Physical Modeling in MATLAB®

Allen B. Downey

Version 3.0.0

www.dbooks.org

Physical Modeling in MATLAB®

Copyright 2012, 2019 Allen B. Downey

Green Tea Press
9 Washburn Ave
Needham MA 02492

Permission is granted to copy, distribute, and/or modify this document under the terms of the Creative Commons Attribution-NonCommercial 4.0 Unported License, which is available at `http://creativecommons.org/licenses/by-nc/4.0/`.

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

MATLAB®is a registered trademark of The Mathworks, Inc. The Mathworks does not warrant the accuracy of this book; they probably don't even like it.

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.
- Simulation is an approach to modeling that uses computer programs to implement models and generate predictions. This book shows how simulations are used to run experiments, answer questions, and guide decision-making.

To make this book accessible to the widest possible audience, I try to minimize the "prerequisites".

In particular, 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 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 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.

www.dbooks.org

You will learn to use numerical methods to search for roots of non-linear equations, to solve differential equations, and to search for optimal solutions. You will learn how to use these methods first; then in Chapter 14 you will learn more about how they work. But if you can't stand the suspense, you can look "under the hood" whenever you want.

I have tried to present a small set of tools that provides the most versatility and power, to explain them as clearly as possible, and to give you chances to practice what you learn.

I hope you enjoy the book and find it valuable.

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 9.

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://www.mathworks.com/downloads/web_downloads/`.

To install Octave, we strongly 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://www.anaconda.com/download`.

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

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

Then you can launch it from the command line like this:

```
octave
```

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

0.1 Working with the code

The code for each chapter in this book is in a Zip file you can download from `https://github.com/AllenDowney/ModSimMatlab/raw/master/ModSimMatlabCode.zip`.

Once you have the Zip file, you can unzip it on the command line by running

```
unzip ModSimMatlabCode.zip
```

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

If you open any of these files in MATLAB, you should be able to read the code. To run it, press the green Run button.

You might get a message like, "File not found in the current folder". MATLAB will give you the option to Change Folder or Add to Path. If you change folders, you will be able to run this file until you change folder again. If you add to the path, you will always be able to run this file.

However, as you add more folders to the path, you are more likely to run into problems with name collisions (see Section 4.17). I recommend you change folders when necessary and avoid adding folders to the path.

www.dbooks.org

Contributor's list

If you have suggestions and corrections, please send them to:
`mod_sim_matlab@greenteapress.com`.

People who have found errors and helped us 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, Craig Scratchley.

Matt Wiens revised several sections of the book.

Contents

Preface **iii**

0.1 Working with the code v

1 Modeling and simulation **1**

1.1 Modeling 1

1.2 A glorified calculator 3

1.3 Math functions 4

1.4 Documentation 5

1.5 Variables 6

1.6 Assignment statements 7

1.7 The workspace 8

1.8 Why variables? 8

1.9 Errors 9

1.10 Glossary 11

1.11 Exercises 12

2 Scripts **15**

2.1 M-files 15

2.2 Why scripts? 16

2.3 Fibonacci 17

2.4 Floating-point numbers 18

2.5 Comments 19

www.dbooks.org

2.6 Documentation 20
2.7 Assignment and equality 21
2.8 Glossary 21
2.9 Exercises 22

3 Loops **25**

3.1 Updating variables 25
3.2 Bug taxonomy 26
3.3 Absolute and relative error 27
3.4 for loops 27
3.5 plotting 29
3.6 Sequences 30
3.7 Series 30
3.8 Generalization 31
3.9 Incremental development 32
3.10 Glossary 33
3.11 Exercises 34

4 Vectors **37**

4.1 Checking preconditions 37
4.2 if statements 38
4.3 Relational operators 39
4.4 Logical operators 40
4.5 Vectors 41
4.6 Vector arithmetic 41
4.7 Everything is a matrix 42
4.8 Elementwise operators 43
4.9 Indices 44
4.10 Indexing errors 45
4.11 Vectors and sequences 45

4.12 Plotting vectors . . . 46
4.13 Reduce . . . 46
4.14 Apply . . . 47
4.15 Search . . . 47
4.16 Spoiling the fun . . . 48
4.17 Name Collisions . . . 48
4.18 Glossary . . . 49
4.19 Exercises . . . 50

5 Functions **53**

5.1 Documentation . . . 55
5.2 What could go wrong? . . . 56
5.3 Multiple input variables . . . 57
5.4 Logical functions . . . 58
5.5 Incremental development . . . 59
5.6 Nested loops . . . 60
5.7 Conditions and flags . . . 61
5.8 Encapsulation and generalization . . . 62
5.9 `continue` . . . 64
5.10 Mechanism and leap of faith . . . 65
5.11 Why functions? . . . 65
5.12 Glossary . . . 66
5.13 Exercises . . . 67

6 Zero-finding **69**

6.1 Nonlinear equations . . . 69
6.2 Zero-finding . . . 70
6.3 `fzero` . . . 71
6.4 What could go wrong? . . . 73
6.5 Choosing an initial guess . . . 74

www.dbooks.org

6.6 Vectorizing functions . . . 75
6.7 More name collisions . . . 75
6.8 Debugging your head . . . 76
6.9 Glossary . . . 77
6.10 Exercises . . . 77

7 Functions of Vectors 79

7.1 Functions and files . . . 79
7.2 Vectors as input variables . . . 80
7.3 Vectors as output variables . . . 81
7.4 Vectorizing functions . . . 81
7.5 Sums and differences . . . 83
7.6 Products and ratios . . . 84
7.7 Existential quantification . . . 84
7.8 Universal quantification . . . 85
7.9 Logical vectors . . . 86
7.10 Debugging in four acts . . . 87
7.11 Glossary . . . 88

8 Ordinary Differential Equations 89

8.1 Differential equations . . . 89
8.2 Euler's method . . . 90
8.3 Implementing Euler's method . . . 91
8.4 `ode45` . . . 92
8.5 Time dependence . . . 94
8.6 What could go wrong? . . . 96
8.7 Labeling axes . . . 97
8.8 Glossary . . . 97
8.9 Exercises . . . 98

9 Systems of ODEs **101**

9.1 Matrices . . . 101

9.2 Row and column vectors . . . 102

9.3 The transpose operator . . . 103

9.4 Lotka-Volterra . . . 104

9.5 Output matrices . . . 106

9.6 Phase plot . . . 108

9.7 What could go wrong? . . . 109

9.8 Glossary . . . 109

9.9 Exercises . . . 110

10 Second-order Systems **111**

10.1 Newtonian motion . . . 111

10.2 Freefall . . . 112

10.3 ODE events . . . 114

10.4 Air resistance . . . 115

10.5 Exercises . . . 117

11 Two dimensions **119**

11.1 Spatial vectors . . . 119

11.2 Adding vectors . . . 120

11.3 ODEs in two dimensions . . . 121

11.4 Drag force . . . 124

11.5 What could go wrong? . . . 126

11.6 Glossary . . . 128

11.7 Exercises . . . 128

www.dbooks.org

12 Optimization **131**

12.1 Optimal baseball 131

12.2 Trajectory 132

12.3 Range versus angle 132

12.4 `fminsearch` 134

12.5 Animation 135

12.6 Exercises 137

13 Case studies **139**

13.1 Celestial mechanics 139

13.2 Conservation of Energy 140

13.3 Bungee jumping 140

13.4 Bungee revisited 141

13.5 Spider-Man 142

14 How does it work? **145**

14.1 How `ode45` works 145

14.2 How `fzero` works 146

14.3 How `fminsearch` works 148

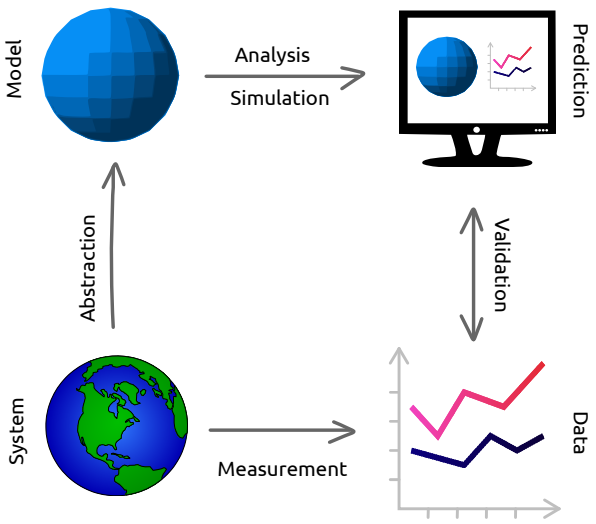
Chapter 1

Modeling and simulation

This chapter presents the modeling process and introduces MATLAB, the programming language we will use to represent models and run simulations.

1.1 Modeling

This book is about modeling and simulation of physical systems. The following diagram shows what I mean by "modeling":



www.dbooks.org

Starting in the lower left, the **system** is something in the real world we are interested in. Often, it is 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**, which 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 **design** intended to achieve a purpose.

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.

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 is likely to be too simple, and test whether it is 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 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 to data from the real world provides **external validation**, which is generally the strongest test.

The focus of this book is simulation, and the primary tool we will use is MATLAB.

1.2 A glorified calculator

At heart, MATLAB is a glorified calculator. When you start MATLAB you will see a window entitled MATLAB that contains smaller windows entitled Current Folder, Command Window, and Workspace. In Octave, Current Folder is called File Browser.

The Command Window runs the **interpreter**, which allows you to type **commands**, then executes them and prints the result.

Initially, the Command Window contains a welcome message with information about the version of the software you are running, followed by a **prompt**:

```
>>
```

This 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.

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

www.dbooks.org

Speaking of spaces, you might have noticed that MATLAB puts a blank line between `ans =` and the result. In my examples I will 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 by an asterisk, `*`; division by a forward slash, `/`.

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

Another common operator is exponentiation, which uses the `^` symbol, sometimes pronounced "carat" 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 also changed the spacing 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 is worth spending time to make your code pretty; it will save you time debugging!

1.3 Math functions

MATLAB knows how to compute pretty much every math function you've heard of. 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`, `tan`—among many others—work in radians.[1]

Some functions take more than one argument, in which case they 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 y and x 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 is not meant to be a reference manual. To learn about other functions, you should read the documentation.

1.4 Documentation

MATLAB comes with two forms of online documentation, `help` and `doc`.

The help command works in the Command Window; just type `help` followed by the name of a command.

[1] MATLAB also provides trig functions that work in degrees. For example, `sind`,`cosd`, and `tand` compute the sine, cosine, and tangent of an angle given in degrees.

www.dbooks.org

```
>> help sin
```

```
 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" will likely not make sense until we get to vectors and matrices a few chapters from now.

The `doc` pages are usually better. If you type `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 arrays, so they may not make complete sense yet, but you can get a preview of what's coming.

1.5 Variables

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

```
>> 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

>> exp(i * pi)
ans = -1.0000 + 0.0000i
```

As the second example shows, many MATLAB functions work with complex numbers. This example demonstrates Euler's Equality:

$$e^{i\pi} = -1$$

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.

1.6 Assignment statements

You can create your own variables, and give them values, with an **assignment statement**. The assignment operator is the equals sign, `=`.

```
>> 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.

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 upper case 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.

```
>> fibonacci0 = 1;

>> LENGTH = 10;

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

The first two examples demonstrate the use of the semi-colon, 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**.

www.dbooks.org

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

1.7 The workspace

When you create a new variable, it appears in the Workspace window, and it 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
```

1.8 Why variables?

Some reasons to use variables are:

- To avoid recomputing a value that is used repeatedly. For example, if your computation uses e frequently, you might want to compute it once and save the result[2].

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

- To make the connection between the code and the underlying mathematics more apparent. If you are 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 $a = \pi r^2$.

- To break a long computation into a sequence of steps. Suppose you are evaluating a big, hairy expression like this:

```
ans = ((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 type `...` at the end of the first line and continue on the next.

But often it is 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
ans = 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 them easier to debug.

1.9 Errors

It's early, but now would be a good time to start making errors. Whenever you learn a new feature, you should try to make as many errors as possible, as soon as possible.

[2] You don't have to do this in Octave; it is predefined.

www.dbooks.org

When you make deliberate errors, you see what the error messages are. Later, when you make accidental errors, you will know what the messages mean.

A common error for beginning programmers is leaving out the * for multiplication.

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

```
 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 error message indicates that the expression in invalid and suggests several things that might be wrong. In this case, one of its guesses is right; we are 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 is common to write something like $\sin \pi$, but not in MATLAB.

```
>> sin pi
```

```
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 (rather than as an expression). In this case the error message is helpful, but 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.

This example also demonstrates 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 is worth the effort to understand them.

Here's another common error. If you were translating this mathematical expression into MATLAB:

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

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)
```

1.10 Glossary

interpreter: The program that reads and executes MATLAB code.

command: A line of MATLAB code executed by the interpreter.

prompt: The symbols the interpreter prints to indicate that it is waiting for you to type a command.

operator: One of the symbols, like `*` and `+`, that represent mathematical operations.

operand: A number or variable that appears in an expression along with operators.

expression: A sequence of operands and operators that specifies a mathematical computation and yields a value.

value: The numerical result of a computation.

evaluate: To compute the value of an expression.

order of operations: The rules that specify which operations in an expression are performed first.

function: A named computation; for example `log10` is the name of a function that computes logarithms in base 10.

call: To cause a function to execute and compute a result.

function call: A kind of command that executes a function.

argument: An expression that appears in a function call to specify the value the function operates on.

www.dbooks.org

nested function call: An expression that uses the result from one function call as an argument for another.

variable: A named value.

assignment statement: A command that creates a new variable (if necessary) and gives it a value.

string: A value that consists of a sequence of characters (as opposed to a number).

1.11 Exercises

Exercise 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 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 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 m/s, much less than 86 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 this sidewalk.

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

www.dbooks.org

Chapter 2

Scripts

In this chapter we introduce scripts, floating-point numbers, and as a warm-up exercise, Fibonacci numbers.

2.1 M-files

So far we have typed all of our programs "at the prompt", which is fine if you are not writing more than a few lines. Beyond that, you will want to store your program in a **script** and then execute the script.

A script is a file that contains MATLAB code. These files are also called "M-files" because they use the extension `.m`, which is short for MATLAB.

You can create and edit scripts with any text editor or word processor, but the simplest way is by clicking the New Script button in the upper left corner. A window appears running a text editor specially designed for MATLAB.

Type the following code in the editor:

```
x = 5
```

Then press the Save button. A dialog window should appears where you can choose the file name and the folder where it should go. Change the name to `myscript.m` and save it into any folder you like.

Now press 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 your script from the Command Window: type `myscript` at the prompt and press Enter. MATLAB executes your script and displays the result.

www.dbooks.org

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

When you run a script, MATLAB executes the commands in the M-File, one after another, exactly as if you had typed them at the prompt.

