# Physical Modeling in MATLAB®

Allen B. Downey

Version 1.1.6

Physical Modeling in MATLAB®

Copyright 2011 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 3.0 Unported License, which is available at `http://creativecommons.org/licenses/by-nc/3.0/`.

The original form of this book is LaTeX source code. Compiling this code has the effect of generating a device-independent representation of a textbook, which can be converted to other formats and printed.

This book was typeset by the author using latex, dvips and ps2pdf, 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

Most books that use MATLAB are aimed at readers who know how to program. This book is for people who have never programmed before.

As a result, the order of presentation is unusual. The book starts with scalar values and works up to vectors and matrices very gradually. This approach is good for beginning programmers, because it is hard to understand composite objects until you understand basic programming semantics. But there are problems:

- The MATLAB documentation is written in terms of matrices, and so are the error messages. To mitigate this problem, the book explains the necessary vocabulary early and deciphers some of the messages that beginners find confusing.
- Many of the examples in the first half of the book are not idiomatic MATLAB. I address this problem in the second half by translating the examples into a more standard style.

The book puts a lot of emphasis on functions, in part because they are an important mechanism for controlling program complexity, and also because they are useful for working with MATLAB tools like `fzero` and `ode45`.

I assume that readers know calculus, differential equations, and physics, but not linear algebra. I explain the math as I go along, but the descriptions might not be enough for someone who hasn't seen the material before.

There are small exercises within each chapter, and a few larger exercises at the end of some chapters.

If you have suggestions and corrections, please send them to `downey@allendowney.com`.

Allen B. Downey
Needham, MA

## Contributor's list

The following are some of the people who have contributed to this book:

- Michael Lintz spotted the first (of many) typos.
- Kaelyn Stadtmueller reminded me of the importance of linking verbs.
- Roydan Ongie knows a matrix when he sees one (and caught a typo).
- Keerthik Omanakuttan knows that acceleration is not the second derivative of acceleration.
- Pietro Peterlongo pointed out that Binet's formula is an exact expression for the $n$th Fibonacci number, not an approximation.
- Li Tao pointed out several errors.
- Steven Zhang pointed out an error and a point of confusion in Chapter 11.
- Elena Oleynikova pointed out the "gotcha" that script file names can't have spaces.
- Kelsey Breseman pointed out that numbers as footnote markers can be confused with exponents, so now I am using symbols.
- Philip Loh sent me some updates for recent revisions of MATLAB.
- Harold Jaffe spotted a typo.
- Vidie Pong pointed out the problem with spaces in filenames.

# Contents

**Preface** **iii**

**1 Variables and values** **1**

1.1 A glorified calculator . . . 1
1.2 Math functions . . . 2
1.3 Documentation . . . 3
1.4 Variables . . . 4
1.5 Assignment statements . . . 5
1.6 Why variables? . . . 6
1.7 Errors . . . 7
1.8 Floating-point arithmetic . . . 8
1.9 Comments . . . 9
1.10 Glossary . . . 10
1.11 Exercises . . . 11

**2 Scripts** **13**

2.1 M-files . . . 13
2.2 Why scripts? . . . 14
2.3 The workspace . . . 15
2.4 More errors . . . 16
2.5 Pre- and post-conditions . . . 17
2.6 Assignment and equality . . . 17

2.7 Incremental development . . . . . 18
2.8 Unit testing . . . . . 19
2.9 Glossary . . . . . 20
2.10 Exercises . . . . . 20

**3 Loops** **23**

3.1 Updating variables . . . . . 23
3.2 Kinds of error . . . . . 24
3.3 Absolute and relative error . . . . . 24
3.4 for loops . . . . . 25
3.5 plotting . . . . . 26
3.6 Sequences . . . . . 27
3.7 Series . . . . . 28
3.8 Generalization . . . . . 28
3.9 Glossary . . . . . 29
3.10 Exercises . . . . . 30

**4 Vectors** **33**

4.1 Checking preconditions . . . . . 33
4.2 `if` . . . . . 34
4.3 Relational operators . . . . . 35
4.4 Logical operators . . . . . 35
4.5 Vectors . . . . . 36
4.6 Vector arithmetic . . . . . 37
4.7 Everything is a matrix . . . . . 37
4.8 Indices . . . . . 39
4.9 Indexing errors . . . . . 40
4.10 Vectors and sequences . . . . . 41
4.11 Plotting vectors . . . . . 42
4.12 Reduce . . . . . 42

4.13 Apply . . . 43
4.14 Search . . . 43
4.15 Spoiling the fun . . . 45
4.16 Glossary . . . 45
4.17 Exercises . . . 46

**5 Functions** **49**
5.1 Name Collisions . . . 49
5.2 Functions . . . 50
5.3 Documentation . . . 51
5.4 Function names . . . 52
5.5 Multiple input variables . . . 53
5.6 Logical functions . . . 54
5.7 An incremental development example . . . 55
5.8 Nested loops . . . 56
5.9 Conditions and flags . . . 57
5.10 Encapsulation and generalization . . . 58
5.11 A misstep . . . 60
5.12 `continue` . . . 61
5.13 Mechanism and leap of faith . . . 62
5.14 Glossary . . . 63
5.15 Exercises . . . 63

**6 Zero-finding** **65**
6.1 Why functions? . . . 65
6.2 Maps . . . 65
6.3 A note on notation . . . 66
6.4 Nonlinear equations . . . 66
6.5 Zero-finding . . . 68
6.6 `fzero` . . . 69

6.7 What could go wrong? . . . . . 71
6.8 Finding an initial guess . . . . . 72
6.9 More name collisions . . . . . 73
6.10 Debugging in four acts . . . . . 74
6.11 Glossary . . . . . 75
6.12 Exercises . . . . . 76

**7 Functions of vectors** **79**
7.1 Functions and files . . . . . 79
7.2 Physical modeling . . . . . 80
7.3 Vectors as input variables . . . . . 81
7.4 Vectors as output variables . . . . . 82
7.5 Vectorizing your functions . . . . . 82
7.6 Sums and differences . . . . . 84
7.7 Products and ratios . . . . . 85
7.8 Existential quantification . . . . . 85
7.9 Universal quantification . . . . . 86
7.10 Logical vectors . . . . . 87
7.11 Glossary . . . . . 88

**8 Ordinary Differential Equations** **89**
8.1 Differential equations . . . . . 89
8.2 Euler's method . . . . . 90
8.3 Another note on notation . . . . . 91
8.4 `ode45` . . . . . 92
8.5 Multiple output variables . . . . . 94
8.6 Analytic or numerical? . . . . . 95
8.7 What can go wrong? . . . . . 96
8.8 Stiffness . . . . . 98
8.9 Glossary . . . . . 99
8.10 Exercises . . . . . 100

**9 Systems of ODEs** **103**

9.1 Matrices . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
9.2 Row and column vectors . . . . . . . . . . . . . . . . . . . . . 104
9.3 The transpose operator . . . . . . . . . . . . . . . . . . . . . . 105
9.4 Lotka-Voltera . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
9.5 What can go wrong? . . . . . . . . . . . . . . . . . . . . . . . 108
9.6 Output matrices . . . . . . . . . . . . . . . . . . . . . . . . . 108
9.7 Glossary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
9.8 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110

**10 Second-order systems** **113**

10.1 Nested functions . . . . . . . . . . . . . . . . . . . . . . . . . 113
10.2 Newtonian motion . . . . . . . . . . . . . . . . . . . . . . . . 114
10.3 Freefall . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
10.4 Air resistance . . . . . . . . . . . . . . . . . . . . . . . . . . . 116
10.5 Parachute! . . . . . . . . . . . . . . . . . . . . . . . . . . . . 118
10.6 Two dimensions . . . . . . . . . . . . . . . . . . . . . . . . . 118
10.7 What could go wrong? . . . . . . . . . . . . . . . . . . . . . . 120
10.8 Glossary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122
10.9 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122

**11 Optimization and Interpolation** **125**

11.1 ODE Events . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125
11.2 Optimization . . . . . . . . . . . . . . . . . . . . . . . . . . . 126
11.3 Golden section search . . . . . . . . . . . . . . . . . . . . . . 127
11.4 Discrete and continuous maps . . . . . . . . . . . . . . . . . . 130
11.5 Interpolation . . . . . . . . . . . . . . . . . . . . . . . . . . . 131
11.6 Interpolating the inverse function . . . . . . . . . . . . . . . . 132
11.7 Field mice . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 134
11.8 Glossary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 135
11.9 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 135

**12 Vectors as vectors** **137**

12.1 What's a vector? . . . . . . . . . . . . . . . . . . . . . . . . . . 137
12.2 Dot and cross products . . . . . . . . . . . . . . . . . . . . . . . . 139
12.3 Celestial mechanics . . . . . . . . . . . . . . . . . . . . . . . . . . 140
12.4 Animation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 141
12.5 Conservation of Energy . . . . . . . . . . . . . . . . . . . . . . . . 142
12.6 What is a model for? . . . . . . . . . . . . . . . . . . . . . . . . . 143
12.7 Glossary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 144
12.8 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 144

# Chapter 1

# Variables and values

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

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

```
>>
```

which is the MATLAB **prompt**; that is, this symbol prompts you to enter a command.

The simplest kind of command is a mathematical **expression**, which is made up of **operands** (like numbers, for example) and **operators** (like the plus sign, `+`).

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 the example above, MATLAB printed `>>`; I typed `2 + 1` and then hit Enter, and MATLAB printed `ans = 3`. And when I say "printed," I really mean "displayed on the screen," which might be confusing, but it's the way people talk.

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
```

Speaking of spaces, you might have noticed that MATLAB puts some space between `ans =` and the result. In my examples I will leave it out to save paper.

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

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

The order of operations is what you would expect from basic algebra: 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!

The other 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
```

As in basic algebra, exponentiation happens before multiplication and division, but again, you can use parentheses to override the order of operations.

## 1.2 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**. All the trig functions in MATLAB work in radians.

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.

## 1.3 Documentation

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

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

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

    See also asin, sind.
```

```
    Overloaded functions or methods (ones with the same name in other
       directories) help sym/sin.m

    Reference page in Help browser
       doc sin
```

Unfortunately, this documentation is not beginner-friendly.

One gotcha is that the name of the function appears in the help page in capital letters, but if you type it like that in MATLAB, you get an error:

```
>> SIN(1)
??? Undefined command/function 'SIN'.
```

Another problem is that the help page uses vocabulary you don't know yet. For example, "the elements of X" won't 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 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 sense yet, but you can get a preview of what's coming.

## 1.4 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 $\pi$, 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
```

*Technically `pi` is a function, not a variable, but for now it's best to pretend.

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.

## 1.5 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 = 'allen'
first_name = allen
```

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**.

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 it is probably not a good idea to change the value of `pi`!

## 1.6 Why variables?

The most common reasons to use variables are

- To avoid recomputing a value that is used repeatedly. For example, if you are performing computations involving $e$, you might want to compute it once and save the result.

  ```
  >> 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 $\pi r^2$.

- 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 (see the First Theorem of Debugging).

## 1.7 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.

When you make deliberate errors, you get to see what the error messages look like. 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: Unexpected MATLAB expression.
```

The error message indicates that, after seeing the operand `pi`, MATLAB was "expecting" to see an operator, like `*`. Instead, it got a variable name, which is the "unexpected expression" indicated by the vertical line, `|` (which is called a "pipe").

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
??? Function 'sin' is not defined for values of class '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 `sin` function generates a reasonable error message, but in other cases the results can be baffling. For example, what do you think is going on here?

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

There is a reason for this "feature", but rather than get into that now, let me 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 hate error messages and do everything they can to make them go away. Experienced programmers know that error messages are your friend. They can be hard to understand, and even misleading, but it is worth making some effort to understand them.

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

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

You might be tempted to write something like this:

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

But that would be wrong. So very wrong.

## 1.8 Floating-point arithmetic

In mathematics, there are several kinds of numbers: integer, real, rational, irrational, imaginary, complex, etc. MATLAB only has one kind of number, called **floating-point**.

You might have noticed that MATLAB expresses values in decimal notation. So, for example, the rational number 1/3 is represented by the floating-point value

```
>> 1/3
ans = 0.3333
```

which is only approximately correct. It's not quite as bad as it seems; MATLAB uses more digits than it shows by default. You can change the `format` to see the other digits.

```
>> format long
>> 1/3
ans = 0.333333333333333
```

Internally, MATLAB uses the IEEE double-precision floating-point format, which provides about 15 significant digits of precision (in base 10). Leading and trailing zeros don't count as "significant" digits, so MATLAB can represent large and small numbers with the same precision.

Very large and very small values are displayed in scientific notation.

```
>> 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 $9.33 \times 10^{157}$. The exact solution is a 158-digit integer, but 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 MATLAB can handle large numbers, there is a limit. The predefined variables `realmax` and `realmin` contain the largest and smallest numbers that 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`, but in this case MATLAB gives you a warning because division by zero is usually considered undefined.

```
>> 1/0
Warning: Divide by zero.

ans = Inf
```

A warning is like an error message without teeth; the computation is allowed to continue. Allowing `Inf` to propagate through a computation doesn't always do what you expect, but if you are careful with how you use it, `Inf` can be quite useful.

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

```
>> 0/0
Warning: Divide by zero.

ans = NaN
```

## 1.9 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
```

[†]The names of these variables are misleading; floating-point numbers are sometimes, wrongly, called "real".

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, but 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 tradeoff: 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.

## 1.10 Glossary

**interpreter:** The program that reads and executes MATLAB code.

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

**prompt:** The symbol 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.

**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).

**floating-point:** The kind of number MATLAB works with. All floating-point numbers can be represented with about 16 significant decimal digits (unlike mathematical integers and reals).

**scientific notation:** A format for typing and displaying large and small numbers; e.g. `3.0e8`, which represents $3.0 \times 10^8$ or 300,000,000.

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

## 1.11 Exercises

**Exercise 1.1** *Write a MATLAB expression that evaluates the following math expression. You can assume that the variables* `mu`, `sigma` *and* `x` *already exist.*

$$\frac{e^{-\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)^2}}{\sigma\sqrt{2\pi}} \tag{1.1}$$

*Note: you can't use Greek letters in MATLAB; when translating math expressions with Greek letters, it is common to write out the name of the letter (assuming you know it).*

# Chapter 2

# Scripts

## 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 selecting New→Script from the File menu. A window appears running a text editor specially designed for MATLAB.

Type the following code in the editor

```
x = 5
```

and then press the (outdated) floppy disk icon, or select Save from the File menu. Either way, a dialog box appears where you can choose the file name and the directory where it should go. Change the name to `myscript.m` and leave the directory unchanged.

By default, MATLAB will store your script in a directory that is on the **search path**, which is the list of directories MATLAB searches for scripts.

Go back to the Command Window and type `myscript` (without the extension) at the prompt. MATLAB executes your script and displays the result.

```
>> 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.

If something goes wrong and MATLAB can't find your script, you will get an error message like:

```
>> myscript
??? Undefined function or variable 'myscript'.
```

In this case you can either save your script again in a directory that is on the search path, or modify the search path to include the directory where you keep your scripts. You'll have to consult the documentation for the details (sorry!).

The filename can be anything you want, but you should try to choose something meaningful and memorable. You should be very careful to choose a name that is not already in use; if you do, you might accidentally replace one of MATLAB's functions with your own. Finally, the name of the file cannot contain spaces. If you create a file named `my script.m`, MATLAB doesn't complain until you try to run it:

```
>> my script
??? Undefined function or method 'my' for input arguments
of type 'char'.
```

The problem is that it is looking for a scipt named `my`. The problem is even worse if the first word of the filename is a function that exists. Just for fun, create a script named `abs val.m` and run it.

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

**Exercise 2.1** *The Fibonacci sequence, denoted $F$, is described by the equations $F_1 = 1$, $F_2 = 1$, and for $i \geq 3$, $F_i = F_{i-1} + F_{i-2}$. The elements of this sequence occur naturally in many plants, particularly those with petals or scales arranged in the form of a logarithmic spiral.*

*The following expression computes the nth Fibonacci number:*

$$F_n = \frac{1}{\sqrt{5}} \left[ \left( \frac{1+\sqrt{5}}{2} \right)^n - \left( \frac{1-\sqrt{5}}{2} \right)^n \right] \tag{2.1}$$

*Translate this expression into MATLAB and store your code in a file named* `fibonacci1`*. At the prompt, set the value of* `n` *to 10 and then run your script. The last line of your script should assign the value of $F_n$ to* `ans`*. (The correct value of $F_{10}$ is 55).*

## 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.

On the other hand, it can be a pain to switch back and forth between the Command Window and the Editor. Try to arrange your windows so you can see the Editor and the Command Window at the same time, and use the Tab key or the mouse to switch between them.

- 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!

Unfortunately, the great power of scripts comes with great responsibility, which is that you have to make sure that the code you are running is the code you think you are running.

First, whenever you edit your script, you have to save it before you run it. If you forget to save it, you will be running the old version.

Also, 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. MATLAB comes with a lot of predefined functions. It is easy to write a script that has the same name as a MATLAB function, and if you are not careful, you might find yourself running the MATLAB function instead of your script.

Either way, 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:

> You must always be 100% sure that the code you are running is the code you think you are running.

## 2.3 The workspace

The variables you create are stored in 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 variables.

```
>> clear y
>> who

Your variables are:

x  z
```

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
```

(Strictly speaking, the name of a variable is an expression, so evaluating it should assign a value to `ans`, but MATLAB seems to handle this as a special case.)

## 2.4 More errors

Again, when you try something new, you should make a few mistakes on purpose so you'll recognize them later.

The most common error with scripts is to run a script without creating the necessary variables. For example, `fibonacci1` requires you to assign a value to `n`. If you don't:

```
>> fibonacci1
??? Undefined function or variable "n".

Error in ==> fibonacci1 at 4
diff = t1^(n+1) - t2^(n+1);
```

The details of this message might be different for you, depending on what's in your script. But the general idea is that `n` is undefined. Notice that MATLAB tells you what line of your program 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; it is, in a sense, in the workspace.

Which brings us to the Fourth Theorem of Debugging:

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

The object of the game is to find the cause and fix it—not just to make the error message go away.

## 2.5 Pre- and post-conditions

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`:

```
% Computes 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 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.

## 2.6 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: The expression to the left of the equals sign is not a valid
target for an assignment.
```

In this case the error message is pretty helpful, as long as you know what a "target" is.

Another difference 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. For example, $y = y + 1$ is a statement that happens to be false for all real 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."

To test your understanding of assignment statements, try this exercise:

**Exercise 2.2** *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.*

## 2.7 Incremental development

When you start writing scripts that are more than a few lines, you might find yourself spending more and 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 lots of 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.

When this process works, you will find that your changes usually work the first time, or the problem is obvious. That's a good thing, and it brings us to the Fifth 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 spend on scaffolding.
- When you are getting started, it is usually not 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 5.7.

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.

## 2.8 Unit testing

In large software projects, **unit testing** is the process of testing software components in isolation before putting them together.

The programs we have seen so far are not big enough to need unit testing, but the same principle applies 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. Also, we 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 unit 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 Sixth Theorem of Debugging:

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

## 2.9 Glossary

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

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

**search path:** The list of directories 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.

**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.

**unit testing:** A process of testing software by testing each component in isolation.

## 2.10 Exercises

**Exercise 2.3** *Imagine that you are the owner of a car rental company with two locations, Albany and Boston. Some of your customers do "one-way rentals," picking up a car in Albany and returning it in Boston, or the other way around. Over time, you have observed that each week 5% of the cars in Albany are*

*dropped off in Boston, and 3% of the cars in Boston get dropped off in Albany. At the beginning of the year, there are 150 cars at each location.*

*Write a script called* `car_update` *that updates the number of cars in each location from one week to the next. The precondition is that the variables* `a` *and* `b` *contain the number of cars in each location at the beginning of the week. The postcondition is that* `a` *and* `b` *have been modified to reflect the number of cars that moved.*

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

*Note: cars are countable things, so* `a` *and* `b` *should always be integer values. You might want to use the* `round` *function to compute the number of cars that move during each week.*

*If you execute your script repeatedly, you can simulate the passage of time from week to week. What do you think will happen to the number of cars? Will all the cars end up in one place? Will the number of cars reach an equilibrium, or will it oscillate from week to week?*

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

# Chapter 3

# Loops

## 3.1 Updating variables

In Exercise 2.3, you might have been tempted to write something like

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

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 Cars!

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

```
anew = a - 0.05*a + 0.03*b
bnew = b + 0.05*a - 0.03*b
a = anew
b = bnew
```

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:

```
atob = 0.05*a - 0.03*b
a = a - atob
b = b + atob
```

It is easy to look at this code and confirm that it obeys Conversation of Cars. Even if the value of `atob` is wrong, at least the total number of cars is right. And that brings us to the Seventh 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 Kinds of error

There are four kinds of error:

**Syntax error:** You have written a MATLAB 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 MATLAB 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 MATLAB 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 `a` 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 roundoff 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.

Run time 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 roundoff 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**.

An absolute error is just the difference between the correct value and the approximation. We usually write the magnitude of the error, ignoring its sign, because it doesn't matter whether the approximation is too high or too low.

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 $359,536.87$. The absolute error is 3,343.13.

At first glance, that sounds like a lot—we're off by three thousand—but it is worth taking into account 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, which is the error expressed as a fraction (or percentage) of the exact value. 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 `car_update` that simulates one week in the life of a rental car company. To simulate an entire year, we have to run it 52 times:

```
for i=1:52
    car_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 52.

The colon operator, `:`, specifies a **range** of integers. In the spirit of unit testing, 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 a common convention 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 actually affect the execution of the program. The end of the loop is officially marked by the `end` statement.

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 3.1** *Create a script named* `car_loop` *that uses a* `for` *loop to run* `car_update` *52 times. Remember that before you run* `car_update`*, you have to assign values to* `a` *and* `b`*. For this exercise, start with the values* `a = 150` *and* `b = 150`*.*

*If everything goes smoothly, your script will display a long stream of numbers on the screen. But 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)
```

A Figure Window should appear with a graph and a single, blue dot at $x$ position 1 and $y$ position 2. To make the dot more visible, you can specify a different shape:

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

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

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:

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

You should see a figure with two points. MATLAB scales the plot automatically so that the axes run from the lowest value in the plot to the highest. So in this example the points are plotted in the corners.

**Exercise 3.2** *Modify* `car_loop` *so that each time through the loop it plots the value of* `a` *versus the value of* `i`*.*

*Once you get that working, modify it so it plots the values of* `a` *with red circles and the values of* `b` *with blue diamonds.*

*One more thing: if you use* `hold on` *to prevent MATLAB from clearing the figure, you might want to clear the figure yourself, from time to time, with the command* `clf`*.*

## 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 all of 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 $i$th element **directly**, as a function of $i$, without using the previous element. In math notation, $A_i = A_1 r^{i-1}$.

**Exercise 3.3** *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

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

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

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

`A1` is the first element of the sequence, so each time through the loop `a` is the $i$th 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 $i$th 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 3.4** *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 could replace the constant, 10, with a variable named `n`:

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

Now the script can compute any number of terms, with the precondition that you have to set `n` before you execute the script. Here's how you could run 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 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`.

**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.

## 3.10 Exercises

**Exercise 3.5** *We have already seen the Fibonacci sequence, $F$, which is defined recurrently as*

$$F_i = F_{i-1} + F_{i-2}$$

*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 2.*

*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!*

**Exercise 3.6** *Write a script named* `fib_plot` *that loops* `i` *through a range from 1 to 20, uses* `fibonacci2` *to compute Fibonacci numbers, and plots* $F_i$ *for each* $i$ *with a series of red circles.*

# Chapter 4

# Vectors

## 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 = Empty matrix: 1-by-0
```

If the matrix is empty, you might expect it to be "0-by-0," 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`

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, and 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, well, `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." (In some logic notations, the tilde is the symbol for "not.")

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 a syntax error:

```
if x=5
??? if x=5
        |
Error: The expression to the left of the equals sign is not a valid
target for an assignment.
```

MATLAB thinks you are making an assignment to a variable named `if x`!

## 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 a set of numbers that correspond to positive integers. What we called a “range” in the previous chapter was actually a vector.

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 = 1:5

X = 1     2     3     4     5
```

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 scalar and what is a vector.

Just as with sequences, the numbers that make up the vector are called **elements**.

## 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     9    10
```

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    13    15
```

But adding vectors only works if the operands are the same size. Otherwise:

```
>> W = 1:3

W = 1     2     3

>> X+W
??? Error using ==> plus
Matrix dimensions must agree.
```

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.

You can see this if you use the `whos` command to display the variables in the workspace. `whos` is similar to `who` except that it also displays the size and type of each variable.

First I'll make one of each kind of value:

```
>> scalar = 5

scalar = 5

>> vector = 1:5

vector = 1     2     3     4     5

>> matrix = ones(2,3)

matrix =

     1     1     1
     1     1     1
```

`ones` is a function that 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

  scalar       1x1                            8  double array
  vector       1x5                           40  double array
  matrix       2x3                           32  double array
```

According to MATLAB, everything is a double array: "double" is another name for double-precision floating-point numbers, and "array" is another name for a matrix.

The only difference is the size, which is specified by the number of rows and columns. The thing we called `scalar` is, according to MATLAB, a matrix with one row and one column. Our `vector` is really a matrix with one row and 5 columns. And, of course, `matrix` is a matrix.

The point of all this is that you can think of your values as scalars, vectors, and matrices, and I think you should, as long as you remember that MATLAB thinks everything is a matrix.

Here's another example where the error message only makes sense if you know what is happening under the hood:

```
>> X = 1:5

X = 1     2     3     4     5
```

```
>> Y = 1:5

Y = 1     2     3     4     5

>> Z = X*Y
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

First of all, `mtimes` is the MATLAB function that performs matrix multiplication. The reason the "inner matrix dimensions must agree" is that the way matrix multiplication is defined in linear algebra, the number of rows in `X` has to equal the number of columns in `Y` (those are the inner dimensions).

If you don't know linear algebra, this doesn't make much sense. When you saw `X*Y` you probably expected it to multiply each the the elements of `X` by the corresponding element of `Y` and put the results into a new vector. That operation is called **elementwise** multiplication, and the operator that performs it is `.*`:

```
>> X .* Y

ans = 1     4     9    16    25
```

We'll get back to the elementwise operators later; you can forget about them for now.

## 4.8 Indices

You can select elements of a vector with parentheses:

```
>> Y = 6:10

Y = 6     7     8     9    10

>> Y(1)

ans = 6

>> Y(5)

ans = 10
```

This means that the first element of `Y` is 6 and the fifth element is 10. 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
```

Loops and vectors go together like the storm and rain. For example, this loop displays the elements of `Y`.

```
for i=1:5
     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
```

There. Now that will work for a vector of any length.

## 4.9 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 this:

```
>> Y(0)
??? Subscript indices must either be real positive integers or
logicals.
```

"Subscript indices" is MATLAB's longfangled way to say "indices." "Real positive integers" means that complex numbers are out. And you can forget about "logicals" for now.

If the index is too big, you get this:

```
>> Y(6)
??? Index exceeds matrix dimensions.
```

There's the "m" word again, but other than that, this message is pretty clear.

Finally, don't forget that the index has to be an integer:

```
>> Y(1.5)
??? Subscript indices must either be real positive integers or
logicals.
```

## 4.10 Vectors and sequences

Vectors and sequences go together like ice cream and apple pie. 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 $i \geq 3$. In MATLAB, that looks like

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

Notice that I am using a capital letter for the vector `F` and lower-case letters for the scalars `i` and `n`. At the end, the script extracts the final element of `F` and stores it in `ans`, since the result of this script is supposed to be the $n$th Fibonacci number, not the whole sequence.

If you had any trouble with Exercise 3.5, 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. The only drawbacks are

- You have to be careful with the range of the loop. In this 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`:

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

  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 really only want the $n$th Fibonacci number, then storing the whole sequence wastes some storage space. But if wasting space makes your code easier to write and debug, that's probably ok.

**Exercise 4.1** *Write a loop that computes the first* `n` *elements of the geometric sequence* $A_{i+1} = A_i/2$ *with* $A_1 = 1$*. Notice that the 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.*

## 4.11 Plotting vectors

Plotting and vectors go together like the moon and June, whatever that means. If you call `plot` with a single 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.

If you call `plot` with two vectors as arguments, MATLAB plots the second one as a function of the first; that is, it treats the first vector as a sequence of $x$ values and the second as corresponding $y$ value and plots a sequence of $(x, y)$ points.

```
X = 1:5
Y = 6:10
plot(X, Y)
```

By default, MATLAB draws a blue line, but you can override that setting with the same kind of string 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.

In this example, I stuck with the convention of naming the first argument `X` (since it is plotted on the $x$-axis) and the second `Y`. There is nothing special about these names; you could just as well plot `X` as a function of `Y`. MATLAB always treats the first vector as the "independent" variable, and the second as the "dependent" variable (if those terms are familiar to you).

## 4.12 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.

**Exercise 4.2** *Write a similar loop that multiplies all the elements of a vector together. You might want to call the accumulator* `product`*, and you might want to think about the initial value you give it before the loop.*

## 4.13 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
```

**Exercise 4.3** *Write a loop that computes a vector* `Y` *that contains the sines of the elements of* `X`*. To test your loop, write a script that*

1. *Uses* `linspace` *(see the documentation) to assign to* `X` *a vector with 100 elements running from 0 to* $2\pi$*.*
2. *Uses your loop to store the sines in* `Y`*.*
3. *Plots the elements of* `Y` *as a function of the elements of* `X`*.*

## 4.14 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, if a vector contains the computed altitude of a falling object, you might want to know the index where the object touches down (assuming that the ground is at altitude 0).

To create some fake data, we'll use an extended version of the colon operator:

```
X = 10:-1:-10
```

The values in this range run from 10 to -10, with a **step size** of -1. The step size is the interval between elements of the range.

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
```

One 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 script ends).

This example demonstrates the basic idea of a search, but it also demonstrates a dangerous use of the `if` statement. Remember that floating-point values are often only approximately right. That means that if you look for a perfect match, you might not find it. For example, try this:

```
X = linspace(1,2)
for i=1:length(X)
    Y(i) = sin(X(i))
end
plot(X, Y)
```

You can see in the plot that the value of $\sin x$ goes through 0.9 in this range, but if you search for the index where `Y(i) == 0.9`, you will come up empty.

```
for i=1:length(Y)
    if Y(i) == 0.9
        ans = i
        break
    end
end
```

The condition is never true, so the body of the `if` statement is never executed.

Even though the plot shows a continuous line, don't forget that `X` and `Y` are sequences of discrete (and usually approximate) values. As a rule, you should (almost) never use the `==` operator to compare floating-point values. There are a number of ways to get around this limitation; we will get to them later.

**Exercise 4.4** *Write a loop that finds the index of the first negative number in a vector and stores it in* `ans`*. If there are no negative numbers, it should set* `ans` *to -1 (which is not a legal index, so it is a good way to indicate the special case).*

## 4.15 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 4.13

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

Also, most built-in MATLAB functions work with vectors:

```
X = linspace(0, 2*pi)
Y = sin(X)
plot(X, Y)
```

Finally, the `find` function can perform search operations, but understanding it requires a couple of concepts we haven't got to, so for now you are better off on your own.

I started with simple loops because I wanted to demonstrate the basic concepts and give you a chance to practice. At some point you will probably have to write a loop for which there is no MATLAB shortcut, but you have to work your way up from somewhere.

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.

**Exercise 4.5** *Write an expression that computes the sum of the squares of the elements of a vector.*

## 4.16 Glossary

**compound:** 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).

**index:** 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.

## 4.17 Exercises

**Exercise 4.6** *The ratio of consecutive Fibonacci numbers, $F_{n+1}/F_n$, converges to a constant value as $n$ increases. Write a script that computes a vector with the first $n$ elements of a Fibonacci sequence (assuming that the variable* `n` *is defined), and then computes a new vector that contains the ratios of consecutive Fibonacci numbers. Plot this vector to see if it seems to converge. What value does it converge on?*

**Exercise 4.7** *A certain famous system of differential equations can be approximated by a system of difference equations that looks like this:*

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

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

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

- *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 time step $dt = 0.01$.*

- *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 $\sigma$, $b$ and $r$ and see if you can get a sense of how each variable affects the system.*

**Exercise 4.8** *The logistic map is often cited as an example of how complex, chaotic behaviour can arise from simple non-linear dynamical equations [some of this description is adapted from the Wikipedia page on the logistic map]. It was popularized in a seminal 1976 paper by the biologist Robert May.*

*It has been used to model the biomass of a species in the presence of limiting factors such as food supply and disease. In this case, there are two processes at work: (1) A reproductive process increases the biomass of the species in proportion to the current population. (2) A starvation process causes the biomass to decrease at a rate proportional to the carrying capacity of the environment less the current population.*

*Mathematically this can be written as*

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

*where $X_i$ is a number between zero and one that represents the biomass at year $i$, and $r$ is a positive number that represents a combined rate for reproduction and starvation.*

- *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 population.*

- *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 $r$.*

- *One way to characterize the effect of $r$ is to make a plot with $r$ on the x-axis and biomass on the y axis, and to show, for each value of $r$, the values of biomass that occur in steady state. See if you can figure out how to generate this plot.*

# Chapter 5

# Functions

## 5.1 Name Collisions

Remember that all of your scripts run in the same workspace, so if one script changes the value of a variable, all your 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.

## 5.2 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 word of the file has to be the word `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.

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.

## 5.3 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)
    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 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.

## 5.4 Function names

There are three "gotchas" that come up when you start naming 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
```

Then here's what you'll get:

```
>> y = myfunc(1);
>> y = something_else(1);
??? Undefined command/function 'something_else'.
```

The second gotcha is that the name of the file can't have spaces. For example, if you write a function and name the file `my func.m`, which the MATLAB editor will happily allow you to do, 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!

## 5.5 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
```

If you remember the Pythagorean Theorem, you probably figured out that this function computes the length of the hypotenuse of a right triangle if the lengths of the adjacent sides are `a` and `b`. (There is a MATLAB function called `hypot` that does the same thing.)

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

```
>> c = hypotenuse(3)
??? Input argument "b" is undefined.

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

This error message is 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.

## 5.6 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; if both are false, you can assume it is a matrix (at least for now).

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; in some cases the approximation is an integer but the actual value is not.

## 5.7 An incremental development example

Let's say that 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.

So the first draft of this program is `x=5`, which 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
```

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

## 5.8 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 loops 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 am using a feature we haven't seen before. The bracket operator creates a new matrix which, when it is displayed, shows the three values on one line:

```
>> 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
```

Sharp-eyed readers will 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.

## 5.9 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

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
```

I chose the ranges for a and b to be small (so the amount of output is manageable), but to contain at least one Pythagorean triple. A constant challenge of debugging is to generate enough output to demonstrate that the code is working (or not) without being overwhelmed.

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 elegant and simple:

```
>> find_triples

ans = 3     4     5
```

## 5.10 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.

In order to put the code we have written inside a function, we have to indent the whole thing. The MATLAB editor provides a shortcut for doing that, the Increase Indent command under the Text menu. Just don't forget to unselect the text before you start typing!

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.

## 5.11 A misstep

When you change the signature of a function, you have to change all the places that call the function, too. For example, suppose I decided to add a third input variable to `hypotenuse`:

```
function res = hypotenuse(a, b, d)
    res = (a.^d + b.^d) ^ (1/d);
end
```

When `d` is 2, this does the same thing it did before. There is no practical reason to generalize the function in this way; it's just an example. Now when you run `find_triples`, you get:

```
>> find_triples(20)
??? Input argument "d" is undefined.

