of them is near 4. Then we could call `fzero` like this:

```
>> fzero(@error_func, 4)
ans = 3.0000
```

Success! We found one of the zeros.

The first argument is a *function handle* that specifies the error function. The `@` symbol allows us to name the function without calling it. The interesting thing here is that you're not actually calling `error_func` directly; you're just telling `fzero` where it is. In turn, `fzero` calls your error function—more than once, in fact.

The second argument is the initial value. If we provide a different value, we get a different root (at least sometimes).

```
>> fzero(@error_func, -2)
ans = -1
```

Alternatively, if you know two values that bracket the root, you can provide both.

```
>> fzero(@error_func, [2,4])
ans = 3
```

The second argument is a vector that contains two elements.

You might be curious to know how many times `fzero` calls your function, and where. If you modify `error_func` so that it displays the value of `x` when it is called and then run `fzero` again, you get

```
>> fzero(@error_func, [2,4])
x = 2
x = 4
x = 2.75000000000000
x = 3.03708133971292
x = 2.99755211623500
x = 2.99997750209270
x = 3.00000000025200
x = 3.00000000000000
x = 3
x = 3
ans = 3
```

Not surprisingly, it starts by computing $f(2)$ and $f(4)$. Then it computes a point in the interval, 2.75, and evaluates f there. After each iteration, the interval gets smaller and the guess gets closer to the true root. The `fzero` function stops when the interval is so small that the estimated zero is correct to about 15 digits.

If you'd like to know more about how `fzero` works, see Chapter 15.2.

7.1.2 What Could Go Wrong?

The most common problem people have with `fzero` is leaving out the `@`. In that case, you get something like so:

```
>> fzero(error_func, [2,4])
Not enough input arguments.

Error in error_func (line 2)
    res = x^2 - 2*x -3;
```

The error occurs because MATLAB treats the first argument as a function call, so it calls `error_func` with no arguments.

Another common problem is writing an error function that never assigns a value to the output variable. In general, functions should *always* assign a value to the output variable, but MATLAB doesn't enforce this rule, so it's easy to forget.

For example, if you write

```
function res = error_func(x)
    y = x^2 - 2*x -3
end
```

and then call it from the Command Window,

```
>> error_func(4)
y = 5
```

it looks like it worked, but don't be fooled. This function assigns a value to `y`, and it displays the result, but when the function ends, `y` disappears along with the function's workspace. If you try to use it with `fzero`, you get

```
>> fzero(@error_func, [2,4])
y = -3

Error using fzero (line 231)
FZERO cannot continue because user-supplied function_handle ==>
error_func failed with the error below.

Output argument "res" (and maybe others) not assigned during call
to "error_func".
```

If you read it carefully, this is a pretty good error message, provided you understand that "output argument" and "output variable" are the same thing.

You would have seen the same error message when calling `error_func` from the interpreter, if you had assigned the result to a variable:

```
>> x = error_func(4)
y = 5

Output argument "res" (and maybe others) not assigned during
call to "error_func".
```

Another thing can go wrong: if you provide an interval for the initial guess and it doesn't actually contain a root, you get

```
>> fzero(@error_func, [0,1])
Error using fzero (line 272)
The function values at the interval endpoints must differ in sign.
```

There is one other thing that can go wrong when you use `fzero`, but this one is less likely to be your fault. It's possible that `fzero` won't be able to find a root.

Generally, `fzero` is robust, so you may never have a problem, but you should remember that there is no guarantee that `fzero` will work, especially if you provide a single value as an initial guess. Even if you provide an interval that brackets a root, things can still go wrong if the error function is discontinuous.

7.1.3 Choosing an Initial Value

The better your initial value is, the more likely it is that `fzero` will work, and the fewer iterations it will need.

When you're solving problems in the real world, you'll usually have some intuition about the answer. This intuition is often enough to provide a good initial guess.

If not, another way to choose an initial guess is to plot the error function and approximate the zeros visually. If you have a function like `error_func` that takes a scalar input variable and returns a scalar output variable, you can plot it with `ezplot`:

```
>> ezplot(@error_func, [-2,5])
```

The first argument is a function handle; the second is the interval you want to plot the function in. By examining the plot, you can estimate the locations of the two roots.

7.1.4 Vectorizing Functions

When you call `ezplot`, you might get the following warning (or error, if you're using Octave):

```
Warning: Function failed to evaluate on array inputs;
vectorizing the function may speed up its evaluation and
avoid the need to loop over array elements.
```

This means that MATLAB tried to call `error_func` with a vector, and it failed. The problem is that it uses the `*` and `^` operators. With vectors, those operators don't do what we want, which is *element-wise* multiplication and exponentiation (see "Vector Arithmetic" on page 38).

If you rewrite `error_func` like this:

```
function res = error_func(x)
    res = x.^2 - 2.*x -3;
end
```

the warning message goes away, and `ezplot` runs faster.

7.2 Debugging

When you start writing longer programs, you might spend more time debugging. So we'll end this chapter with some debugging tips.

7.2.1 More Name Collisions

Functions and variables occupy the same workspace, which means that whenever a name appears in an expression, MATLAB starts by looking for a variable with that name; if there isn't one, it looks for a function.

As a result, if you have a variable with the same name as a function, the variable *shadows* the function; metaphorically, you can't see the function because the variable is in the way.

For example, if you assign a value to `sin` and then try to use the `sin` function, you might get an error:

```
>> sin = 3;
>> sin(5)
Index exceeds the number of array elements (1).

'sin' appears to be both a function and a variable.
If this is unintentional, use 'clear sin' to remove
the variable 'sin' from the workspace.
```

Since the value we assigned to `sin` is a number, and a number is considered a 1×1 matrix, MATLAB tries to access the fifth element of the matrix and finds that there isn't one.

In this case, MATLAB is able to detect the error, and the error message is pretty helpful. But if the value of `sin` were a vector, or if the argument were smaller, you would be in trouble. For example,

```
>> sin = 3;
>> sin(1)
ans = 3
```

Just to review, the sine of 1 is not 3!

You can avoid these problems by choosing function names carefully. Use long, descriptive names for functions, not single letters like `f`. To be even clearer, use function names that end in `func`. And before you define a function, check whether MATLAB already has a function with the same name.

7.2.2 Debugging Your Head

When you're working with a new function or a new language feature for the first time, you should test it in isolation before you put it into your program.

For example, suppose you know that `x` is the sine of some angle and you want to find the angle. You find the MATLAB function `asin`, and you're pretty sure it computes the inverse sine function. Pretty sure is not good enough; you want to be very sure.

Since we know $\sin 0 = 0$, we could try

```
>> asin(0)
ans = 0
```

which is correct. Now, we also know that the sine of 90°is 1, so if we try `asin(1)`, we expect the answer to be 90, right?

```
>> asin(1)
ans = 1.5708
```

Oops. We forgot that the trig functions in MATLAB work in radians, not degrees. The answer we got is $\pi/2$, which is 90°, in radians.

With this kind of testing, you're not really checking for errors in your program; you're checking your understanding. If you make an error because you are confused about how MATLAB works, it might take a long time to find, because when you look at the code, it looks right.

Which brings us to the Seventh Theorem of Debugging:

> The worst bugs aren't in your code; they're in your head.

7.3 Chapter Review

This chapter introduced `fzero`, a function we can use to solve nonlinear equations. To use `fzero`, you have to write an error function and pass a function handle as an argument. Using functions in this way can be tricky at first, but get comfortable with it, because we are going to use it a lot.

Here are some terms from this chapter you might want to remember.

If we can solve an equation by performing algebraic operations and deriving an explicit way to compute a value, the result is an *analytic solution*. Otherwise, we can use a *numerical method*, which finds a *numerical solution* to the equation, which is usually an approximation.

To solve nonlinear equations, we often rewrite them as functions and then find one or more *zeros* of the function, that is, arguments that make the value of the function 0.

A *function handle* is a way of referring to a function by name (and passing it as an argument) without calling it.

Finally, *shadowing* is a kind of name collision in which a new definition causes an existing definition to become invisible. In MATLAB, variable names can shadow built-in functions (with hilarious results).

In the next chapter, we'll write functions that take vectors as inputs and return vectors as outputs.

7.4 Exercises

Before you go on, you might want to work on the following exercises.

Exercise 7.1. 1. Write a function called `cheby6` that evaluates the sixth Chebyshev polynomial. It should take an input variable, x, and return

$$32x^6 - 48x^4 + 18x^2 - 1$$

2. Use `ezplot` to display a graph of this function in the interval from -1 to 1. Estimate the location of any zeros in this range.

3. Use `fzero` to find as many different roots as you can. Does `fzero` always find the root that is closest to the initial value?

Exercise 7.2. When a duck is floating on water, how much of its body is submerged?[1]

To estimate a solution to this problem, we'll assume that the submerged part of a duck is well approximated by a section of a sphere. If a sphere with radius r is submerged in water to a depth d, the volume of the sphere below the water line is

$$V = \frac{\pi}{3}(3rd^2 - d^3) \quad \text{as long as} \quad d < 2r$$

We'll also assume that the density of a duck is $\rho = 0.3\,\mathrm{g/cm^3}$ (0.3 times the density of water) and that its mass is $\frac{4}{3}\pi r^3 \rho\,\mathrm{g}$.

Finally, according to the law of buoyancy, an object floats at the level where the weight of the displaced water equals the total weight of the object.

Here are some suggestions for how to proceed:

1. Write an equation relating ρ, d, and r.
2. Rearrange the equation so the right-hand side is zero. Our goal is to find values of d that are roots of this equation.
3. Write a MATLAB function that evaluates this function. Test it, then make it a quiet function.
4. Make a guess about the value of d_0 to use as an initial value.
5. Use `fzero` to find a root near d_0.
6. Check to make sure the result makes sense. In particular, check that $d < 2r$, because otherwise the volume equation doesn't work!
7. Try different values of ρ and r and see if you get the effect you expect. What happens as ρ increases? Goes to infinity? Goes to zero? What happens as r increases? Goes to infinity? Goes to zero?

[1] This exercise is adapted from C. F. Gerald and P. O. Wheatley, *Applied Numerical Analysis*, 4th edition (Boston: Addison-Wesley, 1989).

Chapter 8

Functions of Vectors

Now that we have functions and vectors, we'll put them together to write functions that take vectors as input variables and return vectors as output variables. You'll also see two patterns for computing with vectors: existential and universal quantification.

8.1 Functions and Vectors

In this section we'll look at common patterns involving functions and vectors, and you will learn how to write a single function that can work with vectors as well as scalars.

8.1.1 Vectors as Input Variables

Since many of the built-in functions take vectors as arguments, it should come as no surprise that you can write functions that take vectors as input. Here's a simple (but not very useful) example:

```
function res = display_vector(X)
    for i=1:length(X)
        display(X(i))
    end
end
```

There's nothing special about this function. The only difference from the scalar functions we've seen is that this one uses a capital letter to remind us that `X` is a vector.

Using `display_vector` doesn't actually return a value; it just displays the elements of the vector it gets as an input variable:

```
>> display_vector(1:3)
    1
    2
    3
```

Here's a more interesting example that encapsulates the code from Listing 4.2 on page 42 to add up the elements of a vector:

```
function res = mysum(X)
    total = 0;
    for i=1:length(X)
        total = total + X(i);
    end
    res = total;
end
```

I called this function `mysum` to avoid a collision with the built-in function `sum`, which does pretty much the same thing.

Here's how you call it from the Command Window:

```
>> total = mysum(1:3)
total = 6
```

Because this function has an output variable, I made a point of assigning it to a variable.

8.1.2 Vectors as Output Variables

There's also nothing wrong with assigning a vector to an output variable. Here's an example that encapsulates the code from Listing 4.3 on page 43:

```
function res = mysquare(X)
    for i=1:length(X)
        Y(i) = X(i)^2;
    end
    res = Y;
end
```

This function squares each element of `X` and stores it as an element of `Y`. Then it assigns `Y` to the output variable, `res`. Here's how we use this function:

```
>> V = mysquare(1:3)
V = 1     4     9
```

The input variable is a vector with the elements `1,2,3`. The output variable is a vector with the elements `1,4,9`.

8.1.3 Vectorizing Functions

Functions that work on vectors will almost always work on scalars as well, because MATLAB considers a scalar to be a vector with length 1.

```
>> mysum(17)
ans = 17

>> mysquare(9)
ans = 81
```

Unfortunately, the converse isn't always true. If you write a function with scalar inputs in mind, it might not work on vectors.

But it might! If the operators and functions you use in the body of your function work on vectors, then your function will probably work on vectors. For example, here's the very first function we wrote:

```
function res = myfunc(x)
    s = sin(x);
    c = cos(x);
    res = abs(s) + abs(c);
end
```

And lo! It turns out to work on vectors:

```
>> Y = myfunc(1:3)
Y = 1.3818    1.3254    1.1311
```

Some of the other functions we wrote don't work on vectors, but they can be patched up with just a little effort. For example, here's `hypotenuse` from Section 5.1:

```
function res = hypotenuse(a, b)
    res = sqrt(a^2 + b^2);
end
```

This doesn't work on vectors because the `^` operator tries to do matrix exponentiation, which only works on square matrices.

```
>> hypotenuse(1:3, 1:3)
Error using  ^  (line 51)
Incorrect dimensions for raising a matrix to a power.
Check that the matrix is square and the power is a scalar.
To perform element-wise matrix powers, use '.^'.
```

But if you replace `^` with the element-wise operator (`.^`), it works!

```
>> A = [3,5,8];
>> B = [4,12,15];
>> C = hypotenuse(A, B)

C = 5    13    17
```

The function matches up corresponding elements from the two input vectors, so the elements of `C` are the hypotenuses of the pairs $(3, 4)$, $(5, 12)$, and $(8, 15)$, respectively.

In general, if you write a function using only element-wise operators and functions that work on vectors, the new function will also work on vectors.

8.1.4 Sums and Differences

Another common vector operation is *cumulative sum*, which takes a vector as an input and computes a new vector that contains all of the partial sums of the original. In math notation, if V is the original vector, the elements of the cumulative sum, C, are

$$C_i = \sum_{j=1}^{i} V_j$$

In other words, the ith element of C is the sum of the first i elements from V. MATLAB provides a function named `cumsum` that computes cumulative sums:

```
>> X = 1:5

X = 1     2     3     4     5

>> C = cumsum(X)

C = 1     3     6    10    15
```

The inverse operation of `cumsum` is `diff`, which computes the difference between successive elements of the input vector.

```
>> D = diff(C)

D = 2     3     4     5
```

Notice that the output vector is shorter by one than the input vector. As a result, MATLAB's version of `diff` is not exactly the inverse of `cumsum`. If it were, we would expect `cumsum(diff(X))` to be `X`:

```
>> cumsum(diff(X))

ans = 1     2     3     4
```

But it isn't.

Exercise 8.1. Write a function named `mydiff` that computes the inverse of `cumsum` so that `cumsum(mydiff(X))` and `mydiff(cumsum(X))` both return `X`.