When you run a script, you should not include the extension `.m`. If you try, you will get an error message like this:

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

When you name a new script, try to choose something meaningful and memorable.

Do not choose a name that is not already in use; if you do, you will 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'.
```

Keeping track of your scripts can be a pain. To keep things simple, for now, I suggest putting all of your scripts in one folder.

2.2 Why scripts?

The most common reasons to use scripts are:

- When you are 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.
- If you choose good names for your scripts, you will be able to remember which script does what, and you might be able to reuse a script from one project to the next.
- If you run a script repeatedly, it is faster to type the name of the script than to retype the code!

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 will find debugging a frustrating exercise! And that brings us to the Third Theorem of Debugging:

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

2.3 Fibonacci

The Fibonacci sequence, denoted F, is described 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 semi-colon 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'.
```

www.dbooks.org

```
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 get an error.

MATLAB tells you what line of the script the error is in, and displays the line.

This information can be helpful, but beware! MATLAB is telling you where the error was discovered, not where the error is. In this case, the error is not in the script at all, which 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 source of the problem.

2.4 Floating-point numbers

MATLAB uses IEEE double-precision floating-point numbers, which are accurate to about 15 digits of precision. Most integers can be represented exactly, but most fractions cannot.

For example, if you compute the fraction `2/3`:

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

The result is only approximate — the correct answer has an infinite number of 6s.

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

```
>> 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 50!, 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 is 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 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 floating-point 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

Along with the commands that make up a program, it is useful to include comments that provide additional information about the program. The percent symbol `%` separates the comments from the code.

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

www.dbooks.org

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.

Comments have no effect on the execution of the program. They are there for human readers. Good comments make programs more readable; bad comments are useless or (even worse) misleading.

Avoid comments that are redundant with the code:

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

Good comments provide additional information that is 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 is a trade-off: longer code can become harder to read. Also, if you are 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 contain a comment that explains what it does, and what the requirements are for the workspace. 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 is 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

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 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 an assignment statement is only temporary, in the following sense. 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 one, 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 "gets" rather than "equals." So `x = y+1` is pronounced "`x` gets the value of `y` plus one."

2.8 Glossary

M-file: A file that contains a MATLAB program.

www.dbooks.org

script: An M-file that contains a sequence of MATLAB commands.

search path: The list of folder where MATLAB looks for M-files.

workspace: A set of variables and their values.

precondition: Something that must be true when the script starts, in order for it to work correctly.

postcondition: Something that will be true when the script completes.

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

floating-point: A way to represent numbers in a computer.

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.

comment: Part of a program that provides additional information about the program, but does not affect its execution.

2.9 Exercises

Exercise 3

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` and test it.

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

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

>> swap

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

Exercise 4

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

You observe that every day 5% of the bikes in Boston are dropped off in Cambridge, and 3% 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` 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 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 `a` and `b` over time.

www.dbooks.org

Chapter 3

Loops

This chapter introduces one of the most important programming language features, the `for` loop, the mathematical concepts sequence and series, and a process for writing programs, incremental development.

3.1 Updating variables

In Exercise 4, 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 `a`, so when the second line runs, it gets the old value of `b` and the new value of `a`. As a result, the change in `a` is not always the same as the change in `b`, which violates the principle of Conversation of Bikes!

One solution is to use temporary variables `anew` and `bnew`:

```
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:

www.dbooks.org

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

It is easy to look at this code and confirm that it obeys Conversation 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

There are four kinds of bugs:

Syntax error: You have written a command that cannot execute because it violates one of the rules of syntax. For example, 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. 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. Only you know what the program is supposed to do, so only you can check it. From MATLAB's point of view, there's nothing wrong with the program; the bug is in your head!

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, called **absolute** and **relative**.

Absolute error: 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.

Relative error: The error 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, $3000 matters a lot if we are talking about my annual salary, but not at all if we are 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 .00921, which is just less than 1%. For many purposes, being off by 1% is good enough.

3.4 for loops

A **loop** is a part of a program that executes repeatedly; a **for loop** is the kind of loop that uses the `for` statement.

The simplest use of a `for` loop is to execute one or more lines a fixed number of times. For example, in the last chapter we wrote a script named `bike_update` that simulates a day in the life of a bike share service. To simulate an entire month, we have to run it 30 times:

www.dbooks.org

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

The first line looks like an assignment statement, and it *is* like an assignment statement, except that it runs more than once. The first time it runs, 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 is 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 are inside the loop, but the indentation does not affect the execution of the program. The `end` statement marks the end of the loop.

To see the loop in action you can run a loop 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 much more common to put it in a script.

Exercise 5

Create a script named `bike_loop` that uses a `for` loop to run `bike_update` 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 is probably too long to fit, and even if it fit, it would be hard to interpret. A graph would be much better!

3.5 plotting

`plot` is a versatile function for plotting points and lines on a two-dimensional graph. Unfortunately, it is so versatile that it can be hard to use (and hard to read the documentation). We will 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. Other shapes include `+`, `*`, `x`, `s` (for square), `d` (for diamond), `^` (for a triangle).

You can also specify the color:

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

`r` stands for red; the other colors include **g**reen, **b**lue, **c**yan, **m**agenta, **y**ellow, and blac**k**.

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.

You should see a figure with three circles. MATLAB scales the plot automatically so that the axes runs from the lowest values in the plot to the highest.

Exercise 6

Modify `bike_loop` 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.

www.dbooks.org

3.6 Sequences

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.

`for` loops are 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 ratio $A_{i+1} = A_i/2$, 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
```

Each time through the loop, we find the next value of `a` by dividing the previous value by 2. Notice that the loop range starts at 2 because the initial value of `a` corresponds to A_1, so the first time through the loop we are computing A_2.

Each time through the loop, we replace the previous element with the next, so 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 will 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 is also possible to compute the ith element **directly**, as a function of i, without using the previous element. In math notation, $A_i = A_1 r^{i-1}$.

Exercise 7

Write a script named `sequence` 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

$$\sum_{i=1}^{10} A_i \tag{3.1}$$

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

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

`A1` is the first element of the sequence; each time through the loop `a` is the ith element.

The way we are using `total` is sometimes called an **accumulator**; that is, a variable that accumulates a result a little bit at a time. Before the loop we initialize it to 0. Each time through the loop we add in the ith element. At the end of the loop `total` contains the sum of the elements. Since that's the value we were looking for, we assign it to `ans`.

Exercise 8

This example computes the terms of the series directly; as an exercise, write a script named `series` 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 have 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, with a variable named `n`:

```
A1 = 1;
total = 0;
for i=1:n
```

www.dbooks.org

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

The code above 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 will 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. 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 will 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:

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

- When you are getting started, it might not be obvious how to choose the steps that get from `x=5` to the program you are trying to write. There is an extended example in Section 5.5.

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

3.10 Glossary

absolute error: The difference between an approximation and an exact answer.

relative error: The difference between an approximation and an exact answer, expressed as a fraction or percentage of the exact answer.

loop: A part of a program that runs repeatedly.

loop variable: A variable, defined in a `for` statement, that gets assigned a different value each time through the loop.

range: The set of values assigned to the loop variable, often specified with the colon operator; for example `1:5`.

www.dbooks.org

body: The statements inside the for loop that are run repeatedly.

sequence: In mathematics, a set of numbers that correspond to the positive integers.

element: A member of the set of numbers in a sequence.

recurrently: A way of computing the next element of a sequence based on previous elements.

directly: A way of computing an element in a sequence without using previous elements.

series: The sum of the elements in a sequence.

accumulator: A variable that is used to accumulate a result a little bit at a time.

generalization: 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: A way of programming by making a series of small, testable changes.

scaffolding: Code you write to help you program or debug, but which is not part of the finished program.

3.11 Exercises

Exercise 9

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 can 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 1/16. 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 10

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

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

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` 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 will 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 will have to update `prev1` and `prev2`; think carefully about the order of the updates!

www.dbooks.org

Chapter 4

Vectors

This chapter presents conditional statements and vectors, and three patterns for working with data: reduce, apply, and search.

4.1 Checking preconditions

Some of the loops in the previous chapter don't work if the value of `n` isn't set correctly before the loop runs. For example, this loop computes the sum of the first `n` elements of a geometric sequence:

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

It works for any positive value of `n`, but what if `n` is negative? In that case, you get:

```
total = 0
```

Why? Because the expression `1:-1` means "all the numbers from 1 to -1, counting up by 1." It's not immediately obvious what that should mean, but MATLAB's interpretation is that there aren't any numbers that fit that description, so the result is

```
>> 1:-1
ans = 1x0 empty double row vector
```

www.dbooks.org

If the matrix is empty, you might expect it to be `0x0`, but there you have it.

In any case, if you loop over an empty range, the loop never runs at all, which is why in this example the value of `total` is zero for any negative value of `n`.

If you are sure that you will never make a mistake, and that the preconditions of your functions will always be satisfied, then you don't have to check. But for the rest of us, it is dangerous to write a script, like this one, that quietly produces the wrong answer (or at least a meaningless answer) if the input value is negative. A better alternative is to use an `if` statement.

4.2 if statements

The `if` statement allows you to check for certain conditions and execute statements if the conditions are met. In the previous example, we could write:

```
if n<0
    ans = NaN
end
```

The syntax is similar to a `for` loop. The first line specifies the condition we are interested in; in this case we are asking if `n` is negative. If it is, 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. Don't make me tell you again.

In this example, the "right" thing to do if `n` is negative is to set `ans = NaN`, which is a standard way to indicate that the result is undefined (not a number).

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

```
    ans = total
end
```

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

In this example, one of the statements in the `else` clause is a `for` loop. Putting one compound statement inside another is legal and common, and sometimes called **nesting**.

4.3 Relational operators

The operators that compare values, like `<` and `>` are called **relational operators** because they test the relationship between two values. The result of a relational operator is one of the **logical values**: either 1, which represents "true," or 0, which represents "false."

Relational operators often appear in `if` statements, but you can also evaluate them at the prompt:

```
>> x = 5;
>> x < 10
ans = 1
```

You can assign a logical value to a variable:

```
>> flag = x > 10
flag = 0
```

A variable that contains a logical value is often called a **flag** because it flags the status of some condition.

The other relational operators are `<=` and `>=`, which are self-explanatory, `==`, for "equal," and `~=`, for "not equal."

Don't forget that `==` is the operator that tests equality, and `=` is the assignment operator. If you try to use `=` in an `if` statement, you get an error:

```
>> if x=5
 if x=5
     |
Error: Incorrect use of '=' operator.
To assign a value to a variable, use '='.
To compare values for equality, use '=='.

Did you mean:
>> x = 5
```

In this case, the error message is pretty helpful.

www.dbooks.org

4.4 Logical operators

To test if a number falls in an interval, you might be tempted to write something like `0 < x < 10`, but that would be wrong, so very wrong. Unfortunately, in many cases, you will get the right answer for the wrong reason. For example:

```
>> x = 5;
>> 0 < x < 10            % right for the wrong reason
ans = 1
```

But don't be fooled!

```
>> x = 17
>> 0 < x < 10            % just plain wrong
ans = 1
```

The problem is that MATLAB is evaluating the operators from left to right, so first it checks if `0<x`. It is, so the result is 1. Then it compares the logical value 1 (not the value of `x`) to 10. Since `1<10`, the result is true, even though `x` is not in the interval.

For beginning programmers, this is an evil, evil bug!

One way around this problem is to use a nested `if` statement to check the two conditions separately:

```
ans = 0
if 0<x
    if x<10
        ans = 1
    end
end
```

But it is more concise to use the AND operator, `&&`, to combine the conditions.

```
>> x = 5;
>> 0<x && x<10
ans = 1

>> x = 17;
>> 0<x && x<10
ans = 0
```

The result of AND is true if *both* of the operands are true. The OR operator, `||`, is true if *either or both* of the operands are true.

4.5 Vectors

The values we have seen so far are all single numbers, which are called **scalars** to contrast them with **vectors** and **matrices**, which are collections of numbers.

A vector in MATLAB is similar to a sequence in mathematics; it is set of numbers that correspond to positive integers. 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
```

In general, anything you can do with a scalar, you can also do with a vector. You can assign a vector value to a variable:

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

Variables that contain vectors are often capital letters. That's just a convention; MATLAB doesn't require it, but for beginning programmers, it is a useful way to remember what is a vector and what is a scalar.

4.6 Vector arithmetic

You can perform arithmetic with vectors, too. If you add a scalar 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` is 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     3

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

www.dbooks.org

The error message in this case is confusing, because we are thinking of these values as vectors, not matrices. The problem is a slight mismatch between math vocabulary and MATLAB vocabulary.

4.7 Everything is a matrix

In math (specifically in linear algebra) a vector is a one-dimensional sequence of values and a matrix is two-dimensional. And, if you want to think of it that way, a scalar is zero-dimensional.

In MATLAB, everything is a matrix (except strings). You can see this if you use the `whos` command to display the variables in the workspace. `whos` is similar to `who`, but it also displays the size of each value and other information.

To demonstrate, I'll make one of each kind of value:

```
>> scalar = 5
scalar = 5

>> vector = [1 2 3 4 5]
vector = 1     2     3     4     5

>> matrix = ones(2,3)
matrix =
     1     1     1
     1     1     1
```

The built-in function `ones` builds a new matrix with the given number of rows and columns, and sets all the elements to 1. Now let's see what we've got.

```
>> whos
  Name        Size             Bytes  Class     Attributes

  matrix      2x3                 48  double
  scalar      1x1                  8  double
  vector      1x5                 40  double
```

According to MATLAB, everything is a `double`, which is another name for a double-precision floating-point number.

But they have difference sizes:

- The size of `scalar` is `1x1`, which means it has 1 row and 1 column.
- `vector` has 1 row and 5 columns.
- And `matrix` has 2 rows and 3 columns.

The point of all this is that you can think of your values as scalars, vectors, and matrices, and I think you should. But in MATLAB they are all matrices.

4.8 Elementwise operators

If you have two vectors with the same length, you can add and subtract them:

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

>> Y = [4 5 6]
Y = 1     4     6

>> X + Y
ans = 5     7     9

>> X - Y
ans = -3    -3    -3
```

These operations are performed **elementwise**; that is, MATLAB adds or subtracts corresponding elements of the two vectors, and the result is a vector with the same size.

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

```
>> X / Y
ans = 0.4156
```

MATLAB is performing a matrix operation called right division, which I will not try to explain. If you want to divide the elements of `X` by the elements of `Y`, you have to use `./`, which is elementwise division:

```
>> X ./ Y
ans = 0.2500    0.4000    0.5000
```

Multiplication has the same problem. If you use `*`, MATLAB does matrix multiplication. With these two vectors, matrix multiplication is not defined, and 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 elementwise multiplication, use '.*'.
```

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

www.dbooks.org

```
>> X .* Y
ans = 4    10    18
```

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

4.9 Indices

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 any kind of expression.