Error in ==> hypotenuse at 2
    res = (a.^d + b.^d) ^ (1/d);

Error in ==> find_triples at 7
            c = hypotenuse(a, b);
```

So that makes it pretty easy to find the error. This is an example of a development technique that is sometimes useful: rather than search the program for all the places that use `hypotenuse`, you can run the program and use the error messages to guide you.

But this technique is risky, especially if the error messages make suggestions about what to change. If you do what you're told, you might make the error message go away, but that doesn't mean the program will do the right thing. MATLAB doesn't know what the program is *supposed* to do, but you should.

And that brings us to the Eighth Theorem of debugging:

> Error messages sometimes tell you what's wrong, but they seldom tell you what to do (and when they try, they're usually wrong).

## 5.12 continue

As one final improvement, let's modify the function so that 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, then 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, then $a$ 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
        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 (without executing the rest of the body), 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 wanted).

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
```

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

$$(F_n F_{n+3}, 2F_{n+1}F_{n+2}, F_{n+1}^2 + F_{n+2}^2)$$

is a Pythagorean triple for all $n \geq 1$.

**Exercise 5.1** *Write a function named* `fib_triple` *that takes an input variable* `n`*, uses* `fibonacci2` *to compute the first* `n` *Fibonacci numbers, and then checks whether this formula produces a Pythagorean triple for each number in the sequence.*

## 5.13 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.

## 5.14 Glossary

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

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

**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.

## 5.15 Exercises

**Exercise 5.2** *Take any of the scripts you have written so far, encapsulate the code in an appropriately-named function, and generalize the function by adding one or more input variables.*

*Make the function silent and then call it from the Command Window and confirm that you can display the output value.*

# Chapter 6

# Zero-finding

## 6.1 Why functions?

The previous chapter explained some of the benefits of functions, including

- 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 you should consider using functions is that many of the tools provided by MATLAB require you to write functions. For example, in this chapter we will use `fzero` to find solutions of nonlinear equations. Later we will use `ode45` to approximate solutions to differential equations.

## 6.2 Maps

In mathematics, a **map** is a correspondence between one set called the **range** and another set called the **domain**. For each element of the range, the map specifies the corresponding element of the domain.

You can think of a sequence as a map from positive integers to elements. You can think of a vector as a map from indices to elements. In these cases the maps are **discrete** because the elements of the range are countable.

You can also think of a function as a map from inputs to outputs, but in this case the range is **continuous** because the inputs can take any value, not just integers. (Strictly speaking, the set of floating-point numbers is discrete, but since floating-point numbers are meant to represent real numbers, we think of them as continuous.)

## 6.3 A note on notation

In this chapter I need to start talking about mathematical functions, and I am going to use a notation you might not have seen before.

If you have studied functions in a math class, you have probably seen something like

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

which is supposed to mean that $f$ is a function that maps from $x$ to $x^2 - 2x - 3$. The problem is that $f(x)$ is also used to mean the value of $f$ that corresponds to a particular value of $x$. So I don't like this notation. I prefer

$$f : x \rightarrow x^2 - 2x - 3$$

which means "f is the function that maps from $x$ to $x^2 - 2x - 3$." In MATLAB, this would be expressed like this:

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

I'll explain soon why this function is called `error_func`. Now, back to our regularly-scheduled programming.

## 6.4 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 $x = \sqrt{a}$. 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**, 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 $x = \sqrt{2x+3}$.

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.

Here's what would happen:

```
>> 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 as well as it did in this example, and it is not used very often in practice. We'll see one of the more practical alternatives in a minute.

## 6.5 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; 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

  $$f : x \to x^2 - 2x - 3$$

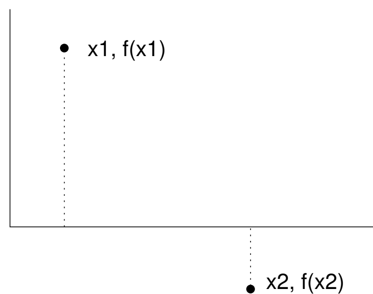
  Any value of $x$ that makes $f(x) = 0$ is also a solution of the original equation.

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

For example, if we can find two values $x_1$ and $x_2$ such that $f(x_1) > 0$ and $f(x_2) < 0$, then we can be certain that there is at least one zero between $x_1$ and $x_2$ (provided that we know that $f$ is continuous). In this case we would say that $x_1$ and $x_2$ bracket a zero.

Here's what this scenario might look like on a graph:




If this was all you knew about $f$, where would you go looking for a zero? If you said "halfway between $x_1$ and $x_2$," then 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," then 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... well, you probably didn't say that.

Finally, if you said, "I would use a built-in MATLAB function that combines the best features of several efficient and robust numerical methods," then you are ready to go on to the next section.

## 6.6 fzero

`fzero` is a built-in MATLAB function that combines the best features of several efficient and robust numerical methods.

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
```

But let's pretend that we don't know exactly 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 here is actually a vector that contains two elements. The bracket operator is a convenient way (one of several) to create a new vector.

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 $f(4)$. After each iteration, the interval that brackets the root gets smaller; `fzero` stops when the interval is so small that the estimated zero is correct to 16 digits. If you don't need that much precision, you can tell `fzero` to give you a quicker, dirtier answer (see the documentation for details).

## 6.7 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])
??? Input argument "x" is undefined.

Error in ==> error_func at 2
    x
```

Which is a very confusing error message. The problem is that MATLAB treats the first argument as a function call, so it calls `error_func` with no arguments. Since `error_func` requires one argument, the message indicates that the input argument is "undefined," although it might be clearer to say that you haven't provided a value for it.

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
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 "/home/downey/error_func.m (error_func)".
```

If you read it carefully, this is a pretty good error message (with the quibble that "output argument" is not a good synonym for "output variable").

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

```
>> x = error_func(4)

y = 5

??? Output argument "res" (and maybe others) not assigned during
call to "/home/downey/error_func.m (error_func)".

Error in ==> error_func at 2
    y = x^2 - 2*x -3
```

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

- Functions should always assign values to their output variables*.
- When you call a function, you should always 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
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.

## 6.8 Finding 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 for zero-finding.

*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.

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`:

```
>> 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 default `ezplot` calls your function 100 times (each time with a different value of `x`, of course). So you probably want to make your function silent before you plot it.

## 6.9 More name collisions

Functions and variables occupy the same "name-space," 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 matrix dimensions.
```

In this example, the problem is clear. Since the value of `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. Of course, if there were more distance between the assignment and the "function call," this message would be pretty confusing.

But the only thing worse than getting an error message is *not* getting an error message. If the value of `sin` was a vector, or if the value of `x` was smaller, you would really be in trouble.

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

ans = 3
```

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

The converse error can also happen if you try to access an undefined variable that also happens to be the name of a function. For example, if you have a function named `f`, and then try to increment a variable named `f` (and if you forget to initialize `f`), you get

```
>> f = f+1
??? Error:  "f" previously appeared to be used as a function or
command, conflicting with its use here as the name of a variable.
 A possible cause of this error is that you forgot to initialize the
 variable, or you have initialized it implicitly using load or eval.
```

At least, that's what you get if you are lucky. If this happens inside a function, MATLAB tries to call `f` as a function, and you get this

```
??? Input argument "x" is undefined.

Error in ==> f at 3
y = x^2 - a
```

There is no universal way to avoid these kind of collisions, but you can improve your chances by choosing variable names that don't shadow existing functions, and by choosing function names that you are unlikely to use as variables. That's why in Section 6.3 I called the error function `error_func` rather than `f`. I often give functions names that end in `func`, so that helps, too.

## 6.10 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 sometimes you have to spend some time to build scaffolding.

**ruminating:** Take some time to think! What kind of error is it: syntax, run-time, 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.

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 Ninth 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.

## 6.11 Glossary

**analytic solution:** A way of solving an equation by performing algebraic operations and deriving an explicit way to compute a value that is only known implicitly.

**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.

**map:** A correspondence between the elements of one set (the range) and the elements of another (the domain). You can think of sequences, vectors and functions as different kinds of maps.

**range:** The set of values a map maps from.

**domain:** The set of values a map maps to.

**discrete set:** A set, like the integers, whose elements are countable.

**continuous set:** A set, like the real numbers, whose elements are not countable. You can think of floating-point numbers as a continuous set.

**zero (of a function):** A value in the range of a function that maps to 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).

## 6.12 Exercises

**Exercise 6.1** *1. Write a function called* `cheby6` *that evaluates the 6th Chebyshev polynomial. It should take an input variable, x, and return*

$$32x^6 - 48x^4 + 18x^2 - 1 \tag{6.1}$$

*2. Use* `ezplot` *to display a graph of this function in the interval from 0 to 1. Estimate the location of any zeros in this range.*

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

**Exercise 6.2** *The density of a duck, $\rho$, is $0.3g/cm^3$ (0.3 times the density of water).*

*The volume of a sphere*[†] *with radius $r$ is $\frac{4}{3}\pi r^3$.*

*If a sphere with radius $r$ is submerged in water to a depth $d$, the volume of the sphere below the water line is*

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

*An object floats at the level where the weight of the displaced water equals the total weight of the object.*

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

*Assuming that a duck is a sphere with radius 10 cm, at what depth does a duck float?*

*Here are some suggestions about how to proceed:*

- *Write an equation relating* $\rho$, $d$ *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* $\rho$ *and* $r$ *and see if you get the effect you expect. What happens as* $\rho$ *increases? Goes to infinity? Goes to zero? What happens as* $r$ *increases? Goes to infinity? Goes to zero?*

# Chapter 7

# Functions of vectors

## 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 all the 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 6.2, I would create a file named `duck.m` and start with a top-level function named `duck` that takes no input variables and returns no output value.

Then I would write a function named `error_func` to evaluate the error function for `fzero`. To test `error_func` I would call it from `duck` and then call `duck` from the Command Window.

Here's what my first draft might look like:

```
function res = duck()
    error = error_func(10)
end

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

The line `res = h` 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`.

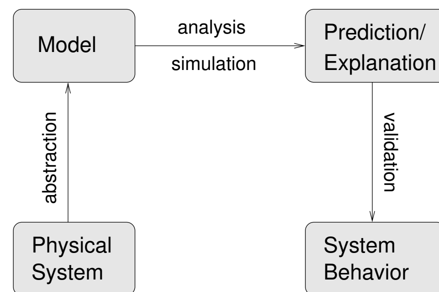
For this problem I might only need two functions, but if there were more, I could write and test them one at a time, and then combine them into a working program.

## 7.2 Physical modeling

Most of the examples so far have been about mathematics; Exercise 6.2, the "duck problem," is the first example we have seen of a physical system. If you didn't work on this exercise, you should at least go back and read it.

This book is supposed to be about **physical modeling**, so it might be a good idea to explain what that is. Physical modeling is a process for making predictions about physical systems and explaining their behavior. A **physical system** is something in the real world that we are interested in, like a duck.

The following figure shows the steps of this process:




A **model** is a simplified description of a physical system. The process of building a model is called **abstraction**. In this context, "abstract" is the opposite of "realistic;" an abstract model bears little direct resemblance to the physical system it models, in the same way that abstract art does not directly depict objects in the real world. A realistic model is one that includes more details and corresponds more directly to the real world.

Abstraction involves making justified decisions about which factors to include in the model and which factors can be simplified or ignored. For example, in the duck problem, we took into account the density of the duck and the buoyancy of water, but we ignored the buoyancy of the duck due to displacement of air and the dynamic effect of paddling feet. We also simplified the geometry of the duck by assuming that the underwater parts of a duck are similar to a segment of a sphere. And we used coarse estimates of the size and weight of the duck.

Some of these decisions are justifiable. The density of the duck is much higher than the density of air, so the effect of buoyancy in air is probably small. Other

decisions, like the spherical geometry, are harder to justify, but very helpful. The actual geometry of a duck is complicated; the sphere model makes it possible to generate an approximate answer without making detailed measurements of real ducks.

A more realistic model is not necessarily better. Models are useful because they can be analyzed mathematically and simulated computationally. Models that are too realistic might be difficult to simulate and impossible to analyze.

A model is successful if it is good enough for its purpose. If we only need a rough idea of the fraction of a duck that lies below the surface, the sphere model is good enough. If we need a more precise answer (for some reason) we might need a more realistic model.

Checking whether a model is good enough is called **validation**. The strongest form of validation is to make a measurement of an actual physical system and compare it to the prediction of a model.

If that is infeasible, there are weaker forms of validation. One is to compare multiple models of the same system. If they are inconsistent, that is an indication that (at least) one of them is wrong, and the size of the discrepancy is a hint about the reliability of their predictions.

We have only seen one physical model so far, so parts of this discussion may not be clear yet. We will come back to these topics later, but first we should learn more about vectors.

## 7.3 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 all. 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 have a return 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 4.12 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 a return value, I made a point of assigning it to a variable.

## 7.4 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 4.13:

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

Ideally I would have changed the name of the output variable to `Res`, as a reminder that it is supposed to get a vector value, but I didn't.

Here's how `myapply` works:

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

V = 1     4     9
```

**Exercise 7.1** *Write a function named* `find_target` *that encapsulates the code, from Section 4.14, that finds the location of a target value in a vector.*

## 7.5 Vectorizing your 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
```

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
```

At this point, I want to take a minute to acknowledge that I have been a little harsh in my presentation of MATLAB, because there are a number of features that I think make life harder than it needs to be for beginners. But here, finally, we are seeing features that show MATLAB's strengths.

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 5.5:

```
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 ==> mpower
Matrix must be square.
```

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
```

In this case, it 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, then the new function will also work on vectors.

## 7.6 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:

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

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

```
>> V = 1:5

V = 1     2     3     4     5

>> C = cumsum(V)

C = 1     3     6    10    15
```

**Exercise 7.2** *Write a function named* `cumulative_sum` *that uses a loop to compute the cumulative sum of the input vector.*

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(V))

ans = 1     2     3     4
```

But it isn't.

**Exercise 7.3** *Write a function named* `mydiff` *that computes the inverse of* `cumsum`*, so that* `cumsum(mydiff(X))` *and* `mydiff(cumsum(X))` *both return* `X`*.*

## 7.7 Products and ratios

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

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

In MATLAB, that looks like:

```
>> V = 1:5

V = 1      2      3      4      5

>> P = cumprod(V)

P = 1      2      6     24    120
```

**Exercise 7.4** *Write a function named* `cumulative_prod` *that uses a loop to compute the cumulative product of the input vector.*

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 7.5** *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 4.6).*

## 7.8 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 exit at the end of the loop, that means we didn't find what we were looking for (because if we had, we would have hit the `return` statement).

## 7.9 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** and denoted with the symbol $\forall$ which is pronounced "for all." So this expression

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

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

A slightly silly way to 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

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

as

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

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

**Exercise 7.6** *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.*

## 7.10 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. They are also 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: 1-by-0
```

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

## 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.

**physical modeling:** A process for making predictions about physical systems and explaining their behavior.

**physical system:** Something in the real world that we are interested in studying.

**model :** A simplified description of a physical system that lends itself to analysis or simulation.

**abstraction:** The process of building a model by making decisions about what factors to simplify or ignore.

**validation:** Checking whether a model is adequate for its purpose.

**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

## 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, when bacteria grow in particularly bacteria-friendly conditions, the rate of growth at any point in time is proportional to the current population. What we might like to know is the population as a function of time. Toward that end, let's define $f$ to be a function that maps from time, $t$, to population $y$. We don't know what it is, but we can write a differential equation that describes it:

$$\frac{df}{dt} = af$$

where $a$ is a constant that characterizes how quickly the population increases.

Notice that both sides of the equation are functions. To say that two functions are equal is to say that their values are equal at all times. In other words:

$$\forall t : \frac{df}{dt}(t) = af(t)$$

This is an **ordinary** differential equation (ODE) because all the derivatives involved are taken with respect to the same variable. If the equation 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.

This equation 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.

Linear, first order ODEs can be solved analytically; that is, we can express the solution as a function of $t$. This particular ODE has an infinite number of solutions, but they all have this form:

$$f(t) = be^{at}$$

For any value of $b$, this function satisfies the ODE. If you don't believe me, take its derivative and check.

If we know the population of bacteria at a particular point in time, we can use that additional information to determine which of the infinite solutions is the (unique) one that describes a particular population over time.

For example, if we know that $f(0) = 5$ billion cells, then we can write

$$f(0) = 5 = be^{a0}$$

and solve for $b$, which is 5. That determines what we wanted to know:

$$f(t) = 5e^{at}$$

The extra bit of information that determines $b$ is called the **initial condition** (although it isn't always specified at $t = 0$).

Unfortunately, most interesting physical systems are described by nonlinear DEs, most of which can't be solved analytically. The alternative is to solve them numerically.

## 8.2 Euler's method

The simplest numerical method for ODEs is Euler's method. Here's a test to see if you are as smart as Euler. Let's say that 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 $\Delta t$ has elapsed?

If you said $y + r\Delta t$, congratulations! You just invented Euler's method (but you're still not as smart as Euler).

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 $\Delta t$ is small.

But let's assume (for now) that the ODE we are interested in can be written so that

$$\frac{df}{dt}(t) = g(t, y)$$

where $g$ is some function that maps $(t, y)$ onto $r$; that is, given the time and current population, it computes the rate of change. Then we can advance from one point in time to the next using these equations:

$$T_{n+1} = T_n + \Delta t \tag{8.1}$$

$$F_{n+1} = F_n + g(t, y)\ \Delta t \tag{8.2}$$

Here $\{T_i\}$ is a sequence of times where we estimate the value of $f$, and $\{F_i\}$ is the sequence of estimates. For each index $i$, $F_i$ is an estimate of $f(T_i)$. The interval $\Delta t$ is called the **time step**.

Assuming that we start at $t = 0$ and we have an initial condition $f(0) = y_0$ (where $y_0$ denotes a particular, known value), we set $T_1 = 0$ and $F_1 = y_0$, and then use Equations 8.1 and 8.2 to compute values of $T_i$ and $F_i$ until $T_i$ gets to the value of $t$ we are interested in.

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 $\Delta t$ and then run it again with time step $\Delta t/2$. If the results are the same, they are probably accurate; otherwise, cut the time step again.

Euler's method is **first order**, which means that each time you cut the time step in half, you expect the estimation error to drop by half. With a second-order method, you expect the error to drop by a factor of 4; third-order drops by 8, etc. The price of higher order methods is that they have to evaluate $g$ more times per time step.

## 8.3 Another note on notation

There's a lot of math notation in this chapter so I want to pause to review what we have so far. Here are the variables, their meanings, and their types:

| Name | Meaning | Type |
|---|---|---|
| $t$ | time | scalar variable |
| $\Delta t$ | time step | scalar constant |
| $y$ | population | scalar variable |
| $r$ | rate of change | scalar variable |
| $f$ | The unknown function specified, implicitly, by an ODE. | function $t \to y$ |
| $df/dt$ | The first time derivative of $f$ | function $t \to r$ |
| $g$ | A "rate function," derived from the ODE, that computes rate of change for any $t$, $y$. | function $t, y \to r$ |
| $T$ | a sequence of times, $t$, where we estimate $f(t)$ | sequence |
| $F$ | a sequence of estimates for $f(t)$ | sequence |

So $f$ is a function that computes the population as a function of time, $f(t)$ is the function evaluated at a particular time, and if we assign $f(t)$ to a variable, we usually call that variable $y$.

Similarly, $g$ is a "rate function" that computes the rate of change as a function of time and population. If we assign $g(t, y)$ to a variable, we call it $r$.

$df/dt$ is the first derivative of $f$, and it maps from $t$ to a rate. If we assign $df/dt(t)$ to a variable, we call it $r$.

It is easy to get $df/dt$ confused with $g$, but notice that they are not even the same type. $g$ is more general: it can compute the rate of change for any (hypothetical) population at any time; $df/dt$ is more specific: it is the actual rate of change at time $t$, given that the population is $f(t)$.

## 8.4 `ode45`

A limitation of Euler's method is that the time step is constant from one iteration to the next. But some parts of the solution are harder to estimate than others; if the time step is small enough to get the hard parts right, it is doing more work than necessary on the easy parts. The ideal solution is to adjust the time step as you go along. Methods that do that are called **adaptive**, and one of the best adaptive methods is the Dormand-Prince pair of Runge-Kutta formulas. You don't have to know what that means, because the nice people at Mathworks have implemented it in a function called `ode45`. The `ode` stands for "ordinary differential equation [solver];" the 45 indicates that it uses a combination of 4th and 5th order formulas.

In order to use `ode45`, you have to write a MATLAB function that evaluates $g$ as a function of $t$ and $y$.

As an example, suppose that the rate of population growth for rats depends on the current population and the availability of food, which varies over the course of the year. The governing equation might be something like

$$\frac{df}{dt}(t) = af(t)\left[1 + \sin(\omega t)\right]$$

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

$a$ and $\omega$ are **parameters**. A parameter is a value that quantifies a physical aspect of the scenario being modeled. For example, in Exercise 6.2 we used parameters `rho` and `r` to quantify the density and radius of a duck. Parameters are often constants, but in some models they vary in time.

In this example, $a$ characterizes the reproductive rate, and $\omega$ is the frequency of a periodic function that describes the effect of varying food supply on reproduction.

This equation specifies a relationship between a function and its derivative. In order to estimate values of $f$ numerically, we have to transform it into a rate function.

The first step is to introduce a variable, $y$, as another name for $f(t)$

$$\frac{df}{dt}(t) = ay\left[1 + \sin(\omega t)\right]$$

This equation means that if we are given $t$ and $y$, we can compute $df/dt(t)$, which is the rate of change of $f$. The next step is to express that computation as a function called $g$:

$$g(t, y) = ay\left[1 + \sin(\omega t)\right]$$

Writing the function this way is useful because we can use it with Euler's method or `ode45` to estimate values of $f$. All we have to do is write a MATLAB function that evaluates $g$. Here's what that looks like using the values $a = 0.01$ and $\omega = 2\pi/365$ (one cycle per year):

```
function res = rats(t, y)
    a = 0.01;
    omega = 2 * pi / 365;
    res = a * y * (1 + sin(omega * t));
end
```

You can test this function from the Command Window by calling it with different values of `t` and `y`; the result is the rate of change (in units of rats per day):

```
>> r = rats(0, 2)

r = 0.0200
```

So if there are two rats on January 1, we expect them to reproduce at a rate that would produce 2 more rats per hundred days. But if we come back in April, the rate has almost doubled:

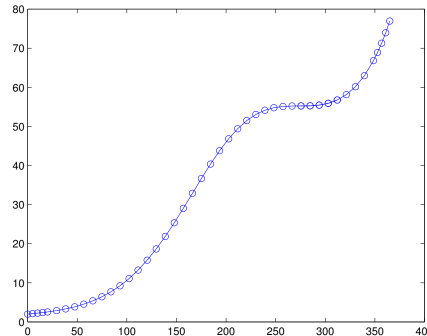
```
>> r = rats(120, 2)

r = 0.0376
```

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

```
>> ode45(@rats, [0, 365], 2)
```

The first argument is a handle for the function that computes $g$. The second argument is the interval we are interested in, one year. The third argument is the initial population, $f(0) = 2$.

When you call `ode45` without assigning the result to a variable, MATLAB displays the result in a figure:




The x-axis shows time from 0 to 365 days; the y-axis shows the rat population, which starts at 2 and grows to almost 80. The rate of growth is slow in the winter and summer, and faster in the spring and fall, but it also accelerates as the population grows.

## 8.5 Multiple output variables

`ode45` is one of many MATLAB functions that return more than one output variable. The syntax for calling it and saving the results is

```
>> [T, Y] = ode45(@rats, [0, 365], 2);
```

The first return value is assigned to `T`; the second is assigned to `Y`. Each element of `T` is a time, $t$, where `ode45` estimated the population; each element of `Y` is an estimate of $f(t)$.

If you assign the output values to variables, `ode45` doesn't draw the figure; you have to do it yourself:

```
>> plot(T, Y, 'bo-')
```

If you plot the elements of `T`, you'll see that the space between the points is not quite even. They are closer together at the beginning of the interval and farther apart at the end.

To see the population at the end of the year, you can display the last element from each vector:

```
>> [T(end), Y(end)]

ans = 365.0000   76.9530
```

`end` is a special word in MATLAB; when it appears as an index, it means "the index of the last element." You can use it in an expression, so `Y(end-1)` is the second-to-last element of `Y`.

How much does the final population change if you double the initial population? How much does it change if you double the interval to two years? How much does it change if you double the value of $a$?

## 8.6 Analytic or numerical?

When you solve an ODE analytically, the result is a function, $f$, that allows you to compute the population, $f(t)$, for any value of $t$. When you solve an ODE numerically, you get two vectors. You can think of these vectors as a discrete approximation of the continuous function $f$: "discrete" because it is only defined for certain values of $t$, and "approximate" because each value $F_i$ is only an estimate of the true value $f(t)$.

So those are the limitations of numerical solutions. The primary advantage is that you can compute numerical solutions to ODEs that don't have analytic solutions, which is the vast majority of nonlinear ODEs.

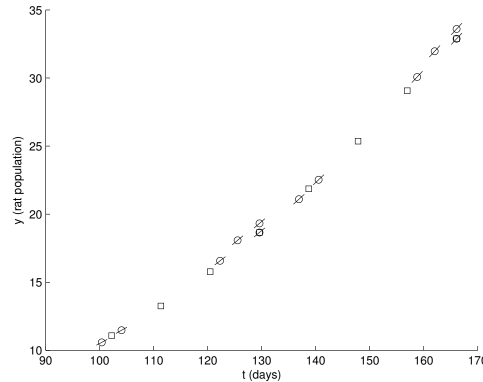
If you are curious to know more about how `ode45` works, you can modify `rats` to display the points, $(t, y)$, where `ode45` evaluates $g$. Here is a simple version:

```
function res = rats(t, y)
    plot(t, y, 'bo')
    a = 0.01;
    omega = 2 * pi / 365;
    res = a * y * (1 + sin(omega * t));
end
```

Each time `rats` is called, it plots one data point; in order to see all of the data points, you have to use `hold on`.

```
>> clf; hold on
>> [T, Y] = ode45(@rats, [0, 10], 2);
```

This figure shows part of the output, zoomed in on the range from Day 100 to 170:




The circles show the points where `ode45` called `rats`. The lines through the circles show the slope (rate of change) calculated at each point. The rectangles show the locations of the estimates $(T_i, F_i)$. Notice that `ode45` typically evaluates $g$ several times for each estimate. This allows it to improve the estimates, for one thing, but also to detect places where the errors are increasing so it can decrease the time step (or the other way around).

## 8.7 What can 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 `rats` without providing arguments.

```
>> ode45(rats, [0,365], 2)
??? Input argument "y" is undefined.

Error in ==> rats at 4
    res = a * y * (1 + sin(omega * t));
```

Again, the error message is confusing, because it looks like the problem is in `rats`. You've been warned!

Also, remember that the function you write will be called by `ode45`, which means it has to have the signature `ode45` expects: it should take two input variables, `t` and `y`, in that order, and return one output variable, `r`.

If you are working with a rate function like this:

$$g(t, y) = ay$$

You might be tempted to write this:

```
function res = rate_func(y)        % WRONG
    a = 0.1
    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 = 0.1
    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 = rats(t, y)
    a = 0.01;
    omega = 2 * pi / 365;
    r = a * y * (1 + sin(omega * t))    % WRONG
end
```

And then call it from `ode45`, you get

```
>> ode45(@rats, [0,365], 2)
??? Output argument "res" (and maybe others) not assigned during
call to "/home/downey/rats.m (rats)".

Error in ==> rats at 2
    a = 0.01;

Error in ==> funfun/private/odearguments at 110
f0 = feval(ode,t0,y0,args{:});   % ODE15I sets args{1} to yp0.

Error in ==> ode45 at 173
[neq, tspan, ntspan, next, t0, tfinal, tdir, y0, f0, odeArgs,
odeFcn, ...
```

This might be a scary message, but if you read the first line and ignore the rest, you'll get the idea.

Yet another mistake that people make with `ode45` is leaving out the brackets on the second argument. In that case, MATLAB thinks there are four arguments, and you get

```
>> ode45(@rats, 0, 365, 2)
??? Error using ==> funfun/private/odearguments
When the first argument to ode45 is a function handle, the
tspan argument must have at least two elements.

Error in ==> ode45 at 173
[neq, tspan, ntspan, next, t0, tfinal, tdir, y0, f0, odeArgs,
odeFcn, ...
```

Again, if you read the first line, you should be able to figure out the problem (`tspan` stands for "time span", which we have been calling the interval).

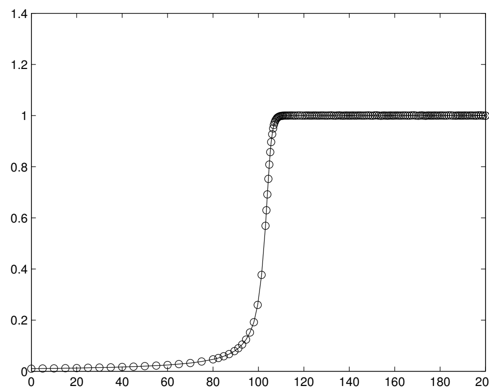
## 8.8 Stiffness

There is yet another problem you might encounter, but if it makes you feel better, it might not be your fault: the problem you are trying to solve might be **stiff***.

I won't give a technical explanation of stiffness here, except to say that for some problems (over some intervals with some initial conditions) the time step needed to control the error is very small, which means that the computation takes a long time. Here's one example:

$$\frac{df}{dt} = f^2 - f^3$$

If you solve this ODE with the initial condition $f(0) = \delta$ over the interval from 0 to $2/\delta$, with $\delta = 0.01$, you should see something like this:




After the transition from 0 to 1, the time step is very small and the computation goes slowly. For smaller values of $\delta$, the situation is even worse.

*The following discussion is based partly on an article from Mathworks available at `http://www.mathworks.com/company/newsletters/news_notes/clevescorner/may03_cleve.html`

In this case, the problem is easy to fix: instead of `ode45` you can use `ode23s`, an ODE solver that tends to perform well on stiff problems (that's what the "s" stands for).

In general, if you find that `ode45` is taking a long time, you might want to try one of the stiff solvers. It won't always solve the problem, but if the problem is stiffness, the improvement can be striking.

**Exercise 8.1** *Write a rate function for this ODE and use* `ode45` *to solve it with the given initial condition and interval. Start with* $\delta = 0.1$ *and decrease it by multiples of 10. If you get tired of waiting for a computation to complete, you can press the Stop button in the Figure window or press Control-C in the Command Window.*

*Now replace* `ode45` *with* `ode23s` *and try again!*

## 8.9 Glossary

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

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

**partial DE:** A DE that includes derivatives with respect to more than one variable

**first order (ODE):** A DE that includes only first derivatives.

**linear:** 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.

**first order (numerical method):** A method whose error is expected to halve when the time step is halved.

**adaptive:** A method that adjusts the time step to control error.

**stiffness:** A characteristic of some ODEs that makes some ODE solvers run slowly (or generate bad estimates). Some ODE solvers, like `ode23s`, are designed to work on stiff problems.

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

## 8.10 Exercises

**Exercise 8.2** *Suppose that you are given an 8 ounce cup of coffee at 90 °C and a 1 ounce container of cream at room temperature, which is 20 °C. You have learned from bitter experience that the hottest coffee you can drink comfortably is 60 °C.*

*Assuming that you take cream in your coffee, and that you would like to start drinking as soon as possible, are you better off adding the cream immediately or waiting? And if you should wait, then how long?*

*To answer this question, you have to model the cooling process of a hot liquid in air. Hot coffee transfers heat to the environment by conduction, radiation, and evaporative cooling. Quantifying these effects individually would be challenging and unnecessary to answer the question as posed.*

*As a simplification, we can use Newton's Law of Cooling*[†]*:*

$$\frac{df}{dt} = -r(f - e)$$

*where $f$ is the temperature of the coffee as a function of time and $df/dt$ is its time derivative; $e$ is the temperature of the environment, which is a constant in this case, and $r$ is a parameter (also constant) that characterizes the rate of heat transfer.*

*It would be easy to estimate $r$ for a given coffee cup by making a few measurements over time. Let's assume that that has been done and $r$ has been found to be 0.001 in units of inverse seconds, $1/s$.*

- *Using mathematical notation, write the rate function, $g$, as a function of $y$, where $y$ is the temperature of the coffee at a particular point in time.*
- *Create an M-file named* `coffee` *and write a function called* `coffee` *that takes no input variables and returns no output value. 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 $g(t, y)$. Notice that in this case $g$ does not actually depend on $t$; nevertheless, your function has to take $t$ as the first input argument in order to work with* `ode45`*.*

  *Test your function by adding a line like* `rate_func(0,90)` *to* `coffee`*, the call* `coffee` *from the Command Window.*

[†] http://en.wikipedia.org/wiki/Heat_conduction

- *Once you get* `rate_func(0,90)` *working, modify* `coffee` *to use* `ode45` *to compute the temperature of the coffee (ignoring the cream) for 60 minutes. Confirm that the coffee cools quickly at first, then more slowly, and reaches room temperature (approximately) after about an hour.*

- *Write a function called* `mix_func` *that computes the final temperature of a mixture of two liquids. It should take the volumes and temperatures of the liquids as parameters.*

  *In general, the final temperature of a mixture depends on the specific heat of the two substances*[‡]*. But if we make the simplifying assumption that coffee and cream have the same density and specific heat, then the final temperature is* $(v_1 y_1 + v_2 y_2)/(v_1 + v_2)$*, where* $v_1$ *and* $v_2$ *are the volumes of the liquids, and* $y_1$ *and* $y_2$ *are their temperatures.*

  *Add code to* `coffee` *to test* `mix_func`*.*

- *Use* `mix_func` *and* `ode45` *to compute the time until the coffee is drinkable if you add the cream immediately.*

- *Modify* `coffee` *so it takes an input variable* $t$ *that determines how many seconds the coffee is allowed to cool before adding the cream, and returns the temperature of the coffee after mixing.*

- *Use* `fzero` *to find the time* $t$ *that causes the temperature of the coffee after mixing to be 60* $^\circ C$*.*

- *What do these results tell you about the answer to the original question? Is the answer what you expected? What simplifying assumptions does this answer depend on? Which of them do you think has the biggest effect? Do you think it is big enough to affect the outcome? Overall, how confident are you that this model can give a definitive answer to this question? What might you do to improve it?*

[‡] `http://en.wikipedia.org/wiki/Heat_capacity`

# Chapter 9

# Systems of ODEs

## 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" matrix with the given size:

```
>> 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:

```
>> 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:

```
>> 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 9.1** *What effect does the transpose operator have on row vectors, column vectors, and scalars?*

## 9.4 Lotka-Voltera

The Lotka-Voltera model describes the interactions between two species in an ecosystem, a predator and its prey. A common example is rabbits and foxes.

The model is governed by the following system of differential equations:

$$R_t = aR - bRF$$
$$F_t = ebRF - cF$$

where

- $R$ is the population of rabbits,
- $F$ is the population of foxes,
- $a$ is the natural growth rate of rabbits in the absence of predation,
- $c$ is the natural death rate of foxes in the absence of prey,
- $b$ is the death rate of rabbits per interaction with a fox,
- $e$ is the efficiency of turning eaten rabbits into foxes.

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

And here's what the rate function looks like with the parameters $a = 0.1$, $b = 0.01$, $c = 0.1$ and $e = 0.2$:

```
function res = lotka(t, V)
    % unpack the elements of V
    r = V(1);
    f = V(2);

    % set the parameters
    a = 0.1;
    b = 0.01;
    c = 0.1;
    e = 0.2;
```

```
    % compute the derivatives
    drdt = a*r - b*r*f;
    dfdt = e*b*r*f - c*f;

    % pack the derivatives into a vector
    res = [drdt; dfdt];
end
```

As usual, the first input variable is time. The second input variable is a vector with two elements, $R(t)$ and $F(t)$. I gave it a capital letter to remind me that it is a vector. The body of the function includes four **paragraphs**, each explained by a comment.

The first paragraph **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 paragraph, where we compute the derivatives, resemble the mathematical equations we were given, which helps prevent errors.

The second paragraph 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 paragraph **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.

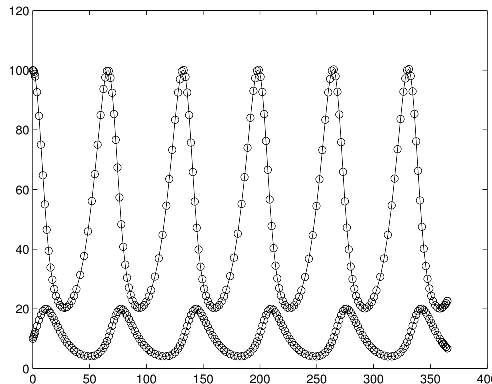
Now we can run `ode45` like this:

```
ode45(@lotka, [0, 365], [100, 10])
```

As always, the first argument is a function handle, the second is the time interval, and the third is the initial condition. The initial condition is a vector: the first element is the number of rabbits at $t = 0$, the second element is the number of foxes.

The order of these 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 `lotka`, where you unpack the input vector and repack the output vector. MATLAB doesn't know what these values mean; it is up to you as the programmer to keep track.

But if you get the order right, you should see something like this:




The x-axis is time in days; the y-axis is population. The top curve shows the population of rabbits; the bottom curve shows foxes. This result is one of several patterns this system can fall into, depending on the starting conditions and the parameters. As an exercise, try experimenting with different values.

## 9.5 What can go wrong?

The output vector from the rate function has to be a column vector; otherwise you get

```
??? Error using ==> funfun/private/odearguments
LOTKA must return a column vector.