8.1.5 Products and Ratios

The multiplicative version of `cumsum` is `cumprod`, which computes the *cumulative product*. In math notation, that's

$$P_i = \prod_{j=1}^{i} V_j$$

In MATLAB, that looks like

```
>> V = 1:5
```

```
V = 1     2     3     4     5

>> P = cumprod(V)

P = 1     2     6    24   120
```

MATLAB doesn't provide the multiplicative version of `diff`, which would be called `ratio`, and which would compute the ratio of successive elements of the input vector.

Exercise 8.2. Write a function named `myratio` that computes the inverse of `cumprod`, so that `cumprod(myratio(X))` and `myratio(cumprod(X))` both return `X`.

You can use a loop, or if you want to be clever, you can take advantage of the fact that $e^{\ln a + \ln b} = ab$.

8.2 Computing with Vectors

In this section, we'll look at two common patterns for working with vectors and connect them to the corresponding ideas from mathematics, existential and universal quantification. And you'll learn about logical vectors, which contain the Boolean values 0 and 1.

8.2.1 Existential Quantification

It's often useful to check the elements of a vector to see if there are any that satisfy a condition. For example, you might want to know if there are any positive elements. In mathematical terms, checking whether something exists is called *existential quantification*, and it's denoted with the symbol $\exists$, which is pronounced "there exists." For example,

$$\exists x \text{ in } S : x > 0$$

means, "there exists some element x in the set S such that $x > 0$." In MATLAB, it's natural to express this idea with a logical function, like `exists`, that returns `1` if there is such an element and `0` if there is not.

```
function res = exists(X)
    for i=1:length(X)
        if X(i) > 0
            res = 1;
            return
        end
```

```
    end
    res = 0;
end
```

We haven't seen the `return` statement before ; it's similar to `break` except that it breaks out of the whole function, not just the loop. That behavior is what we want here because as soon as we find a positive element, we know the answer (it exists!) and we can end the function immediately without looking at the rest of the elements.

If we get to the end of the loop, that means we didn't find what we were looking for, so the result is `0`.

8.2.2 Universal Quantification

Another common operation on vectors is to check whether *all* of the elements satisfy a condition, which is called *universal quantification*, denoted with the symbol $\forall$ and pronounced "for all." So the expression

$$\forall x \text{ in } S : x > 0$$

means "for all elements x in the set S, $x > 0$."

One way to evaluate this expression in MATLAB is to reduce the problem to existential quantification, that is, to rewrite

$$\forall x \text{ in } S : x > 0$$

to the following:

$$\sim\exists x \text{ in } S : x \leq 0$$

where $\sim\exists$ means "does not exist." In other words, checking that all the elements are positive is the same as checking that there are no elements that are nonpositive.

Exercise 8.3. Write a function named `forall` that takes a vector and returns `1` if all of the elements are positive and `0` if there are any nonpositive elements.

8.2.3 Logical Vectors

When you apply a logical operator to a vector, the result is a *logical vector*: a vector whose elements are the logical values 1 and 0. Let's look at an example:

```
>> V = -3:3

V = -3    -2    -1     0     1     2     3

>> L = V>0

L =  0     0     0     0     1     1     1
```

In this example, `L` is a logical vector whose elements correspond to the elements of `V`. For each positive element of `V`, the corresponding element of `L` is `1`.

Logical vectors can be used like flags to store the state of a condition. And they are often used with the `find` function, which takes a logical vector and returns a vector that contains the indices of the elements that are "true."

Applying `find` to `L` from the example above yields

```
>> find(L)

ans = 5     6     7
```

which indicates that elements 5, 6, and 7 have the value 1.

If there are no "true" elements, the result is an empty vector.

```
>> find(V>10)

ans = Empty matrix: 1x0
```

This example computes the logical vector and passes it as an argument to `find` without assigning it to an intermediate variable. You can read this version abstractly as "find the indices of elements of `V` that are greater than 10."

We can also use `find` to write `exists` more concisely:

```
function res = exists(X)
    L = find(X>0)
    res = length(L) > 0
end
```

Exercise 8.4. Write a version of `forall` using `find`.

8.3 Debugging in Four Acts

When you're debugging a program, and especially if you're working on a hard bug, there are four things to try:

Reading Examine your code, read it back to yourself, and check that it means what you meant to say.

Running Experiment by making changes and running different versions. Often, if you display the right thing at the right place in the program, the problem becomes obvious, but you might have to invest time building scaffolding.

Ruminating Take some time to think! What kind of error is it: syntax, runtime, or logical? What information can you get from the error messages or from the output of the program? What kind of error could cause the problem you're seeing? What did you change last, before the problem appeared?

Retreating At some point, the best thing to do is back off, undoing recent changes, until you get back to a program that works and that you understand. Then you can start rebuilding.

Beginning programmers sometimes get stuck on one of these activities and forget the others. Each activity comes with its own failure mode. For example, reading your code might help if the problem is a typographical error, but not if the problem is a conceptual misunderstanding. If you don't understand what your program does, you can read it 100 times and never see the error, because the error is in your head.

Running experiments can help, especially if you run small, simple tests. But if you run experiments without thinking or reading your code, you might fall into a pattern I call "random walk programming," which is the process of making random changes until the program does the right thing. Needless to say, random walk programming can take a long time.

The way out is to take more time to think. Debugging is like an experimental science. You should have at least one hypothesis about what the problem is. If there are two or more possibilities, try to think of a test that would eliminate one of them.

Taking a break sometimes helps with the thinking. So does talking. If you explain the problem to someone else (or even yourself), you will sometimes find the answer before you finish asking the question.

But even the best debugging techniques will fail if there are too many errors or if the code you are trying to fix is too big and complicated. Sometimes the best option is to retreat, simplifying the program until you get to something that works, and then rebuild.

Beginning programmers are often reluctant to retreat, because they can't stand to delete a line of code (even if it's wrong). If it makes you feel better, copy your program into another file before you start stripping it down. Then you can paste the pieces back in, a little bit at a time.

To summarize, here's the Eighth Theorem of Debugging:

> Finding a hard bug requires reading, running, ruminating, and sometimes retreating. If you get stuck on one of these activities, try the others.

8.4 Chapter Review

This chapter presents patterns for working with vectors, including existential and universal quantification. We learned how to write functions that take vectors as input variables and return vectors as output variables. And we learned about *logical vectors*, which contain the values 1 and 0 to represent true and false.

Some of the functions in this chapter are not idiomatic MATLAB; many of them can be done more simply using built-in MATLAB operators and functions, rather than writing them yourself. But these examples demonstrate concepts you will need to know when you work on more complicated problems.

In the next chapter, we will apply the tools we have learned so far to the central goal of this book, modeling physical systems.

Chapter 9

Ordinary Differential Equations

In the previous chapter, we found the equilibrium point where a duck would float on water. This kind of problem is called *static* because it does not move. This chapter introduces *dynamic* problems, which involve things that change over time.

Also, you'll learn about a mathematical tool for describing physical systems, *differential equations*, and two computational tools for solving them, Euler's method and `ode45`.

But first I have a quick suggestion about organizing code in files.

9.1 Functions and Files

So far we've only put one function in each file. It's also possible to put more than one function in a file, but only the first one, the *top-level function*, can be called from the Command Window. The other *helper functions* can be called from anywhere inside the file, but not from the Command Window or any other file.

Keeping multiple functions in one file is convenient, but it makes debugging difficult because you can't call helper functions from the Command Window.

To help with this problem, I often use the top-level function to develop and test my helper functions. For example, to write a program for Section 7.2, I would create a file named *duck.m* and start with a top-level function named `duck` that takes no input variables and returns no output value.

Then I would write a function named `error_func` to evaluate the error function for `fzero`. To test `error_func`, I would call it from `duck` and then call `duck` from the Command Window.

Here's what my first draft might look like:

```
function res = duck()
    error = error_func(10)
end

function res = error_func(d)
    rho = 0.3;      % density in g / cm^3
    r = 10;         % radius in cm
    res = d;
end
```

This program is not complete, but it is enough code to test. Once this program is working, I would finish writing `error_func`. And once I'd finished and tested `error_func`, I would modify `duck` to use `fzero`.

This problem might only require two functions, but if there were more, I could write and test them one at a time and then combine them into a working program.

Now, let's get back to differential equations.

9.2 Differential Equations

A *differential equation (DE)* is an equation that describes the derivatives of an unknown function. "Solving a DE" means finding a function whose derivatives satisfy the equation.

For example, suppose we would like to predict the population of yeast growing in a nutrient solution. Assume that we know the initial population is 5 billion yeast cells. When yeast grow in particularly yeast-friendly conditions, the rate of growth at any point in time is proportional to the current population. If we define $y(t)$ to be the population at a time t, we can write the following equation for the rate of growth:

$$\frac{dy}{dt}(t) = ay(t)$$

where $\frac{dy}{dt}(t)$ is the derivative of $y(t)$ and a is a constant that characterizes how quickly the population grows. This equation is *differential* because it relates a function to one of its derivatives.

It is an *ordinary differential equation (ODE)* because all the derivatives involved are taken with respect to the same variable. If it related derivatives with respect to different variables (partial derivatives), it would be a *partial differential equation (PDE)*.

This equation is *first order* because it involves only first derivatives. If it involved second derivatives, it would be second order, and so on.

Lastly, it's *linear* because each term involves t, y, or dy/dt raised to the first power; if any of the terms involved products or powers of t, y, or dy/dt it would be *nonlinear*.

Now suppose we want to predict the yeast population in the future. We can do that using Euler's method.

9.3 Euler's Method

Here's a test to see if you're as smart as Leonhard Euler. Let's say you arrive at time (t) and measure the current population (y) and the rate of change (r). What do you think the population will be after some period of time Δt has elapsed?

If you said $y + r\Delta t$, congratulations! You just invented Euler's method.

This estimate is based on the assumption that r is constant, but in general it's not, so we only expect the estimate to be good if r changes slowly and Δt is small.

What if we want to make a prediction when Δt is large? One option is to break Δt into smaller pieces, called *time steps*. Then we can use the following equations to get from one time step to the next:

$$\begin{aligned} T_{i+1} &= T_i + dt \\ Y_{i+1} &= Y_i + \frac{df}{dt}(t)\, dt \end{aligned}$$

Here, T_i is a sequence of times where we estimate the value of y, and Y_i is the sequence of estimates. For each index i, Y_i is an estimate of $y(T_i)$.

If the rate doesn't change too fast and the time step isn't too big, Euler's method is accurate enough for most purposes.

9.4 Implementing Euler's Method

As an example we'll use Euler's method to solve the equation from page 88,

$$\frac{dy}{dt}(t) = ay(t)$$

with the initial condition $y(0) = 5$ billion cells and the growth parameter $a = 0.2$ per hour.

As a first step, create a file named *euler.m* with a top-level function and a helper function:

```
function res = euler()
    T(1) = 0;
    Y(1) = 5;
    r = rate_func(T(1), Y(1))
end

function res = rate_func(t, y)
   a = 0.2;
   dydt = a * y;
   res = dydt;
end
```

In `euler` we initialize the initial conditions and then call `rate_func`, so called because it computes the rate of growth in the population.

After testing these functions, we can add code to `euler` to compute these difference equations:

$$\begin{aligned} T_{i+1} &= T_i + \Delta t \\ Y_{i+1} &= Y_i + r\Delta t \end{aligned}$$

where r is the rate of growth computed by `rate_func`. Listing 9.1 has the code we need:

Listing 9.1: A function implementing Euler's method

```
function res = euler()
    T(1) = 0;
    Y(1) = 5;
    dt = 0.1;

    for i=1:40
        r = rate_func(T(i), Y(i));
        T(i+1) = T(i) + dt;
        Y(i+1) = Y(i) + r * dt;
```

```
    end
    plot(T, Y)
end
```

Before the loop, we create two vectors, `T` and `Y`, and set the first element of each with the initial conditions; `dt`, which is the size of the time steps, is 0.1 hours.

Inside the loop, we compute the growth rate based on the current time, `T(i)`, and population, `Y(i)`. You might notice that the rate depends only on population, but we pass time as an input variable anyway, for reasons you'll see soon.

After computing the growth rate, we add an element both `T` and `Y`. Then, when the loop exits, we plot `Y` as a function of `T`.

If you run the code, you should get a plot of population over time, as shown in Figure 9.1.

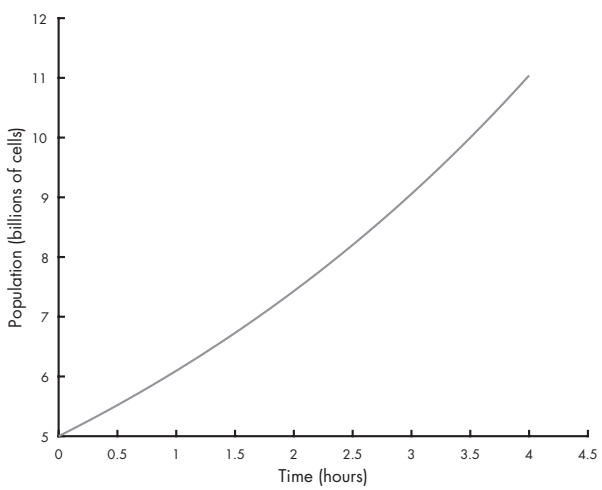

Figure 9.1: Solution to a simple differential equation by Euler's method

As you can see, the population doubles in a little less than 4 hours.

9.5 Solving ODEs with ode45

A limitation of Euler's method is that it assumes that the derivative is constant between time steps, and that's not generally true. Fortunately, there are better methods that estimate the derivative between time steps, and they are much more accurate.

MATLAB provides a function called `ode45` that implements one of these methods. In this section I'll explain how to use it; you can read more about how it works in "How ode45 Works" on page 149.

In order to use `ode45`, you have to write a function that evaluates dy/dt as a function of t and y. Fortunately, we already have one, called `rate_func`:

```
function res = rate_func(t, y)
   a = 0.2;
   dydt = a * y;
   res = dydt;
end
```

We can call `ode45` from the Command Window like this:

```
[T, Y] = ode45(@rate_func, [0, 4], 5);
plot(T, Y)
```

The first argument is a function handle, as we saw in Chapter 7. The second argument is the time interval where we want to evaluate the solution; in this case the interval is from $t = 0$ to $t = 4$ hours. The third argument is the initial population, 5 billion cells.

The `ode45` function is the first function we've seen that returns *two* output variables. In order to store them, we have to assign them to two variables, `T` and `Y`. Figure 9.2 shows the results.

The solid line is the estimate we computed with Euler's method; the dashed line is the solution from `ode45`.

For the first 2–3 hours, the two solutions are visually indistinguishable. During the last hour, they diverge slightly; at 4 hours, the difference is less than 1 percent.

For many purposes, the difference between Euler's method and `ode45` is the least of our worries. In this example, we probably don't know the initial population with perfect accuracy or the growth constant, `a`. Also, the assumption that the growth rate only depends on population is probably not true. Any of these modeling errors could be bigger than 1 percent.

However, for some problems, Euler's method can be off by a lot more than 1 percent. In those cases `ode45` is almost always more accurate, for two reasons: first, it computes the rate function several times per time step; second, if the time step is too big, `ode45` can detect the problem and shrink the time step. For more details, see "How ode45 Works" on page 149.