```
>> i = 1;

>> 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`.

A limitation of this example is that 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
```

Now that works for a vector of any length.

4.10 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 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.

4.11 Vectors and sequences

Vectors and sequences go together nicely. For example, another way to evaluate the Fibonacci sequence is by storing successive values in a vector. Again, the definition of the Fibonacci sequence is $F_1 = 1$, $F_2 = 1$, and $F_i = F_{i-1} + F_{i-2}$ for $i > 2$. In MATLAB, that looks like

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

I use a capital letter for the vector `F` and lower-case letters for the scalars `i` and `n`.

If you had any trouble with Exercise 10, you have to appreciate the simplicity of this version. The MATLAB syntax is similar to the math notation, which makes it easier to check correctness.

However, you have to be careful with the range of the loop. In the previous version, the loop runs from `3` to `n`, and each time we assign a value to the `i`th element.

It would also work to "shift" the index over by two, running the loop from 1 to `n-2`:

www.dbooks.org

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

Either version is fine, but you have to choose one approach and be consistent. If you combine elements of both, you will get confused. I prefer the version that has `F(i)` on the left side of the assignment, so that each time through the loop it assigns the `i`th element.

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 ok.

Exercise 11

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 may want to shift the index.

4.12 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 the previous section:

```
plot(F)
```

This display is often useful for debugging, especially if your vectors are big enough that displaying the elements on the screen is unwieldy.

By default, MATLAB draws a blue line, but you can override that setting with a style string, as we saw in Section 3.5. For example, the string `'ro-'` tells MATLAB to plot a red circle at each data point; the hyphen means the points should be connected with a line.

4.13 Reduce

A frequent use of loops is to run through the elements of an array and add them up, or multiply them together, or compute the sum of their squares, etc. This kind of operation is called **reduce**, because it reduces a vector with multiple elements down to a single scalar.

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

```
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 Section 3.7. 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.

4.14 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 kind of operation is called **apply**, because you apply the operation to each element in the vector.

For example, the following loop computes a vector `Y` that contains the squares of the elements of `X` (assuming, again, that `X` is already defined).

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

4.15 Search

Yet another use of loops is to search the elements of a vector and return the index of the value you are looking for (or the first value that has a particular property).

For example, the following loop finds the index of the element 0 in `X`:

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

www.dbooks.org

A funny thing about this loop is that it keeps going after it finds what it is looking for. That might be what you want; if the target value appears more than one, this loop provides the index of the *last* one.

But if you want the index of the first one (or you know that there is only one), you can save some unnecessary looping by using the `break` statement.

```
for i=1:length(X)
    if X(i) == 0
        ans = i
        break
    end
end
```

`break` does pretty much what it sounds like. It ends the loop and proceeds immediately to the next statement after the loop (in this case, there isn't one, so the code ends).

4.16 Spoiling the fun

Experienced MATLAB programmers would never write the kind of loops in this chapter, because MATLAB provides simpler and faster ways to perform many reduce, filter and search operations.

For example, the `sum` function computes the sum of the elements in a vector and `prod` computes the product.

Many apply operations can be done with elementwise operators. The following statement is more concise than the loop in Section 4.14

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

And `find` can perform search operations:

```
>> X = [3 2 1 0]
X = 3     2     1     0

>> find(X==0)
ans = 4
```

If you understand loops and you are are comfortable with the shortcuts, feel free to use them! Otherwise, you can always write out the loop.

4.17 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 (increasingly familiar) 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 the motivations for functions.

4.18 Glossary

compound statement: A statement, like `if` and `for`, that contains other statements in an indented body.

nesting: Putting one compound statement in the body of another.

relational operator: An operator that compares two values and generates a logical value as a result.

logical value: A value that represents either "true" or "false". MATLAB uses the values 1 and 0, respectively.

flag: A variable that contains a logical value, often used to store the status of some condition.

scalar: A single value.

vector: A sequence of values.

matrix: A two-dimensional collection of values (also called "array" in some MATLAB documentation).

www.dbooks.org

ind An integer value used to indicate one of the values in a vector or matrix (also called subscript in some MATLAB documentation).

element: One of the values in a vector or matrix.

elementwise: An operation that acts on the individual elements of a vector or matrix (unlike some linear algebra operations).

reduce: A way of processing the elements of a vector and generating a single value; for example, the sum of the elements.

apply: A way of processing a vector by performing some operation on each of the elements, producing a vector that contains the results.

search: A way of processing a vector by examining the elements in order until one is found that has the desired property.

name collision: The scenario where two scripts that use the same variable name interfere with each other.

4.19 Exercises

Exercise 12

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 13

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 14

The following set of equations is based on a famous example of a chaotic system, the Lorenz attractor[1]:

$$x_{i+1} = x_i + \sigma\,(y_i - x_i)\,dt \tag{4.1}$$

$$y_{i+1} = y_i + [x_i(r - z_i) - y_i]\,dt \tag{4.2}$$

$$z_{i+1} = z_i + (x_i y_i - b z_i)\,dt \tag{4.3}$$

[1] See `https://en.wikipedia.org/wiki/Lorenz_system`.

- 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 values $\sigma = 10$, $b = 8/3$, and $r = 28$, and with $dt = 0.01$.

- Read the documentation for `plot3` and `comet3` and plot the results in 3 dimensions.

- Once the code is working, use semi-colons to suppress the output and then run the program with sequence length 100, 1000, and 10000.

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

- 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 15

The logistic map[2] is described by the following equation:

$$X_{i+1} = rX_i(1 - X_i) \tag{4.4}$$

where X_i is a number between zero and one and r is a positive number that represents.

- Write a script named `logmap` 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.

- 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?

[2] See `https://en.wikipedia.org/wiki/Logistic_map`

www.dbooks.org

Chapter 5

Functions

This chapter introduces the most important idea in computer programming: functions!

A **function** is like a script, except

- 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.

```
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 this case 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.

www.dbooks.org

Once you have 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 is 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 are debugging a new function, you might want to display intermediate results like this, but once it is working, you will want to add semi-colons to make it a **silent function**. Most built-in functions are silent; they compute a result, but they don't display anything (except sometimes warning messages).

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.

5.1 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 is a good idea to include

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

www.dbooks.org

needs to know. You can put additional comments inside the function that explain the details.

- 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.
- Any preconditions and postconditions.

5.2 What could go wrong?

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 is easy to get confused.

In the spirit of making errors on purpose, change the name of the function in `myfunc` to `something_else`, and then run it again.

If this is what you put in `myfunc.m`:

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

Here's what you'll get:

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

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

`myfunc` still works because that's the name of the file. `something_else` doesn't work because 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 write a function and rename the file to `my func.m`, and then try to run it, you get:

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

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!

5.3 Multiple input variables

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

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

This function computes the length of the hypotenuse of a right triangle if the lengths of the adjacent sides are `a` and `b`.

If we call it from the Command Window with arguments 3 and 4, we can confirm that the length of the third side is 5.

```
>> c = hypotenuse(3, 4)
c = 5
```

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

www.dbooks.org

```
>> c = hypotenuse(3)
Not enough input arguments.

Error in hypotenuse (line 2)
    res = sqrt(a^2 + b^2);
```

This error message is slightly confusing, because it suggests that the problem is in `hypotenuse` rather than in the function call. Keep that in mind when you are debugging.

If you provide too many arguments, you get

```
>> c = hypotenuse(3, 4, 5)
Error using hypotenuse
Too many input arguments.
```

Which is a better message.

5.4 Logical functions

In Section 4.4 we used logical operators to compare values. MATLAB also provides **logical functions** that check for certain conditions and return logical values: 1 for "true" and 0 for "false".

For example, `isprime` checks to see whether a number is prime.

```
>> isprime(17)
ans = 1

>> isprime(21)
ans = 0
```

The functions `isscalar` and `isvector` check whether a value is a scalar or vector.

To check whether a value you have computed is an integer, you might be tempted to use `isinteger`. But that would be wrong, so very wrong. `isinteger` checks whether a value belongs to one of the integer types (a topic we have not discussed); it doesn't check whether a floating-point value happens to be integral.

```
>> c = hypotenuse(3, 4)
c = 5

>> isinteger(c)
ans = 0
```

To do that, we have to write our own logical function, which we'll call `isintegral`:

```
function res = isintegral(x)
    if round(x) == x
        res = 1;
    else
        res = 0;
    end
end
```

This function is good enough for most applications, but remember that floating-point values are only approximately right: sometimes the approximation is an integer when the actual value is not; sometimes the approximation is not an integer when the actual value is.

5.5 Incremental development

Suppose we want to write a program to search for "Pythagorean triples": sets of integral values, like 3, 4, and 5, that are the lengths of the sides of a right triangle. In other words, we would like to find integral values a, b, and c such that $a^2 + b^2 = c^2$.

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

- Write a script named `find_triples` and start with a simple statement like `x=5`.
- Write a loop that enumerates values of a from 1 to 3, and displays them.
- Write a nested loop that enumerates values of b from 1 to 4, and displays them.
- Inside the loop, call `hypotenuse` to compute c and display it.
- Use `isintegral` to check whether c is an integral value.
- Use an if statement to print only the triples a, b, and c that pass the test.
- Transform the script into a function.
- Generalize the function to take input variables that specify the range to search.

Starting with `x=5` might seem silly, but if you start simple and add a little bit at a time, you will avoid a lot of debugging.

Here's the second draft:

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

www.dbooks.org

At each step, the program is testable: it produces output (or another visible effect) that you can check.

5.6 Nested loops

The third draft contains a nested loop:

```
for a=1:3
    a
    for b=1:4
        b
    end
end
```

The inner loop gets executed 3 times, once for each value of `a`, so here's what the output looks like (I adjusted the spacing to make the structure clear):

```
>> find_triples

a = 1   b = 1
        b = 2
        b = 3
        b = 4

a = 2   b = 1
        b = 2
        b = 3
        b = 4

a = 3   b = 1
        b = 2
        b = 3
        b = 4
```

The next step is to compute c for each pair of values a and b.

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

To display the values of `a`, `b`, and `c`, I store them in a vector; here's what the output looks like:

```
>> find_triples

ans = 1.0000    1.0000    1.4142
ans = 1.0000    2.0000    2.2361
ans = 1.0000    3.0000    3.1623
ans = 1.0000    4.0000    4.1231
ans = 2.0000    1.0000    2.2361
ans = 2.0000    2.0000    2.8284
ans = 2.0000    3.0000    3.6056
ans = 2.0000    4.0000    4.4721
ans = 3.0000    1.0000    3.1623
ans = 3.0000    2.0000    3.6056
ans = 3.0000    3.0000    4.2426
ans = 3         4         5
```

You might notice that we are wasting some effort here. After checking $a = 1$ and $b = 2$, there is no point in checking $a = 2$ and $b = 1$. We can eliminate the extra work by adjusting the range of the second loop:

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

If you are following along, run this version to make sure it has the expected effect.

5.7 Conditions and flags

The next step is to check for integral values of c. This loop calls `isintegral` and prints the resulting logical value.

```
for a=1:3
    for b=a:4
        c = hypotenuse(a, b);
        flag = isintegral(c);
        [c, flag]
    end
end
```

By not displaying `a` and `b` I made it easy to scan the output to make sure that the values of `c` and `flag` look right.

```
>> find_triples
```

www.dbooks.org

```
ans = 1.4142         0
ans = 2.2361         0
ans = 3.1623         0
ans = 4.1231         0
ans = 2.8284         0
ans = 3.6056         0
ans = 4.4721         0
ans = 4.2426         0
ans = 5              1
```

The next step is to use `flag` to display only the successful triples:

```
for a=1:3
    for b=a:4
        c = hypotenuse(a, b);
        flag = isintegral(c);
        if flag
            [a, b, c]
        end
    end
end
```

Now the output is minimal:

```
>> find_triples

ans = 3     4     5
```

5.8 Encapsulation and generalization

As a script, this program has the side-effect of assigning values to `a`, `b`, `c`, and `flag`, which would make it hard to use 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
            c = hypotenuse(a, b);
            flag = isintegral(c);
            if flag
                [a, b, c]
            end
```

```
        end
    end
end
```

The empty parentheses in the signature are not strictly necessary, but they make it apparent that there are no input variables. Similarly, when I call the new function, I like to use parentheses to remind me that it is a function, not a script:

```
>> find_triples()
```

The output variable isn't strictly necessary, either; it never gets assigned a value. But I put it there as a matter of habit, and also 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
            c = hypotenuse(a, b);
            flag = isintegral(c);
            if flag
                [a, b, c]
            end
        end
    end
end
```

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

```
>> find_triples(15)

ans = 3     4     5
ans = 5    12    13
ans = 6     8    10
ans = 8    15    17
ans = 9    12    15
```

Some of these are more interesting than others. The triples $5, 12, 13$ and $8, 15, 17$ are "new," but the others are just multiples of the $3, 4, 5$ triangle we already knew.

www.dbooks.org

5.9 continue

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 the result is greater than 1, a and b share a common factor and we can use the `continue` statement to skip to the next pair:

```
function res = find_triples (n)
    for a=1:n-1
        for b=a:n
            if gcd(a,b) > 1
                continue
            end
            c = hypotenuse(a, b);
            if isintegral(c)
                [a, b, c]
            end
        end
    end
end
```

`continue` 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 two loops, it might not be obvious which loop to jump to, but the rule is to jump to the inner-most loop (which is what we want).

I also simplified the program slightly by eliminating `flag` and using `isintegral` as the condition of the `if` statement.

Here are the results with `n=40`:

```
>> find_triples(40)

ans =  3     4     5
ans =  5    12    13
ans =  7    24    25
ans =  8    15    17
ans =  9    40    41
ans = 12    35    37
ans = 20    21    29
```

5.10 Mechanism and leap of faith

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 (often the last line) 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 or creating a figure).
5. The program resumes from where it left off. The value of the function call is the value of the output variable.

When you are 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 take the "leap of faith": assume that the function works correctly, and go on to the next statement after the function call.

When you use built-in functions, it is natural to take the leap of faith, in part because you expect that most MATLAB functions work, and in part because you don't generally have access to the code in the body of the function.

But when you start writing your own functions, you will probably 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.

5.11 Why functions?

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.

www.dbooks.org

But experienced programmers use functions extensively, for several good reasons:

- Each function has its own workspace, so using functions helps avoid name collisions.
- Functions 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.
- 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.

Another reason to use functions is that many of the tools provided by MATLAB require them. For example, in the next chapter we will use `fzero` to find solutions of nonlinear equations. Later we will use `ode45` to approximate solutions to differential equations.

5.12 Glossary

side-effect: An effect, like modifying the workspace, that is not the primary purpose of a script.

input variable: A variable in a function that gets its value, when the function is called, from one of the arguments.

output variable: A variable in a function that is used to return a value from the function to the caller.

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 or generate a figure, or have any other side-effects.

logical function: A function that returns a logical value (1 for "true" or 0 for "false").

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.

generalization: Making a function more versatile by replacing specific values with input variables.

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.

5.13 Exercises

Exercise 16

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

$$(F_i F_{i+3}, 2F_{i+1}F_{i+2}, F_{i+1}^2 + F_{i+2}^2) \tag{5.1}$$

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.

www.dbooks.org

Chapter 6

Zero-finding

In this chapter we'll use the MATLAB function `fzero` to find roots of nonlinear equations.

6.1 Nonlinear equations

What does it mean to "solve" an equation? That may seem like an obvious question, but I want to take a minute to think about it, starting with a simple example: let's say that we want to know the value of a variable, x, but all we know about it is the relationship $x^2 = a$.

If you have 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 I will call an **analytic solution**,

www.dbooks.org

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 equation, 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 is not clear that this move did any good at all. But suppose that we had some reason to expect there to be a solution near 4. We could start with $x = 4$ as an "initial guess," and then use the equation $x = \sqrt{2x + 3}$ iteratively to compute successive approximations of the solution.[1]

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 5 iterations, the relative error is about 0.1%, which is good enough for most purposes.

Techniques that generate numerical solutions are called **numerical methods**. The nice thing about the method I just demonstrated is that it is simple, but it doesn't always work, and it is not often used in practice. We'll see better alternatives soon.

6.2 Zero-finding

A nonlinear equation like $x^2 - 2x = 3$ is a statement of equality that is true for some values of x and false for others. A value that makes it true is a solution;

[1] To understand why this works, see `https://en.wikipedia.org/wiki/Fixed-point_iteration`.

any other value is a non-solution. But for any given non-solution, there is no sense of whether it is close or far from a solution, or where we might look to find one.

To address this limitation, it is useful to rewrite non-linear equations as zero-finding problems:

- The first step is to define an "error function" that computes how far a given value of x is from being a solution.

  In this example, the error function is

  $$f(x) = x^2 - 2x - 3 \tag{6.1}$$

  Any value of x that makes $f(x) = 0$ is also a solution of the original equation.

- The next step is to find values of x that make $f(x) = 0$. These values are called zeros of the function, also called **roots**.

Zero-finding lends itself to numerical solution because we can use the values of f, evaluated at various values of x, to make reasonable inferences about where to look for zeros.

And one of the best numerical methods is available as a built-in MATLAB function, `fzero`.

6.3 fzero

In order to use `fzero`, you have to define a MATLAB function that computes the error function you derived from the original nonlinear equation, and you have to provide an initial guess at the location of a zero.

We've already seen an example of an error function:

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

You can call `error_func` from the Command Window, and confirm that there are zeros at 3 and -1.

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

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

www.dbooks.org

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 names the M-file that evaluates the error function. The `@` symbol allows us to name the function without calling it. The interesting thing here is that you are not actually calling `error_func` directly; you are just telling `fzero` where it is. In turn, `fzero` calls your error function — more than once, in fact.

The second argument is the initial guess. If we provide a different initial guess, 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` every time 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. `fzero` stops when the interval is so small that the estimated zero is correct to about 15 digits.

If you would like to know more about how `fzero` works, see Section 14.2.

6.4 What could go wrong?

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

```
>> 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 is 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 you called `error_func` from the interpreter, if you had assigned the result to a variable:

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

www.dbooks.org

```
y = 5

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

You can avoid all of this if you remember these two rules:

- Functions should assign values to their output variables.[2]
- When you call a function, you should do something with the result (either assign it to a variable or use it as part of an expression, etc.).

When you write your own functions and use them yourself, it is easy for mistakes to go undetected. But when you use your functions with MATLAB functions like `fzero`, you have to get it right!

Yet another thing that 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 is possible that `fzero` won't be able to find a root.

`fzero` is generally pretty 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.

6.5 Choosing an initial guess

The better your initial guess (or interval) is, the more likely it is that `fzero` will work, and the fewer iterations it will need.

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

Another approach is to plot the function and see if you can 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`:

[2] Well, ok, there are exceptions, including `find_triples`. Functions that don't return a value are sometimes called "commands", because they do something (like display values or generate a figure) but either don't have an output variable or don't make an assignment to it.

```
>> 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 location of the two roots.

6.6 Vectorizing functions

With this example, you might get the following warning[3]:

```
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 `*` and `^` operators; as we saw in Section 4.8, those operators don't do what we want, which is *elementwise* multiplication and exponentiation.

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, for what it's worth.

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

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

'sin' appears to be both a function and a variable.
```

[3] In Octave it's an error, so you have to vectorize the function.

www.dbooks.org

```
If this is unintentional, use 'clear sin' to remove
the variable 'sin' from the workspace.
```

Since the value we assigned to `sin` is a scalar, and a scalar is really a 1x1 matrix, MATLAB tries to access the 5th 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` was a vector, or if the value of `x` was 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`.
- Before you define a function, check whether MATLAB already has a function with the same name.

6.8 Debugging your head

When you are 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 are 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 degrees 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. So the correct answer is $\pi/2$, which we can confirm by dividing through by `pi`:

```
>> asin(1) / pi
ans = 0.5000
```

With this kind of testing, you are not really checking for errors in MATLAB, you are 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 are in your head.

6.9 Glossary

analytic solution: A way of solving an equation by performing algebraic operations and deriving an explicit way to compute a value.

numerical solution: A way of solving an equation by finding a numerical value that satisfies the equation, often approximately.

numerical method: A method (or algorithm) for generating a numerical solution.

zero (of a function): An argument that makes the result of a function 0.

function handle: In MATLAB, a function handle is a way of referring to a function by name (and passing it as an argument) without calling it.

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).

6.10 Exercises

Exercise 17

1. Write a function called `cheby6` that evaluates the 6th Chebyshev polynomial. It should take an input variable, x, and return

$$32x^6 - 48x^4 + 18x^2 - 1 \tag{6.2}$$

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.

www.dbooks.org

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

Exercise 18

When a duck is floating on water, how much of its body is submerged?

To estimate a solution to this problem[4], 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 ρ, is $0.3g/cm^3$ (0.3 times the density of water), and that its mass is $\frac{4}{3}\pi r^3\rho$.

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:

- Write an equation relating ρ, d, and r.
- Rearrange the equation so the right-hand side is zero. Our goal is to find values of d that are roots of this equation.
- Write a MATLAB function that evaluates this function. Test it, then make it a quiet function.
- Make a guess about the value of d_0 to use as a starting place.
- Use `fzero` to find a root near d_0.
- Check to make sure the result makes sense. In particular, check that $d < 2r$, because otherwise the volume equation doesn't work!
- 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?

[4]This example is adapted from Gerald and Wheatley, *Applied Numerical Analysis*, Fourth Edition, Addison-Wesley, 1989.

Chapter 7

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. And you'll see two patterns for computing with vectors, existential and universal quantification. But first, let's talk about organizing functions and files.

7.1 Functions and files

So far we have only put one function in each file. It is 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 any other file.

Large programs almost always require more than one function; 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 Exercise 18, 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)
```

www.dbooks.org

```
end

function res = error_func(d)
    rho = 0.3;      % density in g / cm^3
    r = 10;         % radius in cm
    res = d;
end
```

The line `res = d` isn't finished yet, but this is enough code to test. Once I finished and tested `error_func`, I would modify `duck` to use `fzero`.

For this problem we might only need 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.

7.2 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. Here's a simple (silly) example:

```
function res = display_vector(X)
    X
end
```

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

This is another example of a function that doesn't actually return a value; it just displays the value of the input variable:

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

Here's a more interesting example that encapsulates the code from Section 4.13 that adds 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 it `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.

7.3 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 Section 4.14:

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

Here's how `myapply` works:

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

Exercise 19

Write a function named `myfind` that encapsulates the code, from Section 4.15, that finds the location of a target value in a vector.

7.4 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

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

www.dbooks.org

Unfortunately, the converse is not 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 is 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 5.3:

```
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 elementwise matrix powers, use '.^'.
```

But if you replace `^` with the elementwise 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 elementwise operators and functions that work on vectors, the new function will also work on vectors.

7.5 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, then the elements of the cumulative sum, C, are:

$$C_i = \sum_{j=1}^{i} V_j \tag{7.1}$$

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, then we would expect `cumsum(diff(X)` to be `X`:

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

ans = 1     2     3     4
```

But it isn't.

Exercise 20

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

www.dbooks.org

7.6 Products and ratios

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

$$P_i = \prod_{j=1}^{i} V_j \tag{7.2}$$

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 21

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 $e^{\ln a + \ln b} = ab$.

If you apply `myratio` to a vector that contains Fibonacci numbers, you can confirm that the ratio of successive elements converges on the golden ratio, $(1 + \sqrt{5})/2$ (see Exercise 13).

7.7 Existential quantification

It is 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 logic, this condition is called **existential quantification**, and it is denoted with the symbol $\exists$, which is pronounced "there exists." For example, this expression

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

means, "there exists some element x in the set S such that $x > 0$." In MATLAB it is 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 is 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.

7.8 Universal quantification

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

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

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

One way evaluate this expression in MATLAB is to count the number of elements that satisfy the condition. A better way is to reduce the problem to existential quantification; that is, to rewrite

$$\forall x \text{ in } S : x > 0 \tag{7.3}$$

as

$$\sim \exists x \text{ in } S : x \leq 0 \tag{7.4}$$

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 non-positive.

www.dbooks.org

Exercise 22

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 non-positive elements.

7.9 Logical vectors

When you apply a logical operator to a vector, the result is a **logical vector**; that is, a vector whose elements are the logical values 1 and 0.

```
>> 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` 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 23

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

7.10 Debugging in four acts

When you are debugging a program, and especially if you are 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 starting 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.

www.dbooks.org

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.

7.11 Glossary

top-level function: The first function in an M-file; it is the only function that can be called from the Command Window or from another file.

helper function: A function in an M-file that is not the top-level function; it only be called from another function in the same file.

existential quantification: A logical condition that expresses the idea that "there exists" an element of a set with a certain property.

universal quantification: A logical condition that expresses the idea that all elements of a set have a certain property.

logical vector: A vector, usually the result of applying a logical operator to a vector, that contains logical values 1 and 0.

Chapter 8

Ordinary Differential Equations

In this chapter you'll learn about a mathematical tool for describing physical systems, differential equations, and two computation tools for solving them, Euler's method and `ode45`.

8.1 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 time t, we can write the following equation for the rate of growth:

$$\frac{dy}{dt}(t) = ay(t) \tag{8.1}$$

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.

www.dbooks.org

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.

This equation is **first-order** because it involves only first derivatives. If it involved second derivatives, it would be second order, and so on.

And it is **linear** because each term involves t, f, or df/dt raised to the first power; if any of the terms involved products or powers of t, f, and df/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.

8.2 Euler's method

Here's a test to see if you are as smart as 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.

So 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:

$$T_{i+1} = T_i + dt \tag{8.2}$$

$$Y_{i+1} = Y_i + \frac{df}{dt}(t)\ dt \tag{8.3}$$

$$\tag{8.4}$$

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. One way to check is to run it once with time step dt and then run it again with time step $dt/2$. If the results are the same, they are probably accurate; otherwise, we can cut the time step again.

8.3 Implementing Euler's method

As an example I'll use Euler's method to solve the equation from Section 8.1:

$$\frac{dy}{dt}(t) = ay(t)$$

With the initial condition $y(0) = 5$ billion cells and the growth parameter $a = 0.2$ per hour.

As a first step, I created 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` I initialize the initial conditions and then call `rate_func`. In `rate_func` I compute the rate of change in the population.

After testing these functions, I added code to `euler` to implement these difference equations:

$$T_{i+1} = T_i + \Delta t \tag{8.5}$$
$$Y_{i+1} = Y_i + r\Delta t \tag{8.6}$$
$$\tag{8.7}$$

where r is the rate of population growth computed by `rate_func`. Here's the code:

```
function res = euler()
    T(1) = 0;
    Y(1) = 5;
    dt = 0.1;

    for i=1:100
        r = rate_func(T(i), Y(i));
```

www.dbooks.org

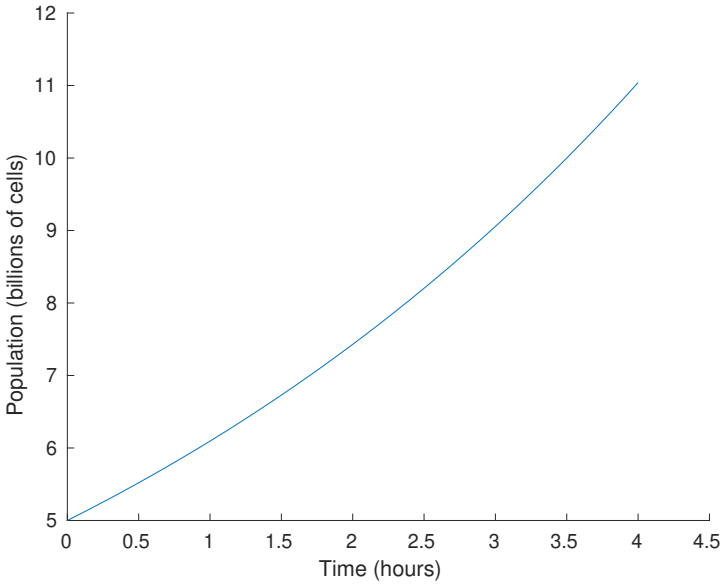


Figure 8.1: Solution to a simple differential equation by Euler's method.

```
        T(i+1) = T(i) + dt;
        Y(i+1) = Y(i) + r * dt;
    end
    plot(T, Y)
end
```

The result is a plot of population over time, shown in Figure 8.1. The population doubles in a little less than 4 hours.

8.4 ode45

A limitation of Euler's method is that it assumes that the derivative is constant between time steps, and that is 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 Section 14.1.

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, `rate_func`:

```
function res = rate_func(t, y)
```

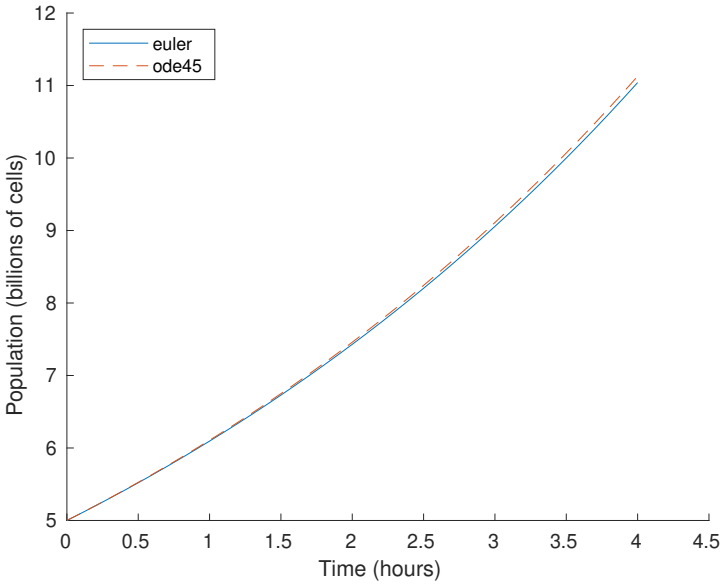


Figure 8.2: Solutions to a simple differential equation by Euler's method and `ode45`.