Error in ==> ode45 at 173
[neq, tspan, ntspan, next, t0, tfinal, tdir, y0, f0, odeArgs,
odeFcn, ...
```

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`. Again, 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.

## 9.6 Output matrices

As we saw before, if you call `ode45` without assigning the results to variables, it plots the results. If you assign the results to variables, it suppresses the figure. Here's what that looks like:

```
>> [T, M] = ode45(@lotka, [0, 365], [100, 10]);
```

You can think of the left side of this assignment as a vector of variables.

As in previous examples, T is a vector of time values where ode45 made estimates. But unlike previous examples, the second output variable is a matrix containing one column for each variable (in this case, $R$ and $F$) and one row for each time value.

```
>> 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);
```

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."

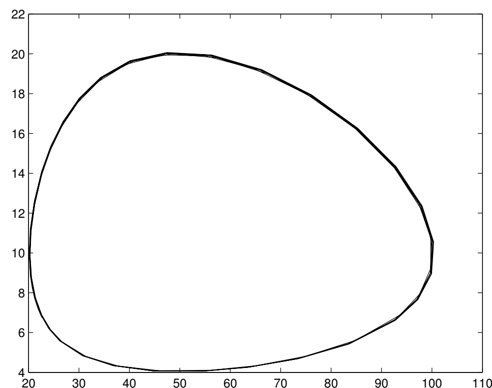
```
>> size(R)

ans = 185     1

>> size(F)

ans = 185     1
```

So R and F are column vectors.

If you plot these vectors against each other, like this

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

You get a **phase plot** that looks like this:



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 during the time interval. This path is called a **trajectory**.

Since the behavior of this system is periodic, the resulting 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 Glossary

**row vector:** In MATLAB, 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 set of equations written in terms of a set of variables such that the equations are intertangled.

**paragraph:** A chunk of code that makes up part of a function, usually with an explanatory comment.

**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.

**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.

## 9.8 Exercises

**Exercise 9.2** *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 the 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..."*

*The system is described by this system of differential equations:*

$$x_t = \sigma(y - x) \tag{9.1}$$

$$y_t = x(r - z) - y \tag{9.2}$$

$$z_t = xy - bz \tag{9.3}$$

*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. *The first step is to write a function named* `lorenz` *that 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.*

2. *The next step is to test your function by calling it from the command line 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`.

3. *Assuming that Step 2 works, you can use* `ode45` *to estimate the solution for the time interval* $t_0 = 0$, $t_e = 30$ *with the initial condition* $x = 1$, $y = 2$ *and* $z = 3$.

4. *Use* `plot3` *to plot the trajectory of* $x$, $y$ *and* $z$.

[*] http://en.wikipedia.org/wiki/Lorenz_attractor

# Chapter 10

# Second-order systems

## 10.1 Nested functions

In the Section 7.1, we saw an example of an M-file with more than one function:

```
function res = duck()
    error = error_func(10)
end

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

Because the first function ends before the second begins, they are at the same level of indentation. Functions like these are **parallel**, as opposed to **nested**. A nested function is defined inside another, like this:

```
function res = duck()
    error = error_func(10)

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

The top-level function, `duck`, is the **outer function** and `error_func` is an **inner function**.

Nesting functions is useful because the variables of the outer function can be accessed from the inner function. This is not possible with parallel functions.

In this example, using a nested function makes it possible to move the parameters rho and r out of error_func.

```
function res = duck(rho)
    r = 10;
    error = error_func(10)

    function res = error_func(h)
        res = ...
    end
end
```

Both rho and r can be accessed from error_func. By making rho an input argument, we made it easier to test duck with different parameter values.

## 10.2 Newtonian motion

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

$$F = ma$$

where $F$ is the net force acting on a object, $m$ is the mass of the object, and $a$ is the resulting acceleration of the object. In a simple case where the object is moving along a straight line, $F$ and $a$ are scalars, but in general they are vectors.

Even more generally, if $F$ and $a$ vary in time, then they can be thought of as functions that return vectors; that is, $F$ is a function and the result of evaluating $F(t)$ is a vector that describes the net force at time $t$. So a more explicit way to write Newton's law is

$$\forall t : \vec{F}(t) = m\vec{a}(t)$$

The arrangement of 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 compute $a$.

So if you know acceleration, $a$, as a function of time, how do you find the position of the object, $p$? Well, we know that acceleration is the second derivative of position, so we can write a differential equation

$$p_{tt} = a$$

Where $a$ and $p$ are functions of time that return vectors, and $p_{tt}$ is the second time derivative of $p$.

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

$$\begin{aligned} p_t &= v \\ v_t &= a \end{aligned}$$

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

## 10.3 Freefall

Let's start with a simple example, an object in freefall in a vacuum (where there's no air resistance). Near the surface of the earth, the acceleration of gravity is $g = -9.8\ m/s^2$, where the minus sign indicates that gravity pulls down.

If the object falls straight down (in the same direction as gravity), we can describe its position with a scalar value, altitude. So this will be a one-dimensional problem, at least for now.

Here is a rate function we can use with `ode45` to solve this problem:

```
function res = freefall(t, X)
    p = X(1);       % the first element is position
    v = X(2);       % the second element is velocity

    dpdt = v;
    dvdt = acceleration(t, p, v);

    res = [dpdt; dvdt];    % pack the results in a column vector
end

function res = acceleration(t, p, v)
    g = -9.8;       % acceleration of gravity in m/s^2
    res = g;
end
```

The first function is the rate function. It gets `t` and `X` as input variables, where the elements of `X` are understood to be position and velocity. The return value from `freefall` is a (column) vector that contains the derivatives of position and velocity, which are velocity and acceleration, respectively.

Computing $p_t$ is easy because we are given velocity as an element of `X`. The only thing we have to compute is acceleration, which is what the second function does.

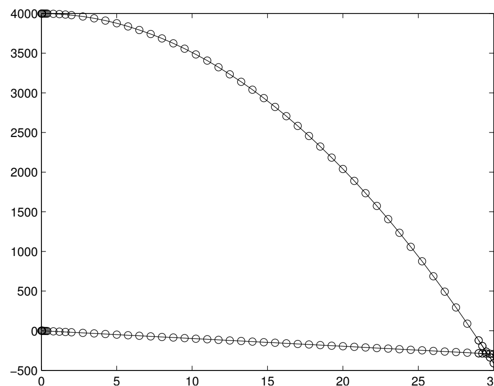
`acceleration` computes acceleration as a function of time, position and velocity. In this example, the net acceleration is a constant, so we don't really have to include all this information yet, but we will soon.

Here's how to run `ode45` with this rate function:

```
>> ode45(@freefall, [0, 30], [4000, 0])
```

As always, the first argument is the function handle, the second is the time interval (30 seconds) and the third is the initial condition: in this case, the initial altitude is 4000 meters and the initial velocity is 0. So you can think of the "object" a a skydiver jumping out of an airplane at about 12,000 feet.

Here's what the result looks like:




The bottom line shows velocity starting at zero and dropping linearly. The top line shows position starting at 4000 m and dropping parabolically (but remember that this parabola is a function of time, not a ballistic trajectory).

Notice that `ode45` doesn't know where the ground is, so the skydiver keeps going through zero into negative altitude. We will address this issue later.

## 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 $v^2$:

$$F_{drag} = cv^2$$

Where $c$ is a drag constant that depends on the density of air, the cross-sectional area of the object and the surface properties of the object. For purposes of this problem, let's say that $c = 0.2$.

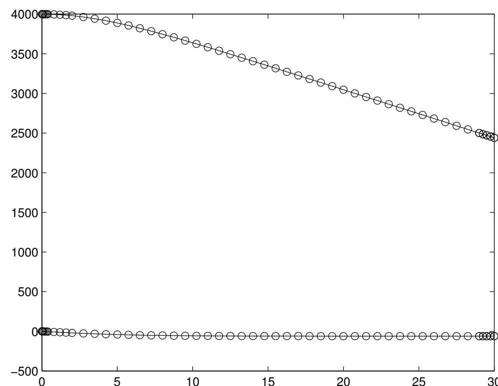
To convert from force to acceleration, we have to know mass, so let's say that the skydiver (with equipment) weighs 75 kg.

Here's a version of `acceleration` that takes air resistance into account (you don't have to make any changes in `freefall`:

```
function res = acceleration(t, p, v)
    a_grav = -9.8;              % acceleration of gravity in m/s^2
    c = 0.2;                    % drag constant
    m = 75;                     % mass in kg
    f_drag = c * v^2;           % drag force in N
    a_drag = f_drag / m;        % drag acceleration in m/s^2
    res = a_grav + a_drag;      % total acceleration
end
```

The sign of the drag force (and acceleration) is positive as long as the object is falling, the direction of the drag force is up. The net acceleration is the sum of gravity and drag. 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.

Here's what the result looks like with air resistance:




Big difference! With air resistance, velocity increases until the drag acceleration equals $g$; after that, velocity is a constant, known as "terminal velocity," and position decreases linearly (and much more slowly than it would in a vacuum). To examine the results more closely, we can assign them to variables

```
>> [T, M] = ode45(@freefall, [0, 30], [4000, 0]);
```

And then read the terminal position and velocity:

```
>> M(end,1)

ans = 2.4412e+03          % altitude in meters
```

```
>> M(end,2)

ans = -60.6143           % velocity in m/s
```

**Exercise 10.1** *Increase the mass of the skydiver, and confirm that terminal velocity increases. This relationship is the source of the intuition that heavy objects fall faster; in air, they do!*

## 10.5 Parachute!

In the previous section, we saw that the terminal velocity of a $75kg$ skydiver is about 60 $m/s$, which is about 130 mph. If you hit the ground at that speed, you would almost certainly be killed. That's where parachutes come in.

**Exercise 10.2** *Modify* `acceleration` *so that after 30 seconds of free-fall the skydiver deploys a parachute, which (almost) instantly increases the drag constant to 2.7.*

*What is the terminal velocity now? How long (after deployment) does it take to reach the ground?*

## 10.6 Two dimensions

So far we have used `ode45` for a system of first-order equations and for a single second-order equation. The next logical step is a system of second-order equations, and the next logical example is a projectile. A "projectile" is an object propelled through space, usually toward, and often to the detriment of, a target.

If a projectile stays in a plane, we can think of the system as two-dimensional, with $x$ representing the horizontal distance traveled and $y$ representing the height or altitude. So now instead of a skydiver, think of a circus performer being fired out of a cannon.

According to the Wikipedia*, the record distance for a human cannonball is 56.5 meters (almost 186 feet).

Here is a general framework for computing the trajectory of a projectile in two dimensions using `ode45`:

```
function res = projectile(t, W)
    P = W(1:2);
    V = W(3:4);

    dPdt = V;
```

*http://en.wikipedia.org/wiki/Human_cannonball

```
    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
    res = [0; g];
end
```

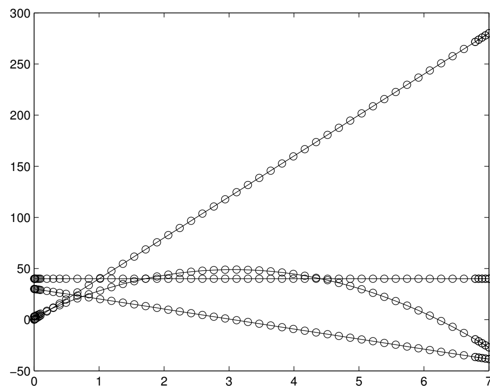
The second argument of the rate function is a vector, `W`, with four elements. The first two are assigned to `P`, which represents position; the last two are assigned to `V`, which represents velocity. `P` and `V` are vectors with elements for the $x$ and $y$ components.

The result from `acceleration` is also a vector; ignoring air resistance (for now), the acceleration in the $x$ direction is 0; in the $y$ direction it's $g$. Other than that, this code is similar to what we saw in Section 10.3.

If we launch the human projectile from an initial height of 3 meters, with velocities 40 m/s and 30 m/s in the $x$ and $y$ directions, the `ode45` call looks like this:

```
ode45(@projectile, [0,10], [0, 3, 40, 30]);
```

And the result looks like this:




You might have to think a little to figure out which line is which. It looks like the flight time is about 6 seconds.

**Exercise 10.3** *Extract the $x$ and $y$ components of position, plot the trajectory of the projectile, and estimate the distance traveled.*

**Exercise 10.4** *Add air resistance to this simulation. In the skydiver scenario, we estimated that the drag constant was* 0.2*, but that was based on the assumption that the skydiver is falling flat. A human cannonball, flying head-first, probably has a drag constant closer to* 0.1*. What initial velocity is needed to*

*achieve the record flight distance of 65.6 meters? Hint: what is the optimal launch angle?*

## 10.7 What could go wrong?

What could go wrong? Well, `vertcat` for one. To explain what that means, I'll start with **catenation**, which is the operation of joining two matrices into a larger matrix. "Vertical catenation" joins the matrices by stacking them on top of each other; "horizontal catenation" lays them side by side.

Here's an example of horizontal catenation 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 catenation. The vertical catenation 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
All matrices on a row in the bracketed expression must have the
 same number of rows.

>> c = [a; b]
??? Error using ==> vertcat
All rows in the bracketed expression must have the same
number of columns.
```

In this example, `a` is a row vector and `b` is a column vector, so they can't be catenated in either direction.

Reading the error messages, you probably guessed that `horzcat` is the function that performs horizontal catenation, and likewise with `vertcat` and vertical catenation.

These operations are relevant to `projectile` because of the last line, which packs `dPdt` and `dVdt` into the output variable:

```
function res = projectile(t, W)
    P = W(1:2);
    V = W(3:4);

    dPdt = V;
    dVdt = acceleration(t, P, V);

    res = [dPdt; dVdt];
end
```

As long as both `dPdt` and `dVdt` are column vectors, the semi-colon performs vertical catenation, 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 `projectile` to be a column vector, so if you are working with `ode45`, it is probably a good idea to make *everything* a column vector.

In general, if you run into problems with `horzcat` and `vertcat`, use `size` to display the dimensions of the operands, and make sure you are clear on which way your vectors go.

## 10.8 Glossary

**parallel functions:** Two or more functions defined side-by-side, so that one ends before the next begins.

**nested function:** A function defined inside another function.

**outer function:** A function that contains another function definition.

**inner function:** A function defined inside another function definition. The inner function can access the variables of the outer function.

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

## 10.9 Exercises

**Exercise 10.5** *The flight of a baseball is governed by three forces: gravity, drag due to air resistance, and Magnus force due to spin. If we ignore wind and Magnus force, the path of the baseball stays in a plane, so we can model it as a projectile in two dimensions.*

*A simple model of the drag of a baseball is:*

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

*where $F_d$ is a vector that represents the force on the baseball due to drag, $C_d$ is the drag coefficient (0.3 is a reasonable choice), $\rho$ is the density of air (1.3 kg/$m^3$ at sea level), $A$ is the cross sectional area of the baseball (0.0042 $m^2$), $v$ is the magnitude of the velocity vector, and $\hat{V}$ is a unit vector in the direction of the velocity vector. The mass of the baseball is 0.145 kg.*

*For more information about drag, see* `http://en.wikipedia.org/wiki/Drag_(physics)`.

- *Write a function that takes the initial velocity of the baseball and the launch angle as input variables, uses* `ode45` *to compute the trajectory, and returns the range (horizontal distance in flight) as an output variable.*
- *Write a function that takes the initial velocity of the baseball as an input variable, computes the launch angle that maximizes the range, and returns the optimal angle and range as output variables. How does the optimal angle vary with initial velocity?*
- *When the 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? Make a prediction about what effect this will have on the optimal launch angle, and then use your simulation to test your prediction.*

- *The Green Monster in Fenway Park is about 12 m high and about 97 m from home plate along the left field line. What is the minimum speed a ball must leave the bat in order to clear the monster (assuming it goes off at the optimal angle)? Do you think it is possible for a person to stand on home plate and throw a ball over the Green Monster?*
- *The actual drag on a baseball is more complicated than what is captured by our simple model. In particular, the drag coefficient varies with velocity. You can get some of the details from* The Physics of Baseball[†]*; you also might find information on the web. Either way, specify a more realistic model of drag and modify your program to implement it. How big is the effect on your computed ranges? How big is the effect on the optimal angles?*

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

# Chapter 11

# Optimization and Interpolation

## 11.1 ODE Events

Normally when you call `ode45` you have to specify a start time and an end time. But in many cases, you don't know ahead of time when the simulation should end. Fortunately MATLAB provides a mechanism for dealing with this problem. The bad news is that it is a little awkward. Here's how it works:

1. Before calling `ode45` you use `odeset` to create an object called `options` that contains values that control how `ode45` works:

   ```
   options = odeset('Events', @events);
   ```

   In this case, the name of the option is `Events` and the value is a function handle. When `ode45` runs, it will invoke `events` after each timestep. You can call this function anything you want, but the name `events` is conventional.

2. The function you provide has to take the same input variables as your rate function. For example, here is an event function that would work with `projectile` from Section 10.6

   ```
   function [value,isterminal,direction] = events(t,X)
       value = X(2);         % Extract the current height.
       isterminal = 1;       % Stop the integration if height crosses zero.
       direction = -1;       % But only if the height is decreasing.
   end
   ```

`events` returns three output variables:

`value` determines when an event occurs. In this case `value` gets the second element of `X`, which is understood to be the height of the projectile. An "event" is a point in time when this value passes through 0.

`direction` determines whether an event occurs when `value` is increasing (`direction=1`), decreasing (`direction=-1`, or both `direction=0`.

`isterminal` determines what happens when an event occurs. If `isterminal=1`, the event is "terminal" and the simulation stops. If `isterminal=0`, the simulation continues, but `ode45` does some additional work to make sure that the solution in the vicinity of the event is accurate, and that one of the estimated values in the result is at the time of the event.

3. When you call `ode45`, you pass `options` as a fourth argument:

```
ode45(@projectile, [0,10], [0, 3, 40, 30], options);
```

**Exercise 11.1** *How would you modify* `events` *to stop when the height of the projectile falls through 3m?*

## 11.2 Optimization

In Exercise 10.5, you were asked to find the optimal launch angle for a batted ball. "Optimal" is a fancy way of saying "best;" what that means depends on the problem. For the Green Monster Problem—finding the optimal angle for hitting a home run in Fenway Park, the meaning of "optimal" is not obvious.

It is tempting to choose the angle that yields the longest range (distance from home plate when it lands). But in this case we are trying to clear a 12m wall, so maybe we want the angle that yields the longest range when the ball falls through 12m.

Although either definition would be good enough for most purposes, neither is quite right. In this case the "optimal" angle is the one that yields the greatest height at the point where the ball reaches the wall, which is 97m from home plate.

So the first step in any optimization problem is to define what "optimal" means. The second step is to define a range of values where you want to search. In this case the range of feasible values is between 0 degrees (parallel to the ground) and 90 degrees (straight up). We expect the optimal angle to be near 45 degrees, but we might not be sure how far from 45 degrees to look. To play it safe, we could start with the widest feasible range.

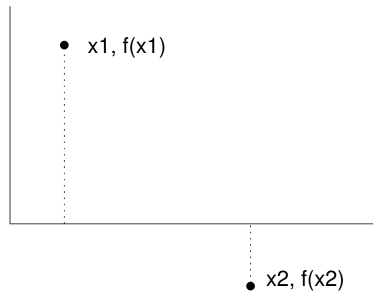
The simplest way to search for an optimal value is to run the simulation with a wide range of values and choose the one that yields the best result. This method

is not very efficient, especially in a case like this where computing the distance in flight is expensive.

A better algorithm is a Golden Section Search.

## 11.3 Golden section search

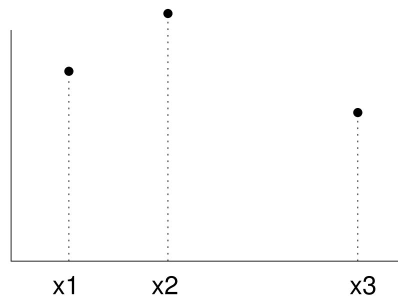
To present the Golden Section Search, I will start with a simplified version I'll call a Silver Section Search. The basic idea is similar to the methods for zero-finding we saw in Section 6.5. In the case of zero-finding, we had a picture like this:




We are given a function, $f$, that we can evaluate, and we want to find a root of $f$; that is, a value of $x$ that makes $f(x) = 0$. If we can find a value, $x_1$, that makes $f(x_1)$ positive and another value, $x_2$, that makes $f(x_2)$ negative, then there has to be a root in between (as long as $f$ is continuous). In this case we say that $x_1$ and $x_2$ "bracket" the root.

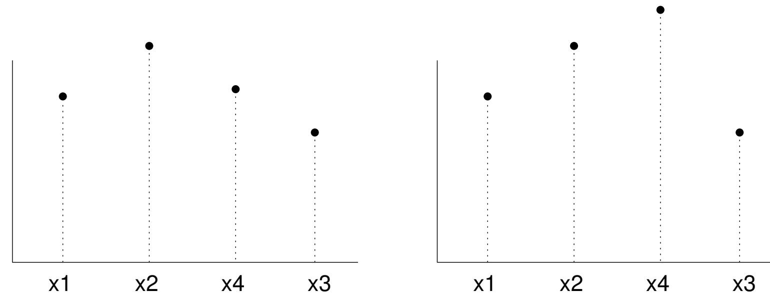
The algorithm proceeds by choosing a third value, $x_3$, between $x_1$ and $x_2$ and then evaluating $y = f(x_3)$. If $y$ is positive, we can form a new pair, $(x_3, x_2)$, that brackets the root. If $y$ is negative then the pair $(x_1, x_3)$ brackets the root. Either way the size of the bracket gets smaller, so our estimate of the location of the root gets better.

So that was root-finding. The Golden Section Search is similar, but we have to start with three values, and the picture looks like this:



This diagram shows that we have evaluated $f$ in three places, $x_1$, $x_2$ and $x_3$, and found that $x_2$ yields the highest value. If $f$ is continuous, then there has to be at least one local maximum between $x_1$ and $x_3$, so we would say that the triple $(x_1, x_2, x_3)$ brackets a maximum.

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 $f(x_2)$:




If $f(x_4)$ is less than than $f(x_2)$ (shown on the left), then the new triple $(x_1, x_2, x_4)$ brackets the maximum. If $f(x_4)$ is greater than $f(x_2)$ (shown on the right), then $(x_2, x_4, x_3)$ brackets the maximum. 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. In the example, I chose to bisect the bigger of the ranges $(x_1, x_2)$ and $(x_2, x_3)$.

Here's what that looks like in MATLAB:

```
function res = optimize(V)
    x1 = V(1);
    x2 = V(2);
    x3 = V(3);

    fx1 = height_func(x1);
    fx2 = height_func(x2);
    fx3 = height_func(x3);

    for i=1:50
        if x3-x2 > x2-x1
            x4 = (x2+x3) / 2;
            fx4 = height_func(x4);
            if fx4 > fx2
                x1 = x2;  fx1 = fx2;
                x2 = x4;  fx2 = fx4;
            else
                x3 = x4;  fx3 = fx4;
            end
```

```
        else
            x4 = (x1+x2) / 2;
            fx4 = height_func(x4);
            if fx4 > fx2
                x3 = x2;  fx3 = fx2;
                x2 = x4;  fx2 = fx4;
            else
                x1 = x4;  fx1 = fx4;
            end
        end

        if abs(x3-x1) < 1e-2
            break
        end
    end
    res = [x1 x2 x3];
end
```

The input variable is a vector that contains three values that bracket a maximum; in this case they are angles in degrees. `optimize` starts by evaluating `height_func` for each of the three values. We assume that `height_func` returns the quantity we want to optimize; for the Green Monster Problem it is the height of the ball when it reaches the wall.

Each time through the `for` loop the function chooses a value of `x4`, evaluates `height_func`, and then updates the triplet `x1`, `x2` and `x3` according to the results.

After the update, it computes the range of the bracket, `x3-x1`, and checks whether it is small enough. If so, it breaks out of the loop and returns the current triplet as a result. In the worst case the loop executes 50 times.

**Exercise 11.2** *I call this algorithm a Silver Section Search because it is almost as good as a Golden Section Search. Read the Wikipedia page about the Golden Section Search (`http://en.wikipedia.org/wiki/Golden_section_search`) and then modify this code to implement it.*

**Exercise 11.3** *You can write functions that take function handles as input variables, just as* `fzero` *and* `ode45` *do. For example,* `handle_func` *takes a function handle called* `func` *and calls it, passing* `pi` *as an argument.*

```
function res = handle_func(func)
    func(pi)
end
```

*You can call* `handle_func` *from the Command Window and pass different function handles as arguments:*

```
>> handle_func(@sin)
```

```
ans = 0

>> handle_func(@cos)

ans = -1
```

*Modify* `optimize` *so that it takes a function handle as an input variable and uses it as the function to be optimized.*

**Exercise 11.4** *The MATLAB function* `fminsearch` *takes a function handle and searches for a local minimum. Read the documentation for* `fminsearch` *and use it to find the optimal launch angle of a baseball with a given velocity.*

## 11.4 Discrete and continuous maps

When you solve an ODE analytically, the result is a function, which you can think of as a continuous map. When you use an ODE solver, you get two vectors (or a vector and a matrix), which you can think of as a discrete map.

For example, in Section 8.4, we used the following rate function to estimate the population of rats as a function of time:

```
function res = rats(t, y)
    a = 0.01;
    omega = 2 * pi / 365;
    res = a * y * (1 + sin(omega * t));
end
```

The result from `ode45` is two vectors:

```
>> [T, Y] = ode45(@rats, [0, 365], 2);
```

`T` contains the time values where `ode45` estimated the population; `Y` contains the population estimates.

Now suppose we would like to know the population on the 180th day of the year. We could search `T` for the value 180:

```
>> find(T==180)

ans = Empty matrix: 0-by-1
```

But there is no guarantee that any particular value appears in `T`. We can find the index where `T` crosses 180:

```
>> I = find(T>180); I(1)

ans = 23
```

I gets the indices of all elements of T greater than 180, so I(1) is the index of the *first* one.

Then we find the corresponding value from Y:

```
>> [T(23), Y(23)]

ans = 184.3451   40.3742
```

That gives us a coarse estimate of the population on Day 180. If we wanted to do a little better, we could also find the last value before Day 180:

```
>> [T(22), Y(22)]

ans = 175.2201   36.6973
```

So the population on Day 180 was between 36.6973 and 40.3742.

But where in this range is the best estimate? A simple option is to choose whichever time value is closer to 180 and use the corresponding population estimate. In the example, that's not a great choice because the time value we want is right in the middle.

## 11.5 Interpolation

A better option is to draw a straight line between the two points that bracket Day 180 and use the line to estimate the value in between. This process is called **linear interpolation**, and MATLAB provides a function named interp1 that does it:

```
>> pop = interp1(T, Y, 180)

pop = 38.6233
```

The first two arguments specify a discrete map from the values in T to the values in Y. The third argument is the time value where we want to interpolate. The result is what we expected, about halfway between the values that bracket it.

interp1 can also take a fourth argument that specifies what kind of interpolation you want. The default is 'linear', which does linear interpolation. Other choices include 'spline' which uses a spline curve to fit two points on either side, and 'cubic', which uses piecewise cubic Hermite interpolation.

```
>> pop = interp1(T, Y, 180, 'spline')

pop = 38.6486

>> pop = interp1(T, Y, 180, 'cubic')

pop = 38.6491
```

In this case we expect the spline and cubic interpolations to be better than linear, because they use more of the data, and we know the function isn't linear. But we have no reason to expect the spline to be more accurate than the cubic, or the other way around. Fortunately, they are not very different.

We can also use `interp1` to project the rat population out beyond the values in `T`:

```
>> [T(end), Y(end)]

ans = 365.0000   76.9530

>> pop = interp1(T, Y, 370, 'cubic')

pop = 80.9971
```

This process is called **extrapolation**. For time values near 365, extrapolation may be reasonable, but as we go farther into the "future," we expect them to be less accurate. For example, here is the estimate we get by extrapolating for a whole year:

```
>> pop = interp1(T, Y, 365*2, 'cubic')

pop = -4.8879e+03
```

And that's wrong. So very wrong.

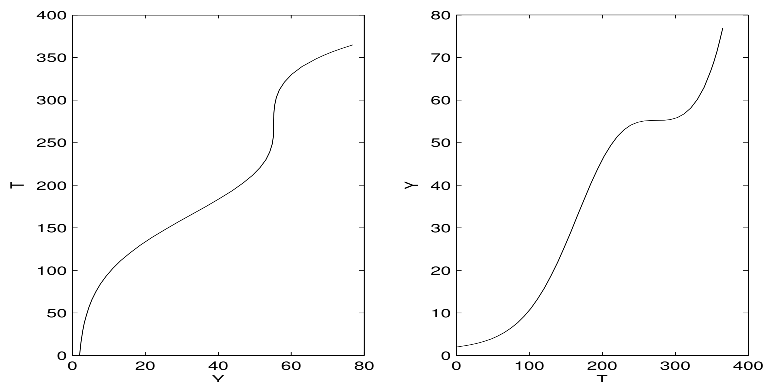
## 11.6 Interpolating the inverse function

We have used `interp1` to find population as a function of time; by reversing the roles of `T` and `Y`, we can also interpolate time as a function of population. For example, we might want to know how long it takes the population to reach 20.

```
>> interp1(Y, T, 20)

ans = 133.4128
```

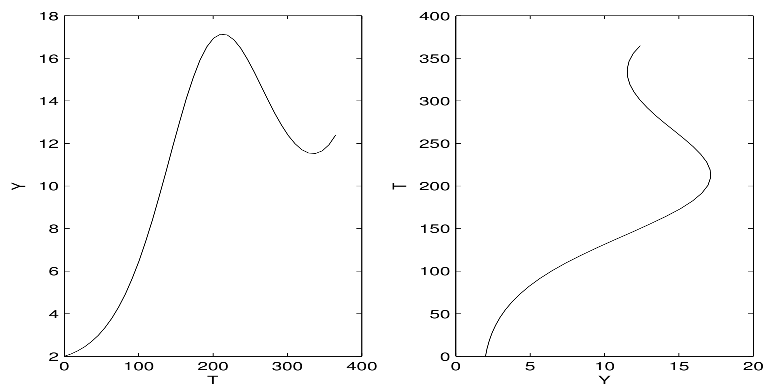
This use of `interp1` might be confusing if you think of the arguments as $x$ and $y$. You might find it helpful to think of them as the range and domain of a map (where the third argument is an element of the range).

The following plot shows $f$ (`Y` plotted as a function of `T`) and the inverse of $f$ (`T` plotted as a function of `Y`).




In this case we can use `interp1` either way because $f$ is a **single-valued mapping**, which means that for each value in the domain, there is only one value in the range that maps to it.

If we reduce the food supply so that the rat population decreases during the winter, we might see something like this:




We can still use `interp1` to map from `T` to `Y`:

```
>> interp1(T, Y, 260)

ans = 15.0309
```

So on Day 260, the population is about 15, but if we ask on what day the population was 15, there are two possible answers, 172.44 and 260.44. If we try to use `interp1`, we get the wrong answer:

```
>> interp1(Y, T, 15)

ans = 196.3833                    % WRONG
```

On Day 196, the population is actually 16.8, so `interp1` isn't even close! The problem is that `T` as a function of `Y` is a **multivalued mapping**; for some values

in the range there are more than one values in the domain. This causes `interp1` to fail. I can't find any documentation for this limitation, so that's pretty bad.

## 11.7 Field mice

As we've seen, one use of interpolation is to interpret the results of a numerical computation; another is to fill in the gaps between discrete measurements.

For example*, suppose that the population of field mice is governed by this rate equation:

$$g(t, y) = ay - b(t)y^{1.7}$$

where $t$ is time in months, $y$ is population, $a$ is a parameter that characterizes population growth in the absence of limitations, and $b$ is a function of time that characterizes the effect of the food supply on the death rate.

Although $b$ appears in the equation as a continuous function, we might not know $b(t)$ for all $t$. Instead, we might only have discrete measurements:

```
t           b(t)
-           ----
0           0.0070
1           0.0036
2           0.0011
3           0.0001
4           0.0004
5           0.0013
6           0.0028
7           0.0043
8           0.0056
```

If we use `ode45` to solve the differential equation, then we don't get to choose the values of $t$ where the rate function (and therefore $b$) gets evaluated. We need to provide a function that can evaluate $b$ everywhere:

```
function res = interpolate_b(t)
    T = 0:8;
    B = [70 36 11 1 4 13 28 43 56] * 1e-4;
    res = interp1(T, B, t);
end
```

Abstractly, this function uses a discrete map to implement a continuous map.