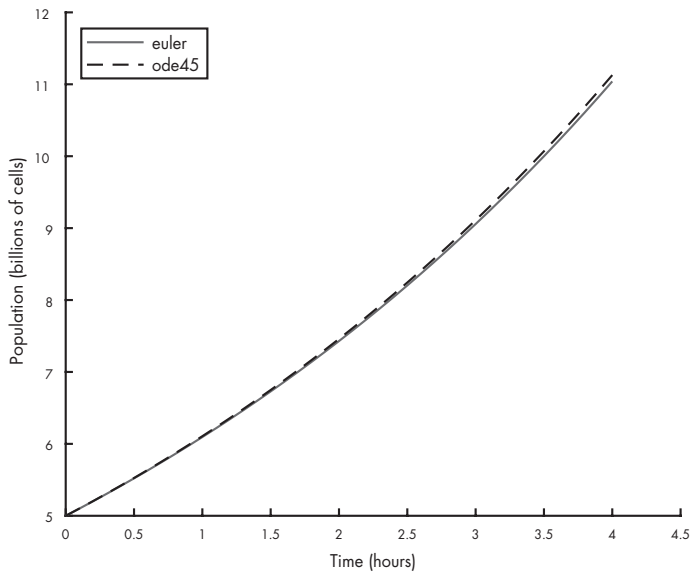

Figure 9.2: Solutions to a simple differential equation using Euler's method and `ode45`

9.6 Time Dependence

Looking at `rate_func` in the previous section, you might notice that it takes `t` as an input variable but doesn't use it. That's because the growth rate does not depend on time—bacteria don't know what time it is.

But rats do. Or, at least, they know what season it is. Suppose that the growth rate for rats depends on the current population *and* the availability of food, which varies over the course of the year. The differential equation might be something like

$$\frac{dy}{dt}(t) = ay(t)\,(1 - \cos(\omega t))$$

where t is time in days and $y(t)$ is the population at time t. Because the growth rate depends on time, this differential equation is *time dependent*.

The variables a and ω are *parameters*, which are values that quantify a physical aspect of the scenario. Parameters are often constants, but in some models they vary in time.

In this example, a characterizes the reproductive rate per day, and ω is the frequency of a periodic function that describes the effect of varying food supply on reproduction.

We'll use the values $a = 0.002$ and $\omega = 2\pi/365$ (one cycle per year). The growth rate is lowest at $t = 0$, on January 1, and highest at $t = 365/2$, on June 30.

Now we can write a function that evaluates the growth rate:

```
function res = rate_func(t, y)
    a = 0.002;
    omega = 2*pi / 365;
    res = a * y * (1 - cos(omega * t));
end
```

To test this function, I put it in a file called *rats.m* with a top-level function called `rats`:

```
function res = rats()
    t = 365/2;
    y = 1000;
    res = rate_func(t, y);
end
```

The top-level function assumes, for purposes of testing, that there are 1000 rats at $t = 365/2$ (June 30) and computes the growth rate under those conditions.

We can run the top-level function like this:

```
>> r = rats

r = 4
```

Under these conditions, the growth rate is 4 new rats per day.

Now that we've tested `rate_func`, we can use `ode45` to solve the differential equation. Here's how to call it from the top-level function in *rats.m*:

```
[T, Y] = ode45(@rate_func, [0, 365], 1000)
plot(T, Y)
```

The first argument is a function handle, again. The second argument is the interval we are interested in, a duration of one year, expressed in units of days. The third argument is the initial population, $y(0) = 1000$.

Figure 9.3 shows the results.

The population grows slowly during the winter, quickly during the summer, and then slowly again in the fall.

To see the population at the end of the year, you can display the last element of `Y`:

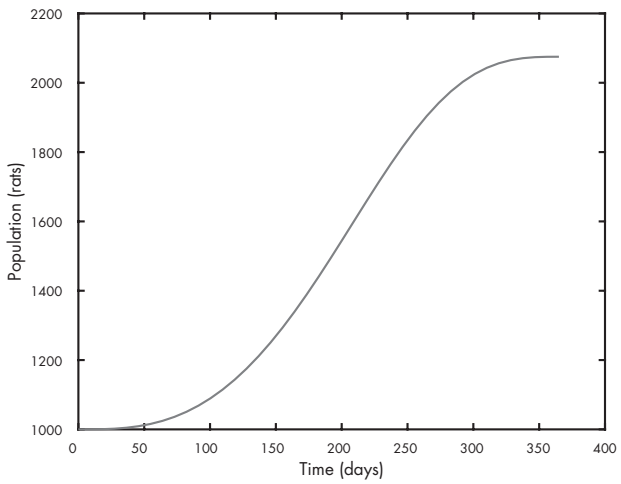

Figure 9.3: Solutions to a simple differential equation by Euler's method and `ode45`

```
Y(end)
2.0751e+03
```

That's a little more than 2000 rats, so the population roughly doubles in one year.

The index here is `end`, which is a special word in MATLAB that means "the index of the last element." You can use it in an expression, so `Y(end - 1)` is the second-to-last element of `Y`.

9.7 What Could Go Wrong?

Don't forget the `@` on the function handle. If you leave it out, like:

```
[T, Y] = ode45(rate_func, [0, 365], 1000)
```

MATLAB treats the first argument as a function call and calls `rate_func` without providing arguments. Then you get an error message:

```
Not enough input arguments.

Error in rats>rate_func (line 18)
    res = a * y * (1 - cos(omega * t));

Error in rats (line 6)
    [T, Y] = ode45(rate_func, [0, 365], 1000);
```

Also, the rate function you write has to take two input variables, `t` and `y`, in that order, and return one output variable, `res`.

If you're working with a rate function like:

$$\frac{dy}{dt}(t) = ay(t)$$

you might be tempted to write this:

```
function res = rate_func(y)         % WRONG
    a = 0.002;
    res = a * y;
end
```

But that would be wrong. So very wrong. Why? Because when `ode45` calls `rate_func`, it provides two arguments. If you only take one input variable, you'll get an error. So you have to write a function that takes `t` as an input variable, even if you don't use it:

```
function res = rate_func(t, y)      % RIGHT
    a = 0.002;
    res = a * y;
end
```

Another common error is to write a function that doesn't make an assignment to the output variable. If you write something like:

```
function res = rate_func(t, y)
    a = 0.002;
    omega = 2*pi / 365;
    r = a * y * (1 - cos(omega * t));     % WRONG
end
```

and then call it from `ode45`, you get

```
Output argument "res" (and maybe others) not assigned during call
to "rate_func".
```

I hope these warnings save you some time debugging.

9.8 Labeling Axes

The plots in this chapter have labels on the axes, and one of them has a legend, but I didn't show you how to do that. Let's do it now.

The functions to label the axes are `xlabel` and `ylabel`:

```
xlabel('Time (hours)')
ylabel('Population (billions of cells)')
```

The function to generate a legend is `legend`:

```
legend('euler', 'ode45')
```

The arguments are the labels for the lines, in the order they were drawn. Usually the legend is in the upper-right corner, but you can move it by providing an optional argument called `Location`:

```
legend('euler', 'ode45', 'Location', 'northwest')
```

Finally, save the figures using `saveas`:

```
saveas(gcf, 'runge.eps', 'epsc')
```

The first argument is the figure we want to save; `gcf` is a MATLAB command that stands for "get current figure," which is the figure we just drew. The second argument is the filename. The extension specifies the format we want, which is Encapsulated PostScript (*.eps*). The third argument tells MATLAB what driver to use. The details aren't important, but `epsc` generates figures in color.

9.9 Chapter Review

This chapter introduced *differential equations (DE)*, which are equations that describe the derivatives of an unknown function. In an *ordinary differential equation (ODE)*, all derivatives are taken with respect to the same variable, as opposed to a *partial differential equation (PDE)*, which includes derivatives with respect to more than one variable.

A *first-order DE* includes only first derivatives, and a *linear DE* includes no products or powers of the function and its derivatives. A differential equation is *time dependent* if the rate function depends on time.

When we solve a differential equation numerically, the *time step* is the interval in time between successive elements of the solution. A *parameter* is a value that appears in a model to quantify some physical aspect of the scenario being modeled.

Until now we have only put one function in each M-file, but in this chapter we wrote a *top-level function*, which is the first function in an M-file, and a *helper function*, which is any function in an M-file that is not the top-level function.

In the next chapter, we'll solve *systems* of ODEs, which are used to describe physical systems with multiple parts that interact. But first, here's an exercise where you can apply what you've learned so far.

9.10 Exercise

Before you go on, you might want to work on the following exercise.

Exercise 9.1. Suppose that you're given an 8-ounce cup of coffee at 90 °C. You have learned from bitter experience that the hottest coffee you can drink comfortably is 60 °C.

If the temperature of the coffee drops by 0.7 °C during the first minute, how long will you have to wait to drink your coffee?

You can answer this question with Newton's Law of Cooling (see https://greenteapress.com/matlab/newton):

$$\frac{dy}{dt}(t) = -k(y(t) - e)$$

where $y(t)$ is the temperature of the coffee at time t, e is the temperature of the environment, and k is a parameter that characterizes the rate of heat transfer from the coffee to the environment.

Let's assume that e is 20 °C and constant; that is, the coffee does not warm up the room.

Let's also assume k is constant. In that case, we can estimate it based on the information we have. If the temperature drops 0.7 °C during the first minute, when the coffee is 90 °C, we can write

$$-0.7 = -k(90 - 20)$$

Solving this equation yields $k = 0.01$.

Here are some suggestions for getting started:

1. Create a file named *coffee.m* and write a function called `coffee` that takes no input variables. Put a simple statement like `x = 5` in the body of the function and invoke `coffee` from the Command Window.

2. Add a helper function called `rate_func` that takes `t` and `y` and computes dy/dt. In this case, `rate_func` does not actually depend on t; nevertheless, your function has to take t as the first input variable in order to work with `ode45`.

3. Test your function by adding a line like `rate_func(0, 90)` to `coffee`, then call `coffee` from the Command Window. Confirm that the initial rate is $-0.7\,°\mathrm{C/min}$.

4. Once you get `rate_func` working, modify `coffee` to use `ode45` to compute the temperature of the coffee for 60 minutes. Confirm that the coffee cools quickly at first, then cools more slowly, and reaches room temperature after about an hour.

5. Plot the results and estimate the time when the temperature reaches $60\,°\mathrm{C}$.

Chapter 10

Systems of ODEs

In the previous chapter we used Euler's method and `ode45` to solve a single first-order differential equation. In this chapter, we'll move on to systems of ODEs and implement a model of a predator-prey system. But first, we have to learn more about matrices.

10.1 Matrices

A *matrix* is a two-dimensional version of a vector. Like a vector, it contains elements that are identified by indices. The difference is that the elements are arranged in rows and columns, so it takes *two* indices to identify an element.

10.1.1 Creating a Matrix

A common way to create a matrix is the `zeros` function, which returns a matrix with the given size filled with zeros. This example creates a matrix with two rows and three columns.

```
>> M = zeros(2, 3)

M =  0     0     0
     0     0     0
```

If you don't know the size of a matrix, you can display it by using `whos`:

```
>> whos M
  Name      Size            Bytes  Class     Attributes
  M         2x3                48  double
```

or the `size` function, which returns a vector:

```
>> V = size(M)

V = 2    3
```

The first element is the number of rows; the second is the number of columns.

To read an element of a matrix, you specify the row and column numbers:

```
>> M(1,2)

ans = 0

>> M(2,3)

ans = 0
```

When you're working with matrices, it takes some effort to remember which index comes first, row or column. I find it useful to repeat "row, column" to myself, like a mantra. You might also find it helpful to remember "down, across" or the abbreviation RC as in "radio control" or RC Cola.

Another way to create a matrix is to enclose the elements in brackets, with semicolons between rows:

```
>> D = [1,2,3 ; 4,5,6]

D =  1     2     3
     4     5     6

>> size(D)

ans = 2     3
```

10.1.2 Row and Column Vectors

Although it's useful to think in terms of numbers, vectors, and matrices, from MATLAB's point of view everything is a matrix. A number is just a matrix that happens to have one row and one column:

```
>> x = 5;
>> size(x)

ans = 1     1
```

And a vector is a matrix with only one row:

```
>> R = 1:5;
>> size(R)

ans = 1     5
```

Well, some vectors have only one row, anyway. Actually, there are two kinds of vectors. The ones we've seen so far are called *row vectors*, because the elements are arranged in a row; the other kind are *column vectors*, where the elements are in a single column.

One way to create a column vector is to create a matrix with only one element per row:

```
>> C = [1;2;3]

C =

     1
     2
     3

>> size(C)

ans = 3     1
```

The difference between row and column vectors is important in linear algebra, but for most basic vector operations, it doesn't matter. For example, when you index the elements of a vector, you don't have to know what kind it is:

```
>> R(2)

ans = 2

>> C(2)

ans = 2
```

10.1.3 The Transpose Operator

The transpose operator, which looks remarkably like an apostrophe, computes the *transpose* of a matrix, which is a new matrix that has all of the elements of the original, but with each row transformed into a column (or you can think of it the other way around).

In this example `D` has two rows:

```
>> D = [1,2,3 ; 4,5,6]

D =  1     2     3
     4     5     6
```

so its transpose has two columns:

```
>> Dt = D'

Dt = 1     4
     2     5
     3     6
```

Exercise 10.1. What effect does the transpose operator have on row vectors, column vectors, and numbers?

10.2 Solving Systems of ODEs

Now that we've seen the basics of matrices, let's see how we can use them to solve systems of differential equations.

10.2.1 Lotka-Volterra

The Lotka-Volterra model describes the interactions between two species in an ecosystem, a predator and its prey. As an example, we'll consider foxes and rabbits.

The model is governed by the following system of differential equations:

$$\begin{aligned} \frac{dx}{dt} &= ax - bxy \\ \frac{dy}{dt} &= -cy + dxy \end{aligned}$$

where x and y are the populations of rabbits and foxes, and a, b, c, and d are parameters that quantify the interactions between the two species (see https://greenteapress.com/matlab/lotka).

At first glance, you might think you could solve these equations by calling ode45 once to solve for x and once to solve for y. The problem is that each equation involves both variables, which is what makes this a *system of equations* and not just a list of unrelated equations. To solve a system, you have to solve the equations simultaneously.

Fortunately, ode45 can handle systems of equations. The difference is that the initial condition is a vector that contains the initial values $x(0)$ and $y(0)$, and the output is a matrix that contains one column for x and one for y.

Listing 10.1 shows the rate function with the parameters $a = 0.1$, $b = 0.01$, $c = 0.1$, and $d = 0.002$:

Listing 10.1: A rate function for Lotka-Volterra

```
function res = rate_func(t, V)
    % unpack the elements of V
    x = V(1);
    y = V(2);

    % set the parameters
    a = 0.1;
    b = 0.01;
    c = 0.1;
    d = 0.002;

    % compute the derivatives
    dxdt = a*x - b*x*y;
    dydt = -c*y + d*x*y;

    % pack the derivatives into a vector
    res = [dxdt; dydt];
end
```

The first input variable, t, is time. Even though the time variable is not used in this rate function, it has to be there in order for this function to work with ode45. The second input variable, V, is a vector with two elements, $x(t)$ and $y(t)$. The body of the function includes four sections, each explained by a comment.