```
    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 Section 6.3. 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.

`ode45` is the first function we have seen that returns *two* output variables. In order to store them, we have to assign them to two variables, `T` and `Y`.

Figure 8.2 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 4-5 hours, the two solutions are indistinguishable. But as the rate of growth increases, Euler's method gets less accurate.

In general, you should use `ode45` instead of Euler's method. It is almost always more accurate.

www.dbooks.org

8.5 Time dependence

Looking at `rate_func` in the previous section, you might notice that it takes `t` as an input variables but doesn't use it. That's because the growth rate does not depend on time; that's because bacteria don't know what time it is.

But rats do. Or, at least, they know what season it is. Suppose that the population 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

$$\frac{dy}{dt}(t) = ay(t)\,(1 - \cos(\omega t)) \tag{8.8}$$

where t is time in days and $y(t)$ is the population at time t.

a and ω are **parameters**. A parameter is a value that quantifies a physical aspect of the scenario being modeled. 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 1.

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
```

Suppose there are 1000 rats at $t = 365/2$. We can compute the growth rate like this:

```
>> r = rats

r = 4
```

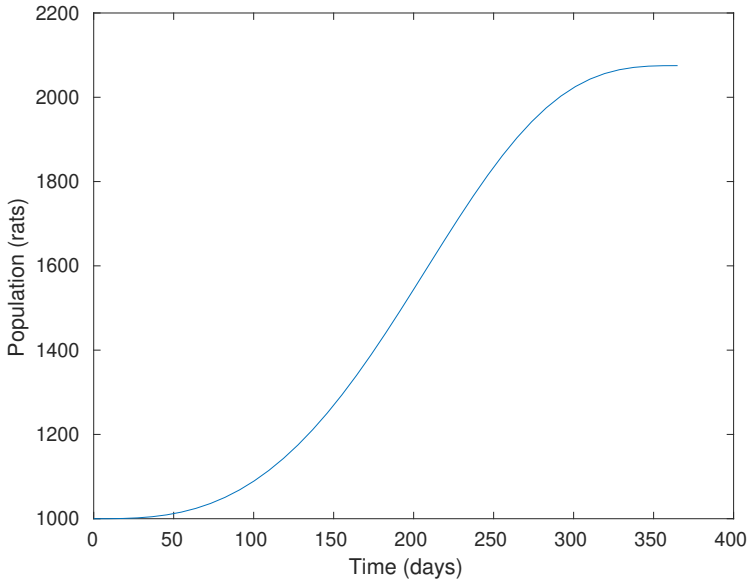


Figure 8.3: Solutions to a simple differential equation by Euler's method and `ode45`.

So if there are 1000 rats on June 31, we expect them to reproduce at a rate that would produce about 4 new rats per day.

Since the growth rate is constantly changing, it is not easy to predict the future rat population, but that is exactly what `ode45` does. Here's how:

```
[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, one year. The third argument is the initial population, $y(0) = 1000$.

Figure 8.3 shows the results. The population grows slowly during the winter, quickly during the summer, and the slowly again in the fall.

To see the population at the end of the year, you can display the last element of `Y`:

```
Y(end)
2.0751e+03
```

That's a little more than 2000 rats, so the population roughly doubles in one year.

`end` is a special word in MATLAB; when it appears as an index, it means "the

www.dbooks.org

index of the last element". You can use it in an expression, so `Y(end-1)` is the second-to-last element of `Y`.

8.6 What could go wrong?

Don't forget the `@` on the function handle. If you leave it out, MATLAB treats the first argument as a function call, and calls `rate_func` without providing arguments.

```
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, remember that the function you write will be called by `ode45`, which means has to take two input variables, `t` and `y`, in that order, and return one output variable, `res`.

If you are working with a rate function like this:

$$\frac{dy}{dt}(t) = ay(t) \tag{8.9}$$

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 this:

```
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.

8.7 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.

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, I saved 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 Encapulated PostScript. The third argument tells MATLAB what "driver" to use. The details aren't important, but `epsc` generates figures in color.

8.8 Glossary

differential equation (DE): An equation that relates the derivatives of an unknown function.

www.dbooks.org

ordinary DE (ODE): A DE in which all derivatives are taken with respect to the same variable.

partial DE (PDE): A DE that includes derivatives with respect to more than one variable

first-order DE: A DE that includes only first derivatives.

linear DE: A DE that includes no products or powers of the function and its derivatives.

time step: The interval in time between successive estimates in the numerical solution of a DE.

parameter: A value that appears in a model to quantify some physical aspect of the scenario being modeled.

8.9 Exercises

Exercise 24

Suppose that you are 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[1]:

$$\frac{dy}{dt}(t) = -k(y(t) - e)$$

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 from 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 0.5 °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] See `https://en.wikipedia.org/wiki/Newton's_law_of_cooling`.

- 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.
- Add a function called `rate_func` that takes `t` and `y` and computes $\frac{dy}{dt}$. 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`.
- 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 $-0.7\,^{\circ}\mathrm{C/min}$.
- 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 more slowly, and reaches room temperature after about an hour.
- Plot the results and estimate the time when the temperature reaches 60 °C.

www.dbooks.org

Chapter 9

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 about matrices.

9.1 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.

One of many ways to create a matrix is the `magic` function, which returns a "magic square" with the given size [1]:

```
>> M = magic(3)

M =  8     1     6
     3     5     7
     4     9     2
```

If you don't know the size of a matrix, you can use `whos` to display it:

```
>> whos
  Name        Size                    Bytes  Class
  M           3x3                        72  double array
```

Or the `size` function, which returns a vector:

[1] See `https://en.wikipedia.org/wiki/Magic_square`.

www.dbooks.org

```
>> V = size(M)

V = 3     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 = 1

>> M(2,1)

ans = 3
```

When you are 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.

Another way to create a matrix is to enclose the elements in brackets, with semi-colons between rows:

```
>> D = [1,2,3 ; 4,5,6]

D =  1     2     3
     4     5     6

>> size(D)

ans = 2     3
```

9.2 Row and column vectors

Although it is useful to think in terms of scalars, vectors and matrices, from MATLAB's point of view, everything is a matrix. A scalar 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, anyway. Actually, there are two kind of vectors. The ones we have 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. 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
```

9.3 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:

www.dbooks.org

```
>> D = [1,2,3 ; 4,5,6]

D =  1     2     3
     4     5     6
```

`D` has two rows, so its transpose has two columns:

```
>> Dt = D'

Dt = 1     4
     2     5
     3     6
```

Exercise 25

What effect does the transpose operator have on row vectors, column vectors, and scalars?

9.4 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:

$$\frac{dx}{dt} = \alpha x - \beta xy \tag{9.1}$$

$$\frac{dy}{dt} = -\gamma y + \delta xy \tag{9.2}$$

$$\tag{9.3}$$

where x and y are the populations of rabbits and foxes, and a, b, c, and d are parameters governing the interactions between the two species.[2]

At first glance you might think you could solve these equations by calling `ode45` once to solve for x as a function of time 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 initial values $x(0)$ and $y(0)$, and the output is a matrix that contains one column for x and one for y.

[2] See `https://en.wikipedia.org/wiki/Lotka-Volterra_equations`.

Here's what the rate function looks like with the parameters $a = 0.1$, $b = 0.01$, $c = 0.1$, and $d = 0.002$:

```
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 is time. Even though the time variable, `t`, is not used in this rate function, its presence is required by the `ode45` solver.

The second input variable 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 scalar variables. This isn't necessary, but giving names to these values helps me 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 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 semi-colon 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.

www.dbooks.org

As always, it is a good idea to test your rate function before you call `ode45`. I created 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.

9.5 Output matrices

`ode45` returns two values: `T`, which is a vector of time values, and `M`, which is a matrix with one column for each population and one row for each time value in `T`.

```
>> size(M)
ans = 185     2
```

This structure – one column per variable – is a common way to use matrices. `plot` understands this structure, so if you do this:

```
>> 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);
```

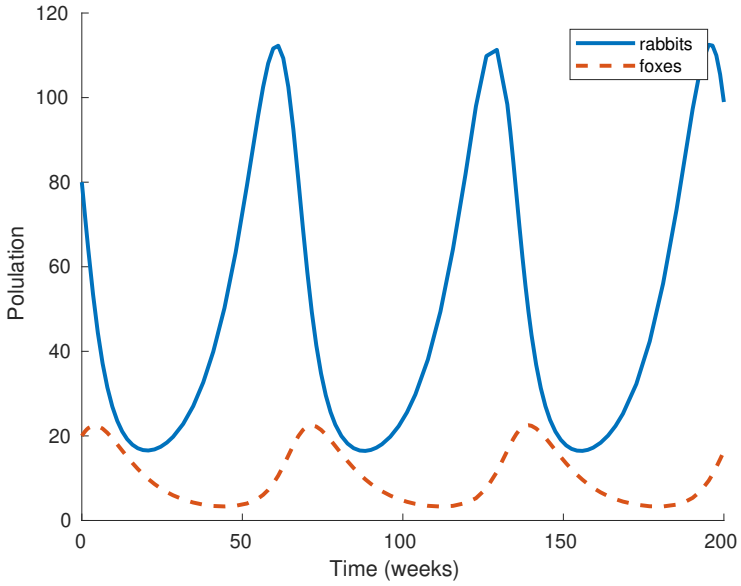


Figure 9.1: Solution for the Lotka-Volterra model.

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 = 185     1

>> size(F)
ans = 185     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 9.1 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.

www.dbooks.org

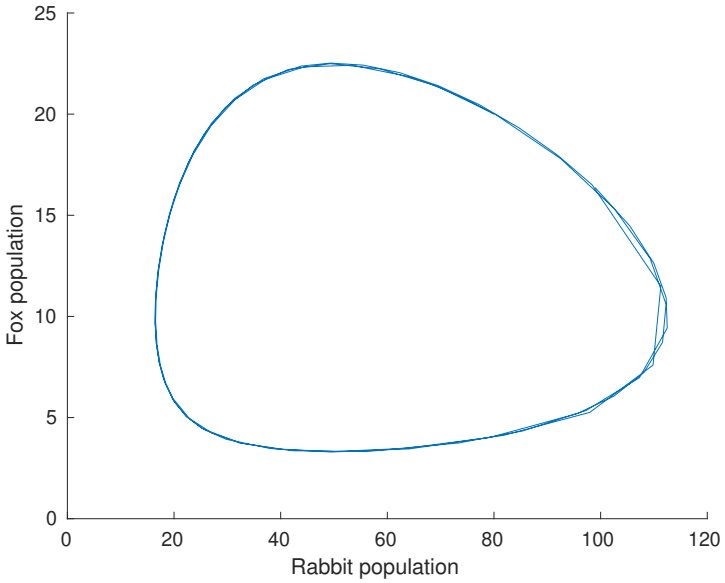


Figure 9.2: Phase plot from the Lotka-Volterra model.

9.6 Phase plot

Instead of plotting the two populations over time, it is sometimes useful to plot them against each other, like this:

```
>> plot(R, F)
```

Figure 9.2 shows the result. 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.

Over time, the state moves around the plane; this figure shows the path traced by the state over time; this path is called a **trajectory**.

Since the behavior of this system is periodic, the trajectory is a loop.

If there are 3 variables in the system, we need 3 dimensions to show the state of the system, so the trajectory is a 3-D curve. You can use `plot3` to trace trajectories in 3 dimensions, but for 4 or more variables, you are on your own.

9.7 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.

Error in ode45 (line 115)
  odearguments(FcnHandlesUsed, solver_name, ode, tspan, y0,
               options, varargin);

Error in lotka (line 7)
    [T, M] = ode45(@rate_func, tspan, V_init);
```

Which is pretty good as error messages go. 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 initial conditions you provide when you call `ode45` have to be the same as the order, inside `rate_func`, where you unpack the input vector and repack the output vector.

9.8 Glossary

row vector: A matrix that has only one row.

column vector: A matrix that has only one column.

transpose: An operation that transforms the rows of a matrix into columns (or the other way around, if you prefer).

system of equations: A collection of equations written in terms of the same set of variables.

unpack: To copy the elements of a vector into a set of variables.

pack: To copy values from a set of variables into a vector.

state: If a system can be described by a set of variables, the values of those variables are called the state of the system.

www.dbooks.org

phase plot: A plot that shows the state of a system as point in the space of possible states.

trajectory: A path in a phase plot that shows how the state of a system changes over time.

9.9 Exercises

Exercise 26

Based on the examples we have seen so far, you would think that all ODEs describe population as a function of time, but that's not true.

According to Wikipedia, "The Lorenz attractor, introduced by Edward Lorenz in 1963, is a non-linear three-dimensional deterministic dynamical system derived from the simplified equations of convection rolls arising in the dynamical equations of the atmosphere. For a certain set of parameters the system exhibits chaotic behavior and displays what is today called a strange attractor..."[3]

The system is described by this system of differential equations:

$$x_t = \sigma(y - x) \tag{9.4}$$

$$y_t = x(r - z) - y \tag{9.5}$$

$$z_t = xy - bz \tag{9.6}$$

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 `lorentz.m` with a top-level function named `lorenz` and a helper function named `rate_func`.
2. The rate function should takes `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.

[3] See `https://en.wikipedia.org/wiki/Lorenz_attractor`.

Chapter 10

Second-order Systems

So far we have seen first-order differential equations and systems of first-order ODEs. In this chapter we introduce second-order systems, which are particularly useful for modeling Newtonian motion.

10.1 Newtonian motion

Newton's second law of motion is often written like this:

$$F = ma \tag{10.1}$$

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 force, which is true, but in most physical simulations it is 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 a differential equation

$$\frac{d^2r}{dt^2} = a \tag{10.2}$$

where $\frac{d^2r}{dt^2}$ is the second time derivative of r.

www.dbooks.org

Because this equation includes a second derivative, it is a second-order ODE. `ode45` can't solve the equation this form, but by introducing a new variable, `v`, for velocity, we can rewrite it as a system of first-order ODEs:

$$\frac{dr}{dt} = v \tag{10.3}$$

$$\frac{dv}{dt} = a \tag{10.4}$$

$$\tag{10.5}$$

The first equation says that the first derivative of r is v; the second equation says that the first derivative of v is a.

10.2 Freefall

As an example, let's get back to the question from Exercise 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 it is 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 $-9.8\ m/s^2$, 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.

Here is a rate function we can use with `ode45` to solve this 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 takes `t` and `X` as input variables, where the elements of `X` are understood to be the position and velocity of the object. 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 of 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 (30 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:

```
Y = M(:, 1)
plot(T, Y)
```

Figure 10.1 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 zero into negative altitude. We can solve that problem using events.

www.dbooks.org

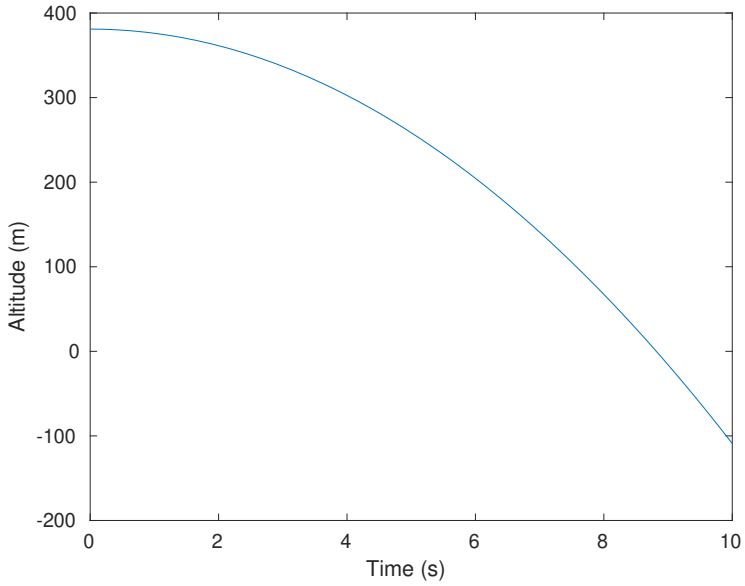


Figure 10.1: Altitude versus time for an object in free fall.

10.3 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. MATLAB provides a way to deal with this problem; here's how it works:

1. Define an **event function** that specifies when the simulation should stop. For example, here is 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 the same input variables as the rate function and returns three output variables: `value` determines whether an **event** can occur, `direction` determines whether it does, and `isterminal` determines what happens.

   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; otherwise the simulation continues.

This event function uses the altitude of the penny as `value`, so an event occurs when the altitude is decreasing and passes through 0. When it does, the simulation ends.

2. 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. 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 timestep. If the event function indicates that a terminal event occurred, `ode45` stop 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 8.817, 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 m/s.

10.4 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.

$$f_d = -sgn(v)bv^2 \tag{10.6}$$

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.

sgn is the sign or signum function, which is 1 for positive values of v and -1 for negative values. So f_d is always in the opposite direction of v.

www.dbooks.org

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 I have estimated[1] that it is about 75×10^{-6} kg/m.

Here's a function that takes `t` and `X` as input variables and returns the total acceleration of the penny due to gravity and drag:

```
function res = acceleration(t, X)
    b = 75e-6;                % drag constant in kg/m
    v = X(2);                 % velocity in m/s
    f_d = -sgn(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
```

The first three lines compute force due to drag. The next two lines compute acceleration due to drag. The last two lines compute total acceleration due to drag and gravity.

Be careful when you are 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 are in. That helps me avoid nonsense like adding forces to accelerations.

To use this function, I made 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
```

Everything else is the same. Figure 10.2 shows the result.

Air resistance makes a big difference! Velocity increases until the drag acceleration equals g; 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).

[1] Based on reports that the terminal velocity of a penny is about 18 m/s.

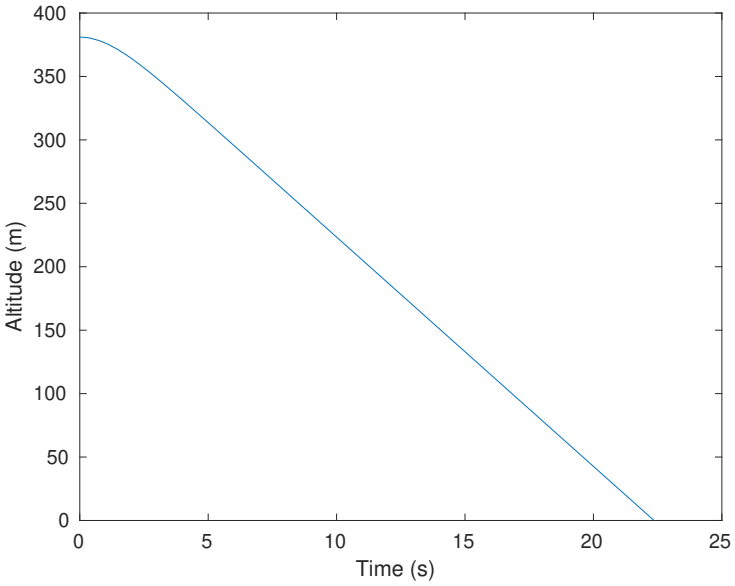


Figure 10.2: Altitude versus time for an penny in free fall with air resistance.

And the final velocity is 18.1 m/s, substantially slower than before (86 m/s).

10.5 Exercises

Exercise 27

The drag constant for a skydiver without a parachute is about 0.2 kg m. Modify the penny code from this chapter to simulate the descent of a 75 kg skydiver from an initial altitude of 4000 m. What is the velocity of the skydiver on impact?

After opening a 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.

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!

Now suppose the skydiver free falls until they get to altitude 1000 m before opening the parachute. How long would it take them to reach the ground?

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)?

www.dbooks.org

Exercise 28

Here's a question from the web site *Ask an Astronomer*[2]:

> "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 Earth reaches the surface of the Sun.
3. Express your answer in days, and plot the results as millions of kilometers versus days.

[2] `https://web.archive.org/web/20180617133223/http://curious.astro.cornell.edu/about-us/39-our-solar-system/the-earth/other-catastrophes/57-how-long-would-it-take-the-earth-to-fall-into-the-sun-intermediate`

Chapter 11

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.

11.1 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 have used MATLAB vectors to represent:

sequences: A sequence is a set of values identified by integer indices; it is natural to store the elements of the sequence as elements of a MATLAB vector.

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: The results from `ode45` are vectors, `T` and `Y`, that represent a time series, that is, a mapping from the time values in `T` to the values in `Y`.

In this chapter we will 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:

www.dbooks.org

```
>> P = [3 4]
```

To interpret this vector, we have to know what 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.

Spatial vectors represent 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 3 and 4. We can compute it using the Pythagorean theorem:

```
>> sqrt(sum(P.*P))
ans = 5
```

Or more simply by 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 is 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 $\hat{\mathbf{P}}$.

11.2 Adding vectors

Vectors are particularly 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:

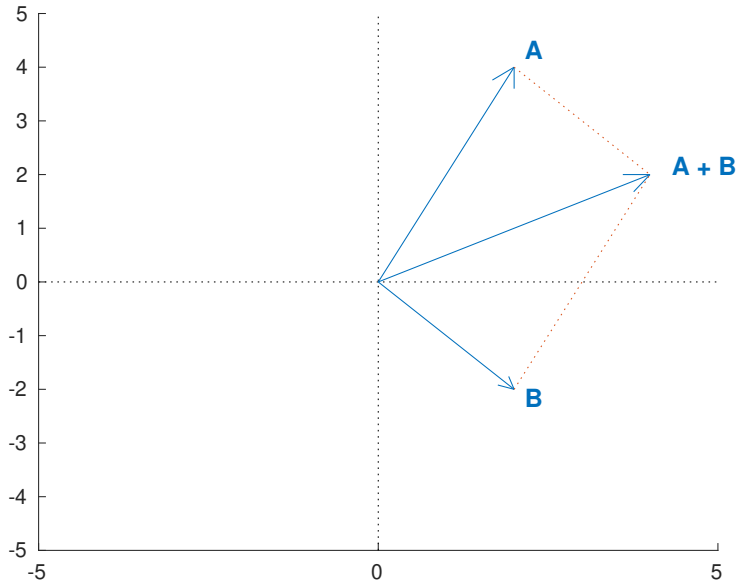


Figure 11.1: The sum of two forces represented by vectors.

```
>> A = [2, 4];
>> B = [2, -2];
```

`A` represents a force pulling northeast; `B` represents a force pulling southeast, as shown in Figure 11.1

To compute the sum of these forces, all we have to do is add the vectors:

```
>> A + B
ans = 4     2
```

This will come in handy later in the chapter.

11.3 ODEs in two dimensions

So far we have 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. Assuming there is no wind and no spin on the ball, it should travel in a vertical plane, so we can think of the system as two-dimensional, with x representing the horizontal distance travelled and y representing height or altitude.

Here's a rate function we can use to simulate this system with `ode45`:

www.dbooks.org

```
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 of 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 home plate, with an initial velocity of 30 m/s in the horizontal and 40 m/s in the vertical direction.

Here's how we can call `ode45` with these initial conditions:

```
    P = [0; 3];       % 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);
```

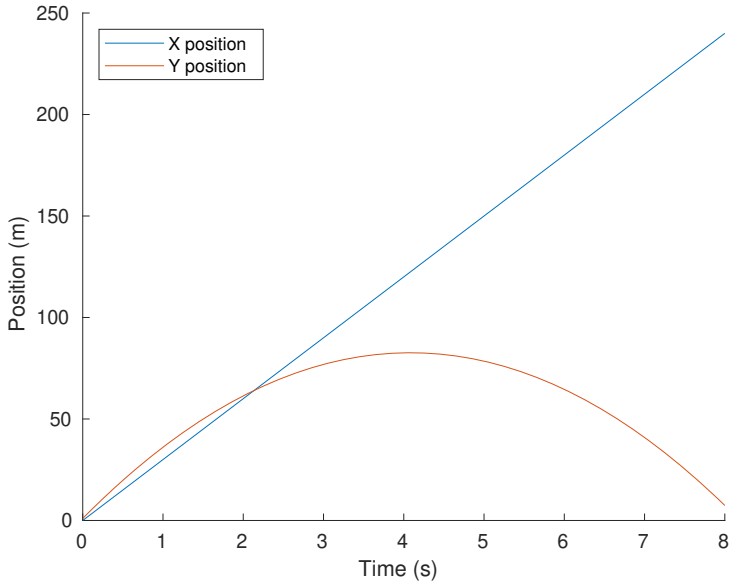


Figure 11.2: Simulated flight of a baseball neglecting drag force.

`P` and `V` are column vectors because we put semi-colons between the elements. So `W` is a column vector with four elements. `tspan` specifies that we want to run the simulation for 6 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 coordinates of position.

Figure 11.2 shows what they look like. 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".

www.dbooks.org

11.4 Drag force

A simple model for the drag force on a baseball is:

$$\mathbf{F_d} = -\frac{1}{2}\,\rho v^2 C_d A \hat{\mathbf{V}} \tag{11.1}$$

where $\mathbf{F_d}$ 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 .

$\mathbf{V}$ is the baseball's velocity vector; v is the magnitude of V and $\hat{\mathbf{V}}$ is a unit vector in the same direction as V. The minus sign at the beginning means that the result is in the opposite direction as V.

The following function computes 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 of 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 Section 11.3, `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, then 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 11.3 shows these quantities graphically. Acceleration due to drag, $\mathbf{D}$, is in the opposite direction of velocity, $\mathbf{V}$. Acceleration of gravity, $\mathbf{G}$, is straight down. Total acceleration, $\mathbf{A}$, is the sum of $\mathbf{D}$ and $\mathbf{G}$.

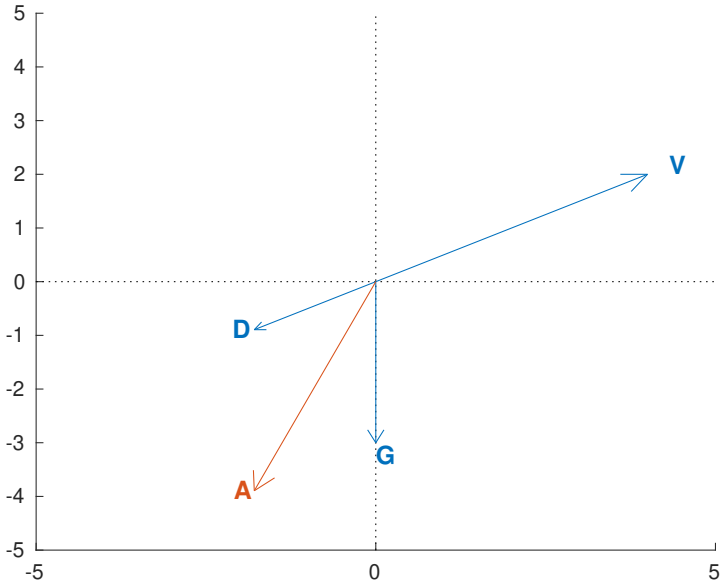


Figure 11.3: Diagram of velocity, **V**, acceleration due to drag force, **D**, acceleration due to gravity, **G**, and total acceleration, **A**.

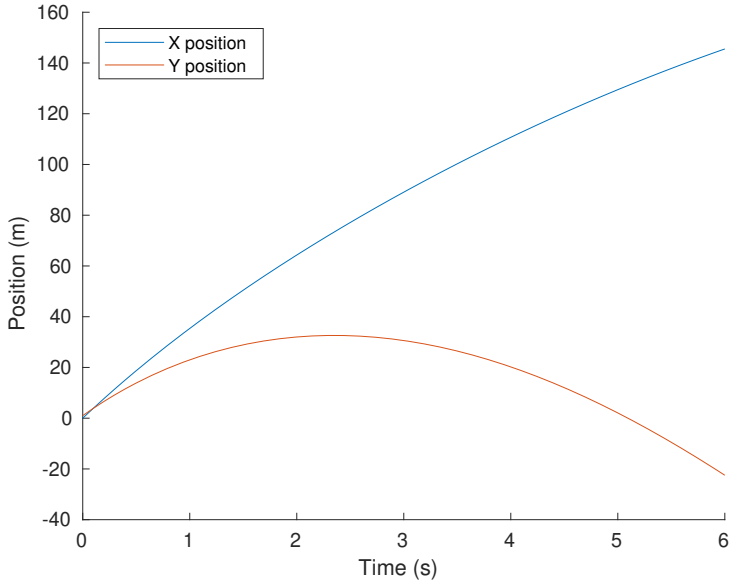


Figure 11.4: Simulated flight of a baseball including drag force.

www.dbooks.org

Figure 11.4 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.

11.5 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 concatenation" 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 semi-colon. Here is 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 probably guessed that `horzcat` is the function that performs horizontal concatenation, and likewise with `vertcat` and vertical concatenation.

In Section 11.3 we use horizontal concatenation to pack `dRdt` 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 `dRdt` and `dVdt` are column vectors, the semi-colon 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.

`ode45` expects the result from `rate_func` to be a column vector, so if you are working with `ode45`, it is probably a good idea to make *everything* a column vector.

www.dbooks.org

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.

11.6 Glossary

spatial vector: A value that represents a multidimensional physical quantity like position, velocity, acceleration or force.

unit vector: A vector with norm 1, used to indicate direction.

norm: The magnitude of a vector. Sometimes called "length," but not to be confused with the number of elements in a MATLAB vector.

concatenation: The operation of joining two matrices end-to-end to form a new matrix.

11.7 Exercises

Exercise 29

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 30

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 *The Physics of Baseball*[1]; the figure you need is reproduced at `https://github.com/AllenDowney/ModSimMatlab/blob/master/code/data/baseball_drag.png`.

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 31

According to Wikipedia, the record distance for a human cannonball is 59.05 meters[2].

[1] Robert K. Adair, Harper Paperbacks, 3rd Edition, 2002.

[2] See `https://en.wikipedia.org/wiki/Human_cannonball`.

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.

www.dbooks.org

Chapter 12

Optimization

In Exercise 31, you were asked to find the best launch angle for a human cannonball, meaning the angle that maximizes the distance traveled before landing. In this chapter, we 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, `fminsearch`.

12.1 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 Section 10.3, 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
```

www.dbooks.org

```
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. Its 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.

12.2 Trajectory

Now we can extract the x and y positions:

```
X = M(:, 1);
Y = M(:, 2);
```

In Section 11.3 we plotted `X` and `Y` separately as functions of time. Alternatively we can plot them against each other, like this:

```
plot(X, Y)
```

Figure 12.1 shows the result, which is the **trajectory** of the baseball from launch, on the left, to landing, on the right.

12.3 Range versus angle

Now we'd like to simulate the trajectory of the baseball with a range of launch angles. First, I'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.