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

**Exercise 11.5** *Write a rate function that uses* `interpolate_b` *to evaluate $g$ and then use* `ode45` *to compute the population of field mice from $t = 0$ to $t = 8$ with an initial population of 100 and $a = 0.9$.*

*Then modify* `interpolate_b` *to use spline interpolation and run* `ode45` *again to see how much effect the interpolation has on the results.*

## 11.8 Glossary

**interpolation:** Estimating the value of a function using known values on either side.

**extrapolation:** Estimating the value of a function using known values that don't bracket the desired value.

**single-valued mapping:** A mapping where each value in the range maps to a single value in the domain.

**multivalued mapping:** A mapping where at least one value in the range maps to more than one value in the domain.

## 11.9 Exercises

**Exercise 11.6** *A golf ball*[†] *hit with backspin generates lift, which might increase the range, but the energy that goes into generating spin probably comes at the cost of lower initial velocity. Write a simulation of the flight of a golf ball and use it to find the launch angle and allocation of spin and initial velocity (for a fixed energy budget) that maximizes the horizontal range of the ball in the air.*

*The lift of a spinning ball is due to the Magnus force*[‡]*, 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.*

[†] See `http://en.wikipedia.org/wiki/Golf_ball`.
[‡] See `http://en.wikipedia.org/wiki/Magnus_effect`.

# Chapter 12

# Vectors as vectors

## 12.1 What's a vector?

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 in the initial conditions in a state vector. Then when `ode45` calls your rate function, it gives you a state vector.

**discrete maps:** If you have two vectors with the same length, you can think of them as a mapping from the elements of one vector to the corresponding elements of the other. For example, in Section 8.5, the results from `ode45` are vectors, `T` and `Y`, that represent a mapping from the time values in `T` to the population values in `Y`.

In this chapter we will see another use of MATLAB vectors: representing spatial vectors. A spatial vector is a value that represents a multidimensional physical quantity like position, velocity, acceleration or force*.

These quantities cannot be described with a single number because they contain multiple components. For example, in a 3-dimensional Cartesian coordinate space, it takes three numbers to specify a position in space; they are usually called $x$, $y$ and $z$ coordinates. As another example, in 2-dimensional polar

*See `http://en.wikipedia.org/wiki/Vector_(spatial)`.

coordinates, you can specify a velocity with two numbers, a magnitude and an angle, often called $r$ and $\theta$.

It is convenient to represent spatial vectors using MATLAB vectors because MATLAB knows how to perform most of the vector operations you need for physical modeling. For example, suppose that you are given the velocity of a baseball in the form of a MATLAB vector with two elements, $v_x$ and $v_y$, which are the components of velocity in the $x$ and $y$ directions.

```
>> V = [30, 40]         % velocity in m/s
```

And suppose you are asked to compute the total acceleration of the ball due to drag and gravity. In math notation, the force due to drag is

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

where $V$ is a spatial vector representing velocity, $v$ is the magnitude of the velocity (sometimes called "speed"), and $\hat{V}$ is a unit vector in the direction of the velocity vector. The other terms, $\rho$, $A$ and $C_d$, are scalars.

The magnitude of a vector is the square root of the sum of the squares of the elements. You could compute it with `hypotenuse` from Section 5.5, or you could use the MATLAB function `norm` (**norm** is another name[†] for the magnitude of a vector):

```
>> v = norm(V)

v = 50
```

$\hat{V}$ is a **unit vector**, which means it should have norm 1, and it should point in the same direction as $V$. The simplest way to compute it is to divide $V$ by its own norm.

```
>> Vhat = V / v

Vhat =  0.6  0.8
```

Then we can confirm that the norm of $\hat{V}$ is 1:

```
>> norm(Vhat)

ans = 1
```

To compute $F_d$ we just multiply the scalar terms by $\hat{V}$.

```
Fd = - 1/2 * C * rho * A * v^2 * Vhat
```

Similarly, we can compute acceleration by dividing the vector $F_d$ by the scalar $m$.

```
Ad = Fd / m
```

[†]Magnitude is also called "length" but I will avoid that term because it gets confused with the `length` function, which returns the number of elements in a MATLAB vector.

To represent the acceleration of gravity, we create a vector with two components:

```
Ag = [0; -9.8]
```

The $x$ component of gravity is 0; the $y$ component is $-9.8m/s^2$.

Finally we compute total acceleration by adding vector quantities:

```
A = Ag + Ad;
```

One nice thing about this computation is that we didn't have to think much about the components of the vectors. By treating spatial vectors as basic quantities, we can express complex computations concisely.

## 12.2 Dot and cross products

Multiplying a vector by a scalar is a straightforward operation; so is adding two vectors. But multiplying two vectors is more subtle. It turns out that there are two vector operations that resemble multiplication: **dot product** and **cross product**.

The dot product of vectors $A$ and $B$ is a scalar:

$$d = ab\cos\theta$$

where $a$ is the magnitude of $A$, $b$ is the magnitude of $B$, and $\theta$ is the angle between the vectors. We already know how to compute magnitudes, and you could probably figure out how to compute $\theta$, but you don't have to. MATLAB provides a function, `dot`, that computes dot products.

```
d = dot(A, B)
```

`dot` works in any number of dimensions, as long as `A` and `B` have the same number of elements.

If one of the operands is a unit vector, you can use the dot product to compute the component of a vector $A$ that is in the direction of a unit vector, $\hat{\imath}$:

```
s = dot(A, ihat)
```

In this example, $s$ is the **scalar projection** of $A$ onto $\hat{\imath}$. The **vector projection** is the vector that has magnitude $s$ in the direction of $\hat{\imath}$:

```
V = dot(A, ihat) * ihat
```

The cross product of vectors $A$ and $B$ is a vector whose direction is perpendicular to $A$ and $B$ and whose magnitude is

$$c = ab\sin\theta$$

where (again) $a$ is the magnitude of $A$, $b$ is the magnitude of $B$, and $\theta$ is the angle between the vectors. MATLAB provides a function, `cross`, that computes cross products.

```
C = cross(A, B)
```

`cross` only works for 3-dimensional vectors; the result is a 3-dimensional vector.

A common use of `cross` is to compute torques. If you represent a moment arm $R$ and a force $F$ as 3-dimensional vectors, then the torque is just

```
Tau = cross(R, F)
```

If the components of `R` are in meters and the components of `F` are in Newtons, then the torques in `Tau` are in Newton-meters.

## 12.3 Celestial mechanics

Modeling celestial mechanics is a good opportunity to compute with spatial vectors. Imagine a star with mass $m_1$ at a point in space described by the vector $P_1$, and a planet with mass $m_2$ at point $P_2$. The magnitude of the gravitational force[‡] between them is

$$f_g = G\frac{m_1 m_2}{r^2}$$

where $r$ is the distance between them and $G$ is the universal gravitational constant, which is about $6.67 \times 10^{-11} Nm^2/kg^2$. Remember that this is the appropriate value of $G$ only if the masses are in kilograms, distances in meters, and forces in Newtons.

The direction of the force on the star at $P_1$ is in the direction toward $P_2$. We can compute relative direction by subtracting vectors; if we compute `R = P2 - P1`, then the direction of `R` is from `P1` to `P2`.

The distance between the planet and star is the length of $R$:

```
r = norm(R)
```

The direction of the force on the star is $\hat{R}$:

```
rhat = R / r
```

**Exercise 12.1** *Write a sequence of MATLAB statements that computes* `F12`*, a vector that represents the force on the star due to the planet, and* `F21`*, the force on the planet due to the star.*

**Exercise 12.2** *Encapsulate these statements in a function named* `gravity_force_func` *that takes* `P1`*,* `m1`*,* `P2`*, and* `m2` *as input variables and returns* `F12`*.*

**Exercise 12.3** *Write a simulation of the orbit of Jupiter around the Sun. The mass of the Sun is about $2.0 \times 10^{30}$ kg. You can get the mass of Jupiter, its distance from the Sun and orbital velocity from* `http://en.wikipedia.org/wiki/Jupiter`*. Confirm that it takes about 4332 days for Jupiter to orbit the Sun.*

[‡]See `http://en.wikipedia.org/wiki/Gravity`

## 12.4 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 an example I wrote for Exercise 12.3:

```
function animate_func(T,M)
    % animate the positions of the planets, assuming that the
    % columns of M are x1, y1, x2, y2.
    X1 = M(:,1);
    Y1 = M(:,2);
    X2 = M(:,3);
    Y2 = M(:,4);

    minmax = [min([X1;X2]), max([X1;X2]), min([Y1;Y2]), max([Y1;Y2])];

    for i=1:length(T)
        clf;
        axis(minmax);
        hold on;
        draw_func(X1(i), Y1(i), X2(i), Y2(i));
        drawnow;
    end
end
```

The input variables are the output from `ode45`, a vector `T` and a matrix `M`. The columns of `M` are the positions and velocities of the Sun and Jupiter, so `X1` and `Y1` get the coordinates of the Sun; `X2` and `Y2` get the coordinates of Jupiter.

`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_func` uses `clf` to clear the figure and `axis` to reset the axes. `hold on` makes it possible to put more than one plot onto the same axes (otherwise MATLAB clears the figure each time you call `plot`).

Each time through the loop, 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.

`draw_func` is the function that actually makes the plot:

```
function draw_func(x1, y1, x2, y2)
    plot(x1, y1, 'r.', 'MarkerSize', 50);
    plot(x2, y2, 'b.', 'MarkerSize', 20);
end
```

The input variables are the position of the Sun and Jupiter. `draw_func` uses `plot` to draw the Sun as a large red marker and Jupiter as a smaller blue one.

**Exercise 12.4** *To make sure you understand how* `animate_func` *works, try commenting out some of the lines to see what happens.*

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.

There are two ways to fix this problem:

1. When you call `ode45` you can give it a vector of points in time where it should generate estimates. Here is an example:

   ```
   end_time = 1000;
   step = end_time/200;
   [T, M] = ode45(@rate_func, [0:step:end_time], W);
   ```

   The second argument is a range vector that goes from 0 to 1000 with a step size determined by `step`. Since `step` is `end_time/200`, there will be about 200 rows in `T` and `M` (201 to be precise).

   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.

2. You can 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.

**Exercise 12.5** *Use* `animate_func` *and* `draw_func` *to vizualize your simulation of Jupiter. Modify it so it shows one day of simulated time in 0.001 seconds of real time—one revolution should take about 4.3 seconds.*

## 12.5 Conservation of Energy

A useful way to check the accuracy of an ODE solver is to see whether it conserves energy. For planetary motion, it turns out that `ode45` does not.

The kinetic energy of a moving body is $mv^2/2$; the kinetic energy of a solar system is the total kinetic energy of the planets and sun. The potential energy of a sun with mass $m_1$ and a planet with mass $m_2$ and a distance $r$ between them is

$$U = -G\frac{m_1 m_2}{r}$$

**Exercise 12.6** *Write a function called* `energy_func` *that takes the output of your Jupiter simulation,* `T` *and* `M`*, 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 confirm that it decreases over the course of the simulation. Your function should also compute the relative change in energy, the difference between the energy at the beginning and end, as a percentage of the starting energy.*

You can reduce the rate of energy loss by decreasing `ode45`'s tolerance option using `odeset` (see Section 11.1):

```
options = odeset('RelTol', 1e-5);
[T, M] = ode45(@rate_func, [0:step:end_time], 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.

**Exercise 12.7** *Run* `ode45` *with a range of values for* `RelTol` *and confirm that as the tolerance gets smaller, the rate of energy loss decreases.*

**Exercise 12.8** *Run your simulation with one of the other ODE solvers MATLAB provides and see if any of them conserve energy.*

## 12.6 What is a model for?

In Section 7.2 I defined a "model" as a simplified description of a physical system, and said that a good model lends itself to analysis and simulation, and makes predictions that are good enough for the intended purpose.

Since then, we have seen a number of examples; now we can say more about what models are for. The goals of a model tend to fall into three categories.

**prediction:** Some models make predictions about physical systems. As a simple example, the duck model in Exercise 6.2 predicts the level a duck floats at. At the other end of the spectrum, global climate models try to predict the weather tens or hundreds of years in the future.

**design:** Models are useful for engineering design, especially for testing the feasibility of a design and for optimization. For example, in Exercise 11.6 you were asked to design the golf swing with the perfect combination of launch angle, velocity and spin.

**explanation:** Models can answer scientific questions. For example, the Lotka-Volterra model in Section 9.4 offers a possible explanation of the dynamics of animal populations systems in terms of interactions between predator and prey species.

The exercises at the end of this chapter include one model of each type.

## 12.7 Glossary

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

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

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

**dot product:** A scalar product of two vectors, proportional to the norms of the vectors and the cosine of the angle between them.

**cross product:** A vector product of two vectors with norm proportional to the norms of the vectors and the sine of the angle between them, and direction perpendicular to both.

**projection:** The component of one vector that is in the direction of the other (might be used to mean "scalar projection" or "vector projection").

## 12.8 Exercises

**Exercise 12.9** *If you put two identical bowls of water into a freezer, one at room temperature and one boiling, which one freezes first?*

*Hint: you might want to do some research on the Mpemba effect.*

**Exercise 12.10** *You have been asked to design a new skateboard ramp; unlike a typical skateboard ramp, this one is free to pivot about a support point. Skateboarders approach the ramp on a flat surface and then coast up the ramp; they are not allowed to put their feet down while on the ramp. If they go fast enough, the ramp will rotate and they will gracefully ride down the rotating ramp. Technical and artistic display will be assessed by the usual panel of talented judges.*

*Your job is to design a ramp that will allow a rider to accomplish this feat, and to create a physical model of the system, a simulation that computes the behavior of a rider on the ramp, and an animation of the result.*

**Exercise 12.11** *A binary star system contains two stars orbiting each other and sometimes planets that orbit one or both stars*[§]*. In a binary system, some orbits are "stable" in the sense that a planet can stay in orbit without crashing into one of the stars or flying off into space.*

*Simulation is a useful tool for investigating the nature of these orbits, as in Holman, M.J. and P.A. Wiegert, 1999, "Long-Term Stability of Planets in Binary Systems,"* Astronomical Journal *117, available from* `http://citeseer.ist.psu.edu/358720.html`.

*Read this paper and then modify your planetary simulation to replicate or extend the results.*

[§]See `http://en.wikipedia.org/wiki/Binary_star`.

# Experiments with MATLAB®

Cleve Moler

Copyright © 2011 Cleve Moler.

All rights reserved. No part of this e-book may be reproduced, stored, or transmitted in any manner without the written permission of the author. For more information, contact moler@mathworks.com.

The programs described in this e-book have been included for their instructional value. These programs have been tested with care but are not guaranteed for any particular purpose. The author does not offer any warranties or representations, nor does he accept any liabilities with respect to the use of the programs. These programs should not be relied on as the sole basis to solve a problem whose incorrect solution could result in injury to person or property.

MATLAB® is a registered trademark of MathWorks, Inc.™.

For more information about relevant MathWorks policies, see:

`http://www.mathworks.com/company/aboutus/policies_statements`

---

October 4, 2011

# Preface

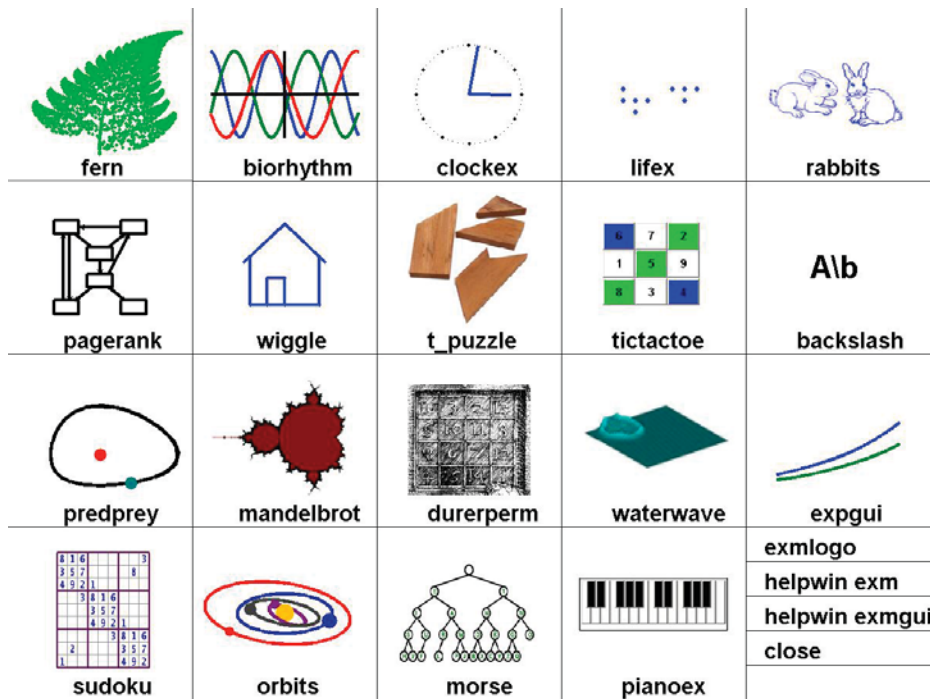



**Figure 1.** *exmgui provides a starting point for some of the experiments.*

Welcome to *Experiments with MATLAB*. This is not a conventional book. It is currently available only via the Internet, at no charge, from

`http://www.mathworks.com/moler`

There may eventually be a hardcopy edition, but not right away.

Although MATLAB is now a full fledged Technical Computing Environment, it started in the late 1970s as a simple "Matrix Laboratory". We want to build on this laboratory tradition by describing a series of experiments involving applied mathematics, technical computing, and MATLAB programming.

We expect that you already know something about high school level material in geometry, algebra, and trigonometry. We will introduce ideas from calculus, matrix theory, and ordinary differential equations, but we do not assume that you have already taken courses in the subjects. In fact, these experiments are useful supplements to such courses.

We also expect that you have some experience with computers, perhaps with word processors or spread sheets. If you know something about programming in languages like C or Java, that will be helpful, but not required. We will introduce MATLAB by way of examples. Many of the experiments involve understanding and modifying MATLAB scripts and functions that we have already written.

You should have access to MATLAB and to our `exm` toolbox, the collection of programs and data that are described in *Experiments with MATLAB*. We hope you will not only use these programs, but will read them, understand them, modify them, and improve them. The `exm` toolbox is the apparatus in our "Laboratory".

You will want to have MATLAB handy. For information about the Student Version, see

```
http://www.mathworks.com/academia/student_version
```

For an introduction to the mechanics of using MATLAB, see the videos at

```
http://www.mathworks.com/academia/student_version/start.html
```

For documentation, including "Getting Started", see

```
http://www.mathworks.com/access/helpdesk/help/techdoc/matlab.html
```

For user contributed programs, programming contests, and links into the world-wide MATLAB community, check out

```
http://www.mathworks.com/matlabcentral
```

To get started, download the `exm` toolbox, use `pathtool` to add `exm` to the MATLAB path, and run `exmgui` to generate figure 1. You can click on the icons to preview some of the experiments.

You will want to make frequent use of the MATLAB help and documentation facilities. To quickly learn how to use the command or function named `xxx`, enter

```
help xxx
```

For more extensive information about `xxx`, use

```
doc xxx
```

We hope you will find the experiments interesting, and that you will learn how to use MATLAB along the way. Each chapter concludes with a "Recap" section that is actually an executable MATLAB program. For example, you can review the Magic Squares chapter by entering

```
magic_recap
```

Better yet, enter

```
edit magic_recap
```

and run the program cell-by-cell by simultaneously pressing the Ctrl-Shift-Enter keys.

A fairly new MATLAB facility is the `publish` command. You can get a nicely formatted web page about `magic_recap` with

```
publish magic_recap
```

If you want to concentrate on learning MATLAB, make sure you read, run, and understand the recaps.

Cleve Moler
October 4, 2011

## Chapter 1

# Iteration

*Iteration is a key element in much of technical computation. Examples involving the Golden Ratio introduce the* MATLAB *assignment statement,* `for` *and* `while` *loops, and the* `plot` *function.*

Start by picking a number, any number. Enter it into MATLAB by typing

```
x = your number
```

This is a MATLAB *assignment statement*. The number you chose is stored in the *variable* `x` for later use. For example, if you start with

```
x = 3
```

MATLAB responds with

```
x =
     3
```

Next, enter this statement

```
x = sqrt(1 + x)
```

The abbreviation `sqrt` is the MATLAB name for the square root function. The quantity on the right, $\sqrt{1+x}$, is computed and the result stored back in the variable `x`, overriding the previous value of `x`.

Somewhere on your computer keyboard, probably in the lower right corner, you should be able to find four arrow keys. These are the *command line editing* keys. The up-arrow key allows you to recall earlier commands, including commands from

Copyright © 2011 Cleve Moler
MATLAB® is a registered trademark of MathWorks, Inc.™
October 4, 2011

previous sessions, and the other arrows keys allow you to revise these commands. Use the up-arrow key, followed by the enter or return key, to iterate, or repeatedly execute, this statement:

```
x = sqrt(1 + x)
```

Here is what you get when you start with `x = 3`.

```
x =
     3
x =
     2
x =
    1.7321
x =
    1.6529
x =
    1.6288
x =
    1.6213
x =
    1.6191
x =
    1.6184
x =
    1.6181
x =
    1.6181
x =
    1.6180
x =
    1.6180
```

These values are 3, $\sqrt{1+3}$, $\sqrt{1+\sqrt{1+3}}$, $\sqrt{1+\sqrt{1+\sqrt{1+3}}}$, and so on. After 10 steps, the value printed remains constant at `1.6180`. Try several other starting values. Try it on a calculator if you have one. You should find that no matter where you start, you will always reach `1.6180` in about ten steps. (Maybe a few more will be required if you have a very large starting value.)

MATLAB is doing these computations to accuracy of about 16 decimal digits, but is displaying only five. You can see more digits by first entering

```
format long
```

and repeating the experiment. Here are the beginning and end of 30 steps starting at $x = 3$.

```
x =
     3
```

```
x =
    2
x =
   1.732050807568877
x =
   1.652891650281070

    ....

x =
   1.618033988749897
x =
   1.618033988749895
x =
   1.618033988749895
```

After about thirty or so steps, the value that is printed doesn't change any more. You have computed one of the most famous numbers in mathematics, $\phi$, the *Golden Ratio*.

In MATLAB, and most other programming languages, the equals sign is the assignment operator. It says compute the value on the right and store it in the variable on the left. So, the statement

```
x = sqrt(1 + x)
```

takes the current value of `x`, computes `sqrt(1 + x)`, and stores the result back in `x`.

In mathematics, the equals sign has a different meaning.

$$x = \sqrt{1 + x}$$

is an *equation*. A solution to such an equation is known as a *fixed point*. (Be careful not to confuse the mathematical usage of *fixed point* with the computer arithmetic usage of *fixed point*.)

The function $f(x) = \sqrt{1 + x}$ has exactly one fixed point. The best way to find the value of the fixed point is to avoid computers all together and solve the equation using the quadratic formula. Take a look at the hand calculation shown in figure 1.1. The positive root of the quadratic equation is the Golden Ratio.

$$\phi = \frac{1 + \sqrt{5}}{2}.$$

You can have MATLAB compute $\phi$ directly using the statement

```
phi = (1 + sqrt(5))/2
```

With `format long`, this produces the same value we obtained with the fixed point iteration,

```
phi =
   1.618033988749895
```

$$x = \sqrt{1+x}$$
$$x^2 = 1 + x$$
$$x^2 - x - 1 = 0$$
$$x = \frac{1 \pm \sqrt{1+4}}{2}$$
$$\phi = \frac{1+\sqrt{5}}{2}$$

**Figure 1.1.** *Compute the fixed point by hand.*

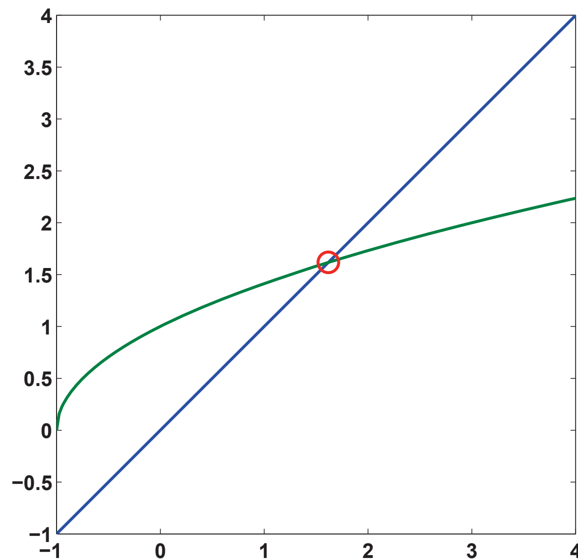



**Figure 1.2.** *A fixed point at $\phi$ = 1.6180.*

Figure 1.2 is our first example of MATLAB graphics. It shows the intersection of the graphs of $y = x$ and $y = \sqrt{1+x}$. The statement

```
x = -1:.02:4;
```

generates a vector `x` containing the numbers from -1 to 4 in steps of .02. The statements

```
y1 = x;
y2 = sqrt(1+x);
plot(x,y1,'-',x,y2,'-',phi,phi,'o')
```

produce a figure that has three components. The first two components are graphs of $x$ and $\sqrt{1+x}$. The `'-'` argument tells the `plot` function to draw solid lines. The last component in the plot is a single point with both coordinates equal to $\phi$. The `'o'` tells the `plot` function to draw a circle.

The MATLAB `plot` function has many variations, including specifying other colors and line types. You can see some of the possibilities with

```
help plot
```

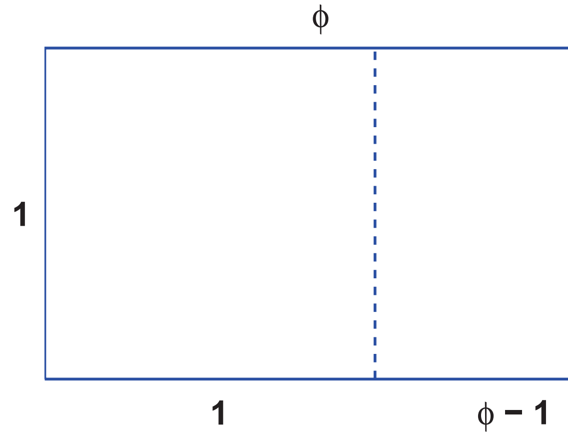



**Figure 1.3.** *The golden rectangle.*

The Golden Ratio shows up in many places in mathematics; we'll see several in this book. The Golden Ratio gets its name from the golden rectangle, shown in figure 1.3. The golden rectangle has the property that removing a square leaves a smaller rectangle with the same shape. Equating the aspect ratios of the rectangles gives a defining equation for $\phi$:

$$\frac{1}{\phi} = \frac{\phi - 1}{1}.$$

Multiplying both sides of this equation by $\phi$ produces the same quadratic polynomial equation that we obtained from our fixed point iteration.

$$\phi^2 - \phi - 1 = 0.$$

The up-arrow key is a convenient way to repeatedly execute a single statement, or several statements, separated by commas or semicolons, on a single line. Two more powerful constructs are the `for` loop and the `while` loop. A `for` loop executes a block of code a prescribed number of times.

```
x = 3
for k = 1:31
   x = sqrt(1 + x)
end
```

produces 32 lines of output, one from the initial statement and one more each time through the loop.

A `while` loop executes a block of code an unknown number of times. Termination is controlled by a logical expression, which evaluates to `true` or `false`. Here is the simplest `while` loop for our fixed point iteration.

```
x = 3
while x ~= sqrt(1+x)
   x = sqrt(1+x)
end
```

This produces the same 32 lines of output as the `for` loop. However, this code is open to criticism for two reasons. The first possible criticism involves the termination condition. The expression `x ~= sqrt(1+x)` is the MATLAB way of writing $x \neq \sqrt{1+x}$. With exact arithmetic, `x` would never be exactly equal to `sqrt(1+x)`, the condition would always be true, and the loop would run forever. However, like most technical computing environments, MATLAB does not do arithmetic exactly. In order to economize on both computer time and computer memory, MATLAB uses *floating point* arithmetic. Eventually our program produces a value of `x` for which the floating point numbers `x` and `sqrt(1+x)` are exactly equal and the loop terminates. Expecting exact equality of two floating point numbers is a delicate matter. It works OK in this particular situation, but may not work with more complicated computations.

The second possible criticism of our simple `while` loop is that it is inefficient. It evaluates `sqrt(1+x)` twice each time through the loop. Here is a more complicated version of the `while` loop that avoids both criticisms.

```
x = 3
y = 0;
while abs(x-y) > eps(x)
   y = x;
   x = sqrt(1+x)
end
```

The semicolons at the ends of the assignment statements involving `y` indicate that no printed output should result. The quantity `eps(x)`, is the spacing of the floating point numbers near `x`. Mathematically, the Greek letter $\epsilon$, or *epsilon*, often represents a "small" quantity. This version of the loop requires only one square root calculation per iteration, but that is overshadowed by the added complexity of the code. Both `while` loops require about the same execution time. In this situation, I prefer the first `while` loop because it is easier to read and understand.

## Help and Doc

MATLAB has extensive on-line documentation. Statements like

```
help sqrt
help for
```

provide brief descriptions of commands and functions. Statements like

```
doc sqrt
doc for
```

provide more extensive documentation in a separate window.

One obscure, but very important, `help` entry is about the various punctuation marks and special characters used by MATLAB. Take a look now at

```
help punct
doc punct
```

You will probably want to return to this information as you learn more about MATLAB.

## Numbers

Numbers are formed from the digits `0` through `9`, an optional decimal point, a leading `+` or `-` sign, an optional `e` followed by an integer for a power of 10 scaling, and an optional `i` or `j` for the imaginary part of a complex number. MATLAB also knows the value of $\pi$. Here are some examples of numbers.

```
42
9.6397238
6.0221415e23
-3+4i
pi
```

## Assignment statements and names

A simple assignment statement consists of a name, an = sign, and a number. The names of variables, functions and commands are formed by a letter, followed by any number of upper and lower case letters, digits and underscores. Single character names, like `x` and `N`, and anglicized Greek letters, like `pi` and `phi`, are often used to reflect underlying mathematical notation. Non-mathematical programs usually employ long variable names. Underscores and a convention known as camel casing are used to create variable names out of several words.

```
x = 42
phi = (1+sqrt(5))/2
Avogadros_constant = 6.0221415e23
camelCaseComplexNumber = -3+4i
```

## Expressions

Power is denoted by `^` and has precedence over all other arithmetic operations. Multiplication and division are denoted by `*`, `/`, and `\` and have precedence over addition and subtraction, Addition and subtraction are denoted by `+` and `-` and

have lowest precedence. Operations with equal precedence are evaluated left to right. Parentheses delineate subexpressions that are evaluated first. Blanks help readability, but have no effect on precedence.

All of the following expressions have the same value. If you don't already recognize this value, you can ask Google about its importance in popular culture.

```
3*4 + 5*6
3 * 4+5 * 6
2*(3 + 4)*3
-2^4 + 10*29/5
3\126
52-8-2
```

## Recap

```
%% Iteration Chapter Recap
% This is an executable program that illustrates the statements
% introduced in the Iteration chapter of "Experiments in MATLAB".
% You can run it by entering the command
%
%    iteration_recap
%
% Better yet, enter
%
%    edit iteration_recap
%
% and run the program cell-by-cell by simultaneously
% pressing the Ctrl-Shift-Enter keys.
%
% Enter
%
%   publish iteration_recap
%
% to see a formatted report.

%% Help and Documentation
%   help punct
%   doc punct

%% Format
    format short
    100/81
    format long
    100/81

    format short
```

```
    format compact

%% Names and assignment statements
    x = 42
    phi = (1+sqrt(5))/2
    Avogadros_constant = 6.0221415e23
    camelCaseComplexNumber = -3+4i

%% Expressions
   3*4 + 5*6
   3 * 4+5 * 6
   2*(3 + 4)*3
   -2^4 + 10*29/5
   3\126
   52-8-2

%% Iteration
% Use the up-arrow key to repeatedly execute
   x = sqrt(1+x)
   x = sqrt(1+x)
   x = sqrt(1+x)
   x = sqrt(1+x)

%% For loop
   x = 42
   for k = 1:12
      x = sqrt(1+x);
      disp(x)
   end

%% While loop
   x = 42;
   k = 1;
   while abs(x-sqrt(1+x)) > 5e-5
      x = sqrt(1+x);
      k = k+1;
   end
   k

%% Vector and colon operator
   k = 1:12
   x = (0.0: 0.1: 1.00)'

%% Plot
   x = -pi: pi/256: pi;
   y = tan(sin(x)) - sin(tan(x));
```

```
   z = 1 + tan(1);
   plot(x,y,'-', pi/2,z,'ro')
   xlabel('x')
   ylabel('y')
   title('tan(sin(x)) - sin(tan(x))')

%% Golden Spiral
   golden_spiral(4)
```

## Exercises

1.1 *Expressions.* Use Matlab to evaluate each of these mathematical expressions.

| | | |
|---|---|---|
| $43^2$ | $-3^4$ | $\sin 1$ |
| $4^{(3^2)}$ | $(-3)^4$ | $\sin 1^\circ$ |
| $(4^3)^2$ | $\sqrt[4]{-3}$ | $\sin \frac{\pi}{3}$ |
| $\sqrt[4]{32}$ | $-2^{-4/3}$ | $(\arcsin 1)/\pi$ |

You can get started with

```
help ^
help sin
```

1.2 *Temperature conversion.*
(a) Write a Matlab statement that converts temperature in Fahrenheit, `f`, to Celsius, `c`.

`c =` *something involving* `f`

(b) Write a Matlab statement that converts temperature in Celsius, `c`, to Fahrenheit, `f`.

`f =` *something involving* `c`

1.3 *Barn-megaparsec.* A *barn* is a unit of area employed by high energy physicists. Nuclear scattering experiments try to "hit the side of a barn". A *parsec* is a unit of length employed by astronomers. A star at a distance of one parsec exhibits a trigonometric parallax of one arcsecond as the Earth orbits the Sun. A *barn-megaparsec* is therefore a unit of volume – a very long skinny volume.

A barn is $10^{-28}$ square meters.
A megaparsec is $10^6$ parsecs.
A parsec is 3.262 light-years.
A light-year is $9.461 \cdot 10^{15}$ meters.
A cubic meter is $10^6$ milliliters.
A milliliter is $\frac{1}{5}$ teaspoon.

Express one barn-megaparsec in teaspoons. In MATLAB, the letter `e` can be used to denote a power of 10 exponent, so $9.461 \cdot 10^{15}$ can be written `9.461e15`.

1.4 *Complex numbers.* What happens if you start with a large negative value of `x` and repeatedly iterate

```
x = sqrt(1 + x)
```

1.5 *Comparison.* Which is larger, $\pi^\phi$ or $\phi^\pi$?

1.6 *Solving equations.* The best way to solve

$$x = \sqrt{1+x}$$

or

$$x^2 = 1 + x$$

is to avoid computers all together and just do it yourself by hand. But, of course, MATLAB and most other mathematical software systems can easily solve such equations. Here are several possible ways to do it with MATLAB. Start with

```
format long
phi = (1 + sqrt(5))/2
```

Then, for each method, explain what is going on and how the resulting `x` differs from `phi` and the other `x`'s.

```
% roots
help roots
x1 = roots([1 -1 -1])

% fsolve
help fsolve
f = @(x) x-sqrt(1+x)
p = @(x) x^2-x-1
x2 = fsolve(f, 1)
x3 = fsolve(f, -1)
x4 = fsolve(p, 1)
x5 = fsolve(p, -1)

% solve (requires Symbolic Toolbox or Student Version)
help solve
help syms
syms x
x6 = solve('x-sqrt(1+x)=0')
x7 = solve(x^2-x-1)
```

1.7 *Symbolic solution.* If you have the Symbolic Toolbox or Student Version, explain what the following program does.