The first section *unpacks* the vector by copying the elements into variables. This isn't necessary, but giving names to these values will help you to remember what's what. It also makes the

third section, where we compute the derivatives, resemble the mathematical equations we were given, which helps prevent errors.

The second section sets the parameters that describe the reproductive rates of rabbits and foxes, and the characteristics of their interactions. If we were studying a real system, these values would come from observations of real animals, but for this example I chose values that yield interesting results.

The third section computes the derivatives of x and y, using the equations we were given.

The last section *packs* the computed derivatives back into a vector. When `ode45` calls this function, it provides a vector as input and expects to get a vector as output.

Sharp-eyed readers will notice something different about this line:

```
    res = [drdt; dfdt];
```

The semicolon between the elements of the vector is not an error. It is necessary in this case because `ode45` requires the result of this function to be a column vector.

As always, it's a good idea to test your rate function before you call `ode45`. Create a file named *lotka.m* with the following top-level function:

```
function res = lotka()
    t = 0;
    V_init = [80, 20];
    rate_func(t, V_init)
end
```

`V_init` is a vector that represents the initial condition, 80 rabbits and 20 foxes. The result from `rate_func` is

```
-8.0000
 1.2000
```

which means that with these initial conditions, we expect the rabbit population to decline initially at a rate of 8 per week and the fox population to increase by 1.2 per week.

Now we can run `ode45` like this:

```
tspan = [0, 200]
[T, M] = ode45(@rate_func, tspan, V_init)
```

The first argument is the function handle for the rate function. The second argument is the time span, from 0 to 200 weeks. The third argument is the initial condition.

10.2.2 Output Matrices

The ode45 function returns two values: T, a vector, and M, a matrix.

```
>> size(T)
ans = 101     1

>> size(M)
ans = 101     2
```

T has 101 rows and 1 column, so it is a column vector with one row for each time step.

M has 101 rows, one for each time step, and 2 columns, one for each variable, x and y.

This structure—one column per variable—is a common way to use matrices. And plot understands this structure, so if you do the following:

```
>> plot(T, M)
```

MATLAB understands that it should plot each column from M versus T.

You can copy the columns of M into other variables like this:

```
>> R = M(:, 1);
>> F = M(:, 2);
```

In this context, the colon represents the range from 1 to end, so M(:, 1) means "all the rows, column 1" and M(:, 2) means "all the rows, column 2."

```
>> size(R)
ans = 101     1

>> size(F)
ans = 101     1
```

So R and F are column vectors.

Now we can plot these vectors separately, which makes it easier to give them different style strings:

```
>> plot(T, R, '-')
>> plot(T, F, '--')
```

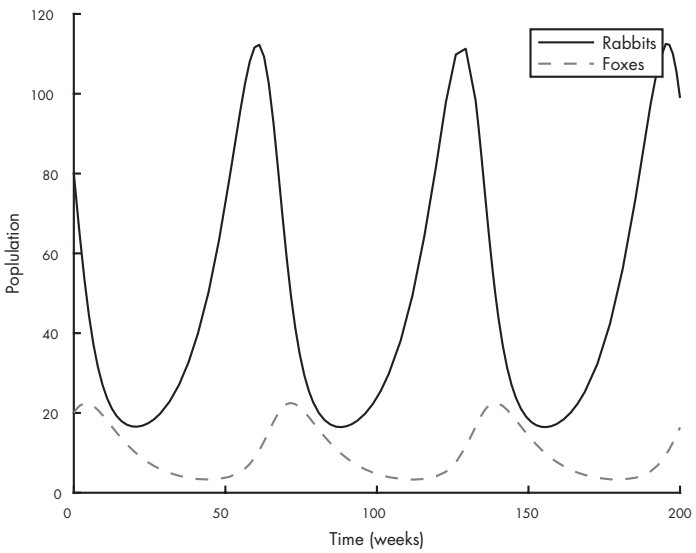

Figure 10.1: Solution for the Lotka-Volterra model

Figure 10.1 shows the results. The x-axis is time in weeks; the y-axis is population. The top curve shows the population of rabbits; the bottom curve shows foxes.

Initially, there are too many foxes, so the rabbit population declines. Then there are not enough rabbits, and the fox population declines. That allows the rabbit population to recover, and the pattern repeats.

This cycle of "boom and bust" is typical of the Lotka-Volterra model.

10.2.3 Phase Plot

Instead of plotting the two populations over time, it is sometimes useful to plot them against each other:

```
>> plot(R, F)
```

Figure 10.2 shows the result, which is called a *phase plot*. Each point on this plot represents a certain number of rabbits (on the x-axis) and a certain number of foxes (on the y-axis). Since these are the only two variables in the system, each point in this plane describes the complete *state* of the system, that is, the values of the variables we're solving for.

Over time, the state moves around the plane. Figure 10.2 shows the path traced by the state over time; this path is called a *trajectory*.

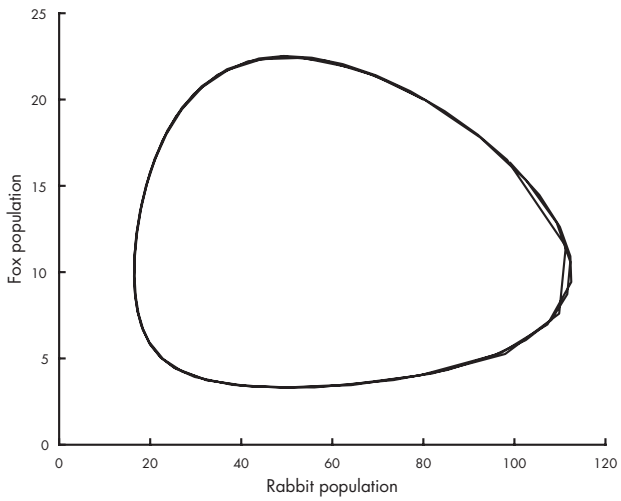

Figure 10.2: Phase plot from the Lotka-Volterra model

Since the behavior of this system is periodic, the trajectory is a loop.

If there are three variables in the system, we need three dimensions to show the state of the system, so the trajectory is a 3D curve. You can use `plot3` to trace trajectories in three dimensions, but for four or more variables, you're on your own.

10.2.4 What Could Go Wrong?

The output vector from the rate function has to be a column vector, otherwise you get an error:

```
Error using odearguments (line 93)
RATE_FUNC must return a column vector.
```

which is a pretty good error message. It's not clear *why* it needs to be a column vector, but that's not our problem.

Another possible error is reversing the order of the elements in the initial conditions or the vectors inside `lotka`. MATLAB doesn't know what the elements are supposed to mean, so it can't catch errors like this; it will just produce incorrect results.

The order of the elements (rabbits and foxes) is up to you, but you have to be consistent. That is, the order of the initial conditions you provide when you call `ode45` has to be the same as the order inside `rate_func` where you unpack the input vector and the same as the order of the derivatives in the output vector.

10.3 Chapter Review

In this chapter, we used `ode45` to solve a system of first-order differential equations. As an exercise, you'll have a chance to solve the famous Lorenz equations, one of the first examples of a chaotic system.

Here are the terms from this chapter you might want to remember.

A *row vector* is a matrix that has only one row, and a *column vector* is a matrix that has only one column. The *transpose operation* transforms the rows of a matrix into columns (or the other way around, if you prefer).

A *system of equations* is a collection of equations written in terms of the same set of variables.

In a rate function, we often have to *unpack* the input variable, copying the elements of a vector into a set of variables. Then we have to *pack* the results into a vector as an output variable.

The *state* of a system is a set of variables that quantify the condition of the system as it changes over time.

When we solve a system of differential equations, we can visualize the results with a *phase plot*, which shows the state of a system as a point in the space of possible states. A *trajectory* is a path in a phase plot that shows how the state of a system changes over time.

In the next chapter, we'll move on to second-order systems, which we use to describe systems with objects moving in space, governed by Newton's laws of motion.

10.4 Exercises

Before you go on, you might want to work on the following exercise.

Exercise 10.2. Based on the examples we've seen so far, you'd think that all ODEs describe population as a function of time, but that's not true.

For example, the Lorenz system is a system of differential equations based on a model of fluid dynamics in the atmosphere (see `https://greenteapress.com/matlab/lorenz`). It turns out to be interesting in part because its solutions are chaotic; that is, small changes in the initial conditions yield big differences in the solutions.

The system is described by these differential equations:

$$\begin{aligned} x_t &= \sigma(y - x) \\ y_t &= x(r - z) - y \\ z_t &= xy - bz \end{aligned}$$

Common values for the parameters are $\sigma = 10$, $b = 8/3$, and $r = 28$.

Use `ode45` to estimate a solution to this system of equations.

1. Create a file named *lorenz.m* with a top-level function named `lorenz` and a helper function named `rate_func`.
2. The rate function should take `t` and `V` as input variables, where the components of `V` are understood to be the current values of `x`, `y`, and `z`. It should compute the corresponding derivatives and return them in a single column vector.
3. Test the function by calling it from the top-level function with values like $t = 0$, $x = 1$, $y = 2$, and $z = 3$. Once you get your function working, you should make it a silent function before calling `ode45`.
4. Use `ode45` to estimate the solution for the time span $[0, 30]$ with the initial condition $x = 1$, $y = 2$, and $z = 3$.
5. Plot the results as a time series, that is, each of the variables as a function of time.
6. Use `plot3` to plot the trajectory of x, y, and z.

Chapter 11

Second-Order Systems

So far we've seen first-order differential equations and systems of first-order ODEs. In this chapter, we'll introduce second-order systems, which are particularly useful for modeling Newtonian motion.

11.1 Newtonian Motion

Newton's second law of motion is often written like this:

$$F = ma$$

where F is the net force acting on an object, m is the mass of the object, and a is the acceleration of the object.

This equation suggests that if you know m and a, you can compute the force. And that's true, but in most physical simulations it's the other way around: based on a physical model, you know F and m, and you compute a.

So if we know acceleration as a function of time, how do we find the position of the object, r? Well, we know that acceleration is the second derivative of position, so we can write the differential equation

$$\frac{d^2r}{dt^2} = a$$

where d^2r/dt^2 is the second time derivative of r.

Because this equation includes a second derivative, it's a second-order ODE. We can't solve the equation using `ode45` in this form, but by introducing a new variable, v, for velocity, we can rewrite it as a system of first-order ODEs:

$$\begin{aligned} \frac{dr}{dt} &= v \\ \frac{dv}{dt} &= a \end{aligned}$$

The first equation says that the first derivative of r is v; the second equation says that the first derivative of v is a.

11.2 Free Fall

As an example of Newtonian motion, let's go back to the question from Section 1.1:

> If you drop a penny from the top of the Empire State Building, how long does it take to reach the sidewalk, and how fast is it going when it gets there?

We'll start with no air resistance; then we'll add air resistance to the model and see what effect it has.

Near the surface of the earth, acceleration due to gravity is $-9.8\,\mathrm{m/s^2}$, where the minus sign indicates that gravity pulls down. If the object falls straight down, we can describe its position with a scalar value y, representing altitude.

Listing 11.1 contains a rate function we can use with `ode45` to solve this problem:

Listing 11.1: A rate function for the falling penny problem

```
function res = rate_func(t, X)
    % unpack position and velocity
    y = X(1);
    v = X(2);

    % compute the derivatives
```

```
    dydt = v;
    dvdt = -9.8;

    % pack the derivatives into a column vector
    res = [dydt; dvdt];
end
```

The rate function in Listing 11.1 takes `t` and `X` as input variables, where the elements of `X` are understood to be the position and velocity of the penny.

It returns a column vector that contains `dydt` and `dvdt`, which are velocity and acceleration, respectively. Since velocity is the second element of `X`, we can simply assign this value to `dydt`. And since the derivative of velocity is acceleration, we can assign the acceleration due to gravity to `dvdt`.

As always, we should test the rate function before we call `ode45`. Here's the top-level function we can use to test it:

```
function penny()
   t = 0;
   X = [381, 0];
   rate_func(t, X)
end
```

The initial condition of `X` is the initial position, which is the height of the Empire State Building, about 381 m, and the initial velocity, which is 0 m/s.

The result from `rate_func` is

```
    0
   -9.8000
```

which is what we expect.

Now we can run `ode45` with this rate function:

```
tspan = [0, 10]
[T, M] = ode45(@rate_func, tspan, X)
```

As always, the first argument is the function handle, the second is the time span (10 seconds), and the third is the initial condition.

The result is a vector, `T`, that contains the time values, and a matrix, `M`, that contains two columns, one for altitude and one for velocity.

We can extract the first column and plot it, like this:

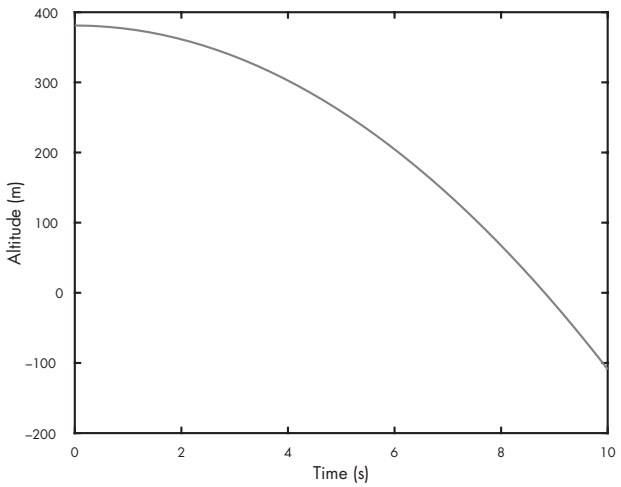

Figure 11.1: Altitude versus time for an object in free fall

```
Y = M(:, 1)
plot(T, Y)
```

Figure 11.1 shows the result. Altitude drops slowly at first and picks up speed. Between 8 and 9 seconds, the altitude reaches 0, which means the penny hits the sidewalk. But `ode45` doesn't know where the ground is, so the penny keeps going through 0 into negative altitude. We'll solve that problem in the next section.

11.3 ODE Events

Normally when you call `ode45` you specify a start time and an end time. But sometimes you don't know ahead of time when the simulation should end. To solve this problem we can define an *event*, something of interest that happens during a simulation, like the penny reaching the ground.

Here are the steps:

1. First we define an *event function* that allows `ode45` to figure out when an event occurs. Here's an event function for the penny example:

```
function [value, isterminal, direction] = event_func(t,X)
    value = X(1);
    isterminal = 1;
```

```
    direction = -1;
end
```

The event function takes the same input variables as the rate function and returns three output variables: `value` determines when an event can occur, `direction` determines whether it does, and `isterminal` determines what happens. More specifically, an event can occur when `value` passes through 0. If `direction` is positive, the event only occurs if `value` is increasing. If `direction` is negative, the event only occurs if `value` is decreasing. If `direction` is 0, the event always occurs. If `isterminal` is 1, the event causes the simulation to end; if it is 0, the simulation continues.

This event function uses the altitude of the penny as `value` so an event can occur when altitude passes through 0. Because `direction` is negative, an event occurs only when altitude is decreasing, and because `isterminal` is 1, the simulation ends if an event occurs.