```
function res = baseball_range(theta)
    P = [0; 1];
    v = 50;
    [vx, vy] = pol2cart(theta, v);
```

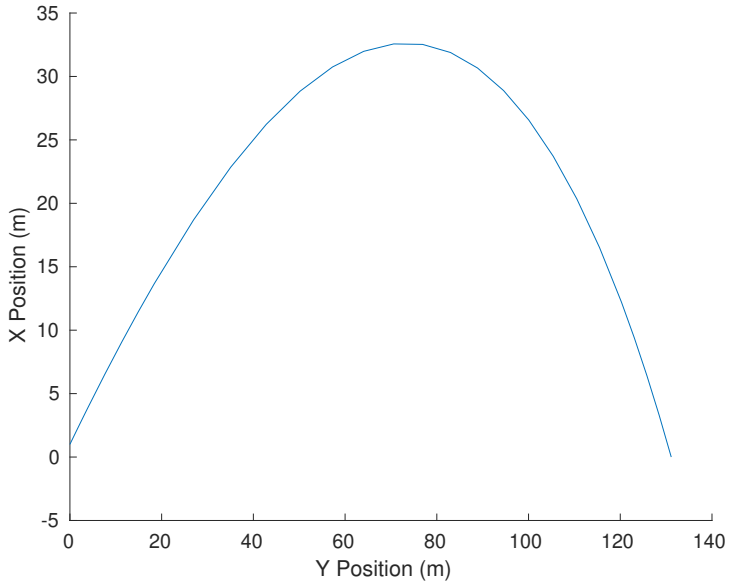


Figure 12.1: Simulated flight of a baseball plotted as a trajectory.

```
    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 is always 50 m/s. I use `pol2cart` to convert the angle and magnitude to Cartesian components, `vx` and `vy`.

After running the simulation, I 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`:

www.dbooks.org

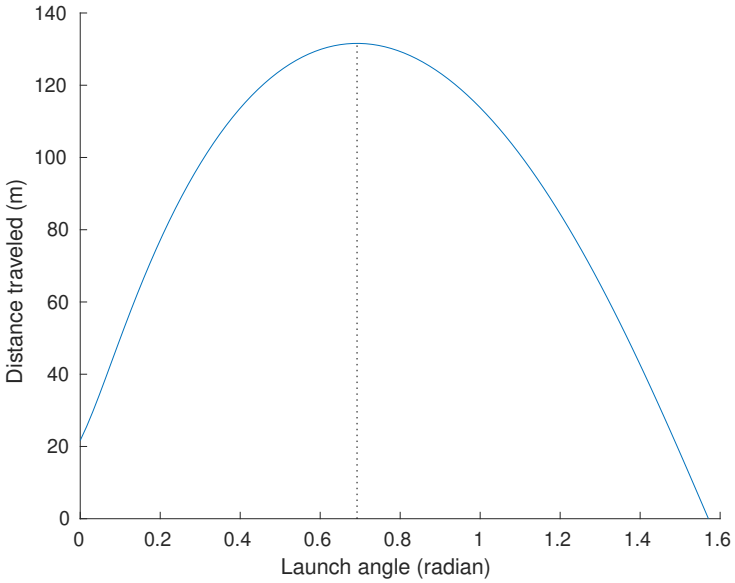


Figure 12.2: Simulated flight of a baseball plotted as a trajectory.

```
    plot(thetas, ranges)
```

Figure 12.2 shows the result. As expected, the ball does not travel far if it is hit nearly horizontal or vertical. The peak is apparently near 0.7 rad.

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`.

12.4 `fminsearch`

`fminsearch` is similar to `fzero`, which we saw in Section6.3. Recall that `fzero` takes a function handle and an initial guess, and returns a root of the function. As an example, to find a root of this 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. If we 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
```

`x` is the location of the minimum; `fval` is the value of the function evaluated at `x`.

We can use `fminsearch` to find the *maximum* of a function by writing a short function that negates the function we want to maximize:

```
function res = min_func(angle)
    res = -baseball_range(angle);
end
```

Now we can 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 about 132 m.

If you are curious about how `fminsearch` works, see Section 14.3.

12.5 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 `movie` to play them back.

The more informal way is to draw a series of plots. Here is a function that animates the results of a baseball simulation:

```
function animate(T,M)
    X = M(:,1);
    Y = M(:,2);
```

www.dbooks.org

```
    minmax = [min([X]), max([X]), min([Y]), max([Y])];

    for i=1:length(T)
        clf
        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`, a vector `T` and a matrix `M`. The columns of `M` are the x and y coordinates of the baseball.

`minmax` is a vector of four elements which 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` 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 step is larger.

One way to fix this problem is to change the way to specify `tspan`. Here is an example:

```
    tspan = 0:0.1:10;
```

The result vector that goes from 0 to 10 with a step size of 0.1. This option does not 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.

12.6 Exercises

Exercise 32

Manny Ramirez is a former member of the Boston Red Sox (an American baseball team) 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 33

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.

www.dbooks.org

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[1], 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.

[1] See `https://en.wikipedia.org/wiki/Magnus_effect`.

Chapter 13

Case studies

This chapter includes additional exercises where you can apply what you have learned.

13.1 Celestial mechanics

Celestial mechanics describes how objects move in outer space. If you did Exercise 28, 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, $\mathbf{P}$, to represent the position of the Earth relative to the Sun.

Given the mass of the Sun, $m1$, and the mass of the Earth, $m2$, the gravitational force between them is

$$\mathbf{F_g} = -G\frac{m_1 m_2}{r^2}\hat{\mathbf{P}}$$

where G is the universal gravitational constant[1], r is the distance of Earth from the Sun, and $\hat{\mathbf{P}}$ is a unit vector in the direction of $\mathbf{P}$.

Write a simulation of Earth orbiting the Sun. You can look up the orbital velocity of the Earth, or 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.

[1] See `https://en.wikipedia.org/wiki/Gravity`.

www.dbooks.org

13.2 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

$$PE = -G\frac{m_1 m_2}{r} \tag{13.1}$$

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 Section 10.3):

```
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 it does more work 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://www.mathworks.com/help/matlab/math/choose-an-ode-solver.html`). 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).

13.3 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: `http://modsimpy.com/dunk`.

Since the record is 70 m, let's design a jump for 80 m. We'll start with the following modeling assumptions:

- Initially the bungee cord hangs from a crane with the attachment point 80 m above a cup of tea.
- Until the cord is fully extended, it applies no force to the jumper. It turns out this might not be a good assumption; we will revisit it.
- 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. See `http://modsimpy.com/hooke`.
- The mass of the jumper is 75 kg.
- The jumper is subject to drag force, as in the previous model, so that their terminal velocity is 60 m/s.

Our objective is 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!

We'll start with the length of the bungee cord, `L` at 25 m and spring constant, `k` at 40 N/m.

Assume that the jumper has a cross-sectional area of 1 m and a terminal velocity of 60 m/s, and weighs 75 kg.

13.4 Bungee revisited

In the previous case study we simulated a bungee jump with a model that took into account gravity, air resistance, and the spring force of the bungee cord, but we ignored the weight of the cord.

It is 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 `http://modsimpy.com/bungee` you'll find a paper[2] that explains this phenomenon and derives the acceleration of the jumper, a, as a function of position, y, and velocity, v:

$$a = g + \frac{\mu v^2/2}{\mu(L+y) + 2L}$$

[2]Heck, Uylings, and Kdzierska, "Understanding the physics of bungee jumping", Physics Education, Volume 45, Number 1, 2010.

www.dbooks.org

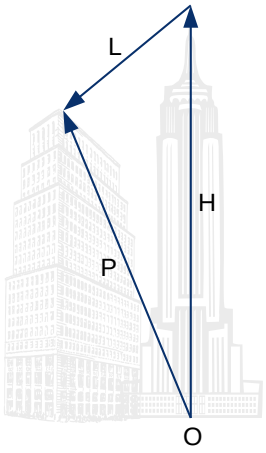


Figure 13.1: Diagram of the initial state for the Spider-Man case study.

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, and the mass of the jumper, M.

If you don't believe that their model is correct, this video might convince you: `http://modsimpy.com/drop`.

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?

13.5 Spider-Man

In this case study 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 13.1.

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
```

The goals of this case study are:

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.

We'll use the following parameters:

1. According to the Spider-Man Wiki[3], Spider-Man weighs 76 kg.
2. Let's assume his terminal velocity is 60 m/s.
3. The length of the web is 100 m.
4. The initial angle of the web is 45° to the left of straight down.
5. The spring constant of the web is 40 N/m when the cord is stretched, and 0 when it's compressed.

[3] `http://modsimpy.com/spider`

www.dbooks.org

Chapter 14

How does it work?

In this chapter we "open the hood", looking more closely at how some of the tools we have used — `ode45`, `fzero`, and `fminsearch` — work.

14.1 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://en.wikipedia.org/wiki/Runge-Kutta_methods`, but I'll try to 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. The details are in where the rate function is called and how the slopes are averages.

As an example, I'll solve the following differential equation:

$$\frac{dy}{dt}(t) = y \sin t$$

Given a differential equation, it is usually straightforward to write a rate function:

```
function res = rate_func(t, y)
    dydt = y * sin(t);
    res = dydt;
end
```

And we can call it like this:

www.dbooks.org

```
    y0 = 1;
    tspan=[0 4];
    options = odeset('Refine', 1);
    [T, Y] = ode45(@rate_func, tspan, y0, options);
```

For this example I 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 I 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 plot a red circle at each location, and a short red line showing the computed slope.

Figure 14.1 shows the result. `ode45` computes 10 time steps (not counting the initial condition) and evaluates the rate function 61 times.

Figure 14.2 shows the same plot, zoomed in on a single time step, from 0.8 to 1.2. We can see that `ode45` evaluates the rate function several times per time step, at several places between the end points.

One important thing to take away from this figure: 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.

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 is uses to adjust the time step. If the 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.

14.2 How `fzero` works

According to the MATLAB documentation, `fzero` uses uses "a combination of bisection, secant, and inverse quadratic interpolation methods".

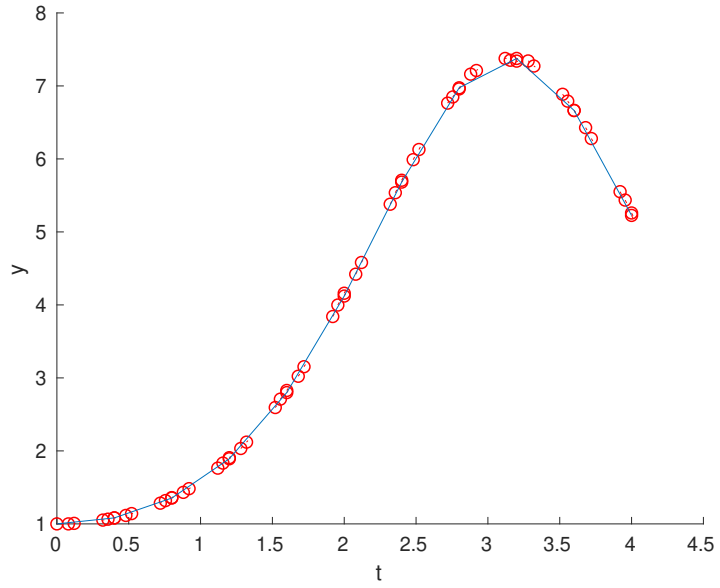


Figure 14.1: Points where `ode45` evaluates the rate function.

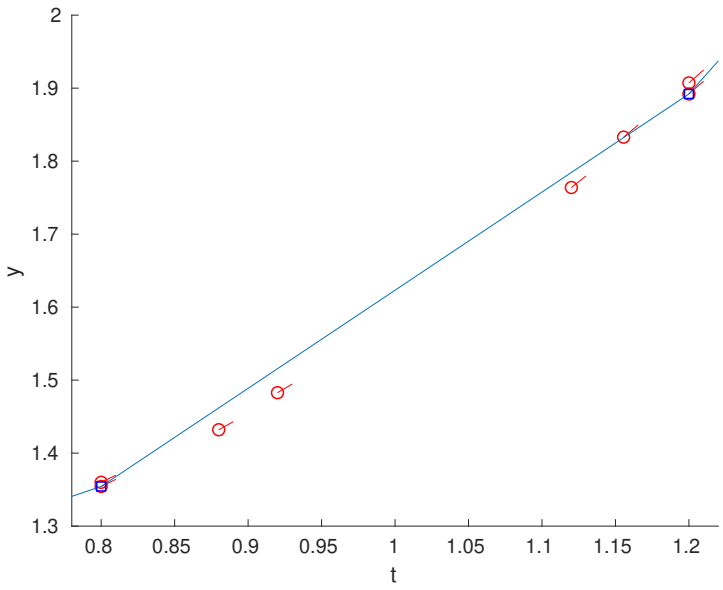


Figure 14.2: Points where `ode45` evaluates the rate function, zoomed in.

www.dbooks.org

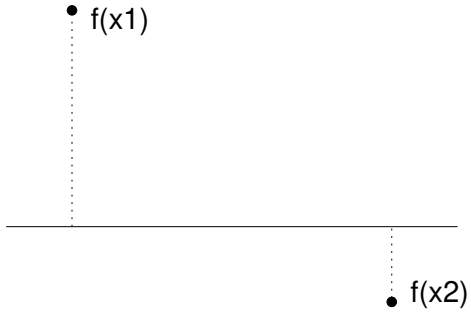


Figure 14.3: Initial state of a root-finding search.