```
x = sym('x')
length(char(x))
for k = 1:10
   x = sqrt(1+x)
   length(char(x))
end
```

1.8 *Fixed points.* Verify that the Golden Ratio is a fixed point of each of the following equations.

$$\phi = \frac{1}{\phi - 1}$$

$$\phi = \frac{1}{\phi} + 1$$

Use each of the equations as the basis for a fixed point iteration to compute $\phi$. Do the iterations converge?

1.9 *Another iteration.* Before you run the following program, predict what it will do. Then run it.

```
x = 3
k = 1
format long
while x ~= sqrt(1+x^2)
   x = sqrt(1+x^2)
   k = k+1
end
```

1.10 *Another fixed point.* Solve this equation by hand.

$$x = \frac{1}{\sqrt{1 + x^2}}$$

How many iterations does the following program require? How is the final value of $x$ related to the Golden Ratio $\phi$?

```
x = 3
k = 1
format long
while x ~= 1/sqrt(1+x^2)
   x = 1/sqrt(1+x^2)
   k = k+1
end
```

1.11 *cos(x).* Find the numerical solution of the equation

$$x = \cos x$$

in the interval $[0, \frac{\pi}{2}]$, shown in figure 1.4.

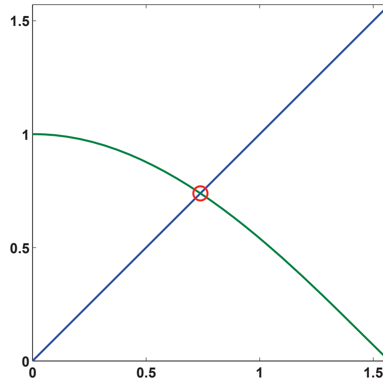



**Figure 1.4.** *Fixed point of* $x = cos(x)$.

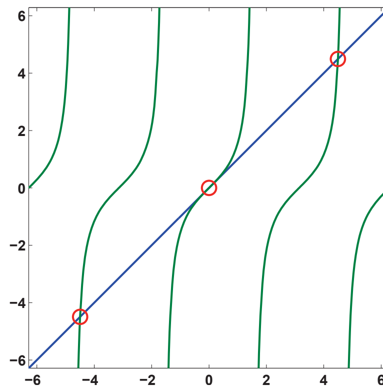



**Figure 1.5.** *Three fixed points of* $x = tan(x)$

1.12 *tan(x).* Figure 1.5 shows three of the many solutions to the equation

$$x = \tan x$$

One of the solutions is $x = 0$. The other two in the plot are near $x = \pm 4.5$. If we did a plot over a large range, we would see solutions in each of the intervals $[(n - \frac{1}{2})\pi, (n + \frac{1}{2})\pi]$ for integer $n$.
(a) Does this compute a fixed point?

```
x = 4.5
```

```
for k = 1:30
   x = tan(x)
end
```

(b) Does this compute a fixed point? Why is the " `+ pi`" necessary?

```
x = pi
while abs(x - tan(x)) > eps(x)
   x = atan(x) + pi
end
```

1.13 *Summation.* Write a mathematical expression for the quantity approximated by this program.

```
s = 0;
t = Inf;
n = 0;
while s ~= t
   n = n+1;
   t = s;
   s = s + 1/n^4;
end
s
```

1.14 *Why.* The first version of MATLAB written in the late 1970's, had `who`, `what`, `which`, and `where` commands. So it seemed natural to add a `why` command. Check out today's `why` command with

```
why
help why
for k = 1:40, why, end
type why
edit why
```

As the `help` entry says, please embellish or modify the `why` function to suit your own tastes.

1.15 *Wiggles.* A glimpse at MATLAB plotting capabilities is provided by the function

```
f = @(x) tan(sin(x)) - sin(tan(x))
```

This uses the '`@`' sign to introduce a simple function. You can learn more about the '`@`' sign with `help function_handle`.

Figure 1.6 shows the output from the statement

```
ezplot(f,[-pi,pi])
```

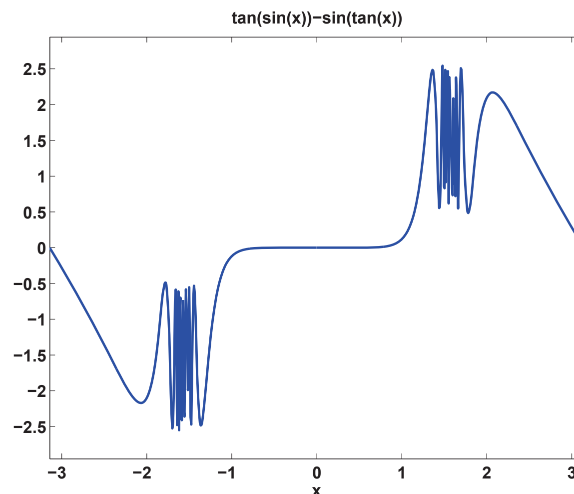



**Figure 1.6.** *A wiggly function.*

(The function name `ezplot` is intended to be pronounced "Easy Plot". This pun doesn't work if you learned to pronounce "z" as "zed".) You can see that the function is very flat near $x = 0$, oscillates infinitely often near $x = \pm\pi/2$ and is nearly linear near $x = \pm\pi$.

You can get more control over the plot with code like this.

```
x = -pi:pi/256:pi;
y = f(x);
plot(x,y)
xlabel('x')
ylabel('y')
title('A wiggly function')
axis([-pi pi -2.8 2.8])
set(gca,'xtick',pi*(-3:1/2:3))
```

(a) What is the effect of various values of `n` in the following code?

```
x = pi*(-2:1/n:2);
comet(x,f(x))
```

(b) This function is bounded. A numeric value near its maximum can be found with

```
max(y)
```

What is its analytic maximum? (To be precise, I should ask "What is the function's *supremum*?")

1.16 *Graphics.* We use a lot of computer graphics in this book, but studying MATLAB graphics programming is not our primary goal. However, if you are curious, the

script that produces figure 1.3 is `goldrect.m`. Modify this program to produce a graphic that compares the Golden Rectangle with TV screens having aspect ratios 4:3 and 16:9.

1.17 *Golden Spiral*

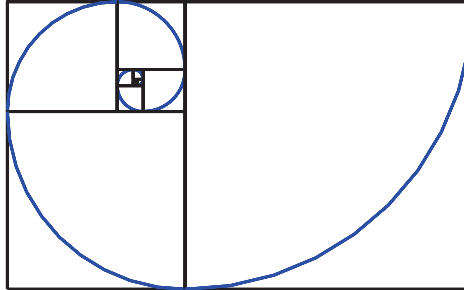


**Figure 1.7.** *A spiral formed from golden rectangles and inscribed quarter circles.*

Our program `golden_spiral` displays an ever-expanding sequence of golden rectangles with inscribed quarter circles. Check it out.

## Chapter 2

# Fibonacci Numbers

*Fibonacci numbers introduce vectors, functions and recursion.*

Leonardo Pisano Fibonacci was born around 1170 and died around 1250 in Pisa in what is now Italy. He traveled extensively in Europe and Northern Africa. He wrote several mathematical texts that, among other things, introduced Europe to the Hindu-Arabic notation for numbers. Even though his books had to be transcribed by hand, they were widely circulated. In his best known book, *Liber Abaci*, published in 1202, he posed the following problem:

> *A man puts a pair of rabbits in a place surrounded on all sides by a wall. How many pairs of rabbits can be produced from that pair in a year if it is supposed that every month each pair begets a new pair which from the second month on becomes productive?*

Today the solution to this problem is known as the *Fibonacci sequence*, or *Fibonacci numbers*. There is a small mathematical industry based on Fibonacci numbers. A search of the Internet for "Fibonacci" will find dozens of Web sites and hundreds of pages of material. There is even a Fibonacci Association that publishes a scholarly journal, the *Fibonacci Quarterly*.

A simulation of Fibonacci's problem is provided by our `exm` program `rabbits`. Just execute the command

```
rabbits
```

and click on the pushbuttons that show up. You will see something like figure 2.1.

If Fibonacci had not specified a month for the newborn pair to mature, he would not have a sequence named after him. The number of pairs would simply

---

Copyright © 2011 Cleve Moler
MATLAB® is a registered trademark of MathWorks, Inc.™
October 4, 2011

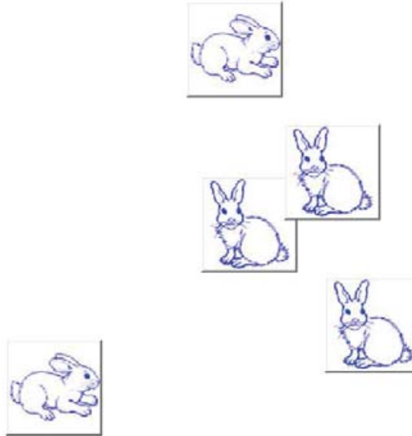


**Figure 2.1.** *Fibonacci's rabbits.*

double each month. After $n$ months there would be $2^n$ pairs of rabbits. That's a lot of rabbits, but not distinctive mathematics.

Let $f_n$ denote the number of pairs of rabbits after $n$ months. The key fact is that the number of rabbits at the end of a month is the number at the beginning of the month plus the number of births produced by the mature pairs:

$$f_n = f_{n-1} + f_{n-2}.$$

The initial conditions are that in the first month there is one pair of rabbits and in the second there are two pairs:

$$f_1 = 1, \quad f_2 = 2.$$

The following MATLAB function, stored in a file `fibonacci.m` with a `.m` suffix, produces a vector containing the first `n` Fibonacci numbers.

```
function f = fibonacci(n)
% FIBONACCI  Fibonacci sequence
% f = FIBONACCI(n) generates the first n Fibonacci numbers.
```

```
f = zeros(n,1);
f(1) = 1;
f(2) = 2;
for k = 3:n
   f(k) = f(k-1) + f(k-2);
end
```

With these initial conditions, the answer to Fibonacci's original question about the size of the rabbit population after one year is given by

```
fibonacci(12)
```

This produces

```
      1
      2
      3
      5
      8
     13
     21
     34
     55
     89
    144
    233
```

The answer is 233 pairs of rabbits. (It would be 4096 pairs if the number doubled every month for 12 months.)

Let's look carefully at `fibonacci.m`. It's a good example of how to create a MATLAB function. The first line is

```
function f = fibonacci(n)
```

The first word on the first line says `fibonacci.m` is a `function`, not a script. The remainder of the first line says this particular function produces one output result, `f`, and takes one input argument, `n`. The name of the function specified on the first line is not actually used, because MATLAB looks for the name of the file with a `.m` suffix that contains the function, but it is common practice to have the two match. The next two lines are comments that provide the text displayed when you ask for `help`.

```
help fibonacci
```

produces

```
FIBONACCI  Fibonacci sequence
f = FIBONACCI(n) generates the first n Fibonacci numbers.
```

The name of the function is in uppercase because historically MATLAB was case insensitive and ran on terminals with only a single font. The use of capital letters may be confusing to some first-time MATLAB users, but the convention persists. It is important to repeat the input and output arguments in these comments because the first line is not displayed when you ask for `help` on the function.

The next line

```
f = zeros(n,1);
```

creates an `n`-by-`1` matrix containing all zeros and assigns it to `f`. In MATLAB, a matrix with only one column is a column vector and a matrix with only one row is a row vector.

The next two lines,

```
f(1) = 1;
f(2) = 2;
```

provide the initial conditions.

The last three lines are the `for` statement that does all the work.

```
for k = 3:n
   f(k) = f(k-1) + f(k-2);
end
```

We like to use three spaces to indent the body of `for` and `if` statements, but other people prefer two or four spaces, or a tab. You can also put the entire construction on one line if you provide a comma after the first clause.

This particular function looks a lot like functions in other programming languages. It produces a vector, but it does not use any of the MATLAB vector or matrix operations. We will see some of these operations soon.

Here is another Fibonacci function, `fibnum.m`. Its output is simply the $n$th Fibonacci number.

```
function f = fibnum(n)
% FIBNUM  Fibonacci number.
% FIBNUM(n) generates the nth Fibonacci number.
if n <= 1
   f = 1;
else
   f = fibnum(n-1) + fibnum(n-2);
end
```

The statement

```
fibnum(12)
```

produces

```
ans =
   233
```

The `fibnum` function is *recursive*. In fact, the term *recursive* is used in both a mathematical and a computer science sense. In mathematics, the relationship $f_n = f_{n-1} + f_{n-2}$ is a *recursion relation* In computer science, a function that calls itself is a *recursive function*.

A recursive program is elegant, but expensive. You can measure execution time with `tic` and `toc`. Try

```
tic, fibnum(24), toc
```

Do *not* try

```
tic, fibnum(50), toc
```

## Fibonacci Meets Golden Ratio

The Golden Ratio $\phi$ can be expressed as an infinite continued fraction.

$$\phi = 1 + \frac{1}{1 + \frac{1}{1+\frac{1}{1+\cdots}}}.$$

To verify this claim, suppose we did not know the value of this fraction. Let

$$x = 1 + \frac{1}{1 + \frac{1}{1+\frac{1}{1+\cdots}}}.$$

We can see the first denominator is just another copy of $x$. In other words.

$$x = 1 + \frac{1}{x}$$

This immediately leads to

$$x^2 - x - 1 = 0$$

which is the defining quadratic equation for $\phi$,

Our `exm` function `goldfract` generates a MATLAB string that represents the first `n` terms of the Golden Ratio continued fraction. Here is the first section of code in `goldfract`.

```
p = '1';
for k = 2:n
   p = ['1 + 1/(' p ')'];
end
display(p)
```

We start with a single '1', which corresponds to `n = 1`. We then repeatedly make the current string the denominator in a longer string.

Here is the output from `goldfract(n)` when `n = 7`.

```
1 + 1/(1 + 1/(1 + 1/(1 + 1/(1 + 1/(1 + 1/(1))))))
```

You can see that there are `n-1` plus signs and `n-1` pairs of matching parentheses.

Let $\phi_n$ denote the continued fraction truncated after $n$ terms. $\phi_n$ is a rational approximation to $\phi$. Let's express $\phi_n$ as a conventional fracton, the ratio of two integers

$$\phi_n = \frac{p_n}{q_n}$$

```
p = 1;
q = 0;
for k = 2:n
   t = p;
   p = p + q;
   q = t;
end
```

Now compare the results produced by `goldfract(7)` and `fibonacci(7)`. The first contains the fraction 21/13 while the second ends with 13 and 21. This is not just a coincidence. The continued fraction for the Golden Ratio is collapsed by repeating the statement

```
p = p + q;
```

while the Fibonacci numbers are generated by

```
f(k) = f(k-1) + f(k-2);
```

In fact, if we let $\phi_n$ denote the golden ratio continued fraction truncated at $n$ terms, then

$$\phi_n = \frac{f_n}{f_{n-1}}$$

In the infinite limit, the ratio of successive Fibonacci numbers approaches the golden ratio:

$$\lim_{n\to\infty} \frac{f_n}{f_{n-1}} = \phi.$$

To see this, compute 40 Fibonacci numbers.

```
n = 40;
f = fibonacci(n);
```

Then compute their ratios.

```
r = f(2:n)./f(1:n-1)
```

This takes the vector containing `f(2)` through `f(n)` and divides it, element by element, by the vector containing `f(1)` through `f(n-1)`. The output begins with

```
2.00000000000000
1.50000000000000
1.66666666666667
1.60000000000000
1.62500000000000
1.61538461538462
1.61904761904762
1.61764705882353
1.61818181818182
```

and ends with

```
1.61803398874990
1.61803398874989
1.61803398874990
1.61803398874989
1.61803398874989
```

Do you see why we chose `n = 40`? Compute

```
phi = (1+sqrt(5))/2
r - phi
```

What is the value of the last element?

The first few of these ratios can also be used to illustrate the rational output format.

```
format rat
r(1:10)
ans =
  2
  3/2
  5/3
  8/5
 13/8
 21/13
 34/21
 55/34
 89/55
```

The population of Fibonacci's rabbit pen doesn't double every month; it is multiplied by the golden ratio every month.

## An Analytic Expression

It is possible to find a closed-form solution to the Fibonacci number recurrence relation. The key is to look for solutions of the form

$$f_n = c\rho^n$$

for some constants $c$ and $\rho$. The recurrence relation

$$f_n = f_{n-1} + f_{n-2}$$

becomes

$$c\rho^n = c\rho^{n-1} + c\rho^{n-2}$$

Dividing both sides by $c\rho^{n-2}$ gives

$$\rho^2 = \rho + 1.$$

We've seen this equation in the chapter on the Golden Ratio. There are two possible values of $\rho$, namely $\phi$ and $1 - \phi$. The general solution to the recurrence is

$$f_n = c_1\phi^n + c_2(1 - \phi)^n.$$

The constants $c_1$ and $c_2$ are determined by initial conditions, which are now conveniently written

$$\begin{aligned} f_0 &= c_1 + c_2 = 1, \\ f_1 &= c_1\phi + c_2(1 - \phi) = 1. \end{aligned}$$

One of the exercises asks you to use the MATLAB backslash operator to solve this 2-by-2 system of simultaneous linear equations, but it is may be easier to solve the system by hand:

$$\begin{aligned} c_1 &= \frac{\phi}{2\phi - 1}, \\ c_2 &= -\frac{(1 - \phi)}{2\phi - 1}. \end{aligned}$$

Inserting these in the general solution gives

$$f_n = \frac{1}{2\phi - 1}(\phi^{n+1} - (1 - \phi)^{n+1}).$$

This is an amazing equation. The right-hand side involves powers and quotients of irrational numbers, but the result is a sequence of integers. You can check this with MATLAB.

```
n = (1:40)';
f = (phi.^(n+1) - (1-phi).^(n+1))/(2*phi-1)
f = round(f)
```

The `.^` operator is an element-by-element power operator. It is not necessary to use `./` for the final division because `(2*phi-1)` is a scalar quantity. Roundoff error prevents the results from being exact integers, so the `round` function is used to convert floating point quantities to nearest integers. The resulting `f` begins with

```
f =
          1
```

```
          2
          3
          5
          8
         13
         21
         34
```

and ends with

```
    5702887
    9227465
   14930352
   24157817
   39088169
   63245986
  102334155
  165580141
```

## Recap

```
%% Fibonacci Chapter Recap
% This is an executable program that illustrates the statements
% introduced in the Fibonacci Chapter of "Experiments in MATLAB".
% You can access it with
%
%    fibonacci_recap
%    edit fibonacci_recap
%    publish fibonacci_recap

%% Related EXM Programs
%
%    fibonacci.m
%    fibnum.m
%    rabbits.m

%% Functions
% Save in file sqrt1px.m
%
%    function y = sqrt1px(x)
%    % SQRT1PX  Sample function.
%    % Usage: y = sqrt1px(x)
%
%       y = sqrt(1+x);

%% Create vector
   n = 8;
```

```
   f = zeros(1,n)
   t = 1:n
   s = [1 2 3 5 8 13 21 34]

%% Subscripts
   f(1) = 1;
   f(2) = 2;
   for k = 3:n
      f(k) = f(k-1) + f(k-2);
   end
   f

%% Recursion
%    function f = fibnum(n)
%    if n <= 1
%       f = 1;
%    else
%       f = fibnum(n-1) + fibnum(n-2);
%    end

%% Tic and Toc
   format short
   tic
      fibnum(24);
   toc

%% Element-by-element array operations
   f = fibonacci(5)'
   fpf = f+f
   ftf = f.*f
   ff = f.^2
   ffdf = ff./f
   cosfpi = cos(f*pi)
   even = (mod(f,2) == 0)
   format rat
   r = f(2:5)./f(1:4)

%% Strings
   hello_world
```

## Exercises

2.1 *Rabbits.* Explain what our `rabbits` simulation demonstrates. What do the different figures and colors on the pushbuttons signify?

2.2 *Waltz.* Which Fibonacci numbers are even? Why?

2.3 *Primes.* Use the MATLAB function `isprime` to discover which of the first 40 Fibonacci numbers are prime. You do not need to use a `for` loop. Instead, check out

```
help isprime
help logical
```

2.4 *Backslash.* Use the MATLAB *backslash* operator to solve the 2-by-2 system of simultaneous linear equations

$$\begin{aligned} c_1 + c_2 &= 1, \\ c_1\phi + c_2(1-\phi) &= 1 \end{aligned}$$

for $c_1$ and $c_2$. You can find out about the backslash operator by taking a peek at the Linear Equations chapter, or with the commands

```
help \
help slash
```

2.5 *Logarithmic plot.* The statement

```
semilogy(fibonacci(18),'-o')
```

makes a logarithmic plot of Fibonacci numbers versus their index. The graph is close to a straight line. What is the slope of this line?

2.6 *Execution time.* How does the execution time of `fibnum(n)` depend on the execution time for `fibnum(n-1)` and `fibnum(n-2)`? Use this relationship to obtain an approximate formula for the execution time of `fibnum(n)` as a function of `n`. Estimate how long it would take your computer to compute `fibnum(50)`. Warning: You probably do not want to actually run `fibnum(50)`.

2.7 *Overflow.* What is the index of the largest Fibonacci number that can be represented *exactly* as a MATLAB double-precision quantity without roundoff error? What is the index of the largest Fibonacci number that can be represented *approximately* as a MATLAB double-precision quantity without overflowing?

2.8 *Slower maturity.* What if rabbits took two months to mature instead of one?

The sequence would be defined by

$$\begin{aligned} g_1 &= 1, \\ g_2 &= 1, \\ g_3 &= 2 \end{aligned}$$

and, for $n > 3$,

$$g_n = g_{n-1} + g_{n-3}$$

(a) Modify `fibonacci.m` and `fibnum.m` to compute this sequence.
(b) How many pairs of rabbits are there after 12 months?
(c) $g_n \approx \gamma^n$. What is $\gamma$?
(d) Estimate how long it would take your computer to compute `fibnum(50)` with this modified `fibnum`.

2.9 *Mortality.* What if rabbits took one month to mature, but then died after six months. The sequence would be defined by

$$\begin{aligned} d_n &= 0, \quad n <= 0 \\ d_1 &= 1, \\ d_2 &= 1 \end{aligned}$$

and, for $n > 2$,

$$d_n = d_{n-1} + d_{n-2} - d_{n-7}$$

(a) Modify `fibonacci.m` and `fibnum.m` to compute this sequence.
(b) How many pairs of rabbits are there after 12 months?
(c) $d_n \approx \delta^n$. What is $\delta$?
(d) Estimate how long it would take your computer to compute `fibnum(50)` with this modified `fibnum`.

2.10 *Hello World.* Programming languages are traditionally introduced by the phrase "hello world". An script in `exm` that illustrates some features in MATLAB is available with

```
hello_world
```

Explain what each of the functions and commands in `hello_world` do.

2.11 *Fibonacci power series.* The Fibonacci numbers, $f_n$, can be used as coefficients in a power series defining a function of $x$.

$$\begin{aligned} F(x) &= \sum_{n=1}^{\infty} f_n x^n \\ &= x + 2x^2 + 3x^3 + 5x^4 + 8x^5 + 13x^6 + ... \end{aligned}$$

Our function `fibfun1` is a first attempt at a program to compute this series. It simply involves adding an accumulating sum to `fibonacci.m`. The header of `fibfun1.m` includes the help entries.

```
function [y,k] = fibfun1(x)
% FIBFUN1  Power series with Fibonacci coefficients.
% y = fibfun1(x) = sum(f(k)*x.^k).
% [y,k] = fibfun1(x) also gives the number of terms required.
```

The first section of code initializes the variables to be used. The value of `n` is the index where the Fibonacci numbers overflow.

```
\excise
\emph{Fibonacci power series}.
The Fibonacci numbers, $f_n$, can be used as coefficients in a
power series defining a function of $x$.
\begin{eqnarray*}
F(x) & = & \sum_{n = 1}^\infty f_n x^n \\
     & = & x + 2 x^2 + 3 x^3 + 5 x^4 + 8 x^5 + 13 x^6 + ...
\end{eqnarray*}
Our function #fibfun1# is a first attempt at a program to
compute this series.  It simply involves adding an accumulating
sum to #fibonacci.m#.
The header of #fibfun1.m# includes the help entries.
\begin{verbatim}
   function [y,k] = fibfun1(x)
   % FIBFUN1  Power series with Fibonacci coefficients.
   % y = fibfun1(x) = sum(f(k)*x.^k).
   % [y,k] = fibfun1(x) also gives the number of terms required.
```

The first section of code initializes the variables to be used. The value of `n` is the index where the Fibonacci numbers overflow.

```
n = 1476;
f = zeros(n,1);
f(1) = 1;
f(2) = 2;
y = f(1)*x + f(2)*x.^2;
t = 0;
```

The main body of `fibfun1` implements the Fibonacci recurrence and includes a test for early termination of the loop.

```
for k = 3:n
   f(k) = f(k-1) + f(k-2);
   y = y + f(k)*x.^k;
   if y == t
      return
   end
```

```
      t = y;
   end
```

There are several objections to `fibfun1`. The coefficient array of size 1476 is not actually necessary. The repeated computation of powers, `x^k`, is inefficient because once some power of `x` has been computed, the next power can be obtained with one multiplication. When the series converges the coefficients `f(k)` increase in size, but the powers `x^k` decrease in size more rapidly. The terms `f(k)*x^k` approach zero, but huge `f(k)` prevent their computation.

A more efficient and accurate approach involves combining the computation of the Fibonacci recurrence and the powers of $x$. Let

$$p_k = f_k x^k$$

Then, since

$$f_{k+1} x^{k+1} = f_k x^k + f_{k-1} x^{k-1}$$

the terms $p_k$ satisfy

$$\begin{aligned} p_{k+1} &= p_k x + p_{k-1} x^2 \\ &= x(p_k + x p_{k-1}) \end{aligned}$$

This is the basis for our function `fibfun2`. The header is essentially the same as `fibfun1`

```
function [yk,k] = fibfun2(x)
% FIBFUN2  Power series with Fibonacci coefficients.
% y = fibfun2(x) = sum(f(k)*x.^k).
% [y,k] = fibfun2(x) also gives the number of terms required.
```

The initialization.

```
   pkm1 = x;
   pk = 2*x.^2;
   ykm1 = x;
   yk = 2*x.^2 + x;
   k = 0;
```

And the core.

```
   while any(abs(yk-ykm1) > 2*eps(yk))
      pkp1 = x.*(pk + x.*pkm1);
      pkm1 = pk;
      pk = pkp1;
      ykm1 = yk;
      yk = yk + pk;
      k = k+1;
   end
```

There is no array of coefficients. Only three of the $p_k$ terms are required for each step. The power function `^` is not necessary. Computation of the powers is incorporated in the recurrence. Consequently, `fibfun2` is both more efficient and more accurate than `fibfun1`.

But there is an even better way to evaluate this particular series. It is possible to find a analytic expression for the infinite sum.

$$\begin{aligned} F(x) &= \sum_{n=1}^{\infty} f_n x^n \\ &= x + 2x^2 + 3x^3 + 5x^4 + 8x^5 + ... \\ &= x + (1+1)x^2 + (2+1)x^3 + (3+2)x^4 + (5+3)x^5 + ... \\ &= x + x^2 + x(x + 2x^2 + 3x^3 + 5x^4 + ...) + x^2(x + 2x^2 + 3x^3 + ...) \\ &= x + x^2 + xF(x) + x^2 F(x) \end{aligned}$$

So

$$(1 - x - x^2)F(x) = x + x^2$$

Finally

$$F(x) = \frac{x + x^2}{1 - x - x^2}$$

It is not even necessary to have a `.m` file. A one-liner does the job.

```
fibfun3 = @(x) (x + x.^2)./(1 - x - x.^2)
```

Compare these three `fibfun`'s.

Chapter 4

# Matrices

MATLAB *began as a matrix calculator.*

The Cartesian coordinate system was developed in the 17th century by the French mathematician and philosopher René Descartes. A pair of numbers corresponds to a point in the plane. We will display the coordinates in a *vector* of length two. In order to work properly with matrix multiplication, we want to think of the vector as a *column* vector, So

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

denotes the point $x$ whose first coordinate is $x_1$ and second coordinate is $x_2$. When it is inconvenient to write a vector in this vertical form, we can anticipate MATLAB notation and use a semicolon to separate the two components,

$$x = (\ x_1;\ x_2\ )$$

For example, the point labeled `x` in figure 4.1 has Cartesian coordinates

$$x = (\ 2;\ 4\ )$$

Arithmetic operations on the vectors are defined in natural ways. Addition is defined by

$$x + y = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \end{pmatrix}$$

Multiplication by a single number, or *scalar*, is defined by

$$sx = \begin{pmatrix} sx_1 \\ sx_2 \end{pmatrix}$$

Copyright © 2011 Cleve Moler
MATLAB® is a registered trademark of MathWorks, Inc.™
October 4, 2011

A 2-by-2 *matrix* is an array of four numbers arranged in two rows and two columns.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix}$$

or

$$A = (\ a_{1,1}\ a_{1,2};\ a_{2,1}\ a_{2,2}\ )$$

For example

$$A = \begin{pmatrix} 4 & -3 \\ -2 & 1 \end{pmatrix}$$

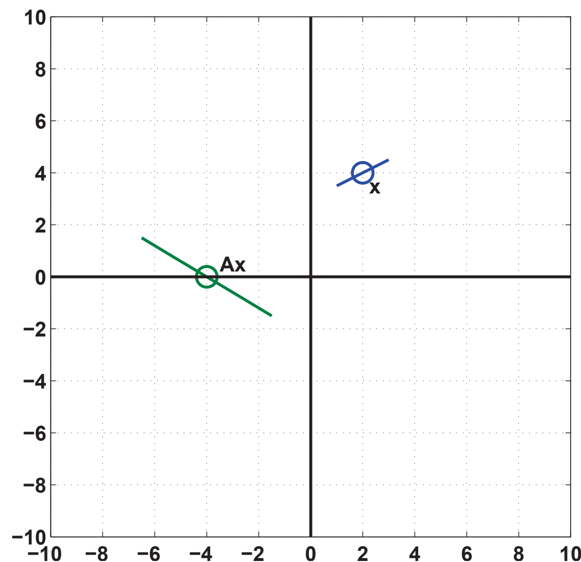



**Figure 4.1.** *Matrix multiplication transforms lines through $x$ to lines through $Ax$.*

Matrix-vector multiplication by a 2-by-2 *matrix* $A$ transforms a vector $x$ to a vector $Ax$, according to the definition

$$Ax = \begin{pmatrix} a_{1,1}x_1 + a_{1,2}x_2 \\ a_{2,1}x_1 + a_{2,2}x_2 \end{pmatrix}$$

For example

$$\begin{pmatrix} 4 & -3 \\ -2 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 4 \end{pmatrix} = \begin{pmatrix} 4 \cdot 2 - 3 \cdot 4 \\ -2 \cdot 2 + 1 \cdot 4 \end{pmatrix} = \begin{pmatrix} -4 \\ 0 \end{pmatrix}$$

The point labeled `x` in figure 4.1 is transformed to the point labeled `Ax`. Matrix-vector multiplications produce *linear* transformations. This means that for scalars $s$ and $t$ and vectors $x$ and $y$,

$$A(sx + ty) = sAx + tAy$$

This implies that points near $x$ are transformed to points near $Ax$ and that straight lines in the plane through $x$ are transformed to straight lines through $Ax$.

Our definition of matrix-vector multiplication is the usual one involving the *dot product* of the *rows* of $A$, denoted $a_{i,:}$, with the vector $x$.

$$Ax = \begin{pmatrix} a_{1,:} \cdot x \\ a_{2,:} \cdot x \end{pmatrix}$$

An alternate, and sometimes more revealing, definition uses *linear combinations* of the *columns* of $A$, denoted by $a_{:,j}$.

$$Ax = x_1 a_{:,1} + x_2 a_{:,2}$$

For example

$$\begin{pmatrix} 4 & -3 \\ -2 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 4 \end{pmatrix} = 2 \begin{pmatrix} 4 \\ -2 \end{pmatrix} + 4 \begin{pmatrix} -3 \\ 1 \end{pmatrix} = \begin{pmatrix} -4 \\ 0 \end{pmatrix}$$

The *transpose* of a column vector is a row vector, denoted by $x^T$. The transpose of a matrix interchanges its rows and columns. For example,

$$x^T = \begin{pmatrix} 2 & 4 \end{pmatrix}$$

$$A^T = \begin{pmatrix} 4 & -2 \\ -3 & 1 \end{pmatrix}$$

Vector-matrix multiplication can be defined by

$$x^T A = A^T x$$

That is pretty cryptic, so if you have never seen it before, you might have to ponder it a bit.

Matrix-matrix multiplication, $AB$, can be thought of as matrix-vector multiplication involving the matrix$A$ and the columns vectors from $B$, or as vector-matrix multiplication involving the row vectors from $A$ and the matrix $B$. It is important to realize that $AB$ is not the same matrix as $BA$.

MATLAB started its life as “Matrix Laboratory”, so its very first capabilities involved matrices and matrix multiplication. The syntax follows the mathematical notation closely. We use square brackets instead of round parentheses, an asterisk to denote multiplication, and `x'` for the transpose of `x`. The foregoing example becomes

```
x = [2; 4]
A = [4 -3; -2 1]
A*x
```

This produces

```
x =
      2
      4
```

```
A =
     4    -3
    -2     1
ans =
    -4
     0
```

The matrices `A'*A` and `A*A'` are not the same.

```
A'*A =
    20   -14
   -14    10
```

while

```
A*A' =
    25   -11
   -11     5
```

The matrix

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

is the 2-by-2 *identity* matrix. It has the important property that for any 2-by-2 matrix $A$,

$$IA = AI = A$$

Originally, MATLAB variable names were not case sensitive, so `i` and `I` were the same variable. Since `i` is frequently used as a subscript, an iteration index, and `sqrt(-1)`, we could not use `I` for the identity matrix. Instead, we chose to use the sound-alike word `eye`. Today, MATLAB is case sensitive and has many users whose native language is not English, but we continue to use `eye(n,n)` to denote the `n`-by-`n` identity. (The Metro in Washington, DC, uses the same pun – "I street" is "eye street" on their maps.)

## 2-by-2 Matrix Transformations

The `exm` toolbox includes a function `house`. The statement

```
X = house
```

produces a 2-by-11 matrix,

```
X =
  -6   -6   -7    0    7    6    6   -3   -3    0    0
  -7    2    1    8    1    2   -7   -7   -2   -2   -7
```

The columns of `X` are the Cartesian coordinates of the 11 points shown in figure 4.2. Do you remember the "dot to dot" game? Try it with these points. Finish off by connecting the last point back to the first. The house in figure 4.2 is constructed from `X` by

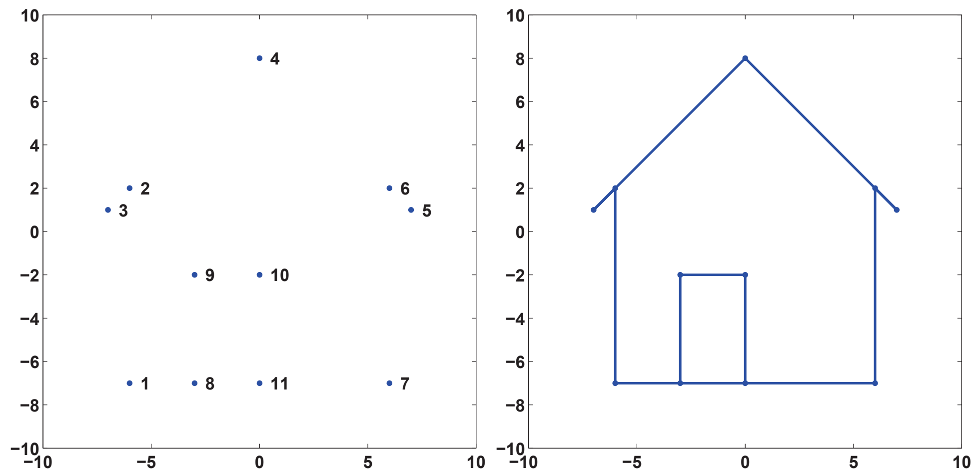



**Figure 4.2.** *Connect the dots.*

```
dot2dot(X)
```

We want to investigate how matrix multiplication transforms this house. In fact, if you have your computer handy, try this now.