2. Next, we use `odeset` to create an object called `options`:

```
options = odeset('Events', @event_func);
```

The name of the option is `Events` and the value is the handle of the event function.

3. Finally, we pass `options` as a fourth argument to `ode45`:

```
[T, M] = ode45(@rate_func, tspan, X, options);
```

When `ode45` runs, it invokes `event_func` after each time step. If the event function indicates that a terminal event occurred, `ode45` stops the simulation.

Let's look at the results from the penny example:

```
>> T(end)
8.8179

>> M(end, :)
0.0000  -86.4153
```

The last value of `T` is about 8.8, which is the number of seconds the penny takes to reach the sidewalk.

The last row of `M` indicates that the final altitude is 0, which is what we wanted, and the final velocity is about $-86\,\mathrm{m/s}$.

11.4 Air Resistance

To make this simulation more realistic, we can add air resistance. For large objects moving quickly through air, the force due to air resistance, called *drag*, is proportional to velocity

squared. For an object falling down, drag is directed up, so if velocity is negative, drag force is positive.

Here's how we can compute the force of drag as a function of velocity in one dimension:

$$f_{\mathrm{d}} = -\mathrm{sign}(v) b v^2$$

where v is velocity and b is a drag constant that depends on the density of air, the cross-sectional area of the object, and the shape of the object.

The sign or signum function returns the value 1 for positive values of v and -1 for negative values. So f_{d} is always in the opposite direction of v.

To convert from force to acceleration we have to know mass, but that's easy to find: the mass of a penny is about 2.5 g. It's not as easy to find the drag constant, but based on reports that the terminal velocity of a penny is about 18 m/s, I've estimated that it's about 75×10^{-6} kg/m.

Listing 11.2 defines a function that takes `t` and `X` as input variables and returns the total acceleration of the penny due to gravity and drag:

Listing 11.2: Calculating acceleration of a penny with drag

```
function res = acceleration(t, X)
    b = 75e-6;                  % drag constant in kg/m
    v = X(2);                   % velocity in m/s
    f_d = -sign(v) * b * v^2;   % drag force in N

    m = 2.5e-3;                 % mass in kg
    a_d = f_d / m;              % drag acceleration in m/s^2

    a_g = -9.8;                 % acceleration of gravity in m/s^2
    res = a_g + a_d;            % total acceleration
end
```

First, we compute force due to drag . Then we compute acceleration due to drag . Lastly, we compute total acceleration due to drag and gravity.

Be careful when you're working with forces and accelerations; make sure you only add forces to forces or accelerations to accelerations. In my code, I use comments to remind myself what units the values have. That helps me avoid errors like adding forces to accelerations.

To use this function, we make a small change in `rate_func`:

```
function res = rate_func(t, X)
    y = X(1);
    v = X(2);

    dydt = v;
    dvdt = acceleration(t, X);   % this line has changed

    res = [dydt; dvdt];
end
```

In the previous version, `dvdt` is always `-9.8`, the acceleration due to gravity. In this version, we call `acceleration` to compute the total acceleration due to gravity and drag .

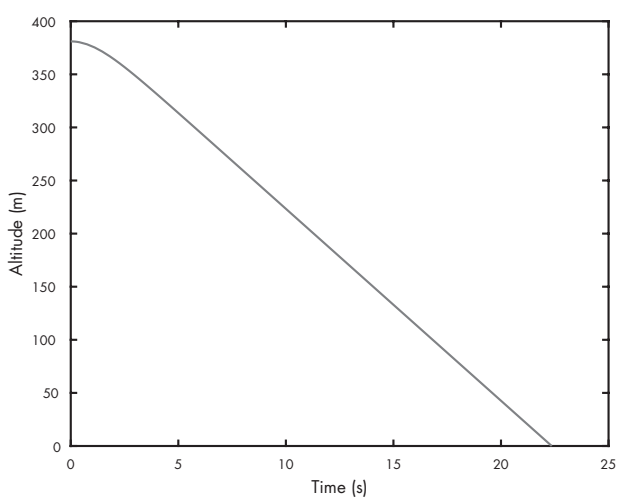

Figure 11.2: Altitude versus time for a penny in free fall with air resistance

Everything else is the same. Figure 11.2 shows the result.

Air resistance makes a big difference! Velocity increases until acceleration due to drag equals acceleration due to gravity; after that, velocity is constant and position decreases linearly (and much more slowly than it would in a vacuum).

With air resistance, the time until the penny hits the sidewalk is 22.4 s, substantially longer than before (8.8 s).

And the final velocity is 18.1 m/s, substantially slower than before (86 m/s).

11.5 Chapter Review

In this chapter, we used Newton's laws of motion to write a differential equation that describes the motion of a falling penny.

We rewrote that equation as a system of first-order differential equations so we could use `ode45` to solve it. Then we ran simulations of a falling penny with and without air resistance, also known as *drag*.

We defined an *event* as something of interest that happens during a simulation, like a collision between moving objects, and we wrote an *event function*, which allows `ode45` to figure out when an event occurs.

In the next chapter, we extend Newtonian motion to two dimensions and model the flight of a baseball.

11.6 Exercises

Before you go on, you might want to work on the following exercise.

Exercise 11.1. In this exercise we'll model the descent of a skydiver, taking into account the change in drag when the parachute opens.

1. Modify the penny code from this chapter to simulate the descent of a 75 kg skydiver from an initial altitude of 4000 m. The drag constant for a skydiver without a parachute is about 0.2 kg/m. What would the velocity of the skydiver be on impact?
2. After opening their parachute, the velocity of the skydiver slows to about 5 m/s. Use your simulation to find the drag constant that yields a terminal velocity of 5 m/s.
3. Increase the mass of the skydiver, and confirm that terminal velocity increases. This phenomenon is the source of the intuition that heavy objects fall faster; in air, they do!
4. Now suppose the skydiver free falls until they get to an altitude of 1000 m before opening the parachute. How long would it take them to reach the ground?
5. What is the lowest altitude where the skydiver can open the parachute and still land at less than 6 m/s (assuming that the parachute opens and deploys instantly)?

Exercise 11.2. Here's a question from the website *Ask an Astronomer* (see https://greenteapress.com/matlab/astro):

> If the Earth suddenly stopped orbiting the Sun, I know eventually it would be pulled in by the Sun's gravity and hit it. How long would it take the Earth to hit the Sun? I imagine it would go slowly at first and then pick up speed.

Use `ode45` to answer this question. Here are some suggestions about how to proceed:

1. Look up the Law of Universal Gravitation and any constants you need. I suggest you work entirely in SI units: meters, kilograms, and newtons.
2. When the distance between the Earth and the Sun gets small, this system behaves badly, so you should use an event function to stop when the surface of the Earth reaches the surface of the Sun.
3. Express your answer in days, and plot the results as millions of kilometers versus days.

Chapter 12

Two Dimensions

In the previous chapter, we solved a one-dimensional problem—a penny falling from the Empire State Building. Now we'll solve a two-dimensional problem—finding the trajectory of a baseball.

To do that, we'll use spatial vectors to represent quantities in two and three dimensions, including force, acceleration, velocity, and position.

12.1 Spatial Vectors

The word *vector* means different things to different people. In MATLAB, a vector is a matrix that has either one row or one column. So far, we've used MATLAB vectors to represent the following:

Sequences A mathematical sequence, like the Fibonacci numbers, is a set of values identified by integer indices; in Chapter 4.5, we used a MATLAB vector to store the elements of a sequence.

State vectors A state vector is a set of values that describes the state of a physical system. When you call `ode45`, you give it initial conditions in a state vector. Then, when `ode45` calls your rate function, it gives you a state vector.

Time series One of the results from `ode45` is a vector that represents a sequence of time values.

In this chapter, we'll see another use of MATLAB vectors: representing *spatial vectors*. A spatial vector represents a multidimensional physical quantity like position, velocity, acceleration, or force.

For example, to represent a position in two-dimensional space, we can use a vector with two elements:

```
>> P = [3 4]
```

To interpret this vector, we have to know the coordinate system it is defined in. Most commonly, we use a Cartesian system where the x-axis points east and the y-axis points north. In that case `P` represents a point 3 units east and 4 units north of the origin.

When a spatial vector is represented in this way, we can use it to compute the magnitude and direction of a physical quantity. For example, the *magnitude* of `P` is the distance from the origin to `P`, which is the hypotenuse of the triangle with sides `P(1)` and `P(2)`. We can compute it using the Pythagorean theorem:

```
>> sqrt(P(1)^2 + P(2)^2)
ans = 5
```

Or we can do the same thing using the function `norm`, which computes the *Euclidean norm* of a vector, which is its magnitude:

```
>> norm(P)
ans = 5
```

There are two ways to get the *direction* of a vector. One convention is to compute the angle between the vector and the x-axis:

```
>> atan2(P(2), P(1))
ans = 0.9273
```

In this example, the angle is about 0.9 rad. But for computational purposes, we often represent direction with a *unit vector*, which is a vector with length 1. To get a unit vector we can divide a vector by its length:

```
function res = hat(V)
    res = V / norm(V)
end
```

This function takes a vector, `V`, and returns a unit vector with the same direction as `V`. It's called `hat` because in mathematical notation, unit vectors are written with a "hat" symbol. For example, the unit vector with the same direction as $\mathbf{P}$ would be written $\hat{\mathbf{P}}$.

12.2 Adding Vectors

Vectors are useful for representing quantities like force and acceleration because we can add them up without having to think explicitly about direction.

As an example, suppose we have two vectors representing forces:

```
>> A = [2, 4];
>> B = [2, -2];
```

`A` represents a force pulling northeast; `B` represents a force pulling southeast, as shown in Figure 12.1:

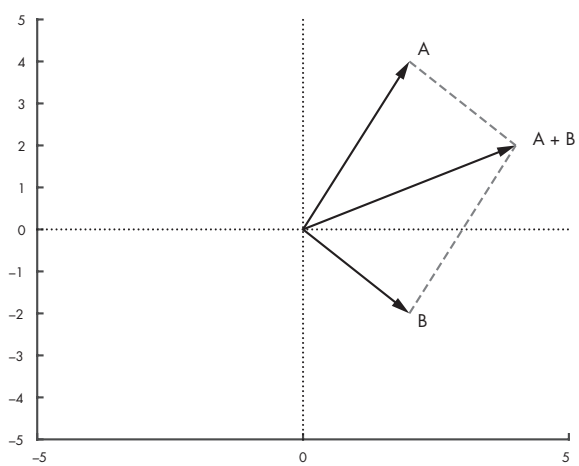

Figure 12.1: The sum of two forces represented by vectors

To compute the sum of these forces, all we have to do is add the vectors:

```
>> A + B
ans = 4     2
```

Later in the chapter, we'll use vector addition to add accelerations due to different forces acting on a baseball.

12.3 ODEs in Two Dimensions

So far we've used `ode45` to solve a system of first-order equations and a single second-order equation. Now we'll take one more step, solving a system of second-order equations.

As an example, we'll simulate the flight of a baseball. If there is no wind and no spin on the ball, the ball travels in a vertical plane, so we can think of the system as two-dimensional, with x representing the horizontal distance traveled from the starting place and y representing height or altitude.

Listing 12.1 shows a rate function we can use to simulate this system with `ode45`:

Listing 12.1: A rate function we can use to model the flight of a baseball

```
function res = rate_func(t, W)
    P = W(1:2);
    V = W(3:4);

    dPdt = V;
    dVdt = acceleration(t, P, V);

    res = [dPdt; dVdt];
end

function res = acceleration(t, P, V)
    g = 9.8;             % acceleration due to gravity in m/s^2
    a_gravity = [0; -g];
    res = a_gravity;
end
```

The second argument of `rate_func` is understood to be a vector, `W`, with four elements. The first two are assigned to `P`, which represents position; the last two are assigned to `V`, which represents velocity. Both `P` and `V` have two elements, representing the x and y components.

The goal of the rate function is to compute the derivative of `W`, so the output has to be a vector with four elements, where the first two represent the derivative of `P` and the last two represent the derivative of `V`. The derivative of `P` is velocity. We don't have to compute it; we were given it as part of `W`. The derivative of `V` is acceleration. To compute it, we call `acceleration`, which takes as input variables time, position, and velocity. In this example, we don't use any of the input variables, but we will soon.

For now we'll ignore air resistance, so the only force on the baseball is gravity. We represent acceleration due to gravity with a vector that has magnitude `g` and direction along the negative y-axis.

Let's assume that a ball is batted from an initial position 1 m above the home plate, with an initial velocity of 40 m/s in the horizontal and 30 m/s in the vertical direction.

Here's how we can call `ode45` with these initial conditions:

```
P = [0; 1];        % initial position in m
V = [40; 30];      % initial velocity in m/s
W = [P; V];        % initial condition

tspan = [0 8]
[T, M] = ode45(@rate_func, tspan, W);
```

P and V are column vectors because we put semicolons between the elements. So W is a column vector with four elements. And tspan specifies that we want to run the simulation for 8 s.

The output variables from ode45 are a vector, T, that contains time values and a matrix, M, with four columns: the first two are position; the last two are velocity.

Here's how we can plot position as a function of time:

```
X = M(:, 1);
Y = M(:, 2);

plot(T, X)
plot(T, Y)
```

X and Y get the first and second columns from M, which are the x and y position coordinates.

Figure 12.2 shows what these coordinates look like as a function of time. The x-coordinate increases linearly because the x velocity is constant. The y-coordinate goes up and down, as we expect.

The simulation ends just before the ball lands, having traveled almost 250 m. That's substantially farther than a real baseball would travel, because we have ignored air resistance, or "drag force."

12.4 Drag Force

A simple model for the drag force on a baseball is

$$\mathbf{F}_{\mathrm{d}} = -\frac{1}{2}\,\rho v C_{\mathrm{d}} A \hat{\mathbf{V}}$$

where $\mathbf{F}_{\mathrm{d}}$ is a vector that represents the force on the baseball due to drag, ρ is the density of air, C_{d} is the drag coefficient, and A is the cross-sectional area.

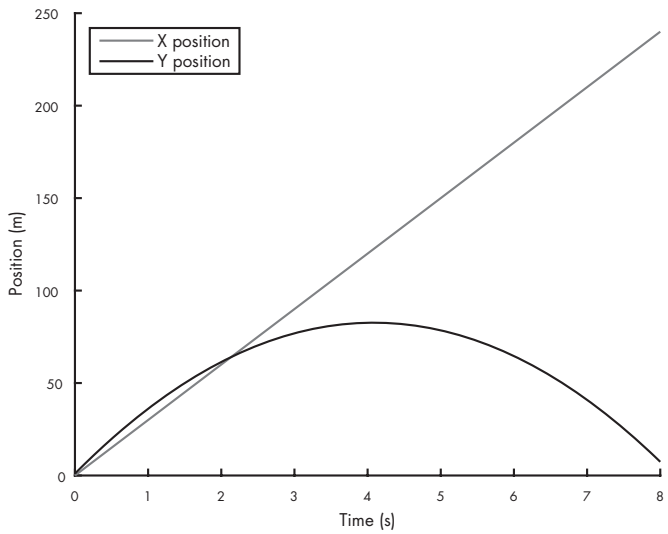

Figure 12.2: Simulated flight of a baseball neglecting drag force

$\mathbf{V}$ is the baseball's velocity vector, v is the magnitude of $\mathbf{V}$, and $\hat{\mathbf{V}}$ is a unit vector in the same direction as $\mathbf{V}$. The minus sign at the beginning means that the result is in the opposite direction to $\mathbf{V}$.