To understand what that means, suppose we are trying to find a root of a function on one variable, $f(x)$, and assume we have evaluated the function at two place, x_1 and x_2, and found that the result have opposite signs. Specifically, assume $f(x_1) > 0$ and $f(x_2) < 0$, as shown in Figure 14.3.

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 was 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**!

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," then 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://en.wikipedia.org/wiki/Brents_method`.

14.3 How `fminsearch` works

According to the MATLAB documentation, `fminsearch` uses the Nelder-Mead simplex algorithm. You can read about it at `https://en.wikipedia.org/wiki/Nelder-Mead_method`, 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 are trying to find the minimum of a function of a single variable, $f(x)$. As a starting place, assume that we have

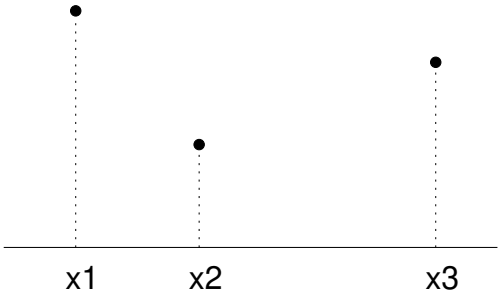


Figure 14.4: Initial state of a golden-section search.

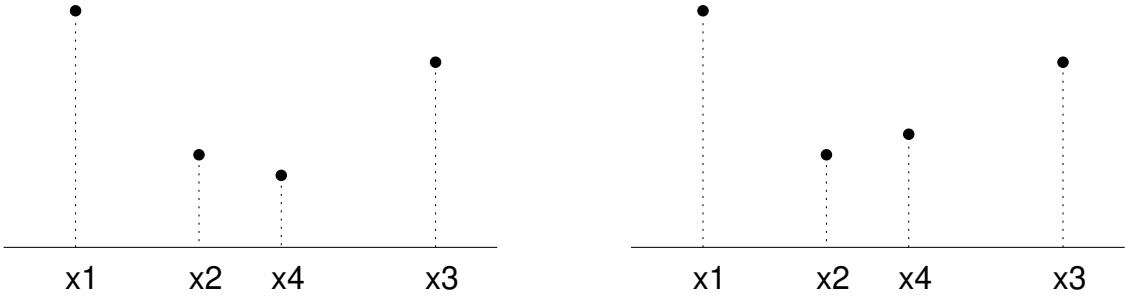


Figure 14.5: Possible states of a golden-section search after evaluating $f(x_4)$.

evaluated the function at three places, x_1, x_2, and x_3, and found that x_2 yields the lowest value. Figure 14.4 shows this initial state.

If f is continuous, there has to be at least one minimum point 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 $f(x_2)$. Figure 14.5 shows the two possible states.

If $f(x_4)$ is less than than $f(x_2)$ (shown on the left), the local 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

www.dbooks.org

x_3, but that turns out not to be the best choice. You can read about a better option at `https://en.wikipedia.org/wiki/Golden-section_search#Probe_point_selection`.

Index

`%`, 15

absolute error, 20
abstraction, 2, 53
acceleration, 96, 99, 104, 107, 121
accumulator, 24, 37
adaptive, 124
addition
    vector, 121
air resistance, 10, 96, 98, 105, 111
altitude, 104
analysis, 2
analytic solution, 57
AND operator, 32
animation, 117
`ans`, 5, 43
apply, 37, 67
argument, 4, 5, 61
arithmetic
    vector, 32
arithmetic operator, 3
array, 36
assignment, 31
    target, 16
assignment operator, 6
assignment statement, 21
axes, 81
`axis`, 117

baseball, 103, 104, 113, 118
bike share system, 17, 19
bisection, 125
body of loop, 21
boom and bust, 90
Boston Red Sox, 110, 118
bracket, 125
`break` statement, 38
bug, 20
bungee cord, 121
bungee jump, 120, 121
buoyancy, 64

cannon, 111
Cartesian coordinates, 115
Cartesian system, 103
case sensitive, 6, 8
celestial mechanics, 119
character, 6
Chebyshev polynomial, 64
`clear`, 7
clear figure, 22, 117
`clf`, 22, 117
coefficient
    drag, 107
coefficient of drag, 111
coffee, 82
collision
    name, 38, 44, 46, 62
colon, 90
colon operator, 21, 29
column, 85
column vector, 86, 90, 92, 96, 110
comma operator, 109
command, 2
    `whos`, 33
    `who`, 6
Command Window, 2, 11
comment, 15, 45
complex number
    Euler's equality, 5
    imaginary unit, 5, 6
compound statement, 21, 30, 43

www.dbooks.org

computation
    explicit, 57
concatenation, 109
condition, 50
conditional statement, 30
conservation of energy, 119
`continue`, 52
cookie, 120
cooling, 82
cross-sectional area, 107
`cumprod`, 69
`cumsum`, 68
cumulative product, 69
cumulative sum, 68

data, 2
debugging, 63, 71
    Eighth Theorem, 72
    Fifth Theorem, 19
    First Theorem, 3
    Fourth Theorem, 13
    list of theorems, 131
    Second Theorem, 8
    Seventh Theorem, 63
    Sixth Theorem, 25
    Third Theorem, 12
definition
    function, 43
denominator, 8, 9
density, 64, 107, 110
derivative, 105
design, 2
`diff`, 68
differential equation, 75, 123
    second-order, 95
direct computation, 23
direction, 103
`disp`, 7
division, 3, 9
division by zero, 14
`doc`, 15
Documentation
    functions, 45
documentation, 45
    `doc`, 4
    `help`, 4
    function, 15
Dormand-Prince, 123
double precision, 33
drag, 98, 107, 111
drag coefficient, 107
`drawnow`, 117
duck, 64

Earth, 100, 119
element, 5, 23, 35, 85
elementwise operator, 34, 62, 68
ellipse, 2
ellipsis, 7
`else` clause, 30
Empire State Building, 10, 96, 121
encapsulation, 51, 66
`end`, 80
`end` statement, 21
energy, 119
equality, 16, 31
equation
    differential, 75
    nonlinear, 57
error, 8
error function, 58
error message, 9, 16
Errors
    absolute, 20
    logical, 20
    numerical, 20
    relative, 21
    runtime, 20
    syntax, 20
Euclidean norm, 104
Euler's equality, 5
Euler's Method, 76
event function, 97, 113
existential quantification, 69
explanation, 2
explicit computation, 57
exponent, 14
exponentiation, 3
expression, 3, 5, 6, 35
    invalid, 8

external validation, 2
ezplot, 61
`ezplot`, 61, 64

`factorial`, 14
Fenway Park, 118
Fibonacci, 15, 36, 40
Fibonacci number, 12, 55
figure, 82
Figure Window, 22
file extension, 11
`find`, 71
first-order differential equation, 75
fixed-point iteration, 58
flag, 31, 50, 52, 71
floating-point, 13, 48
floating-point number, 33
flow of execution, 53
`fminsearch`, 116, 126
fminsearch`fminsearch`, 119
folder, 12
`for` loop, 21
force, 99, 104
    drag, 107
    Magnus, 118
`format`, 13
function, 43
    `gcd`, 52
    `length`, 35
    `prod`, 38
    `sum`, 38
    argument, 4
    documentation, 15
    error, 58
    event, 97
    helper, 65
    logical, 47
    rate, 78, 123
    reasons for, 54
    silent, 44
    top-level, 65
    vectorizing, 62, 67
function call, 5, 53
    nested, 4
function definition, 43
function handle, 59, 78, 80, 89, 97
function name, 45
Functions
    naming, 45
`fzero`, 59, 116, 125

`gcd` function, 52
`gcf`, 82
generalization, 24, 51
geometric sequence, 23
get current figure, 82
`getframe`, 117
golden section search, 126
golf ball, 118
gravity, 96, 105
Green Monster, 118

handle
    function, 59, 78, 89
`dhelpoc`, 15
`help`, 45
Help Window, 15
helper function, 65
`hold`, 22
`horzcat`, 110
human cannonball, 111
hypothesis, 72

`if` statement, 30
incremental development, 25, 48, 65
indentation, 30
index, 35, 85
    `end`, 80
`Inf`, 14
initial condition, 77, 89, 106, 114
input variable, 43, 46, 66, 81
internal validation, 2
interpreter, 2
interval, 31, 61, 80
invalid expression, 8
inverse quadratic interpolation , 125
iterative modeling, 2

kinetic energy, 119

labeling axes, 81

www.dbooks.org

launch angle, 113, 114, 118
Law of Universal Gravitation, 100
Law of universal gravitation, 119
leap of faith, 53
`legend`, 82
`length` function, 35, 37
linear algebra, 33, 87
linear differential equation, 75
logical error, 20
logical function, 47
logical operator, 31
logical value, 30, 47
logical vector, 71
logistic map, 41
loop, 21, 23, 35, 36
    nested, 49
loop body, 21
loop variable, 21
Lorenz attractor, 40, 92
Lotka-Volterra model, 87

M-file, 43
m-file, 11
magic square, 85
magnitude, 103
Magnus force, 118
Manny Ramirez, 118
mass, 99, 119
math function
    exponential, 4
    logarithm, 4
    trigonometric, 4
mathematical function
    square root, 4
Matrices
    transpose, 87
matrix, 5, 32, 33, 85, 97
matrix exponentiation, 67
matrix multiplication, 34
mechanics
    celestial, 119
minimum, 126
model, 2
modeling, 1
multiplication, 3
    matrix, 34
myth, 10

name
    function, 45
    variable, 6
name collision, 38, 44, 46, 62
NaN, 30
`NaN`, 14
Nelder-Mead, 126
nested function call, 4
nested loop, 49
nesting, 30, 31
Newton, 2
Newton's law of cooling, 82
Newton's law of motion, 95
Newtonian motion, 95
nonlinear equation, 57
`norm`, 104
not a number, 14, 30
number
    floating-point, 13, 33
numerator, 8
numerical error, 20
numerical method, 58
numerical solution, 57

ODE event, 97, 113
ODE events, 97
`ode23`, 120
`ode45`, 78, 88, 117, 119, 123
`odeset`, 98, 113, 120, 123
`ones`, 33
operand, 3–5
Operations
    relational, 30
operations
    order of, 3
operator, 3
    AND, 32
    assignment, 6
    colon, 21, 29
    comma, 109
    elementwise, 34, 62
    logical, 31

OR, 32
relational, 30
optimization, 113, 122
`options`, 98
OR operator, 32
orbit, 2
order of operations, 3, 9
ordinary differential equation, 75
output
suppress, 6
output argument, 61
output variable, 43, 51, 60, 66, 78, 81, 106

pack vector, 88
parachute, 100
parameter, 79, 89, 122
parentheses, 3, 8, 35
`pause`, 118
penny, 10, 96
percent sign, 15
phase plot, 91
plot
`ezplot`, 61
`plot`, 22, 36
`plot3`, 91
Plotting
points, 22
plotting vector, 36
`pol2cart`, 115
polar coordinates, 115
position, 96, 103, 119
postcondition, 15, 29, 45
potential energy, 119
precondition, 15, 29, 45
predefined variable, 5
prediction, 2
`prod` function, 38
product
cumulative, 69
prompt, 2
Pythagorean theorem, 103
Pythagorean triple, 48, 55

quantification
existential, 69
universal, 70

rabbit, 87
radian, 115
Ramirez, Manny, 118
random walk programming, 72
range, 21, 29, 36, 113, 114, 122
rate function, 78, 80, 88, 104, 123
rate function , 96
reading, 71
realism, 2
`realmax`, 14
`realmin`, 14
recurrent computation, 23
reduce, 37
relational operator, 30
relative error, 20
relativity, 2
`RelTol`, 120
`res`, 43
retreating, 71
`return` statement, 70
root, 58
row, 85
row vectors, 86
ruminating, 71
Runge-Kutta, 123
running, 71
runtime error, 20

`saveas`, 82
scaffolding, 25
scalar, 32, 37
scientific notation, 14
script, 11, 39, 43
reasons for, 12
script file name, 12
Scripts
M-files, 11
search, 37
search path, 12
secant method, 125
second derivative, 95
second-order differential equation, 95

www.dbooks.org

semi-colon, 6, 13, 86, 89
sequence, 23, 36, 103
    Fibonacci, 12
series, 23
`sgn`, 99
shadow, 62
signum function, 99
silent function, 44
simplicity, 2
`size`, 85
skydiver, 100
spatial vector, 103, 119
sphere, 64
Spider-Man, 121
spring constant, 121, 122
square root, 40, 57
state, 91, 103
statement
    `break`, 38
    `end`, 21
    `return`, 70
    assignment, 6, 21
    compound, 30
step size, 118
string, 6
    style, 37
style string, 22, 37
sum, 23
    cumulative, 68
`sum`, 66
`sum` function, 38
Sun, 100, 119
suppress output, 6
syntax
    . . . , 7
    semi-colon, 6
syntax error, 20
system, 1
system of equations, 88
system of ODEs, 87

tea, 120
terminal velocity, 99, 122
theorems of debugging, 131
time dependence, 79
time series, 103
time span, 80, 97, 106, 117
time step, 76, 78, 124
tolerance, 120
top-level function, 65, 76
trajectory, 91, 114, 122
transcendental number, 14
transpose operator, 87
trigonometry, 4

undefined operation, 14
underscore, 6
unit, 15
unit vector, 104, 107
universal gravitation constant, 119
universal quantification, 70
unpack vector, 88
update, 19

validation, 2
    external, 2
    internal, 2
value
    logical, 30, 47
variable, 5, 6, 33
    assignment, 16
    flag, 31
    input, 43, 46
    loop, 21
    output, 43, 51, 60, 78
    predefined, 5
    reasons for, 7
variable name, 6, 15, 32
vector, 5, 29, 32, 36, 37, 59, 66
    column, 86
    logical, 71
    plotting, 36
    row, 86
    spatial, 103
    unit, 104
vector addition, 121
vector arithmetic, 32
vectorizing, 62, 67
velocity, 96, 105, 113, 118, 119
`vertcat`, 109

vertical concatenation, 109

weasel, 87
`who`, 6
`whos`, 85
`whos` command, 33
Workspace, 2
workspace, 6, 13, 38, 44, 53

`xlabel`, 81

`ylabel`, 81

zero
    division by, 14
zero-finding, 58

www.dbooks.org