```
wiggle(X)
```

Our goal is to see how `wiggle` works.

Here are four matrices.

```
A1 =
   1/2       0
   0         1

A2 =
   1         0
   0         1/2

A3 =
   0         1
   1/2       0

A4 =
   1/2       0
   0        -1
```

Figure 4.3 uses matrix multiplication `A*X` and `dot2dot(A*X)` to show the effect of the resulting linear transformations on the house. All four matrices are diagonal or antidiagonal, so they just scale and possibly interchange the coordinates. The coordinates are not combined in any way. The floor and sides of the house remain at

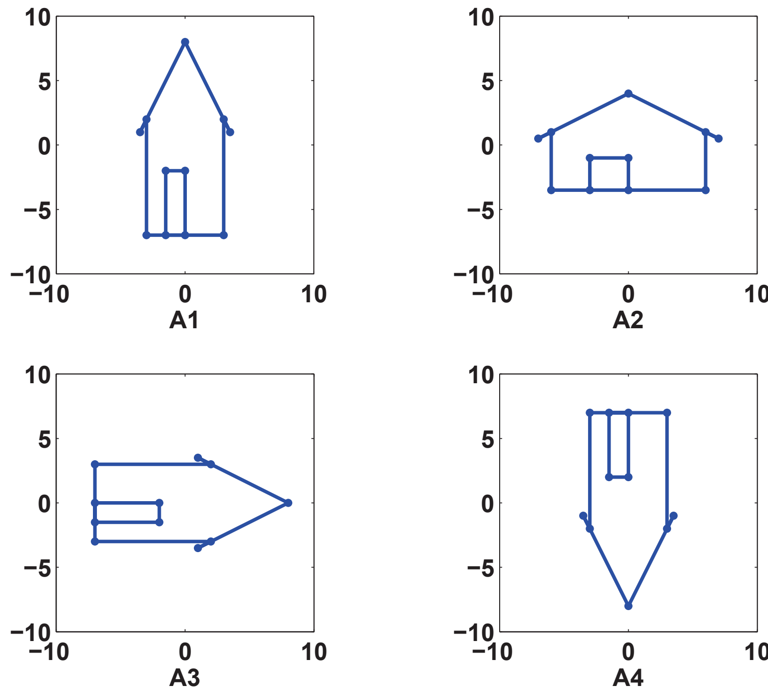



**Figure 4.3.** *The effect of multiplication by scaling matrices.*

right angles to each other and parallel to the axes. The matrix `A1` shrinks the first coordinate to reduce the width of the house while the height remains unchanged. The matrix `A2` shrinks the second coordinate to reduce the height, but not the width. The matrix `A3` interchanges the two coordinates while shrinking one of them. The matrix `A4` shrinks the first coordinate and changes the sign of the second.

The *determinant* of a 2-by-2 matrix

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix}$$

is the quantity

$$a_{1,1}a_{2,2} - a_{1,2}a_{2,1}$$

In general, determinants are not very useful in practical computation because they have atrocious scaling properties. But 2-by-2 determinants can be useful in understanding simple matrix properties. If the determinant of a matrix is positive, then multiplication by that matrix preserves left- or right-handedness. The first two of our four matrices have positive determinants, so the door remains on the left side of the house. The other two matrices have negative determinants, so the door is transformed to the other side of the house.

The MATLAB function `rand(m,n)` generates an $m$-by-$n$ matrix with random entries between 0 and 1. So the statement

```
R = 2*rand(2,2) - 1
```

generates a 2-by-2 matrix with random entries between -1 and 1. Here are four of them.

```
R1 =
    0.0323   -0.6327
   -0.5495   -0.5674

R2 =
    0.7277   -0.5997
    0.8124    0.7188

R3 =
    0.1021    0.1777
   -0.3633   -0.5178

R4 =
   -0.8682    0.9330
    0.7992   -0.4821
```

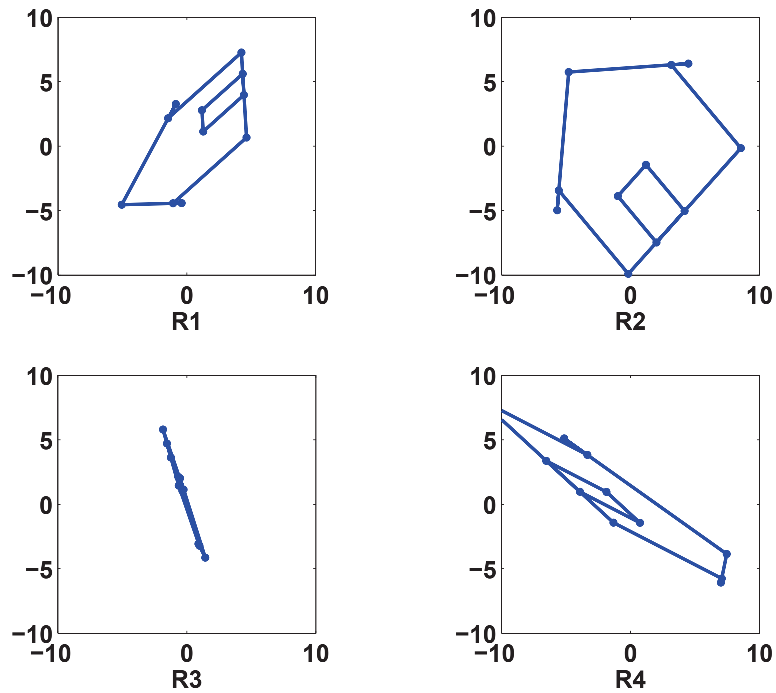



**Figure 4.4.** *The effect of multiplication by random matrices.*

Figure 4.4 shows the effect of multiplication by these four matrices on the house. Matrices R1 and R4 have large off-diagonal entries and negative determinants, so they distort the house quite a bit and flip the door to the right side. The lines are still straight, but the walls are not perpendicular to the floor. Linear transformations preserve straight lines, but they do not necessarily preserve the angles between those lines. Matrix R2 is close to a rotation, which we will discuss shortly. Matrix R3 is nearly *singular*; its determinant is equal to 0.0117. If the determinant were exactly zero, the house would be flattened to a one-dimensional straight line.

The following matrix is a *plane rotation*.

$$G(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

We use the letter $G$ because Wallace Givens pioneered the use of plane rotations in matrix computation in the 1950s. Multiplication by $G(\theta)$ rotates points in the plane through an angle $\theta$. Figure 4.5 shows the effect of multiplication by the plane rotations with $\theta = 15°$, $45°$, $90°$, and $215°$.

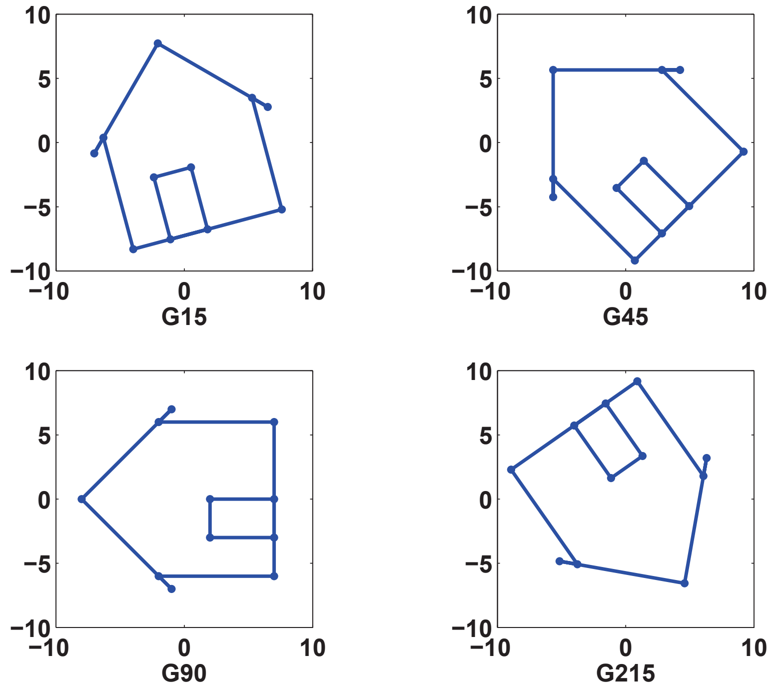



**Figure 4.5.** *The affect of multiplication by plane rotations though* $15°$, $45°$, $90°$, *and* $215°$.

```
G15 =
    0.9659   -0.2588
```

```
    0.2588    0.9659

G45 =
    0.7071   -0.7071
    0.7071    0.7071

G90 =
     0    -1
     1     0

G215 =
   -0.8192    0.5736
   -0.5736   -0.8192
```

You can see that G45 is fairly close to the random matrix R2 seen earlier and that its effect on the house is similar.

MATLAB generates a plane rotation for angles measured in radians with

```
G = [cos(theta) -sin(theta); sin(theta) cos(theta)]
```

and for angles measured in degrees with

```
G = [cosd(theta) -sind(theta); sind(theta) cosd(theta)]
```

Our exm toolbox function wiggle uses dot2dot and plane rotations to produce an animation of matrix multiplication. Here is wiggle.m, without the Handle Graphics commands.

```
function wiggle(X)
thetamax = 0.1;
delta = .025;
t = 0;
while true
   theta = (4*abs(t-round(t))-1) * thetamax;
   G = [cos(theta) -sin(theta); sin(theta) cos(theta)]
   Y = G*X;
   dot2dot(Y);
   t = t + delta;
end
```

Since this version does not have a stop button, it would run forever. The variable t advances steadily by increment of delta. As t increases, the quantity t-round(t) varies between $-1/2$ and $1/2$, so the angle $\theta$ computed by

```
theta = (4*abs(t-round(t))-1) * thetamax;
```

varies in a sawtooth fashion between -thetamax and thetamax. The graph of $\theta$ as a function of $t$ is shown in figure 4.6. Each value of $\theta$ produces a corresponding plane rotation $G(\theta)$. Then

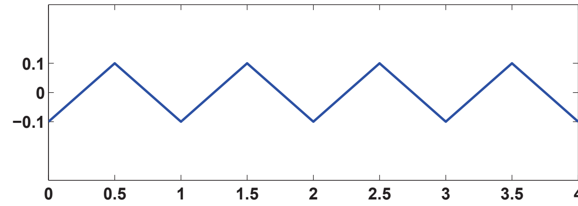



**Figure 4.6.** *Wiggle angle* $\theta$

```
Y = G*X;
dot2dot(Y)
```

applies the rotation to the input matrix `X` and plots the wiggling result.

## Vectors and Matrices

Here is a quick look at a few of the many MATLAB operations involving vectors and matrices. Try to predict what output will be produced by each of the following statements. You can see if you are right by using cut and paste to execute the statement, or by running

```
matrices_recap
```

Vectors are created with square brackets.

```
v = [0 1/4 1/2 3/4 1]
```

Rows of a matrix are separated by semicolons or new lines.

```
A = [8 1 6; 3 5 7; 4 9 2]
```

There are several functions that create matrices.

```
Z = zeros(3,4)
E = ones(4,3)
I = eye(4,4)
M = magic(3)
R = rand(2,4)
[K,J] = ndgrid(1:4)
```

A colon creates uniformly spaced vectors.

```
v = 0:0.25:1
n = 10
y = 1:n
```

A semicolon at the end of a line suppresses output.

```
n = 1000;
y = 1:n;
```

## Matrix arithmetic

Matrix addition and subtraction are denoted by + and - . The operations

```
A + B
```

and

```
A - B
```

require A and B to be the same size, or to be scalars, which are 1-by-1 matrices.

Matrix multiplication, denoted by *, follows the rules of linear algebra. The operation

```
A * B
```

requires the number of columns of A to equal the number of row B, that is

```
size(A,2) == size(B,1)
```

Remember that A*B is usually not equal to B*A

If p is an integer scalar, the expression

```
A^p
```

denotes repeated multiplication of A by itself p times.

The use of the matrix division operations in Matlab,

```
A \ B
```

and

```
A / B
```

is discussed in our "Linear Equations" chapter

## Array arithmetic

.

We usually try to distinguish between *matrices*, which behave according to the rules of linear algebra, and *arrays*, which are just rectangular collections of numbers.

Element-by-element operations array operations are denoted by + , - , .* , ./ , . and .^ . For array multiplication A.*B is equal to B.*A

```
K.*J
v.^2
```

An apostrophe denotes the transpose of a real array and the complex conjugate transpose of a complex array.

```
v = v'
inner_prod = v'*v
outer_prod = v*v'
Z = [1 2; 3+4i 5]'
Z = [1 2; 3+4i 5].'
```

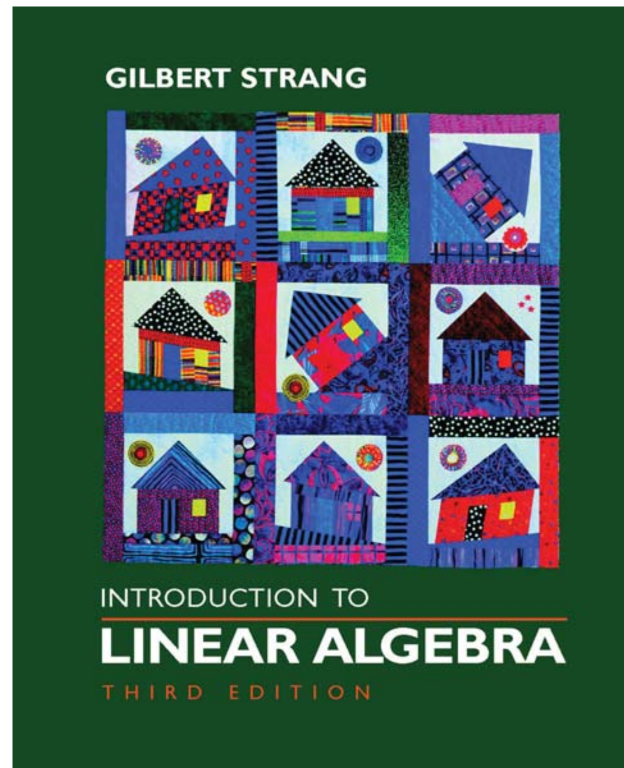



**Figure 4.7.** *The cover of Gilbert Strang's textbook shows a quilt by Chris Curtis.*

## Further Reading

Of the dozens of good books on matrices and linear algebra, we would like to recommend one in particular.

> Gilbert Strang, *Introduction to Linear Algebra*, Wellesley-Cambridge Press, Wellesley, MA, 2003.
> `http://www.wellesleycambridge.com`

Besides its excellent technical content and exposition, it has a terrific cover. The house that we have used throughout this chapter made its debut in Strang's book in 1993. The cover of the first edition looked something like our figure 4.4. Chris Curtis saw that cover and created a gorgeous quilt. A picture of the quilt has appeared on the cover of all subsequent editions of the book.

## Recap

```
%% Matrices Chapter Recap
```

```
% This is an executable program that illustrates the statements
% introduced in the Matrices Chapter of "Experiments in MATLAB".
% You can access it with
%
%    matrices_recap
%    edit matrices_recap
%    publish matrices_recap
%
% Related EXM Programs
%
%    wiggle
%    dot2dot
%    house
%    hand

%% Vectors and matrices
    x = [2; 4]
    A = [4 -3; -2 1]
    A*x
    A'*A
    A*A'

%% Random matrices
    R = 2*rand(2,2)-1

%% Build a house
    X = house
    dot2dot(X)

%% Rotations
    theta = pi/6   % radians
    G = [cos(theta) -sin(theta); sin(theta) cos(theta)]
    theta = 30  % degrees
    G = [cosd(theta) -sind(theta); sind(theta) cosd(theta)]
    subplot(1,2,1)
    dot2dot(G*X)
    subplot(1,2,2)
    dot2dot(G'*X)

%% More on Vectors and Matrices

% Vectors are created with square brackets.

   v = [0 1/4 1/2 3/4 1]

% Rows of a matrix are separated by semicolons or new lines.
```

```
   A = [8 1 6; 3 5 7; 4 9 2]

   A = [8 1 6
        3 5 7
        4 9 2]

%% Creating matrices
   Z = zeros(3,4)
   E = ones(4,3)
   I = eye(4,4)
   M = magic(3)
   R = rand(2,4)
   [K,J] = ndgrid(1:4)

%% Colons and semicolons

% A colon creates uniformally spaced vectors.

   v = 0:0.25:1
   n = 10
   y = 1:n

% A semicolon at the end of a line suppresses output.

   n = 1000;
   y = 1:n;


%% Matrix arithmetic.
% Addition and subtraction, + and -, are element-by-element.
% Multiplication, *, follows the rules of linear algebra.
% Power, ^, is repeated matrix multiplication.

   KJ = K*J
   JK = J*K

%% Array arithmetic
% Element-by-element operations are denoted by
% + , - , .* , ./ , .\ and .^ .

   K.*J
   v.^2

%% Transpose
% An apostrophe denotes the transpose of a real array
```

```
% and the complex conjugate transpose of a complex array.

   v = v'
   inner_prod = v'*v
   outer_prod = v*v'
   Z = [1 2; 3+4i 5]'
   Z = [1 2; 3+4i 5].'
```

## Exercises

4.1 *Multiplication.*
(a) Which 2-by-2 matrices have $A^2 = I$?
(b) Which 2-by-2 matrices have $A^T A = I$?
(c) Which 2-by-2 matrices have $A^T A = AA^T$?

4.2 *Inverse.* Let

$$A = \begin{pmatrix} 3 & 4 \\ 2 & 3 \end{pmatrix}$$

Find a matrix $X$ so that $AX = I$.

4.3 *Powers.* Let

$$A = \begin{pmatrix} 0.99 & 0.01 \\ -0.01 & 1.01 \end{pmatrix}$$

What is $A^n$?

4.4 *Powers.* Let

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

What is $A^n$?

4.5 *Parametrized product.* Let

$$A = \begin{pmatrix} 1 & 2 \\ x & 3 \end{pmatrix} \begin{pmatrix} 4 & 5 \\ 6 & 7 \end{pmatrix}$$

Which elements of $A$ depend upon $x$? Is it possible to choose $x$ so that $A = A^T$?

4.6 *Product of two symmetric matrices.* It turns out that any matrix is the product of two symmetric matrices. Let

$$A = \begin{pmatrix} 3 & 4 \\ 8 & 10 \end{pmatrix}$$

Express $A$ as the product of two symmetric matrices.

4.7 *Givens rotations.*
(a) What is the determinant of $G(\theta)$?
(b) Explain why $G(\theta)^2 = G(2\theta)$.
(c) Explain why $G(\theta)^n = G(n\theta)$.

4.8 $X^8$. Find a real 2-by-2 matrix $X$ so that $X^8 = -I$.

4.9 $G^T$. What is the effect on points in the plane of multiplication by $G(\theta)^T$?

4.10 $\widehat{G}$. (a) What is the effect on points in the plane of multiplication by

$$\widehat{G}(\theta) = \begin{pmatrix} \cos\theta & \sin\theta \\ \sin\theta & -\cos\theta \end{pmatrix}$$

(b) What is the determinant of $\widehat{G}(\theta)$?
(c) What happens if you modify `wiggle.m` to use $\widehat{G}$ instead of $G$?

4.11 *Goldie.* What does the function `goldie` in the `exm` toolbox do?

4.12 *Transform a hand.* Repeat the experiments in this chapter with

```
X = hand
```

instead of

```
X = house
```

Figure 4.8 shows

```
dot2dot(hand)
```

4.13 *Mirror image.* Find a 2-by-2 matrix R so that

```
dot2dot(house)
```

and

```
dot2dot(R*house)
```

as well as

```
dot2dot(hand)
```

and

```
dot2dot(R*hand)
```

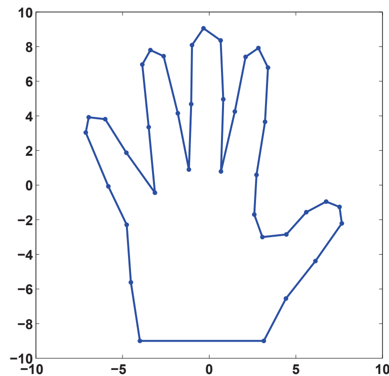



**Figure 4.8.** *A hand.*

are mirror images of each other.

4.14 *Transform your own hand.* Repeat the experiments in this chapter using a plot of your own hand. Start with

```
figure('position',get(0,'screensize'))
axes('position',[0 0 1 1])
axis(10*[-1 1 -1 1])
[x,y] = ginput;
```

Place your hand on the computer screen. Use the mouse to select a few dozen points outlining your hand. Terminate the `ginput` with a carriage return. You might find it easier to trace your hand on a piece of paper and then put the paper on the computer screen. You should be able to see the `ginput` cursor through the paper.

The data you have collected forms two column vectors with entries in the range from -10 to 10. You can arrange the data as two rows in a single matrix with

```
H = [x y]';
```

Then you can use

```
dot2dot(H)
dot2dot(A*H)
wiggle(H)
```

and so on.

You can save your data in the file `myhand.mat` with

```
save myhand H
```

and retrieve it in a later MATLAB session with

```
load myhand
```

4.15 *Wiggler.* Make `wiggler.m`, your own version of `wiggle.m`, with two sliders that control the speed and amplitude. In the initialization, replace the statements

```
thetamax = 0.1;
delta = .025;
```

with

```
thetamax = uicontrol('style','slider','max',1.0, ...
    'units','normalized','position',[.25 .01 .25 .04]);
delta = uicontrol('style','slider','max',.05, ...
    'units','normalized','position',[.60 .01 .25 .04]);
```

The quantities `thetamax` and `delta` are now the *handles* to the two sliders. In the body of the loop, replace `thetamax` by

```
get(thetamax,'value');
```

and replace `delta` by

```
get(delta,'value');
```

Demonstrate your `wiggler` on the `house` and the `hand`.

## Chapter 5

# Linear Equations

*The most important task in technical computing.*

I am thinking of two numbers. Their average is 3. What are the numbers? Please remember the first thing that pops into your head. I will get back to this problem in a few pages.

Solving systems of simultaneous linear equations is the most important task in technical computing. It is not only important in its own right, it is also a fundamental, and often hidden, component of other more complicated computational tasks.

The very simplest linear equations involve only one unknown. Solve

$$7x = 21$$

The answer, of course, is

$$x = \frac{21}{7} = 3$$

Now solve

$$ax = b$$

The answer, of course, is

$$x = \frac{b}{a}$$

But what if $a = 0$? Then we have to look at $b$. If $b \neq 0$ then there is no value of $x$ that satisfies

$$0x = b$$

---

Copyright © 2011 Cleve Moler
MATLAB® is a registered trademark of MathWorks, Inc.™
October 4, 2011

The solution does not exist. On the other hand, if $b = 0$ then any value of $x$ satisfies

$$0x = 0$$

The solution is not unique. Mathematicians have been thinking about existence and uniqueness for centuries. We will see that these concepts are also important in modern technical computing.

Here is a toy story problem.

> Alice buys three apples, a dozen bananas, and one cantaloupe for $2.36. Bob buys a dozen apples and two cantaloupes for $5.26. Carol buys two bananas and three cantaloupes for $2.77. How much do single pieces of each fruit cost?

Let $x_1, x_2$, and $x_3$ denote the unknown price of each fruit. We have three equations in three unknowns.

$$\begin{aligned} 3x_1 + 12x_2 + \ x_3 &= 2.36 \\ 12x_1 \qquad\quad + 2x_3 &= 5.26 \\ 2x_2 + 3x_3 &= 2.77 \end{aligned}$$

Because matrix-vector multiplication has been defined the way it has, these equations can be written

$$\begin{pmatrix} 3 & 12 & 1 \\ 12 & 0 & 2 \\ 0 & 2 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2.36 \\ 5.27 \\ 2.77 \end{pmatrix}$$

Or, simply

$$Ax = b$$

where $A$ is a given 3-by-3 matrix, $b$ is a given 3-by-1 column vector, and $x$ is a 3-by-1 column vector with unknown elements.

We want to solve this equation. If you know anything about matrices, you know that the equation can be solved using $A^{-1}$, the *inverse* of $A$,

$$x = A^{-1}b$$

This is a fine concept theoretically, but not so good computationally We don't really need $A^{-1}$, we just want to find $x$.

If you do not know anything about matrices, you might be tempted to divide both sides of the equation by $A$.

$$x = \frac{b}{A}$$

This is a terrible idea theoretically – you can't divide by matrices – but it is the beginning of a good idea computationally.

To find the solution to a linear system of equations with MATLAB, start by entering the matrix of coefficients.

```
A = [3 12 1; 12 0 2; 0 2 3]
```

Since all the elements of A are integers, the matrix is printed with an integer format.

```
A =
     3    12     1
    12     0     2
     0     2     3
```

Next, enter the right hand side as a column vector.

```
b = [2.36 5.26 2.77]'
```

The elements of b are not integers, so the default format shows four digits after the decimal point.

```
b =
    2.3600
    5.2600
    2.7700
```

MATLAB has an output format intended for financial calculations, like this fruit price calculation. The command

```
format bank
```

changes the output to show only two digits after the decimal point.

```
b =
    2.36
    5.26
    2.77
```

In MATLAB the solution to the linear system of equations

$$Ax = b$$

is found using the *backslash* operator.

```
x = A\b
```

Think of this as "dividing" both sides of the equation by A. The result is

```
x =
    0.29
    0.05
    0.89
```

This gives us the solution to our story problem – apples cost 29 cents each, bananas are a nickel each, and cantaloupes are 89 cents each.

Very rarely, systems of linear equations come in the form

$$xA = b$$

where $b$ and $x$ are row vectors. In this case, the solution is found using the forward slash operator.

```
x = b/A
```

The two operations `A\b` and `b/A` are sometimes called *left* and *right* matrix division. In both cases, the coefficient matrix is in the "denominator". For scalars, left and right division are the same thing. The quantities `7\21` and `21/7` are both equal to `3`.

## Singular matrix

Let's change our story problem a bit. Assume now that Carol buys six apples and one cantaloupe for $2.77. The coefficient matrix and right hand side become

```
A =
     3    12     1
    12     0     2
     6     0     1
```

and

```
b =
    2.36
    5.26
    2.77
```

At first glance, this does not look like much of a change. However,

```
x = A\b
```

produces

```
Warning: Matrix is singular to working precision.
x =
   NaN
  -Inf
   Inf
```

`Inf` and `-Inf` stand for plus and minus infinity and result from division of nonzero numbers by zero. `NaN` stands for "Not-a-Number" and results from doing arithmetic involving infinities.

The source of the difficulty is that the new information about Carol's purchase is inconsistent with the earlier information about Alice's and Bob's purchases. We have said that Carol bought exactly half as much fruit as Bob. But she paid 2.77 when half of Bob's payment would have been only 2.63. The third row of `A` is equal to one-half of the second row, but b(3) is not equal to one-half of b(2). For this particular matrix $A$ and vector $b$, the solution to the linear system of equations $Ax = b$ does not exist.

What if we make Carol's purchase price consistent with Bob's? We leave `A` unchanged and revise `b` with

```
b(3) = 2.63
```

so

```
b =
    2.36
    5.26
    2.63
```

Now we do not have enough information. Our last two equations are scalar multiples of each other.

$$\begin{aligned} 12x_1 + 2x_3 &= 5.26 \\ 6x_1 + \ x_3 &= 2.63 \end{aligned}$$

One possible solution is the solution to the original problem.

```
  x =
      0.29
      0.05
      0.89
A*x =
      2.36
      5.26
      2.63
```

But we can pick an arbitrary value for any component of the solution and then use the first equation and one of the last two equations to compute the other components. The result is a solution to all three equations. For example, here is a solution with its third component equal to zero.

```
y = A(1:2,1:2)\b(1:2);
y(3) = 0

  y =
     0.44
     0.09
     0.00
A*y =
     2.36
     5.26
     2.63
```

There are infinitely many more.

For this particular matrix $A$ and vector $b$, the solution to $Ax = b$ is not unique. The family of possible solutions is generated by the *null vector* of A. This is a nonzero vector $z$ for which

$$Az = 0$$

The general form of the solution is one particular solution, say our vector $x$, plus any arbitrary parameter times the null vector. For any value of $t$ the vector

$$y = x + tz$$

is a solution to

$$Ay = b$$

In MATLAB

```
z = null(A)
A*z
t = rand
y = x + t*z
A*y
```

You can see that `A*z` is zero and `A*y` is equal to `b`.

## "Their average is three."

Let's return to the question that I asked you to consider at the beginning of this chapter. I'm thinking of two numbers. Their average is three. What are the numbers?

What popped into your head? You probably realized that I hadn't given you enough information. But you must have thought of *some* solution. In my experience, the most frequent response is "2 and 4". Let's see what MATLAB responds.

My problem is one linear equation in two unknowns The matrix and right hand side are

```
A = [1/2 1/2]
b = 3
```

We want to solve $Ax = b$. This is now an *underdetermined* system. There are fewer equations than unknowns, so there are infinitely many solutions.

Backslash offers one possible solution

```
x = A\b

x =
     6
     0
```

I bet you didn't think of this solution.

If we try `inv(A)*b` we get an error message because rectangular matrices do not have inverses. But there is something called the *pseudoinverse*. We can try that.

```
x = pinv(A)*b

x =
     3
     3
```

Did that solution occur to you?

These two x's are just two members of the infinite family of solutions. If we wanted MATLAB to find the [2 4]' solution, we would have to pose a problem where the solution is constrained to be a pair of integers, close to each other, but not equal. It is possible to solve such problems, but that would take us too far afield.

## My Rules.

This chapter illustrates two fundamental facts about technical computing.

- The hardest quantities to compute are ones that do not exist.
- The next hardest are ones that are not unique.

## Recap

```
%% Linear Equations Chapter Recap
% This is an executable program that illustrates the statements
% introduced in the Linear Equations Chapter of "Experiments in MATLAB".
% You can access it with
%
%    linear_recap
%    edit linear_recap
%    publish linear_recap

%% Backslash
    format bank
    A = [3 12 1; 12 0 2; 0 2 3]
    b = [2.36 5.26 2.77]'
    x = A\b

%% Forward slash
    x = b'/A'

%% Inconsistent singular system
    A(3,:) = [6 0 1]
    A\b

%% Consistent singular system
    b(3) = 2.63

%% One particular solution
    x = A(1:2,1:2)\b(1:2);
    x(3) = 0
    A*x
```

```
%% Null vector
    z = null(A)
    A*z

%% General solution
    t = rand   % Arbitrary parameter
    y = x + t*z
    A*y
```

## Exercises

5.1 *Two-by-two*. Use backslash to try to solve each of these systems of equations. Indicate if the solution exists, and if it unique.

(a)

$$\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} x = \begin{pmatrix} -1 \\ 2 \end{pmatrix}$$

(b)

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} x = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

(c)

$$\begin{pmatrix} 1 & 2 \\ 3 & 6 \end{pmatrix} x = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

(d)

$$\begin{pmatrix} 1 & 2 \\ 3 & 6 \end{pmatrix} x = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$$

5.2 *Three-by-three*. Use backslash to try to solve each of these systems of equations. Indicate if the solution exists, and if it unique.

(a)

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix} x = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

(b)

$$\begin{pmatrix} 9 & -36 & 30 \\ -36 & 192 & -180 \\ 30 & -180 & 180 \end{pmatrix} x = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

(c)

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 12 & 6 \\ 7 & 8 & 12 \end{pmatrix} x = \begin{pmatrix} 3 \\ 12 \\ 15 \end{pmatrix}$$

(d)

$$\begin{pmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{pmatrix} x = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

(e)

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} x = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

(f)

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} x = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

5.3 *Null vector*. Find a nonzero solution $z$ to

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} z = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

5.4 Matrix equations. Backslash can be used to solve matrix equations of the form

$$AX = B$$

where $B$ has several colums. Do you recognize the solution to the following equation?

$$\begin{pmatrix} 53 & -52 & 23 \\ -22 & 8 & 38 \\ -7 & 68 & -37 \end{pmatrix} X = \begin{pmatrix} 360 & 0 & 0 \\ 0 & 360 & 0 \\ 0 & 0 & 360 \end{pmatrix}$$

5.5 *More Fruit.* Alice buys four apples, two dozen bananas, and two cantaloupes for \$4.14. Bob buys a dozen apples and two cantaloupes for \$5.26. Carol buys a half dozen bananas and three cantaloupes for \$2.97. How much do single pieces of each fruit cost? (You might want to set `format bank`.)

5.6 *Truss.* Figure 5.1 depicts a plane truss having 13 members (the numbered lines) connecting 8 joints (the numbered circles). The indicated loads, in tons, are applied

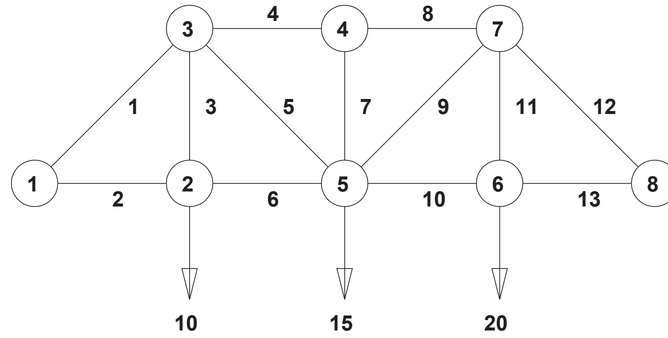



**Figure 5.1.** *A plane truss.*

at joints 2, 5, and 6, and we want to determine the resulting force on each member of the truss.

For the truss to be in static equilibrium, there must be no net force, horizontally or vertically, at any joint. Thus, we can determine the member forces by equating the horizontal forces to the left and right at each joint, and similarly equating the vertical forces upward and downward at each joint. For the eight joints, this would give 16 equations, which is more than the 13 unknown factors to be determined. For the truss to be statically determinate, that is, for there to be a unique solution, we assume that joint 1 is rigidly fixed both horizontally and vertically and that joint 8 is fixed vertically. Resolving the member forces into horizontal and vertical components and defining $\alpha = 1/\sqrt{2}$, we obtain the following system of equations for the member forces $f_i$:

$$\begin{aligned}
\text{Joint 2:} \quad & f_2 = f_6, \\
& f_3 = 10; \\
\text{Joint 3:} \quad & \alpha f_1 = f_4 + \alpha f_5, \\
& \alpha f_1 + f_3 + \alpha f_5 = 0; \\
\text{Joint 4:} \quad & f_4 = f_8, \\
& f_7 = 0; \\
\text{Joint 5:} \quad & \alpha f_5 + f_6 = \alpha f_9 + f_{10}, \\
& \alpha f_5 + f_7 + \alpha f_9 = 15; \\
\text{Joint 6:} \quad & f_{10} = f_{13}, \\
& f_{11} = 20; \\
\text{Joint 7:} \quad & f_8 + \alpha f_9 = \alpha f_{12}, \\
& \alpha f_9 + f_{11} + \alpha f_{12} = 0; \\
\text{Joint 8:} \quad & f_{13} + \alpha f_{12} = 0.
\end{aligned}$$

Solve this system of equations to find the vector $f$ of member forces.

5.7 *Circuit.* Figure 5.2 is the circuit diagram for a small network of resistors.

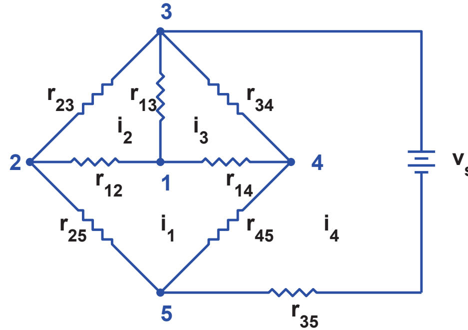



**Figure 5.2.** *A resistor network.*

There are five nodes, eight resistors, and one constant voltage source. We want to compute the voltage drops between the nodes and the currents around each of the loops.

Several different linear systems of equations can be formed to describe this circuit. Let $v_k, k = 1, \ldots, 4$, denote the voltage difference between each of the first four nodes and node number 5 and let $i_k, k = 1, \ldots, 4$, denote the clockwise current around each of the loops in the diagram. *Ohm's law* says that the voltage drop across a resistor is the resistance times the current. For example, the branch between nodes 1 and 2 gives

$$v_1 - v_2 = r_{12}(i_2 - i_1).$$

Using the *conductance*, which is the reciprocal of the resistance, $g_{kj} = 1/r_{kj}$, Ohm's law becomes

$$i_2 - i_1 = g_{12}(v_1 - v_2).$$

The voltage source is included in the equation

$$v_3 - v_s = r_{35} i_4.$$

*Kirchhoff's voltage law* says that the sum of the voltage differences around each loop is zero. For example, around loop 1,

$$(v_1 - v_4) + (v_4 - v_5) + (v_5 - v_2) + (v_2 - v_1) = 0.$$

Combining the voltage law with Ohm's law leads to the *loop* equations for the currents:

$$Ri = b.$$

Here $i$ is the current vector,

$$i = \begin{pmatrix} i_1 \\ i_2 \\ i_3 \\ i_4 \end{pmatrix},$$

$b$ is the source voltage vector,

$$b = \begin{pmatrix} 0 \\ 0 \\ 0 \\ v_s \end{pmatrix},$$

and $R$ is the resistance matrix,

$$\begin{pmatrix} r_{25} + r_{12} + r_{14} + r_{45} & -r_{12} & -r_{14} & -r_{45} \\ -r_{12} & r_{23} + r_{12} + r_{13} & -r_{13} & 0 \\ -r_{14} & -r_{13} & r_{14} + r_{13} + r_{34} & -r_{34} \\ -r_{45} & 0 & -r_{34} & r_{35} + r_{45} + r_{34} \end{pmatrix}.$$

*Kirchhoff's current law* says that the sum of the currents at each node is zero. For example, at node 1,

$$(i_1 - i_2) + (i_2 - i_3) + (i_3 - i_1) = 0.$$

Combining the current law with the conductance version of Ohm's law leads to the *nodal* equations for the voltages:

$$Gv = c.$$

Here $v$ is the voltage vector,

$$v = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix},$$

$c$ is the source current vector,

$$c = \begin{pmatrix} 0 \\ 0 \\ g_{35} v_s \\ 0 \end{pmatrix},$$

and $G$ is the conductance matrix,

$$\begin{pmatrix} g_{12} + g_{13} + g_{14} & -g_{12} & -g_{13} & -g_{14} \\ -g_{12} & g_{12} + g_{23} + g_{25} & -g_{23} & 0 \\ -g_{13} & -g_{23} & g_{13} + g_{23} + g_{34} + g_{35} & -g_{34} \\ -g_{14} & 0 & -g_{34} & g_{14} + g_{34} + g_{45} \end{pmatrix}.$$

You can solve the linear system obtained from the loop equations to compute the currents and then use Ohm's law to recover the voltages. Or you can solve the linear system obtained from the node equations to compute the voltages and then use Ohm's law to recover the currents. Your assignment is to verify that these two approaches produce the same results for this circuit. You can choose your own numerical values for the resistances and the voltage source.

## Chapter 6

# Fractal Fern

*The fractal fern involves 2-by-2 matrices.*

The programs `fern` and `finitefern` in the `exm` toolbox produce the *Fractal Fern* described by Michael Barnsley in *Fractals Everywhere* [?]. They generate and plot a potentially infinite sequence of random, but carefully choreographed, points in the plane. The command