The function in Listing 12.2 computes the drag force on a baseball:

Listing 12.2: A function that calculates the drag force on a baseball

```
function res = drag_force(V)
    C_d = 0.3;       % dimensionless
    rho = 1.3;       % kg / m^3
    A = 0.0042;      % m^2
    v = norm(V);     % m/s

    res = -1/2 * C_d * rho * A * v * V;
end
```

The drag coefficient for a baseball is about 0.3. The density of air at sea level is about $1.3\,\mathrm{kg/m^3}$. The cross-sectional area of a baseball is $0.0042\,\mathrm{m^2}$.

Now we have to update `acceleration` to take drag into account:

```
function res = acceleration(t, P, V)
    g = 9.8;                        % acceleration due to gravity in m/s^2
    a_gravity = [0; -g];
```

```
    m = 0.145;                      % mass in kilograms
    a_drag = drag_force(V) / m;
    res = a_gravity + a_drag;
end
```

As in Listing 12.1, `acceleration` represents acceleration due to gravity with a vector that has magnitude `g` and direction along the negative y-axis. But now it also computes drag force and divides by the mass of the baseball to get acceleration due to drag. Finally, it adds `a_gravity` and `a_drag` to get the total acceleration of the baseball.

Figure 12.3 shows the following quantities graphically: (1) acceleration due to drag, **D**, which is in the opposite direction to (2) velocity, **V**; (3) acceleration due to gravity, **G**, which is straight down; and (4) total acceleration, **A**, which is the sum of **D** and **G**.

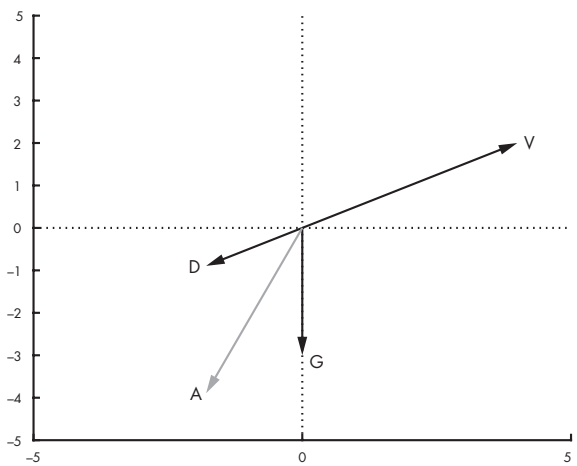

Figure 12.3: Diagram of velocity, **V**; acceleration due to drag force, **D**; acceleration due to gravity, **G**; and total acceleration, **A**

Figure 12.4 shows the results from `ode45`. The ball lands after about 5 s, having traveled less than 150 m, substantially less than what we got without air resistance, about 250 m.

This result suggests that ignoring air resistance is not a good choice for modeling a baseball.

12.5 What Could Go Wrong?

What could go wrong? Well, `vertcat` for one. To explain what that means, I'll start with *concatenation*, which is the operation of joining two matrices into a larger matrix. *Vertical con-*

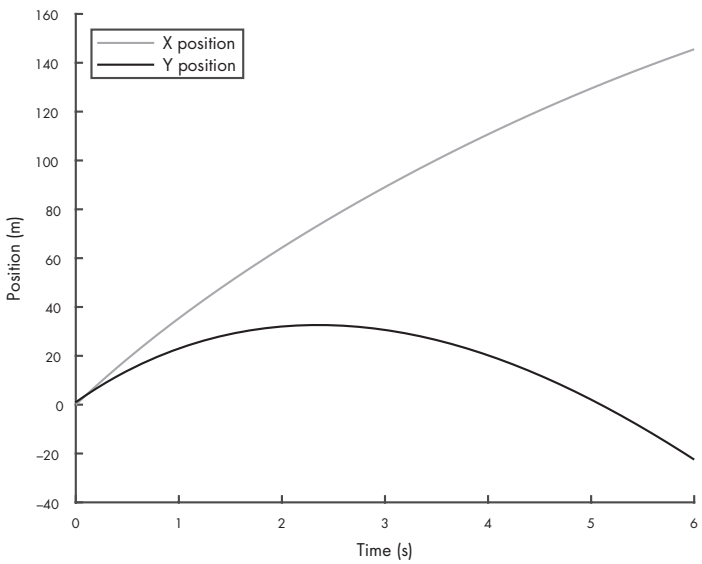

Figure 12.4: Simulated flight of a baseball including drag force

catenation joins the matrices by stacking them on top of each other; *horizontal concatenation* lays them side by side.

Here's an example of horizontal concatenation with row vectors:

```
>> x = 1:3
x = 1     2     3

>> y = 4:5
y = 4     5

>> z = [x, y]
z = 1     2     3     4     5
```

Inside brackets, the comma operator performs horizontal concatenation. The vertical concatenation operator is the semicolon. Here's an example with matrices:

```
>> X = zeros(2, 3)

X =  0     0     0
     0     0     0

>> Y = ones(2, 3)
```

```
Y =  1     1     1
     1     1     1

>> Z = [X; Y]

Z =  0     0     0
     0     0     0
     1     1     1
     1     1     1
```

These operations only work if the matrices are the same size along the dimension where they are glued together. If not, you get

```
>> a = 1:3

a = 1     2     3

>> b = a'

b =  1
     2
     3

>> c = [a, b]
Error using horzcat
Dimensions of matrices being concatenated are not consistent.

>> c = [a; b]
Error using vertcat
Dimensions of matrices being concatenated are not consistent.
```

In this example, `a` is a row vector and `b` is a column vector, so they can't be concatenated in either direction.

Reading the error messages, you might guess that `horzcat` is the function that performs horizontal concatenation, and likewise with `vertcat` and vertical concatenation. You would be correct.

In Listing 12.1 we used vertical concatenation to pack `dPdt` and `dVdt` into the output variable:

```
function res = rate_func(t, W)
    P = W(1:2);
```

```
    V = W(3:4);

    dPdt = V;
    dVdt = acceleration(t, P, V);

    res = [dPdt; dVdt];
end
```

As long as `dPdt` and `dVdt` are column vectors, the semicolon performs vertical concatenation, and the result is a column vector with four elements. But if either of them is a row vector, that's trouble.

The `ode45` function expects the result from `rate_func` to be a column vector, so if you are working with `ode45`, it's probably a good idea to make everything a column vector.

In general, if you run into problems with `horzcat` and `vertcat`, use `size` to display the dimensions of the operands, and make sure you are clear on which way your vectors go.

12.6 Chapter Review

In this chapter, we simulated the flight of a baseball with and without air resistance and saw that the difference is substantial. We can conclude that it's important to model air resistance if we want to make accurate predictions about baseballs and similar projectiles.

Here are some terms from this chapter you might want to remember.

A *spatial vector* is a value that represents a multidimensional physical quantity like position, velocity, acceleration, or force. A spatial vector has a direction and a magnitude. The magnitude is also called the *norm* of the vector. A *unit vector* is a vector with norm 1, which is often used to represent a direction.

Concatenation is the operation of joining two vectors or matrices end-to-end to form a new vector or matrix.

In the next chapter, we'll continue with the baseball example, using `fzero`, which we saw in Chapter 7, and a new tool for optimization, called `fminsearch`. We'll also see a simple way to animate the solution of a differential equation.

12.7 Exercises

Before you go on, you might want to work on the following exercises.

Exercise 12.1. When the Boston Red Sox won the World Series in 2007, they played the Colorado Rockies at their home field in Denver, Colorado. Find an estimate of the density of air in the Mile High City. What effect does this have on drag? What effect does it have on the distance the baseball travels?

Exercise 12.2. The actual drag on a baseball is more complicated than what is captured by our simple model. In particular, the drag coefficient depends on velocity. You can get some of the details from Robert K. Adair's *The Physics of Baseball* (Harper Perennial, 2002); the figure you need is reproduced at https://greenteapress.com/matlab/drag.

Use this data to specify a more realistic model of drag and modify your program to implement it. How big is the effect on the distance the baseball travels?

Exercise 12.3. According to Wikipedia, the record distance for a human cannonball is 59.05 m (see https://greenteapress.com/matlab/cannon).

Modify the example from this chapter to simulate the flight of a human cannonball. You might have to do some research to find the drag coefficient and cross-sectional area for a flying human.

Find the initial velocity (both magnitude and direction) you would need to break this record. You might have to experiment to find the optimal launch angle.

How much acceleration can a human withstand without injury? At this maximum acceleration, how long would the barrel of the cannon have to be to reach the initial velocity needed to break the record?

Chapter 13

Optimization

In in the previous chapter you were asked to find the best launch angle for a human cannonball, meaning the angle that maximizes the distance traveled before landing. This kind of problem, finding minimums and maximums, is called *optimization*.

In this chapter, we'll solve a similar problem, finding the best launch angle for a baseball. We'll solve the problem two ways, first running simulations with a range of values and plotting the results, then using a MATLAB function that automates the process, `fminsearch`.

13.1 Optimal Baseball

In the previous chapter we wrote functions to simulate the flight of a baseball with a known initial velocity. Now we'll use that code to find the launch angle that maximizes *range*, that is, the distance the ball travels before landing.

First, we need an event function to stop the simulation when the ball lands.

```
function [value, isterminal, direction] = event_func(t, W)
    value = W(2);
    isterminal = 1;
    direction = -1;
end
```

This is similar to the event function we saw in Chapter 11.3, except that it uses `W(2)` as the event value, which is the y-coordinate. This event function stops the simulation when the altitude of the ball is 0 and falling.

Now we can call ode45 like this:

```
P = [0; 1];        % initial position in m
V = [40; 30];      % initial velocity in m/s
W = [P; V];        % initial condition

tspan = [0 10];
options = odeset('Events', @event_func);
[T, M] = ode45(@rate_func, tspan, W, options);
```

The initial position of the ball is 1 m above home plate. The ball's initial velocity is 40 m/s in the x-direction and 30 m/s in the y-direction.

The maximum duration of the simulation is 10 s, but we expect an event to stop the simulation first. We can get the final values of the simulation like this:

```
T(end)
M(end, :)
```

The final time is 5.1 s. The final x-position is 131 m; the final y-position is 0, as expected.

13.2 Trajectory

Now we can extract the x- and y-positions:

```
X = M(:, 1);
Y = M(:, 2);
```

In Chapter 12.3 we plotted X and Y separately as functions of time. As an alternative, we can plot them against each other, like this:

```
plot(X, Y)
```

Figure 13.1 shows the result, which is the *trajectory* of the baseball from launch, on the left, to landing, on the right.

13.3 Range Versus Angle

Now we'll simulate the trajectory of the baseball with a range of launch angles. First, we'll take the code we have and wrap it in a function that takes the launch angle as an input variable, runs the simulation, and returns the distance the ball travels (Listing 13.1).

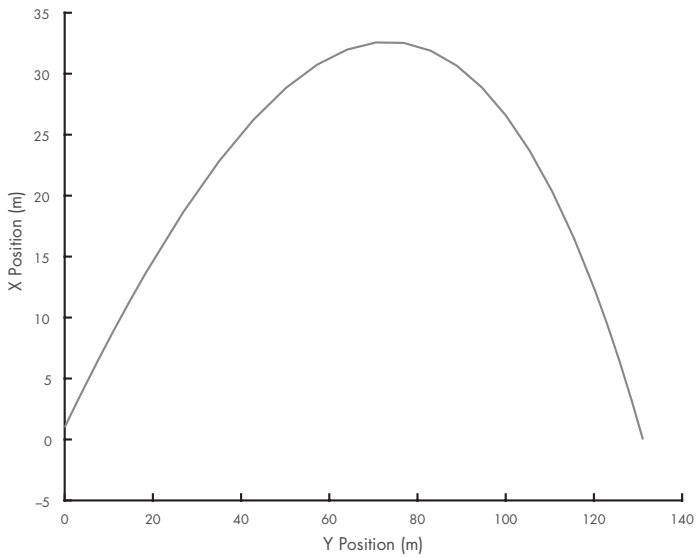

Figure 13.1: Simulated flight of a baseball plotted as a trajectory

Listing 13.1: A function that takes the launch angle of a baseball and returns the distance it travels

```
function res = baseball_range(theta)
    P = [0; 1];
    v = 50;
    [vx, vy] = pol2cart(theta, v);

    V = [vx; vy];      % initial velocity in m/s
    W = [P; V];        % initial condition

    tspan = [0 10];
    options = odeset('Events', @event_func);
    [T, M] = ode45(@rate_func, tspan, W, options);

    res = M(end, 1);
end
```

The launch angle, `theta`, is in radians. The magnitude of velocity, `v`, is always 50 m/s. We use `pol2cart` to convert the angle and magnitude (*polar* coordinates) to Cartesian components, `vx` and `vy`.

After running the simulation we extract the final x-position and return it as an output variable.

We can run this function for a range of angles like this:

```
    thetas = linspace(0, pi/2);
    for i = 1:length(thetas)
        ranges(i) = baseball_range(thetas(i));
    end
```

And then plot `ranges` as a function of `thetas`:

```
    plot(thetas, ranges)
```

Figure 13.2 shows the result. As expected, the ball does not travel far if it's hit nearly horizontal or vertical. The peak is apparently near 0.7 rad.

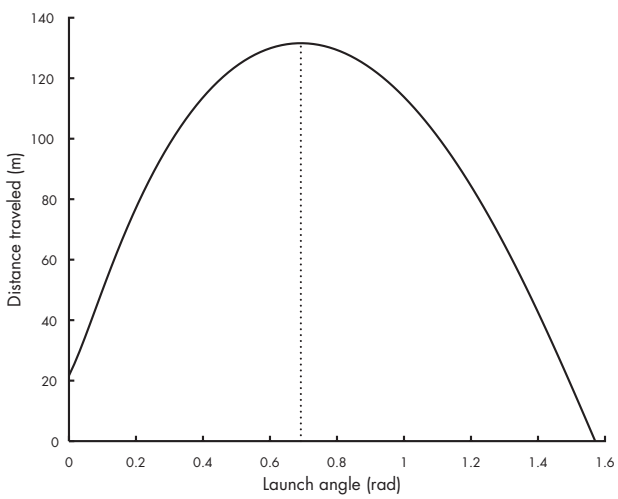

Figure 13.2: Simulated flight of a baseball plotted as a trajectory

Considering that our model is only approximate, this result might be good enough. But if we want to find the peak more precisely, we can use `fminsearch`.