```
fern
```

runs forever, producing an increasingly dense plot. The command

```
finitefern(n)
```

generates `n` points and a plot like Figure 6.1. The command

```
finitefern(n,'s')
```

shows the generation of the points one at a time. The command

```
F = finitefern(n);
```

generates, but does not plot, `n` points and returns an array of zeros and ones for use with sparse matrix and image-processing functions.

The `exm` toolbox also includes `fern.jpg`, a 768-by-1024 color image with half a million points that you can view with a browser or a paint program. You can also view the file with

```
F = imread('fern.png');
image(F)
```

Copyright © 2011 Cleve Moler
MATLAB® is a registered trademark of MathWorks, Inc.™
October 4, 2011

**Figure 6.1.** *Fractal fern.*

If you like the image, you might even choose to make it your computer desktop background. However, you should really run `fern` on your own computer to see the dynamics of the emerging fern in high resolution.

The fern is generated by repeated transformations of a point in the plane. Let $x$ be a vector with two components, $x_1$ and $x_2$, representing the point. There are four different transformations, all of them of the form

$$x \to Ax + b,$$

with different matrices $A$ and vectors $b$. These are known as *affine transformations*. The most frequently used transformation has

$$A = \begin{pmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{pmatrix}, \; b = \begin{pmatrix} 0 \\ 1.6 \end{pmatrix}.$$

This transformation shortens and rotates $x$ a little bit, then adds 1.6 to its second component. Repeated application of this transformation moves the point up and to the right, heading toward the upper tip of the fern. Every once in a while, one of the other three transformations is picked at random. These transformations move

the point into the lower subfern on the right, the lower subfern on the left, or the stem.

Here is the complete fractal fern program.

```
function fern
%FERN  MATLAB implementation of the Fractal Fern
%Michael Barnsley, Fractals Everywhere, Academic Press,1993
%This version runs forever, or until stop is toggled.
%See also: FINITEFERN.

shg
clf reset
set(gcf,'color','white','menubar','none', ...
   'numbertitle','off','name','Fractal Fern')
x = [.5; .5];
h = plot(x(1),x(2),'.');
darkgreen = [0 2/3 0];
set(h,'markersize',1,'color',darkgreen,'erasemode','none');
axis([-3 3 0 10])
axis off
stop = uicontrol('style','toggle','string','stop', ...
   'background','white');
drawnow

p  = [ .85  .92  .99  1.00];
A1 = [ .85  .04; -.04  .85];  b1 = [0; 1.6];
A2 = [ .20 -.26;  .23  .22];  b2 = [0; 1.6];
A3 = [-.15  .28;  .26  .24];  b3 = [0; .44];
A4 = [  0    0 ;   0   .16];

cnt = 1;
tic
while ~get(stop,'value')
   r = rand;
   if r < p(1)
      x = A1*x + b1;
   elseif r < p(2)
      x = A2*x + b2;
   elseif r < p(3)
      x = A3*x + b3;
   else
      x = A4*x;
   end
   set(h,'xdata',x(1),'ydata',x(2));
   cnt = cnt + 1;
   drawnow
```

```
end
t = toc;
s = sprintf('%8.0f points in %6.3f seconds',cnt,t);
text(-1.5,-0.5,s,'fontweight','bold');
set(stop,'style','pushbutton','string','close', ...
   'callback','close(gcf)')
```

Let's examine this program a few statements at a time.

```
shg
```

stands for "show graph window." It brings an existing graphics window forward, or creates a new one if necessary.

```
clf reset
```

resets most of the figure properties to their default values.

```
set(gcf,'color','white','menubar','none', ...
   'numbertitle','off','name','Fractal Fern')
```

changes the background color of the figure window from the default gray to white and provides a customized title for the window.

```
x = [.5; .5];
```

provides the initial coordinates of the point.

```
h = plot(x(1),x(2),'.');
```

plots a single dot in the plane and saves a *handle*, `h`, so that we can later modify the properties of the plot.

```
darkgreen = [0 2/3 0];
```

defines a color where the red and blue components are zero and the green component is two-thirds of its full intensity.

```
set(h,'markersize',1,'color',darkgreen,'erasemode','none');
```

makes the dot referenced by `h` smaller, changes its color, and specifies that the image of the dot on the screen should not be erased when its coordinates are changed. A record of these old points is kept by the computer's graphics hardware (until the figure is reset), but MATLAB itself does not remember them.

```
axis([-3 3 0 10])
axis off
```

specifies that the plot should cover the region

$$-3 \le x_1 \le 3, \quad 0 \le x_2 \le 10,$$

but that the axes should not be drawn.

```
stop = uicontrol('style','toggle','string','stop', ...
   'background','white');
```

creates a toggle user interface control, labeled `'stop'` and colored white, in the default position near the lower left corner of the figure. The handle for the control is saved in the variable `stop`.

```
drawnow
```

causes the initial figure, including the initial point, to actually be plotted on the computer screen.

The statement

```
p  = [ .85  .92  .99  1.00];
```

sets up a vector of probabilities. The statements

```
A1 = [ .85  .04; -.04  .85];  b1 = [0; 1.6];
A2 = [ .20 -.26;  .23  .22];  b2 = [0; 1.6];
A3 = [-.15  .28;  .26  .24];  b3 = [0; .44];
A4 = [  0    0 ;   0   .16];
```

define the four affine transformations. The statement

```
cnt = 1;
```

initializes a counter that keeps track of the number of points plotted. The statement

```
tic
```

initializes a stopwatch timer. The statement

```
while ~get(stop,'value')
```

begins a `while` loop that runs as long as the `'value'` property of the `stop` toggle is equal to 0. Clicking the `stop` toggle changes the value from 0 to 1 and terminates the loop.

```
r = rand;
```

generates a *pseudorandom* value between 0 and 1. The compound `if` statement

```
if r < p(1)
   x = A1*x + b1;
elseif r < p(2)
   x = A2*x + b2;
elseif r < p(3)
   x = A3*x + b3;
else
   x = A4*x;
end
```

picks one of the four affine transformations. Because `p(1)` is `0.85`, the first transformation is chosen 85% of the time. The other three transformations are chosen relatively infrequently.

```
set(h,'xdata',x(1),'ydata',x(2));
```

changes the coordinates of the point `h` to the new $(x_1, x_2)$ and plots this new point. But `get(h,'erasemode')` is `'none'`, so the old point also remains on the screen.

```
cnt = cnt + 1;
```

counts one more point.

```
drawnow
```

tells MATLAB to take the time to redraw the figure, showing the new point along with all the old ones. Without this command, nothing would be plotted until `stop` is toggled.

```
end
```

matches the `while` at the beginning of the loop. Finally,

```
t = toc;
```

reads the timer.

```
s = sprintf('%8.0f points in %6.3f seconds',cnt,t);
text(-1.5,-0.5,s,'fontweight','bold');
```

displays the elapsed time since `tic` was called and the final count of the number of points plotted. Finally,

```
set(stop,'style','pushbutton','string','close', ...
   'callback','close(gcf)')
```

changes the control to a push button that closes the window.

## Recap

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

```
%   fern
%   finitefern

%% fern.jpg
   F = imread('fern.png');
   image(F)

%% A few graphics commands
   shg
   clf reset
   set(gcf,'color','white')
   x = [.5; .5];
   h = plot(x(1),x(2),'.');
   darkgreen = [0 2/3 0];
   set(h,'markersize',1,'color',darkgreen,'erasemode','none');
   set(h,'xdata',x(1),'ydata',x(2));
   axis([-3 3 0 10])
   axis off
   stop = uicontrol('style','toggle','string','stop','background','white');
   drawnow
   cnt = 12345;
   t = 5.432;
   s = sprintf('%8.0f points in %6.3f seconds',cnt,t);
   text(-1.5,-0.5,s,'fontweight','bold');
   set(stop,'style','pushbutton','string','close','callback','close(gcf)')
```

## Exercises

6.1 *Fern color.* Change the fern color scheme to use pink on a black background. Don't forget the stop button.

6.2 *Flip the fern.* Flip the fern by interchanging its $x$- and $y$-coordinates.

6.3 *Erase mode.*
(a) What happens if you resize the figure window while the fern is being generated? Why?

(b) The `exm` program `finitefern` can be used to produce printed output of the fern. Explain why printing is possible with `finitefern.m` but not with `fern.m`.

6.4 *Fern stem.*
(a) What happens to the fern if you change the only nonzero element in the matrix `A4`?
(b) What are the coordinates of the lower end of the fern's stem?

6.5 *Fern tip.* The coordinates of the point at the upper tip end of the fern can be computed by solving a certain 2-by-2 system of simultaneous linear equations. What is that system and what are the coordinates of the tip?

6.6 *Trajectories.* The fern algorithm involves repeated random choices from four different formulas for advancing the point. If the $k$th formula is used repeatedly by itself, without random choices, it defines a deterministic trajectory in the $(x, y)$ plane. Modify `finitefern.m` so that plots of each of these four trajectories are superimposed on the plot of the fern. Start each trajectory at the point $(-1, 5)$. Plot `o`'s connected with straight lines for the steps along each trajectory. Take as many steps as are needed to show each trajectory's limit point. You can superimpose several plots with

```
plot(...)
hold on
plot(...)
plot(...)
hold off
```

6.7 *Sierpinski's triangle.* Modify `fern.m` or `finitefern.m` so that it produces *Sierpinski's triangle.* Start at

$$x = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

At each iterative step, the current point $x$ is replaced with $Ax + b$, where the matrix $A$ is always

$$A = \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix}$$

and the vector $b$ is chosen at random with equal probability from among the three vectors

$$b = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad b = \begin{pmatrix} 1/2 \\ 0 \end{pmatrix}, \text{ and } \quad b = \begin{pmatrix} 1/4 \\ \sqrt{3}/4 \end{pmatrix}.$$

## Chapter 8

# Exponential Function

*The function $e^z$.*

I hope you have a live MATLAB and the `exm` functions handy. Enter the statement

```
expgui
```

Click on the blue line with your mouse. Move it until the green line is on top of the blue line. What is the resulting value of `a`?

The exponential function is denoted mathematically by $e^t$ and in MATLAB by `exp(t)`. This function is the solution to the world's simplest, and perhaps most important, differential equation,

$$\dot{y} = ky$$

This equation is the basis for any mathematical model describing the time evolution of a quantity with a rate of production that is proportional to the quantity itself. Such models include populations, investments, feedback, and radioactivity. We are using $t$ for the independent variable, $y$ for the dependent variable, $k$ for the proportionality constant, and

$$\dot{y} = \frac{dy}{dt}$$

for the rate of growth, or derivative, with respect to $t$. We are looking for a function that is proportional to its own derivative.

Let's start by examining the function

$$y = 2^t$$

Copyright © 2011 Cleve Moler
MATLAB® is a registered trademark of MathWorks, Inc.™
October 4, 2011

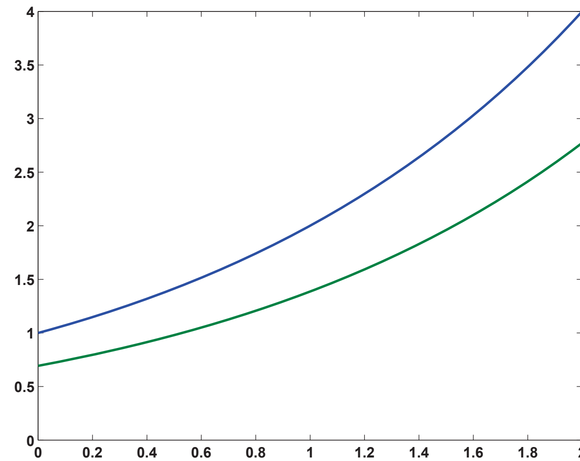



**Figure 8.1.** *The blue curve is the graph of $y = 2^t$. The green curve is the graph of the rate of growth, $\dot{y} = dy/dt$.*

We know what $2^t$ means if $t$ is an integer, $2^t$ is the $t$-th power of 2.

$$2^{-1} = 1/2, \quad 2^0 = 1, \quad 2^1 = 1, \quad 2^2 = 4, \ldots$$

We also know what $2^t$ means if $t = p/q$ is a rational number, the ratio of two integers, $2^{p/q}$ is the $q$-th root of the $p$-th power of 2.

$$\begin{aligned} 2^{1/2} &= \sqrt{2} = 1.4142..., \\ 2^{5/3} &= \sqrt[3]{2^5} = 3.1748..., \\ 2^{355/113} &= \sqrt[113]{2^{355}} = 8.8250... \end{aligned}$$

In principal, for floating point arithmetic, this is all we need to know. All floating point numbers are ratios of two integers. We do not have to be concerned yet about the definition of $2^t$ for irrational $t$. If Matlab can compute powers and roots, we can plot the graph of $2^t$, the blue curve in figure 8.1

What is the derivative of $2^t$? Maybe you have never considered this question, or don't remember the answer. (Be careful, it is *not* $t2^{t-1}$.) We can plot the graph of the approximate derivative, using a step size of something like 0.0001. The following code produces figure 8.1, the graphs of both $y = 2^t$ and its approximate derivative, $\dot{y}$.

```
t = 0:.01:2;
h = .00001;
y = 2.^t;
ydot = (2.^(t+h) - 2.^t)/h;
plot(t,[y; ydot])
```

The graph of the derivative has the same shape as the graph of the original function. Let's look at their ratio, $\dot{y}(t)/y(t)$.

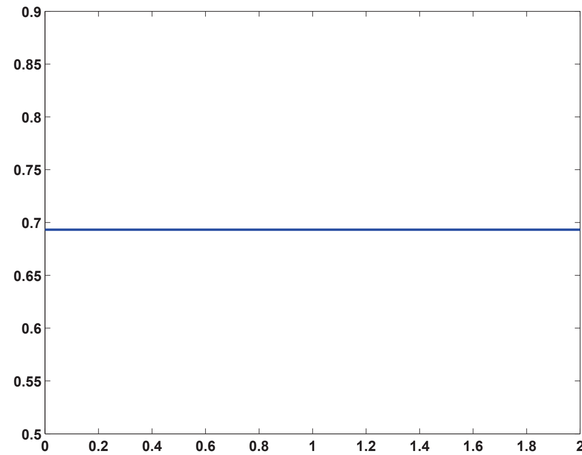



**Figure 8.2.** *The ratio,* $\dot{y}/y$.

```
plot(t,ydot./y)
axis([0 2 .5 .9])
```

We see that the ratio of the derivative to the function, shown in figure 8.2, has a constant value, $\dot{y}/y = 0.6931...$, that does not depend upon t .

Now, if you are following along with a live Matlab, repeat the preceding calculations with $y = 3^t$ instead of $y = 2^t$. You should find that the ratio is again independent of $t$. This time $\dot{y}/y = 1.0986....$ Better yet, experiment with `expgui`.

If we take any value $a$ and look at $y = a^t$, we find that, numerically at least, the ratio $\dot{y}/y$ is constant. In other words, $\dot{y}$ is proportional to $y$. If $a = 2$, the proportionality constant is less than one. If $a = 3$, the proportionality constant is greater than one. Can we find an $a$ so that $\dot{y}/y$ is actually equal to one? If so, we have found a function that is equal to its own derivative.

The approximate derivative of the function $y(t) = a^t$ is

$$\dot{y}(t) = \frac{a^{t+h} - a^t}{h}$$

This can be factored and written

$$\dot{y}(t) = \frac{a^h - 1}{h} a^t$$

So the ratio of the derivative to the function is

$$\frac{\dot{y}(t)}{y(t)} = \frac{a^h - 1}{h}$$

The ratio depends upon $h$, but not upon $t$. If we want the ratio to be equal to 1, we need to find $a$ so that

$$\frac{a^h - 1}{h} = 1$$

Solving this equation for $a$, we find

$$a = (1 + h)^{1/h}$$

The approximate derivative becomes more accurate as $h$ goes to zero, so we are interested in the value of

$$(1 + h)^{1/h}$$

as $h$ approaches zero. This involves taking numbers very close to 1 and raising them to very large powers. The surprising fact is that this limiting process defines a number that turns out to be one of the most important quantities in mathematics

$$e = \lim_{h \to 0} (1 + h)^{1/h}$$

Here is the beginning and end of a table of values generated by repeatedly cutting $h$ in half.

```
format long
format compact
h = 1;
while h > 2*eps
    h = h/2;
    e = (1 + h)^(1/h);
    disp([h e])
end

0.500000000000000   2.250000000000000
0.250000000000000   2.441406250000000
0.125000000000000   2.565784513950348
0.062500000000000   2.637928497366600
0.031250000000000   2.676990129378183
0.015625000000000   2.697344952565099
      ...                 ...
0.000000000000014   2.718281828459026
0.000000000000007   2.718281828459036
0.000000000000004   2.718281828459040
0.000000000000002   2.718281828459043
0.000000000000001   2.718281828459044
0.000000000000000   2.718281828459045
```

The last line of output involves a value of `h` that is not zero, but is so small that it prints as a string of zeros. We are actually computing

$$(1 + 2^{-51})^{2^{51}}$$

which is

$$(1 + \frac{1}{2251799813685248})^{2251799813685248}$$

The result gives us the numerical value of $e$ correct to 16 significant decimal digits. It's easy to remember the repeating pattern of the first 10 significant digits.

$$e = 2.718281828...$$

Let's derive a more useful representation of the exponential function. Start by putting $t$ back in the picture.

$$\begin{aligned} e^t &= (\lim_{h \to 0} (1+h)^{1/h})^t \\ &= \lim_{h \to 0} (1+h)^{t/h} \end{aligned}$$

Here is the *Binomial Theorem.*

$$(a+b)^n = a^n + na^{n-1}b + \frac{n(n-1)}{2!}a^{n-2}b^2 + \frac{n(n-1)(n-2)}{3!}a^{n-3}b^3 + ...$$

If $n$ is an integer, this terminates after $n+1$ terms with $b^n$. But if $n$ is not an integer, the expansion is an infinite series. Apply the binonimial theorem with $a = 1$, $b = h$ and $n = t/h$.

$$\begin{aligned} (1+h)^{t/h} &= 1 + (t/h)h + \frac{(t/h)(t/h-1)}{2!}h^2 + \frac{(t/h)(t/h-1)(t/h-2)}{3!}h^3 + ... \\ &= 1 + t + \frac{t(t-h)}{2!} + \frac{t(t-h)(t-2h)}{3!} + ... \end{aligned}$$

Now let $h$ go to zero. We get the power series for the exponential function.

$$e^t = 1 + t + \frac{t^2}{2!} + \frac{t^3}{3!} + ... + \frac{t^n}{n!} + ...$$

This series is a rigorous mathematical definition that applies to any $t$, positive or negative, rational or irrational, real or complex. The $n+1$-st term is $t^n/n!$. As $n$ increases, the $t^n$ in the numerator is eventually overwhelmed by the $n!$ in the denominator, so the terms go to zero fast enough that the infinite series converges.

It is almost possible to use this power series for actual computation of $e^t$. Here is an experimental MATLAB program.

```
function s = expex(t)
% EXPEX  Experimental version of EXP(T)
s = 1;
term = 1;
n = 0;
r = 0;
while r ~= s
   r = s;
   n = n + 1;
   term = (t/n)*term;
   s = s + term;
end
```

Notice that there are no powers or factorials. Each term is obtained from the previous one using the fact that

$$\frac{t^n}{n!} = \frac{t}{n}\frac{t^{n-1}}{(n-1)!}$$

The potentially infinite loop is terminated when `r == s`, that is when the floating point values of two successive partial sums are equal.

There are "only" two things wrong with this program – its speed and its accuracy. The terms in the series increase as long as $|t/n| \geq 1$, then decrease after $n$ reaches the point where $|t/n| < 1$. So if $|t|$ is not too large, say $|t| < 2$, everything is OK; only a few terms are required and the sum is computed accurately. But larger values of $t$ require more terms and the program requires more time. This is not a very serious defect if $t$ is real and positive. The series converges so rapidly that the extra time is hardly noticeable.

However, if $t$ is real and negative the computed result may be inaccurate. The terms alternate in sign and cancel each other in the sum to produce a small value for $e^t$. Take, for example, $t = -20$. The true value of $e^{-20}$ is roughly $2 \cdot 10^{-9}$. Unfortunately, the largest terms in the series are $(-20)^{19}/19!$ and $(-20)^{20}/20!$, which are opposite in sign and both of size $4 \cdot 10^7$. There is 16 orders of magnitude difference between the size of the largest terms and the size of the final sum. With only 16 digits of accuracy, we lose everything. The computed value obtained from `expex(-20)` is completely wrong.

For real, negative $t$ it is possible to get an accurate result from the power series by using the fact that

$$e^t = \frac{1}{e^{-t}}$$

For complex $t$, there is no such easy fix for the accuracy difficulties of the power series.

In contrast to its more famous cousin, $\pi$, the actual numerical value of $e$ is not very important. It's the exponential function

$$e^t$$

that's important. In fact, Matlab doesn't have the value of $e$ built in. Nevertheless, we can use

```
e = expex(1)
```

to compute an approximate value for $e$. Only seventeen terms are required to get floating point accuracy.

```
e = 2.718281828459045
```

After computing `e`, you could then use `e^t`, but `exp(t)` is preferable.

## Logarithms

The *logarithm* is the *inverse* function of the exponential. If

$$y = e^t$$

then

$$\log_e(y) = t$$

The function $\log_e(y)$ is known as the *natural* logarithm and is often denoted by $\ln y$. More generally, if

$$y = a^t$$

then

$$\log_a(y) = t$$

The function $\log_{10}(y)$ is known as the *common* logarithm. MATLAB uses `log(y)`, `log10(y)`, and `log2(y)` for $\log_e(y)$, $\log_{10}(y)$, and $\log_2(y)$.

## Exponential Growth

The term *exponential growth* is often used informally to describe any kind of rapid growth. Mathematically, the term refers to any time evolution, $y(t)$, where the rate of growth is proportional to the quantity itself.

$$\dot{y} = ky$$

The solution to this equation is determined for all $t$ by specifying the value of $y$ at one particular $t$, usually $t = 0$.

$$y(0) = y_0$$

Then

$$y(t) = y_0 e^{kt}$$

Suppose, at time $t = 0$, we have a million *E. coli* bacteria in a test tube under ideal laboratory conditions. Twenty minutes later each bacterium has fissioned to produce another one. So at $t = 20$, the population is two million. Every 20 minutes the population doubles. At $t = 40$, it's four million. At $t = 60$, it's eight million. And so on. The population, measured in millions of cells, $y(t)$, is

$$y(t) = 2^{t/20}$$

Let $k = \ln 2/20 = .0347$. Then, with $t$ measured in minutes and the population $y(t)$ measured in millions, we have

$$\dot{y} = ky, \quad y(0) = 1$$

Consequently

$$y(t) = e^{kt}$$

This is exponential growth, but it cannot go on forever. Eventually, the growth rate is affected by the size of the container. Initially at least, the size of the population is modelled by the exponential function.

Suppose, at time $t = 0$, you invest \$1000 in a savings account that pays 5% interest, compounded yearly. A year later, at $t = 1$, the bank adds 5% of \$1000 to your account, giving you $y(1) = 1050$. Another year later you get 5% of 1050, which is 52.50, giving $y(2) = 1102.50$. If $y(0) = 1000$ is your initial investment, $r = 0.05$ is the yearly interest rate, $t$ is measured in years, and $h$ is the step size for the compound interest calculation, we have

$$y(t + h) = y(t) + rhy(t)$$

What if the interest is compounded monthly instead of yearly? At the end of the each month, you get .05/12 times your current balance added to your account. The same equation applies, but now with $h = 1/12$ instead of $h = 1$. Rewrite the equation as

$$\frac{y(t + h) - y(t)}{h} = ry(t)$$

and let $h$ tend to zero. We get

$$\dot{y}(t) = ry(t)$$

This defines interest compounded *continuously*. The evolution of your investment is described by

$$y(t) = y(0)e^{rt}$$

Here is a MATLAB program that tabulates the growth of \$1000 invested at 5% over a 20 year period , with interest compounded yearly, monthly, and continuously.

```
format bank
r = 0.05;
y0 = 1000;
for t = 0:20
   y1 = (1+r)^t*y0;
   y2 = (1+r/12)^(12*t)*y0;
   y3 = exp(r*t)*y0;
   disp([t y1 y2 y3])
end
```

The first few and last few lines of output are

```
 t       yearly       monthly     continuous
 0      1000.00      1000.00       1000.00
 1      1050.00      1051.16       1051.27
 2      1102.50      1104.94       1105.17
 3      1157.63      1161.47       1161.83
 4      1215.51      1220.90       1221.40
 5      1276.28      1283.36       1284.03
..      .......      .......       .......
16      2182.87      2221.85       2225.54
```

```
  17      2292.02     2335.52     2339.65
  18      2406.62     2455.01     2459.60
  19      2526.95     2580.61     2585.71
  20      2653.30     2712.64     2718.28
```

Compound interest actually qualifies as exponential growth, although with modest interest rates, most people would not use that term.

Let's borrow money to buy a car. We'll take out a $20,000 car loan at 10% per year interest, make monthly payments, and plan to pay off the loan in 3 years. What is our monthly payment, $p$? Each monthly transaction adds interest to our current balance and subtracts the monthly payment.

$$\begin{aligned} y(t+h) &= y(t) + rhy(t) - p \\ &= (1+rh)y(t) - p \end{aligned}$$

Apply this repeatedly for two, three, then $n$ months.

$$\begin{aligned} y(t+2h) &= (1+rh)y(t+h) - p \\ &= (1+rh)^2 y(t) - ((1+rh)+1)p \\ y(t+3h) &= (1+rh)^3 y(t) - ((1+rh)^2 + (1+rh) + 1)p \\ y(t+nh) &= (1+rh)^n y(0) - ((1+rh)^{n-1} + ... + (1+rh) + 1)p \\ &= (1+rh)^n y(0) - ((1+rh)^n - 1)/(1+rh-1)p \end{aligned}$$

Solve for $p$

$$p = (1+rh)^n/((1+rh)^n - 1)rhy_0$$

Use Matlab to evaluate this for our car loan.

```
y0 = 20000
r = .10
h = 1/12
n = 36
p = (1+r*h)^n/((1+r*h)^n-1)*r*h*y0
```

We find the monthly payment would be

```
p = 645.34
```

If we didn't have to pay interest on the loan and just made 36 monthly payments, they would be

```
y0/n
  = 555.56
```

It's hard to think about continuous compounding for a loan because we would have to figure out how to make infinitely many infinitely small payments.

## Complex exponential

What do we mean by $e^z$ if $z$ is complex? The behavior is very different from $e^t$ for real $t$, but equally interesting and important.

Let's start with a purely imaginary $z$ and set $z = i\theta$ where $\theta$ is real. We then make the *definition*

$$e^{i\theta} = \cos\theta + i\sin\theta$$

This formula is remarkable. It defines the exponential function for an imaginary argument in terms of trig functions of a real argument. There are several reasons why this is a reasonable defintion. First of all, it behaves like an exponential should. We expect

$$e^{i\theta+i\psi} = e^{i\theta}e^{i\psi}$$

This behavior is a consequence of the double angle formulas for trig functions.

$$\begin{aligned}\cos(\theta+\psi) &= \cos\theta\cos\psi - \sin\theta\sin\psi \\ \sin(\theta+\psi) &= \cos\theta\sin\psi + \sin\theta\cos\psi\end{aligned}$$

Secondly, derivatives should be have as expected.

$$\begin{aligned}\frac{d}{d\theta}e^{i\theta} &= ie^{i\theta} \\ \frac{d^2}{d\theta^2}e^{i\theta} &= i^2e^{i\theta} = -e^{i\theta}\end{aligned}$$

In words, the second derivative should be the negative of the function itself. This works because the same is true of the trig functions. In fact, this could be the basis for the defintion because the initial conditions are correct.

$$e^0 = 1 = \cos 0 + i\sin 0$$

The power series is another consideration. Replace $t$ by $i\theta$ in the power series for $e^t$. Rearranging terms gives the power series for $\cos\theta$ and $\sin\theta$.

For MATLAB especially, there is an important connection between multiplication by a complex exponential and the rotation matrices that we considered in the chapter on matrices. Let $w = x + iy$ be any other complex number. What is $e^{i\theta}w$ ? Let $u$ and $v$ be the result of the 2-by-2 matrix multiplication

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Then

$$e^{i\theta}w = u + iv$$

This says that multiplication of a complex number by $e^{i\theta}$ corresponds to a rotation of that number in the complex plane by an angle $\theta$.

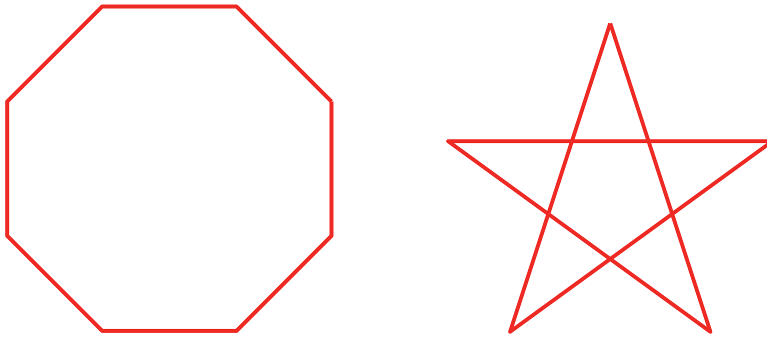


**Figure 8.3.** *Two plots of* $e^{i\theta}$.

When the MATLAB `plot` function sees a complex vector as its first argument, it understands the components to be points in the complex plane. So the octagon in the left half of figure 8.3 can be defined and plotted using $e^{i\theta}$ with

```
theta = (1:2:17)'*pi/8
z = exp(i*theta)
p = plot(z);
```

The quantity `p` is the handle to the plot. This allows us to complete the graphic with

```
set(p,'linewidth',4,'color','red')
axis square
axis off
```

An exercise asks you to modify this code to produce the five-pointed star in the right half of the figure.

Once we have defined $e^{i\theta}$ for real $\theta$, it is clear how to define $e^z$ for a general complex $z = x + iy$,

$$\begin{aligned} e^z &= e^{x+iy} \\ &= e^x e^{iy} \\ &= e^x(\cos y + i \sin y) \end{aligned}$$

Finally, setting $z = i\pi$, we get a famous relationship involving three of the most important quantities in mathematics, $e$, $i$, and $\pi$

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

Let's check that MATLAB and the Symbolic Toolbox get this right.

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

```
>> exp(i*sym(pi))
ans =
  -1
```

## Recap

```
%% Exponential Chapter Recap
% This is an executable program that illustrates the statements
% introduced in the Exponential Chapter of "Experiments in MATLAB".
% You can access it with
%
%    exponential_recap
%    edit exponential_recap
%    publish exponential_recap
%
%  Related EXM programs
%
%    expgui
%    wiggle

%% Plot a^t and its approximate derivative
    a = 2;
    t = 0:.01:2;
    h = .00001;
    y = 2.^t;
    ydot = (2.^(t+h) - 2.^t)/h;
    plot(t,[y; ydot])

%% Compute e
    format long
    format compact
    h = 1;
    while h > 2*eps
        h = h/2;
        e = (1 + h)^(1/h);
        disp([h e])
    end

%% Experimental version of exp(t)
    t = rand
    s = 1;
    term = 1;
    n = 0;
    r = 0;
    while r ~= s
```

```
      r = s;
      n = n + 1;
      term = (t/n)*term;
      s = s + term;
   end
   exp_of_t = s

%% Value of e
   e = expex(1)

%% Compound interest
   fprintf('          t       yearly      monthly    continuous\n')
   format bank
   r = 0.05;
   y0 = 1000;
   for t = 0:20
      y1 = (1+r)^t*y0;
      y2 = (1+r/12)^(12*t)*y0;
      y3 = exp(r*t)*y0;
      disp([t y1 y2 y3])
   end

%% Payments for a car loan
   y0 = 20000
   r = .10
   h = 1/12
   n = 36
   p = (1+r*h)^n/((1+r*h)^n-1)*r*h*y0

%% Complex exponential
   theta = (1:2:17)'*pi/8
   z = exp(i*theta)
   p = plot(z);
   set(p,'linewidth',4,'color','red')
   axis square off

%% Famous relation between e, i and pi
   exp(i*pi)

%% Use the Symbolic Toolbox
   exp(i*sym(pi))
```

## Exercises

8.1 *e cubed*. The value of $e^3$ is close to 20. How close? What is the percentage error?

8.2 *expgui*.
(a) With `expgui`, the graph of $y = a^t$, the blue line, always intercepts the $y$-axis at $y = 1$. Where does the graph of $dy/dx$, the green line, intercept the $y$-axis?
(b) What happens if you replace `plot` by `semilogy` in `expgui`?

8.3 *Computing e*.
(a) If we try to compute $(1+h)^{1/h}$ for small values of $h$ that are inverse powers of 10, it doesn't work very well. Since inverse powers of 10 cannot be represented exactly as binary floating point numbers, the portion of $h$ that effectively gets added to 1 is different than the value involved in the computation of $1/h$. That's why we used inverse powers of 2 in the computation shown in the text. Try this:

```
format long
format compact
h = 1;
while h > 1.e-15
    h = h/10;
    e = (1 + h)^(1/h);
    disp([h e])
end
```

How close do you get to computing the correct value of $e$?
(b) Now try this instead:

```
format long
format compact
h = 1;
while h > 1.e-15
    h = h/10;
    e = (1 + h)^(1/(1+h-1));
    disp([h e])
end
```

How well does this work? Why?

8.4 *expex*. Modify `expex` by inserting

```
disp([term s])
```

as the last statement inside the `while` loop. Change the output you see at the command line.