13.4 fminsearch

The `fminsearch` function is similar to `fzero`, which we saw in Chapter 7. Recall that `fzero` takes a function handle and an initial guess, and it returns a root of the function. As an example, to find a root of the function

```
function res = error_func(x)
    res = x^2 - 2;
end
```

we can call `fzero` like this:

```
>> x = fzero(@error_func, 1)
ans = 1.4142
```

The result is near the square root of 2. Let's call `fminsearch` with the same function:

```
>> x = fminsearch(@error_func, 1)
x = -8.8818e-16
```

The result is close to 0, which is where this function is minimized. Optionally, `fminsearch` returns two values:

```
>> [x, fval] = fminsearch(@error_func, 1)
x = -8.8818e-16

fval = -2
```

The `x` is the location of the minimum, and `fval` is the value of the function evaluated at `x`.

If we want to find the maximum of a function, rather than the minimum, we can still use `fminsearch` by writing a short function that negates the function we want to maximize. In the baseball example, the function we want to maximize is `baseball_range`; we can wrap it in another function like this:

```
function res = min_func(angle)
    res = -baseball_range(angle);
end
```

And then we call `fminsearch` like this:

```
>> [x, fval] = fminsearch(@min_func, pi/4)

x = 0.6921

fval = -131.5851
```

The optimal launch angle for the baseball is 0.69 rad; launched at that angle, the ball travels almost 132 m.

If you're curious about how `fminsearch` works, see "How fminsearch Works" on page 152.

13.5 Animation

Animation is a useful tool for checking the results of a physical model. If something is wrong, animation can make it obvious. There are two ways to do animation in MATLAB. One is to use `getframe` to capture a series of images and then use `movie` to play them back.

The more informal way is to draw a series of plots. Listing 13.2 is a function that animates the results of a baseball simulation:

Listing 13.2: A function that animates the results of a baseball simulation

```
function animate(T,M)
    X = M(:,1);
    Y = M(:,2);

    minmax = [min([X]), max([X]), min([Y]), max([Y])];

    for i=1:length(T)
        clf; hold on
        axis(minmax)
        plot(X(i), Y(i), 'o')
        drawnow;

        if i < length(T)
            dt = T(i+1) - T(i);
            pause(dt);
        end
    end
end
```

The input variables are the output from `ode45`: `T`, which contains the time values, and `M`, which contains the position and velocity of the baseball.

A vector of four elements, `minmax` is used inside the loop to set the axes of the figure. This is necessary because otherwise MATLAB scales the figure each time through the loop, so the axes keep changing, which makes the animation hard to watch.

Each time through the loop, `animate` uses `clf` to clear the figure and `axis` to reset the axes. Then it plots a circle to represent the position of the baseball.

We have to call `drawnow` after `plot` so that MATLAB actually displays each plot. Otherwise it waits until you finish drawing all the figures and then updates the display.

We can call `animate` like this:

```
tspan = [0 10];
W = [0 1 30 40];
[T, M] = ode45(@rate_func, tspan, W);
animate(T, M)
```

One limitation of this kind of animation is that the speed of the animation depends on how fast your computer can generate the plots. Since the results from `ode45` are usually not equally spaced in time, your animation might slow down where `ode45` takes small time steps and speed up where the time steps are larger.

One way to fix this problem is to change the way we specify `tspan`. Here's an example:

```
tspan = 0:0.1:10;
```

The result is a vector that goes from 0 to 10 with a step size of 0.1. Passing `tspan` to `ode45` in this form doesn't affect the accuracy of the results; `ode45` still uses variable time steps to generate the estimates, but then it interpolates them before returning the results.

With equal time steps, the animation should be smoother.

Another option is to use `pause` to play the animation in real time. After drawing each frame and calling `drawnow`, you can compute the time until the next frame and use `pause` to wait:

```
dt = T(i+1) - T(i);
pause(dt);
```

A limitation of this method is that it ignores the time required to draw the figure, so it tends to run slow, especially if the figure is complex or the time step is small.

13.6 Chapter Review

This chapter presented two new tools, `fminsearch` and `animate`. The MATLAB function `fminsearch` searches efficiently for the minimum of a function and can be adapted to search for the maximum, too. The `animate` function is one I wrote to read results from `ode45` and generate an animation; the version in this chapter works with the results from the baseball simulation, but it can be adapted for other simulations.

In the exercises below, you have a chance to extend the example from this chapter and bring together many of the tools we have used so far.

In the next chapter, we move on to a new example, celestial mechanics, which describes the motion of planets and other bodies in outer space.

13.7 Exercises

Before you go on, you might want to work on the following exercises.

Exercise 13.1. Manny Ramirez is a former member of the Boston Red Sox who was famous for his relaxed attitude. The goal of this exercise is to solve the following Manny-inspired problem:

> What is the minimum effort required to hit a home run in Fenway Park?

Fenway Park is a baseball stadium in Boston, Massachusetts. One of its most famous features is the "Green Monster," which is a wall in left field that is unusually close to home plate, only 310 feet away. To compensate for the short distance, the wall is unusually high, at 37 feet.

You can solve this problem in two steps:

1. For a given velocity, find the launch angle that maximizes the height of the ball when it reaches the wall. Notice that this is not quite the same as the angle that maximizes the distance the ball travels.
2. Find the minimal velocity that clears the wall, given that it has the optimal launch angle. Hint: this is actually a root-finding problem, not an optimization problem.

Exercise 13.2. A golf ball hit with backspin generates lift, which might increase the distance it travels, but the energy that goes into generating spin probably comes at the cost of lower initial velocity.

Write a simulation of the flight of a golf ball and use it to find the launch angle and allocation of spin and initial velocity (for a fixed energy budget) that maximizes the horizontal range of the ball in the air.

The lift of a spinning ball is due to the Magnus force (see https://greenteapress.com/matlab/magnus), which is perpendicular to the axis of spin and the path of flight. The coefficient of lift is proportional to the spin rate; for a ball spinning at 3000 rpm it is about 0.1. The coefficient of drag of a golf ball is about 0.2 as long as the ball is moving faster than 20 m/s.

Chapter 14

Springs and Things

The computational tools you have learned so far make up a versatile toolkit for modeling physical systems described by first- and second-order differential equations and systems of equations.

With `ode45` you can compute the state variables of these systems as they change over time. By varying the parameters of the model, you can see what effect they have on the results. Then you can use `fminsearch` and `fzero` to find minimums, maximums, and places where the outputs pass through zero.

These tools are all you need to solve a lot of problems, so this chapter doesn't present new computational tools (whew!). Instead, we'll look at some different physical systems and some forces we haven't dealt with yet, including spring forces and universal gravitation.

The examples in this chapter are a little more open-ended than the previous ones. I will present a motivating problem and some background information, and you will have a chance to implement the models as exercises.

14.1 Bungee Jumping

Suppose you want to set the world record for the highest "bungee dunk," which is a stunt in which a bungee jumper dunks a cookie in a cup of tea at the lowest point of a jump. An example is shown in this video: https://greenteapress.com/matlab/dunk.

Since the record is 70 m, let's design a jump for 80 m. We'll start with the following parameters:

- Initially, the jumper stands on a platform 80 m above a cup of tea. One end of the bungee cord is connected to the platform, the other end is attached to the jumper, and the middle hangs down.
- The mass of the jumper is 75 kg, and they are subject to gravitational acceleration of $9.8\,\mathrm{m/s^2}$.
- In free fall the jumper has a cross-sectional area of 1 m and a terminal velocity of 60 m/s.

To model the force of the bungee cord on the jumper, I'll make the following assumptions:

- Until the cord is fully extended, it applies no force to the jumper. It turns out this might not be a good assumption; we'll revisit it in the next section.
- After the cord is fully extended, it obeys Hooke's Law; that is, it applies a force to the jumper proportional to the extension of the cord beyond its resting length.

We can write Hooke's Law as $F_\mathrm{s} = -kx$ where F_s is the force of the spring (bungee cord) on the jumper in newtons, x is the distance the spring is stretched in meters, and k is a spring constant that represents the strength of the spring in newtons per meter. The minus sign indicates that the direction of the spring force is opposite to the direction the spring is stretched.

Hooke's Law is not a law in the sense that it is always true; really, it is a model of how some things behave under some conditions. Almost everything obeys Hooke's Law when x is small enough, but for large values everything deviates from this ideal behavior, one way or the other.

In reality, the spring constant of a bungee cord depends on x over the range we are interested in, but as a starting place I'll assume k is constant.

Exercise 14.1. Write a simulation of this scenario, based on these parameters and modeling assumptions. Use your simulation to choose the length of the cord, `L`, and its spring constant, `k`, so that the jumper falls all the way to the tea cup, but no farther!

You could start with the length 25 m and the spring constant 40 N/m.

14.2 Bungee Revisited

In the previous section, we modeled the motion of a bungee jumper taking into account gravity, air resistance, and the spring force of the bungee cord. But we ignored the weight of the cord.

It's tempting to say that the cord has no effect because it falls along with the jumper, but that intuition is incorrect. As the cord falls, it transfers energy to the jumper.

At https://greenteapress.com/matlab/bungee, you'll find a paper by Heck, Uylings, and Kedzierska titled "Understanding the physics of bungee jumping"; it explains this phenomenon and derives the acceleration of the jumper, a, as a function of position, y, and velocity, v:

$$a = g + \frac{\mu v^2/2}{\mu(L+y) + 2L}$$

where g is acceleration due to gravity, L is the length of the cord, and μ is the ratio of the mass of the cord, m, to the mass of the jumper, M.

If you don't believe that their model is correct, this video might convince you: https://greenteapress.com/matlab/chain.

Exercise 14.2. Modify your solution to the previous problem to model this effect. How does the behavior of the system change as we vary the mass of the cord? When the mass of the cord equals the mass of the jumper, what is the net effect on the lowest point in the jump?

14.3 Spider-Man

In this example we'll develop a model of Spider-Man swinging from a springy cable of webbing attached to the top of the Empire State Building. Initially, Spider-Man is at the top of a nearby building, as shown in Figure 14.1.

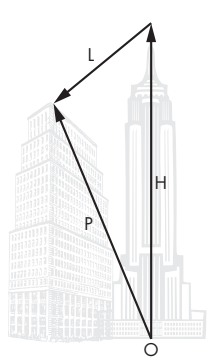

Figure 14.1: Diagram of the initial state for the Spider-Man example

The origin, `O`, is at the base of the Empire State Building. The vector `H` represents the position where the webbing is attached to the building, relative to `O`. The vector `P` is the position of Spider-Man relative to `O`. And `L` is the vector from the attachment point to Spider-Man.

By following the arrows from `O`, along `H`, and along `L`, we can see that

```
H + L = P
```

So we can compute `L` like this:

```
L = P - H
```

Exercise 14.3. As an exercise, simulate this system and estimate the parameters that maximize the distance Spider-Man swings.

1. Implement a model of this scenario to predict Spider-Man's trajectory.
2. Choose the right time for Spider-Man to let go of the webbing in order to maximize the distance he travels before landing.
3. Choose the best angle for Spider-Man to jump off the building, and the best time to let go of the webbing, to maximize range.

Use the following parameters:

- According to the Spider-Man Wiki (https://greenteapress.com/matlab/spider), Spider-Man weighs 76 kg.
- Assume his terminal velocity is 60 m/s.
- The length of the web is 100 m.
- The initial angle of the web is 45° to the left of straight down.
- The spring constant of the web is 40 N/m when the cord is stretched and 0 N/m when it's compressed.

14.4 Celestial Mechanics

Celestial mechanics describes how objects move in outer space. If you did Section 11.2, you simulated the Earth being pulled toward the Sun in one dimension. Now we'll simulate the Earth orbiting the Sun in two dimensions.

To keep things simple, we'll consider only the effect of the Sun on the Earth and ignore the effect of the Earth on the Sun. So we'll place the Sun at the origin and use a spatial vector, **P**, to represent the position of the Earth relative to the Sun.

Given the mass of the Sun, m_1, and the mass of the Earth, m_2, the gravitational force between them is

$$\mathbf{F}_{\mathrm{g}} = -G\frac{m_1 m_2}{r^2}\hat{\mathbf{P}}$$

where G is the universal gravitational constant (see https://greenteapress.com/matlab/gravity), r is the distance of the Earth from the Sun, and $\hat{\mathbf{P}}$ is a unit vector in the direction of $\mathbf{P}$.

Exercise 14.4. Write a simulation of the Earth orbiting the Sun. You can look up the orbital velocity of the Earth or manually search for the initial velocity that causes the Earth to make one complete orbit in one year. Optionally, use `fminsearch` to find the velocity that gets the Earth as close as possible to the starting place after one year.

14.5 Conservation of Energy

A useful way to check the accuracy of an ODE solver is to see whether it conserves energy. For planetary motion, it turns out that `ode45` does not.

The kinetic energy of a moving body is

$$KE = mv^2/2$$

The potential energy of a sun with mass m_1 and a planet with mass m_2 and a distance r between them is

$$PE = -G\frac{m_1 m_2}{r}$$

Exercise 14.5. Write a function called `energy_func` that takes the output of your Earth simulation and computes the total energy (kinetic and potential) of the system for each estimated position and velocity.

Plot the result as a function of time and check whether it increases or decreases over the course of the simulation.

You can reduce the rate of energy loss by decreasing `ode45`'s tolerance option using `odeset` (see "ODE Events" on page 116):

```
options = odeset('RelTol', 1e-5);
[T, M] = ode45(@rate_func, tspan, W, options);
```

The name of the option is `RelTol` for "relative tolerance." The default value is `1e-3`, or 0.001. Smaller values make `ode45` less "tolerant," so uses smaller step sizes to make the errors smaller.

Run `ode45` with a range of values for `RelTol` and confirm that as the tolerance gets smaller, the rate of energy loss decreases.

Along with `ode45`, MATLAB provides several other ODE solvers (see https://greenteapress.com/matlab/solver). Run your simulation with one of the other ODE solvers MATLAB provides and see if any of them conserve energy. You might find that `ode23` works surprisingly well (although technically it does not conserve energy either).

14.6 Chapter Review

This chapter presents examples where you can apply the tools in this book to solve more realistic problems. Some of them are more serious than others, but I hope you had some fun with them.

I think this toolkit is powerful and versatile. If you use it to solve an interesting problem, let me know!

Chapter 15

Under the Hood

In this chapter we "open the hood," looking more closely at how some of the tools we have used—ode45, fzero, and fminsearch—work.

15.1 How ode45 Works

According to the MATLAB documentation, ode45 uses "an explicit Runge-Kutta formula, the Dormand-Prince pair." You can read about it at https://greenteapress.com/matlab/runge, but I'll give you a sense of it here.

The key idea behind all Runge-Kutta methods is to evaluate the rate function several times at each time step and use a weighted average of the computed slopes to estimate the value at the next time step. Different methods evaluate the rate function in different places and compute the average with different weights.

As an example, we'll solve the following differential equation:

$$\frac{dy}{dt}(t) = y \sin t$$

Given a differential equation, it's usually straightforward to write a rate function:

```
function res = rate_func(t, y)
    dydt = y * sin(t);
    res = dydt;
end
```

And we can use it like this:

```
    y0 = 1;
    tspan=[0 4];
    options = odeset('Refine', 1);
    [T, Y] = ode45(@rate_func, tspan, y0, options);
```

For this example we'll use `odeset` to set the `Refine` option to `1`, which tells `ode45` to return only the time steps it computes, rather than interpolating between them.

Now we can modify the rate function to plot the places where it gets evaluated:

```
function res = rate_func(t, y)
    dydt = y * sin(t);
    res = dydt;

    plot(t, y, 'ro')
    dt = 0.01;
    ts = [t t+dt];
    ys = [y y+dydt*dt];
    plot(ts, ys, 'r-')
end
```

When `rate_func` runs, it plots a red circle at each location and a short red line showing the computed slope.

Figure 15.1 shows the result; `ode45` computes 10 time steps (not counting the initial condition) and evaluates the rate function 61 times.

Figure 15.2 shows the same plot, zoomed in on a single time step. The dark squares at 0.8 to 1.2 show the values that were returned as part of the solution. The circles show the places where the rate function was evaluated.

We can see that `ode45` evaluates the rate function several times per time step, at several places between the end points. We can also see that most of the places where `ode45` evaluates the rate function are not part of the solution it returns, and they are not always good estimates of the solution. This is good to know when you are writing a rate function; you should not assume that the time and state you get as input variables will be part of the solution.

In a sense, the rate function is answering a hypothetical question: "*If* the state at a particular time has these particular values, what *would* the slope be?"

At each time step, `ode45` actually computes *two* estimates of the next value. By comparing them, it can estimate the magnitude of the error, which it uses to adjust the time step. If the

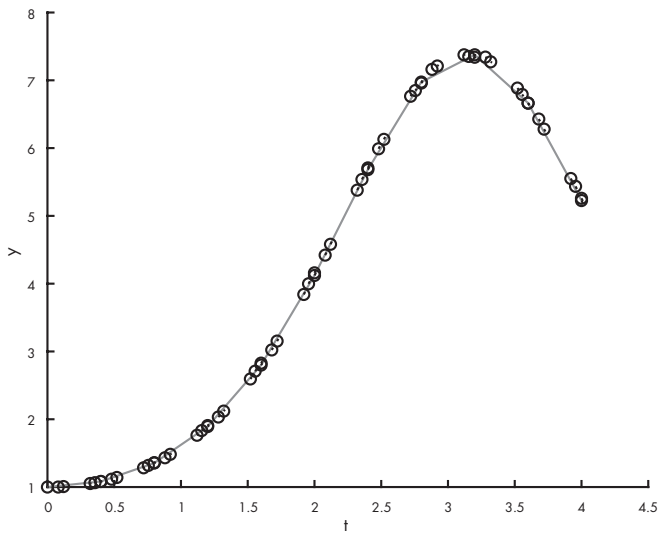

Figure 15.1: Points where `ode45` evaluates the rate function

error is too big, it uses a smaller time step; if the error is small enough, it uses a bigger time step. Because `ode45` is *adaptive* in this way, it minimizes the number of times it calls the rate function to achieve a given level of accuracy.

15.2 How fzero Works

According to the MATLAB documentation, `fzero` uses "a combination of bisection, secant, and inverse quadratic interpolation methods." (See `https://greenteapress.com/matlab/fzero`)

To understand what that means, suppose we're trying to find a root of a function of one variable, $f(x)$, and assume we have evaluated the function at two places, x_1 and x_2, and found that the results have opposite signs. Specifically, assume $f(x_1) > 0$ and $f(x_2) < 0$, as shown in Figure 15.3.

As long as f is continuous, there must be at least one root in this interval. In this case we would say that x_1 and x_2 *bracket* a zero.

If this were all you knew about f, where would you go looking for a root? If you said "halfway between x_1 and x_2," congratulations! You just invented a numerical method called *bisection*!

If you said, "I would connect the dots with a straight line and compute the zero of the line," congratulations! You just invented the *secant method*!

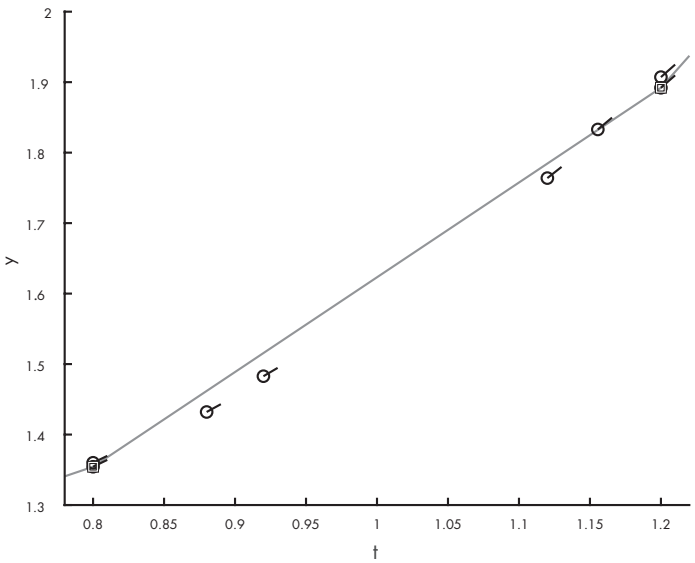

Figure 15.2: Points where `ode45` evaluates the rate function, zoomed in

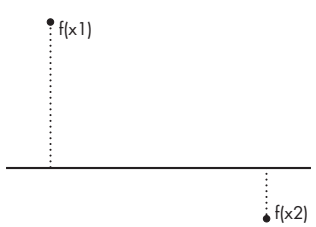

Figure 15.3: Initial state of a root-finding search

And if you said, "I would evaluate f at a third point, find the parabola that passes through all three points, and compute the zeros of the parabola," congratulations, you just invented *inverse quadratic interpolation*!

That's most of how `fzero` works. The details of how these methods are combined are interesting, but beyond the scope of this book. You can read more at `https://greenteapress.com/matlab/brent`.

15.3 How fminsearch Works

According to the MATLAB documentation, `fminsearch` uses the Nelder-Mead simplex algorithm. You can read about it at `https://greenteapress.com/matlab/nelder`, but you might

find it overwhelming.

To give you a sense of how it works, I will present a simpler algorithm, the *golden-section search*. Suppose we're trying to find the minimum of a function of a single variable, $f(x)$.

As a starting place, assume that we have evaluated the function at three places, x_1, x_2, and x_3, and found that x_2 yields the lowest value. Figure 15.4 shows this initial state.

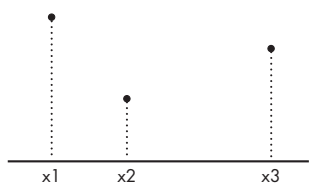

Figure 15.4: Initial state of a golden-section search

We will assume that $f(x)$ is continuous and *unimodal* in this range, which means that there is exactly one minimum between x_1 and x_3.

The next step is to choose a fourth point, x_4, and evaluate $f(x_4)$. There are two possible outcomes, depending on whether $f(x_4)$ is greater than $f(x_2)$ or not. Figure 15.5 shows the two possible states.

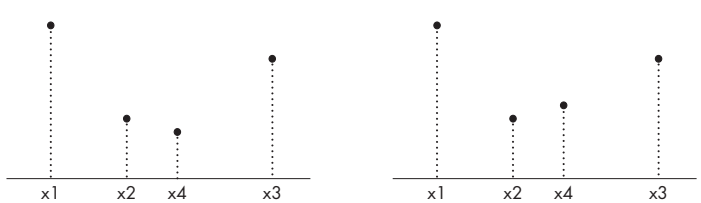

Figure 15.5: Possible states of a golden-section search after evaluating $f(x_4)$

If $f(x_4)$ is less than $f(x_2)$ (shown on the left), the minimum must be between x_2 and x_3, so we would discard x_1 and proceed with the new triple (x_2, x_4, x_3).

If $f(x_4)$ is greater than $f(x_2)$ (shown on the right), the local minimum must be between x_1 and x_4, so we would discard x_3 and proceed with the new triple (x_1, x_2, x_4).

Either way, the range gets smaller and our estimate of the optimal value of x gets better.

This method works for almost any value of x_4, but some choices are better than others. You might be tempted to bisect the interval between x_2 and x_3, but that turns out not to be the best choice. You can read about a better option at `https://greenteapress.com/matlab/golden`.

15.4 Chapter Review

The information in this chapter is not strictly necessary; you can use these methods without knowing much about how they work. But there are two reasons you might want to know.

One reason is pure curiosity. If you use these methods, and especially if you come to rely on them, you might find it unsatisfying to treat them as "black boxes." At the risk of mixing metaphors, I hope you enjoyed opening the hood.

The other reason is that these methods are not infallible; sometimes things go wrong. If you know how they work, at least in a general sense, you might find it easier to debug them.

With that, you have reached the end of the book, so congratulations! I hope you enjoyed it and learned a lot. I think the tools in this book are useful, and the ways of thinking are important, not just in engineering and science, but in practically every field of inquiry.

Models are the tools we use to understand the world: if you build good models, you are more likely to get things right. Good luck!

Appendix A

Glossary

Absolute error The difference between an approximation and an exact answer.

Abstraction The process of ignoring the details of how a function works in order to focus on a simpler model of what the function does.

Accumulator A variable that is used to accumulate a result a little bit at a time.

Analytic solution A way of solving an equation by performing algebraic operations and deriving an explicit way to compute a solution.

Apply A way of processing a vector by performing some operation on each of the elements, producing a vector that contains the results.

Argument An expression that appears in a function call to specify the value the function operates on.

Assignment statement A command that creates a new variable (if necessary) and gives it a value.

Body The statements inside a loop that are run repeatedly.

Call (a function) To cause a function to execute and compute a result.

Column vector A matrix that has only one column.

Command A line of MATLAB code executed by the interpreter.

Comment Part of a program that provides additional information about the program but does not affect its execution.

Compound statement A statement, like `if` and `for`, that contains other statements in an indented body.

Concatenation The operation of joining two vectors or matrices end-to-end to form a new vector or matrix.

Differential equation (DE) An equation that relates the derivatives of an unknown function.

Directly (compute) A way of computing an element in a sequence without using previous elements.

Element (of a matrix) One of the values in a vector or matrix.

Element (of a sequence) One of the numbers in a mathematical sequence.

Elementwise An operation that acts on the elements of a vector or matrix (unlike some linear algebra operations).

Encapsulation The process of wrapping part of a program in a function in order to limit interactions (including name collisions) between the function and the rest of the program.

Evaluate To compute the value of an expression.

Existential quantification A logical condition that expresses the idea that "there exists" an element of a set with a certain property.

Expression A sequence of operands and operators that specifies a mathematical computation and yields a value.

First-order DE A differential equation that includes only first derivatives.

Floating-point A way to represent numbers in a computer.

Function A named computation; for example, `log10` is the name of a function that computes logarithms in base 10.

Function call A command that executes a function.

Function handle A function handle is a way of referring to a function by name (and passing it as an argument) in MATLAB without calling it.

Generalization Making a function more versatile by replacing specific values with input variables.

Helper function A function in an M-file that is not the top-level function; it can only be called from another function in the same file.

Incremental development A way of programming by making a series of small, testable changes.

Index An integer value used to indicate one of the values in a vector or matrix (also called subscript in some MATLAB documentation).

Input variable A variable in a function that gets its value, when the function is called, from one of the arguments.

Interpreter The program that reads and executes MATLAB code.

Linear DE A differential equation that includes no products or powers of the function and its derivatives.

Logical function A function that returns a logical value (1 for "true" or 0 for "false").

Logical vector A vector, often the result of applying a logical operator to a vector, that contains logical values 1 and 0.

Loop A part of a program that runs repeatedly.

Loop variable A variable, defined in a loop, that gets assigned a different value each time through the loop.

M-file A file that contains a MATLAB program.

Matrix A two-dimensional collection of values (also called "array" in some MATLAB documentation).

Name collision The scenario where two scripts that use the same variable name interfere with each other.

Nested function call An expression that uses the result from one function call as an argument for another.

Nesting Putting one compound statement in the body of another.

Norm The magnitude of a vector, sometimes called "length," but not to be confused with the number of elements in a MATLAB vector.

Numerical method A method (or algorithm) for generating a numerical solution.

Numerical solution A way of solving an equation by finding a numerical value that satisfies the equation, often approximately.

Operand A number or variable that appears in an expression along with operators.

Operator One of the symbols, like `*` and `+`, that represent mathematical operations.

Order of operations The rules that specify which operations in an expression are performed first.

Ordinary DE (ODE) A differential equation in which all derivatives are taken with respect to the same variable.

Output variable A variable in a function that is used to return a value from the function to the caller.

Pack To copy values from a set of variables into a vector.

Parameter A value that appears in a model to quantify some physical aspect of the scenario being modeled.

Partial DE (PDE) A differential equation that includes derivatives with respect to more than one variable.

Phase plot A plot that shows the state of a system as a point in the space of possible states.

Postcondition Something that will be true when the script completes.

Precondition Something that must be true when the script starts, in order for it to work correctly.

Projection The component of one vector that is in the direction of the other.

Prompt The symbols the interpreter prints to indicate that it's waiting for the user to type a command.

Range A matrix of values specified with the colon operator, for example, `1:5`.

Recurrently A way of computing the next element of a sequence based on previous elements.

Reduce A way of processing the elements of a vector and generating a single value, for example, the sum of the elements.

Relative error The difference between an approximation and an exact answer, expressed as a fraction or percentage of the exact answer.

Row vector A matrix that has only one row.

Scaffolding Code you write to help you program or debug, but which is not part of the finished program.

Scalar A single value.

Scientific notation A format for typing and displaying large and small numbers, e.g., `3.0e8`, which represents 3.0×10^8 or 300,000,000.

Script An M-file that contains a sequence of MATLAB commands.

Search A way of processing a vector by examining the elements in order until one is found that has the desired property.

Search path The list of folders where MATLAB looks for M-files.

Sequence A set of numbers that correspond to the positive integers, in mathematics.

Series The sum of the elements in a sequence, in mathematics.

Shadow A kind of name collision in which a new definition causes an existing definition to become invisible. In MATLAB, variable names can shadow built-in functions (with hilarious results).

Side effect An effect, like modifying the workspace, that is not the primary purpose of a function or script.

Signature The first line of a function definition, which specifies the names of the function, the input variables, and the output variables.

Silent function A function that doesn't display anything, generate a figure, or have any other side effects.

Spatial vector A value that represents a multidimensional physical quantity like position, velocity, acceleration, or force.

State If a system can be described by a set of variables, the values of those variables are called the state of the system.

String A value that consists of a sequence of characters.

System of equations A collection of equations written in terms of the same set of variables.

Target The variable on the left side of an assignment statement.

Time step The interval in time between successive estimates in the numerical solution of a differential equation.

Top-level function The first function in an M-file; it's the only function that can be called from the Command Window or from another file.

Trajectory A path in a phase plot that shows how the state of a system changes over time.

Transpose An operation that transforms the rows of a matrix into columns (and the other way around).

Unit vector A vector with norm 1, used to indicate direction.

Universal quantification A logical condition that expresses the idea that all elements of a set have a certain property.

Unpack To copy the elements of a vector into a set of variables.

Value The result of a computation, most often a number, string, or matrix.

Variable A named value.

Vector A sequence of values.

Workspace A set of variables and their values.

Zero (of a function) An argument that makes the value of a function 0.

Index