```
format compact
format long
```

Explain what you see when you try `expex(t)` for various real values of `t`.

```
expex(.001)
expex(-.001)
expex(.1)
expex(-.1)
expex(1)
expex(-1)
```

Try some imaginary values of `t`.

```
expex(.1i)
expex(i)
expex(i*pi/3)
expex(i*pi)
expex(2*i*pi)
```

Increase the width of the output window, change the output format and try larger values of `t`.

```
format long e
expex(10)
expex(-10)
expex(10*pi*i)
```

8.5 *Instrument expex.* Investigate both the cost and the accuracy of `expex`. Modify `expex` so that it returns both the sum `s` and the number of terms required `n`. Assess the relative error by comparing the result from `expex(t)` with the result from the built-in function `exp(t)`.

```
relerr = abs((exp(t) - expex(t))/exp(t))
```

Make a table showing that the number of terms required increases and the relative error deteriorates for large $t$, particularly negative $t$.

8.6 *Complex wiggle.* Revise `wiggle` and `dot2dot` to create `wigglez` and `dot2dotz` that use multiplication by $e^{i\theta}$ instead of multiplication by two-by-two matrices. The crux of `wiggle` is

```
G = [cos(theta) sin(theta); -sin(theta) cos(theta)];
Y = G*X;
dot2dot(Y);
```

In `wigglez` this will become

```
w = exp(i*theta)*z;
dot2dotz(w)
```

You can use `wigglez` with a scaled octogon.

```
theta = (1:2:17)'*pi/8
z = exp(i*theta)
wigglez(8*z)
```

Or, with our `house` expressed as a complex vector.

```
H = house;
z = H(1,:) + i*H(2,:);
wigglez(z)
```

8.7 *Make the star.* Recreate the five-pointed star in the right half of figure 8.3. The points of the star can be traversed in the desired order with

```
theta = (0:3:15)'*(2*pi/5) + pi/2
```

## Chapter 13

# Mandelbrot Set

*Fractals, topology, complex arithmetic and fascinating computer graphics.*

Benoit Mandelbrot was a Polish/French/American mathematician who has spent most of his career at the IBM Watson Research Center in Yorktown Heights, N.Y. He coined the term *fractal* and published a very influential book, *The Fractal Geometry of Nature*, in 1982. An image of the now famous Mandelbrot set appeared on the cover of *Scientific American* in 1985. This was about the time that computer graphical displays were becoming widely available. Since then, the Mandelbrot set has stimulated deep research topics in mathematics and has also been the basis for an uncountable number of graphics projects, hardware demos, and Web pages.

To get in the mood for the Mandelbrot set, consider the region in the complex plane of trajectories generated by repeated squaring,

$$z_{k+1} = z_k^2, \quad k = 0, 1, ...$$

For which initial values $z_0$ does this sequence remain bounded as $k \to \infty$? It is easy to see that this set is simply the unit disc, $|z_0| <= 1$, shown in figure 13.1. If $|z_0| <= 1$, the sequence $z_k$ remains bounded. But if $|z_0| > 1$, the sequence is unbounded. The boundary of the unit disc is the unit circle, $|z_0| = 1$. There is nothing very difficult or exciting here.

The definition is the Mandelbrot set is only slightly more complicated. It involves repeatedly adding in the initial point. The Mandelbrot set is the region in the complex plane consisting of the values $z_0$ for which the trajectories defined by

$$z_{k+1} = z_k^2 + z_0, \quad k = 0, 1, ...$$

remain bounded at $k \to \infty$. That's it. That's the entire definition. It's amazing that such a simple definition can produce such fascinating complexity.

Copyright © 2011 Cleve Moler
MATLAB® is a registered trademark of MathWorks, Inc.™
October 4, 2011

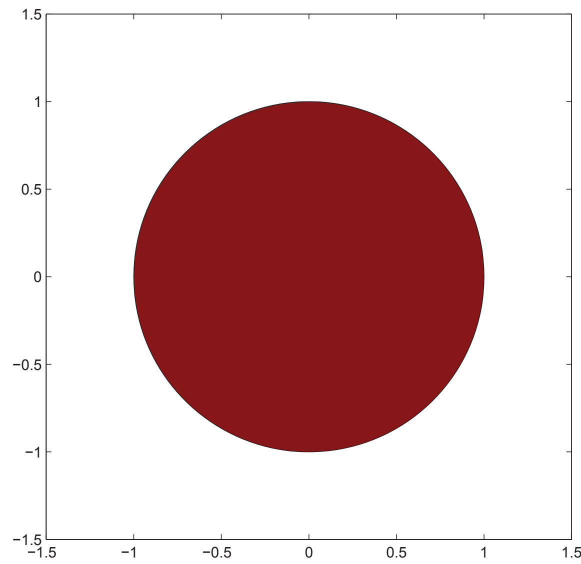



**Figure 13.1.** *The unit disc is shown in red. The boundary is simply the unit circle. There is no intricate fringe.*

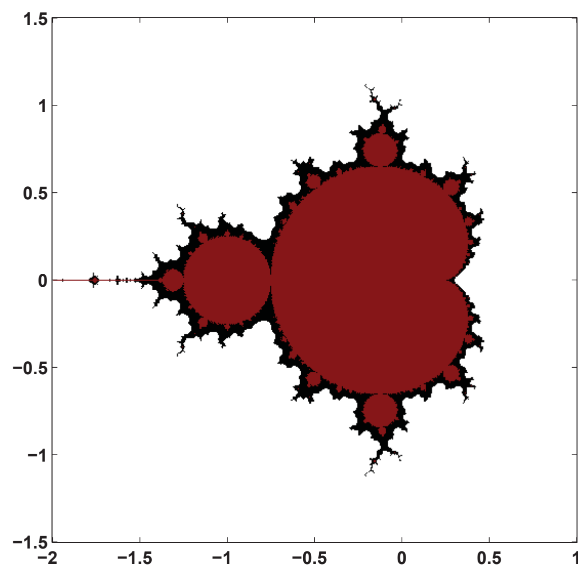



**Figure 13.2.** *The Mandelbrot set is shown in red. The fringe just outside the set, shown in black, is a region of rich structure.*

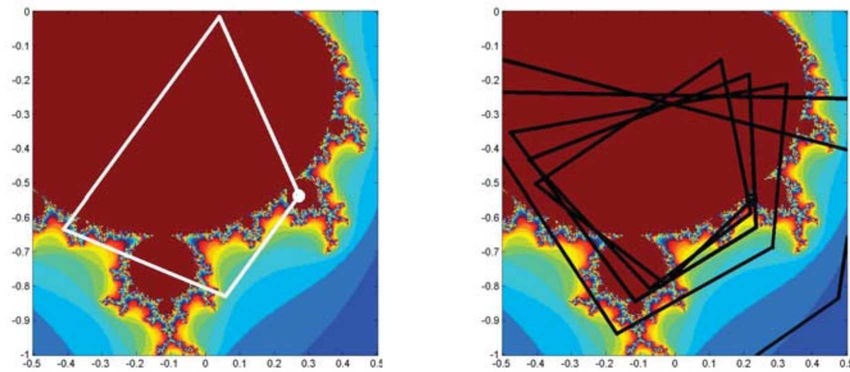



**Figure 13.3.** *Two trajectories. z0 = .25-.54i generates a cycle of length four, while nearby z0 = .22-.54i generates an unbounded trajectory.*

Figure 13.2 shows the overall geometry of the Mandelbrot set. However, this view does not have the resolution to show the richly detailed structure of the fringe just outside the boundary of the set. In fact, the set has tiny filaments reaching into the fringe region, even though the fringe appears to be solid black in the figure. It has recently been proved that the Mandelbrot set is mathematically connected, but the connected region is sometimes so thin that we cannot resolve it on a graphics screen or even compute it in a reasonable amount of time.

To see how the definition works, enter

```
z0 = .25-.54i
z = 0
```

into Matlab. Then use the up-arrow key to repeatedly execute the statement

```
z = z^2 + z0
```

The first few lines of output are

```
 0.2500 - 0.5400i
 0.0209 - 0.8100i
-0.4057 - 0.5739i
 0.0852 - 0.0744i
 0.2517 - 0.5527i
  ...
```

The values eventually settle into a cycle

```
 0.2627 - 0.5508i
 0.0156 - 0.8294i
-0.4377 - 0.5659i
 0.1213 - 0.0446i
```

```
    0.2627 - 0.5508i
     ...
```

This cycle repeats forever. The trajectory remains bounded. This tells us that the starting value value, `z0 = .25-.54i`, is in the Mandelbrot set. The same cycle is shown in the left half of figure 13.3.

On the other hand, start with

```
z0 = .22-.54i
z = 0
```

and repeatedly execute the statement

```
z = z^2 + z0
```

You will see

```
    0.2200 - 0.5400i
   -0.0232 - 0.7776i
   -0.3841 - 0.5039i
    0.1136 - 0.1529i
    0.2095 - 0.5747i
  ...
```

Then, after 24 iterations,

```
     ...
    1.5708 - 1.1300i
    1.4107 - 4.0899i
  -14.5174 -12.0794i
   6.5064e+001 +3.5018e+002i
  -1.1840e+005 +4.5568e+004i
```

The trajectory is blowing up rapidly. After a few more iterations, the floating point numbers overflow. So this `z0` is not in the Mandelbrot set. The same unbounded trajectory is shown in the right half of figure 13.3. We see that the first value, `z0 = .25-.54i`, is in the Mandelbrot set, while the second value, `z0 = .22-.54i`, which is nearby, is not.

The algorithm doesn't have to wait until `z` reachs floating point overflow. As soon as `z` satisfies

```
abs(z) >= 2
```

subsequent iterations will essential square the value of $|z|$ and it will behave like $2^{2^k}$.

Try it yourself. Put these statements on one line.

```
z0 = ...
z = 0;
while abs(z) < 2
  z = z^2+z0;
  disp(z),
end
```

Use the up arrow and backspace keys to retrieve the statement and change z0 to different values near .25-.54i. If you have to hit <ctrl>-c to break out of an infinite loop, then z0 is in the Mandelbrot set. If the while condition is eventually false and the loop terminates without your help, then z0 is not in the set.

The number of iterations required for $z$ to escape the disc of radius 2 provides the basis for showing the detail in the fringe. Let's add an iteration counter to the loop. A quantity we call depth specifies the maximum interation count and thereby determines both the level of detail and the overall computation time. Typical values of depth are several hundred or a few thousand.

```
z0 = ...
z = 0;
k = 0;
while abs(z) < 2 && k < depth
  z = z^2+z0;
  k = k + 1;
end
```

The maximum value of k is depth. If the value of k is less than depth, then z0 is outside the set. Large values of k indicate that z0 is in the fringe, close to the boundary. If k reaches depth then z0 is declared to be inside the Mandelbrot set.

Here is a small table of iteration counts s z0 ranges over complex values near 0.22-0.54i. We have set depth = 512.

| | 0.205 | 0.210 | 0.215 | 0.220 | 0.225 | 0.230 | 0.235 | 0.240 | 0.245 |
|---|---|---|---|---|---|---|---|---|---|
| -0.520 | 512 | 512 | 512 | 512 | 512 | 512 | 44 | 512 | 512 |
| -0.525 | 512 | 512 | 512 | 512 | 512 | 36 | 51 | 512 | 512 |
| -0.530 | 512 | 512 | 512 | 512 | 35 | 31 | 74 | 512 | 512 |
| -0.535 | 512 | 512 | 512 | 512 | 26 | 28 | 57 | 512 | 512 |
| -0.540 | 512 | 139 | 113 | 26 | 24 | 73 | 56 | 512 | 512 |
| -0.545 | 512 | 199 | 211 | 21 | 22 | 25 | 120 | 512 | 512 |
| -0.550 | 33 | 25 | 21 | 20 | 20 | 25 | 63 | 85 | 512 |
| -0.555 | 34 | 20 | 18 | 18 | 19 | 21 | 33 | 512 | 512 |
| -0.560 | 62 | 19 | 17 | 17 | 18 | 33 | 162 | 40 | 344 |

We see that about half of the values are less than depth; they correspond to points outside of the Mandelbrot set, in the fringe near the boundary. The other half of the values are equal to depth, corresponding to points that are regarded as in the set. If we were to redo the computation with a larger value of depth, the entries that are less than 512 in this table would not change, but some of the entries that are now capped at 512 might increase.

The iteration counts can be used as indices into an RGB color map of size depth-by-3. The first row of this map specifies the color assigned to any points on the z0 grid that lie outside the disc of radius 2. The next few rows provide colors for the points on the z0 grid that generate trajectories that escape quickly. The last row of the map is the color of the points that survive depth iterations and so are in the set.

The map used in figure 13.2 emphasizes the set itself and its boundary. The map has 12 rows of white at the beginning, one row of dark red at the end, and black in between. Images that emphasize the structure in the fringe are achieved when the color map varies cyclicly over a few dozen colors. One of the exercises asks you to experiment with color maps.

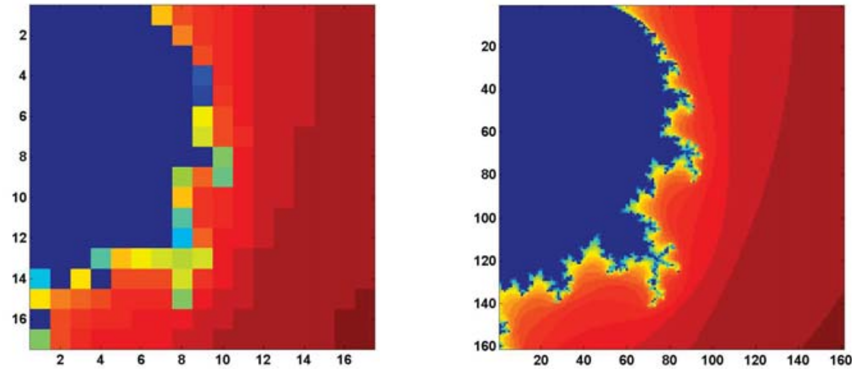



**Figure 13.4.** *Improving resolution.*

## Array operations.

Our script `mandelbrot_recap` shows how MATLAB array arithmetic operates a grid of complex numbers simultaneously and accumulates an array of iteration counters, producing images like those in figure 13.4 The code begins by defining the region in the complex plane to be sampled. A step size of `0.05` gives the coarse resolution shown on the right in the figure.

```
x = 0: 0.05: 0.80;
y = x';
```

The next section of code uses an elegant, but tricky, bit of MATLAB indexing known as Tony's Trick. The quantities `x` and `y` are one-dimensional real arrays of length `n`, one a column vector and the other a row vector. We want to create a two-dimensional $n$-by-$n$ array with elements formed from all possible sums of elements from $x$ and $y$.

$$z_{k,j} = x_k + y_j i, \quad i = \sqrt{-1}, \; k, j = 1, ..., n$$

This can be done by generating a vector `e` of length `n` with all elements equal to one. Then the quantity `x(e,:)` is a two-dimensional array formed by using `x`, which is the same as `x(1,:)`, `n` times. Similarly, `y(:,e)` is a two-dimensional arrray containing `n` copies of the column vector `y`.

```
n = length(x);
e = ones(n,1);
z0 = x(e,:) + i*y(:,e);
```

If you find it hard to remember Tony's indexing trick, the function `meshgrid` does the same thing in two steps.

```
[X,Y] = meshgrid(x,y);
z0 = X + i*Y;
```

Now initialize two more arrays, one for the complex iterates and one for the counts.

```
z = zeros(n,n);
c = zeros(n,n);
```

Here is the Mandelbrot iteration repeated `depth` times. With each iteration we also keep track of the iterates that are still within the circle of radius 2.

```
depth = 32;
for k = 1:depth
   z = z.^2 + z0;
   c(abs(z) < 2) = k;
end
```

We are now finished with `z`. The actual values of `z` are not important, only the counts are needed to create the image. Our grid is small enough that we can actually print out the counts `c`.

```
c
```

The results are

```
c =
  32  32  32  32  32  32  11   7   6   5   4   3   3   3   2   2   2
  32  32  32  32  32  32  32   9   6   5   4   3   3   3   2   2   2
  32  32  32  32  32  32  32  32   7   5   4   3   3   3   2   2   2
  32  32  32  32  32  32  32  32  27   5   4   3   3   3   2   2   2
  32  32  32  32  32  32  32  32  30   6   4   3   3   3   2   2   2
  32  32  32  32  32  32  32  32  13   7   4   3   3   3   2   2   2
  32  32  32  32  32  32  32  32  14   7   5   3   3   2   2   2   2
  32  32  32  32  32  32  32  32  32  17   4   3   3   2   2   2   2
  32  32  32  32  32  32  32  16   8  18   4   3   3   2   2   2   2
  32  32  32  32  32  32  32  11   6   5   4   3   3   2   2   2   2
  32  32  32  32  32  32  32  19   6   5   4   3   2   2   2   2   2
  32  32  32  32  32  32  32  23   8   4   4   3   2   2   2   2   2
  32  32  32  19  11  13  14  15  14   4   3   2   2   2   2   2   2
  22  32  12  32   7   7   7  14   6   4   3   2   2   2   2   2   2
  12   9   8   7   5   5   5  17   4   3   3   2   2   2   2   2   1
  32   7   6   5   5   4   4   4   3   3   2   2   2   2   2   1   1
  17   7   5   4   4   4   4   3   3   2   2   2   2   2   2   1   1
```

We see that points in the upper left of the grid, with fairly small initial `z0` values, have survived 32 iterations without going outside the circle of radius two, while points in the lower right, with fairly large initial values, have lasted only one or two iterations. The interesting grid points are in between, they are on the fringe.

Now comes the final step, making the plot. The `image` command does the job, even though this is not an image in the usual sense. The count values in `c` are used as indices into a 32-by-3 array of RGB color values. In this example, the `jet` colormap is reversed to give dark red as its first value, pass through shades of green and yellow, and finish with dark blue as its 32-nd and final value.

```
image(c)
axis image
colormap(flipud(jet(depth)))
```

Exercises ask you to increase the resolution by decreasing the step size, thereby producing the other half of figure 13.4, to investigate the effect of changing `depth`, and to invesigate other color maps.

## Mandelbrot GUI

The `exm` toolbox function `mandelbrot` is your starting point for exploration of the Mandelbrot set. With no arguments, the statement

```
mandelbrot
```

provides thumbnail icons of the twelve regions featured in this chapter. The statement

```
mandelbrot(r)
```

with `r` between 1 and 12 starts with the `r`-th region. The statement

```
mandelbrot(center,width,grid,depth,cmapindx)
```

explores the Mandelbrot set in a square region of the complex plane with the specified `center` and `width`, using a `grid`-by-`grid` grid, an iteration limit of `depth`, and the color map number `cmapindx`. The default values of the parameters are

```
center = -0.5+0i
width = 3
grid = 512
depth = 256
cmapindx = 1
```

In other words,

```
mandelbrot(-0.5+0i, 3, 512, 256, 1)
```

generates figure 13.2, but with the `jets` color map. Changing the last argument from `1` to `6` generates the actual figure 13.2 with the `fringe` color map. On my laptop, these computations each take about half a second.

A simple estimate of the execution time is proportional to

```
grid^2 * depth
```

So the statement

```
mandelbrot(-0.5+0i, 3, 2048, 1024, 1)
```

could take

$$(2048/512)^2 \cdot (1024/256) = 64$$

times as long as the default. However, this is an overestimate and the actual execution time is about 11 seconds.

Most of the computational time required to compute the Mandelbrot set is spent updating two arrays `z` and `kz` by repeatedly executing the step

```
z = z.^2 + z0;
j = (abs(z) < 2);
kz(j) = d;
```

This computation can be carried out faster by writing a function `mandelbrot_step` in C and creating as a MATLAB *executable* or *c-mex* file. Different machines and operating systems require different versions of a mex file, so you should see files with names like `mandelbrot_step.mexw32` and `mandelbrot_step.glnx64` in the `exm` toolbox.

The `mandelbrot` gui turns on the MATLAB zoom feature. The mouse pointer becomes a small magnifying glass. You can click and release on a point to zoom by a factor of two, or you can click and drag to delineate a new region.

The `mandelbrot` gui provides several uicontrols. Try these as you read along.

The `listbox` at the bottom of the gui allows you to select any of the predefined regions shown in the figures in this chapter.

`depth`. Increase the depth by a factor of `3/2` or `4/3`.

`grid`. Refine the grid by a factor of `3/2` or `4/3`. The depth and grid size are always a power of two or three times a power of two. Two clicks on the `depth` or `grid` button doubles the parameter.

`color`. Cycle through several color maps. `jets` and `hots` are cyclic repetitions of short copies of the basic MATLAB `jet` and `hot` color maps. `cmyk` cycles through eight basic colors, blue, green, red, cyan, magenta, yellow, gray, and black. `fringe` is a noncyclic map used for images like figure 13.2.

`exit`. Close the gui.

The Mandelbrot set is *self similar*. Small regions in the fringe reveal features that are similar to the original set. Figure 13.6, which we have dubbed "Mandelbrot

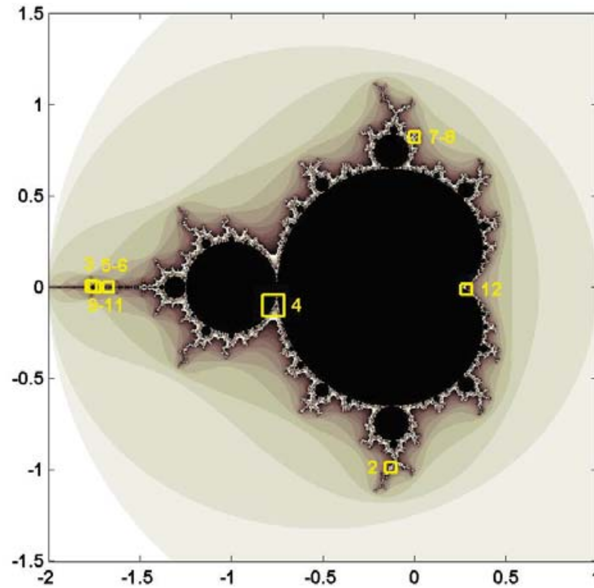



**Figure 13.5.** *The figures in this chapter, and the predefined regions in our mandelbrot program, show these regions in the fringe just outside the Mandelbrot set.*

Junior", is one example. Figure 13.7, which we call the "'Plaza", uses our `flag` colormap to reveal fine detail in red, white and blue.

The portion of the boundary of the Mandelbrot set between the two large, nearly circular central regions is known as "The Valley of the Seahorses". Figure 13.8 shows the result of zooming in on the peninsula between the two nearly circular regions of the set. The figure can be generated directly with the command

```
mandelbrot(-.7700-.1300i,0.1,1024,512)
```

We decided to name the image in figure 13.9 the "West Wing" because it resembles the X-wing fighter that Luke Skywalker flies in Star Wars and because it is located near the leftmost, or far western, portion of the set. The magnification factor is a relatively modest $10^4$, so `depth` does not need to be very large. The command to generate the West Wing is

```
mandelbrot(-1.6735-0.0003318i,1.5e-4,1024,160,1)
```

One of the best known examples of self similarity, the "Buzzsaw", is shown in figure 13.11. It can be generated with

```
mandelbrot(0.001643721971153+0.822467633298876i, ...
```

```
    4.0e-11,1024,2048,2)
```

Taking `width = 4.0e-11` corresponds to a magnification factor of almost $10^{11}$. To appreciate the size of this factor, if the original Mandelbrot set fills the screen on your computer, the Buzzsaw is smaller than the individual transistors in your machine's microprocessor.

We call figure 13.12 "Nebula" because it reminds us of interstellar dust. It is generated by

```
mandelbrot(0.73752777-0.12849548i,4.88e-5,1024,2048,3)
```

The next three images are obtained by carefully zooming on one location. We call them the "Vortex", the "Microbug", and the "Nucleus".

```
mandelbrot(-1.74975914513036646-0.00000000368513796i, ...
   6.0e-12,1024,2048,2)
mandelbrot(-1.74975914513271613-0.00000000368338015i, ...
   3.75e-13,1024,2048,2)
mandelbrot(-1.74975914513272790-0.00000000368338638i, ...
   9.375e-14,1024,2048,2)
```

The most intricate and colorful image among our examples is figure 13.16, the "Geode". It involves a fine grid and a large value of `depth` and consequently requires a few minutes to compute.

```
mandelbrot(0.28692299709-0.01218247138i,6.0e-10,2048,4096,1)
```

These examples are just a tiny sampling of the structure of the Mandelbrot set.

## Further Reading

We highly recommend a real time fractal zoomer called "XaoS", developed by Thomas Marsh, Jan Hubicka and Zoltan Kovacs, assisted by an international group of volunteers. See

```
http://wmi.math.u-szeged.hu/xaos/doku.php
```

If you are expert at using your Web browser and possibly downloading an obscure video codec, take a look at the Wikipedia video

```
http://commons.wikimedia.org/wiki/ ...
   Image:Fractal-zoom-1-03-Mandelbrot_Buzzsaw.ogg
```

It's terrific to watch, but it may be a lot of trouble to get working.

## Recap

```
%% Mandelbrot Chapter Recap
% This is an executable program that illustrates the statements
```

```
% introduced in the Mandelbrot Chapter of "Experiments in MATLAB".
% You can access it with
%
%    mandelbrot_recap
%    edit mandelbrot_recap
%    publish mandelbrot_recap
%
% Related EXM programs
%
%    mandelbrot

%% Define the region.
   x = 0: 0.05: 0.8;
   y = x';

%% Create the two-dimensional complex grid using Tony's indexing trick.
   n = length(x);
   e = ones(n,1);
   z0 = x(e,:) + i*y(:,e);

%% Or, do the same thing with meshgrid.
   [X,Y] = meshgrid(x,y);
   z0 = X + i*Y;

%% Initialize the iterates and counts arrays.
   z = zeros(n,n);
   c = zeros(n,n);

%% Here is the Mandelbrot iteration.
   depth = 32;
   for k = 1:depth
      z = z.^3 + z0;
      c(abs(z) < 2) = k;
   end

%% Create an image from the counts.
   c
   image(c)
   axis image

%% Colors
   colormap(flipud(jet(depth)))
```

## Exercises

13.1 *Explore*. Use the `mandelbrot` gui to find some interesting regions that, as far as you know, have never been seen before. Give them your own names.

13.2 *depth*. Modify `mandelbrot_recap` to reproduce our table of iteration counts for `x = .205:.005:.245` and `y = -.520:-.005:-.560`. First, use `depth = 512`. Then use larger values of `depth` and see which table entries change.

13.3 *Resolution*. Reproduce the image in the right half of figure 13.4.

13.4 *Big picture*. Modify `mandelbrot_recap` to display the entire Mandelbrot set.

13.5 *Color maps*. Investigate color maps. Use `mandelbrot_recap` with a smaller step size and a large value of `depth` to produce an image. Find how `mandelbrot` computes the cyclic color maps called `jets`, `hots` and `sepia`. Then use those maps on your image.

13.6 *p-th power*. In either `mandelbrot_recap` or the `mandelbrot` gui, change the power in the Mandelbrot iteration to

$$z_{k+1} = z_k^p + z_0$$

for some fixed $p \neq 2$. If you want to try programming Handle Graphics, add a button to `mandelbrot` that lets you set $p$.

13.7 *Too much magnification*. When the width of the region gets to be smaller than about $10^{-15}$, our `mandelbrot` gui does not work very well. Why?

13.8 *Spin the color map*. This might not work very well on your computer because it depends on what kind of graphics hardware you have. When you have an interesting region plotted in the figure window, bring up the command window, resize it so that it does not cover the figure window, and enter the command

```
spinmap(10)
```

I won't try to describe what happens – you have to see it for yourself. The effect is most dramatic with the "seahorses2" region. Enter

```
help spinmap
```

for more details.

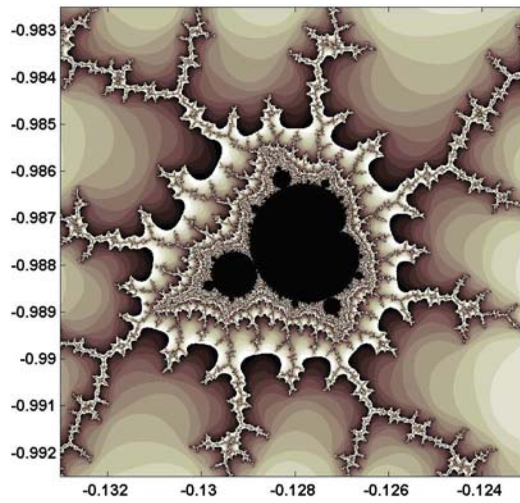



**Figure 13.6.** *Region #2, "Mandelbrot Junior". The fringe around the Mandelbrot set in self similar. Small versions of the set appear at all levels of magnification.*

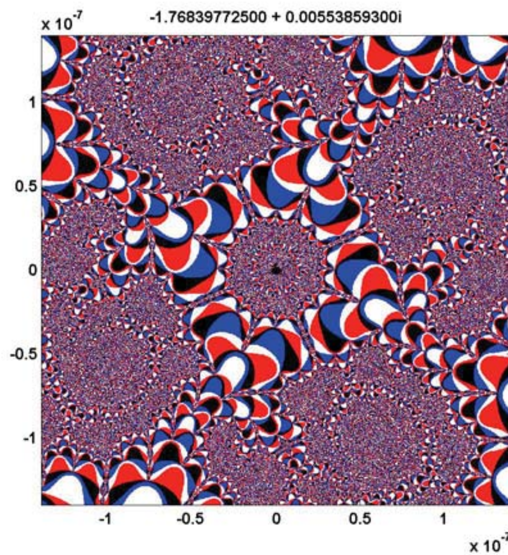



**Figure 13.7.** *Region #3, "Plaza", with the flag color map.*

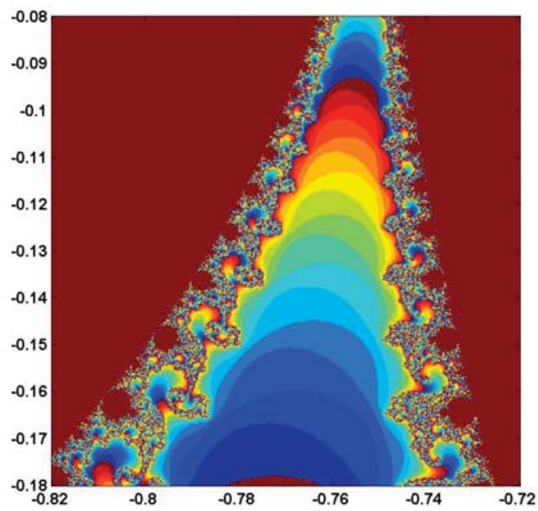



**Figure 13.8.** *Region #4. "Valley of the Seahorses".*

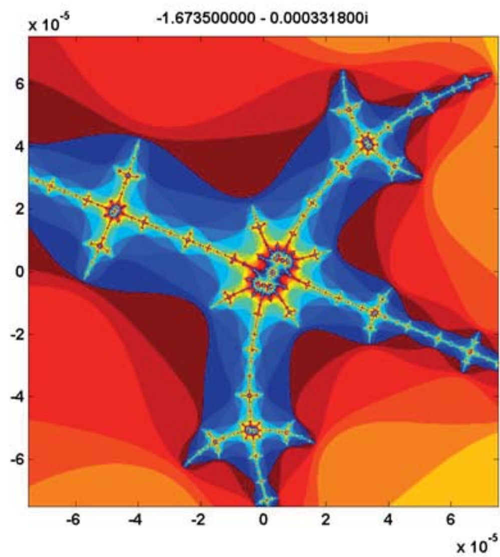



**Figure 13.9.** *Region #5. Our "West Wing" is located just off the real axis in the thin far western portion of the set, near real(z) = -1.67.*

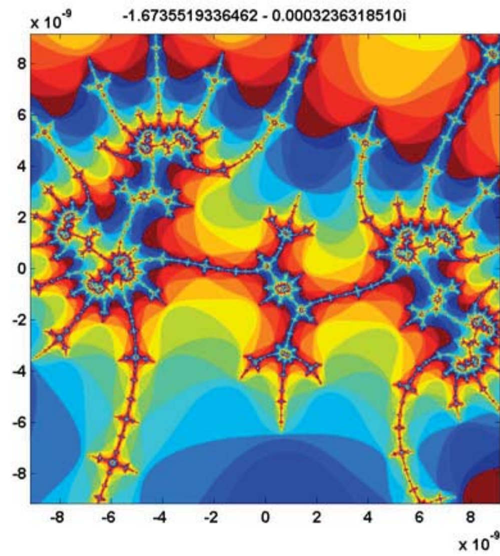



**Figure 13.10.** *Region #6. "Dueling Dragons".*

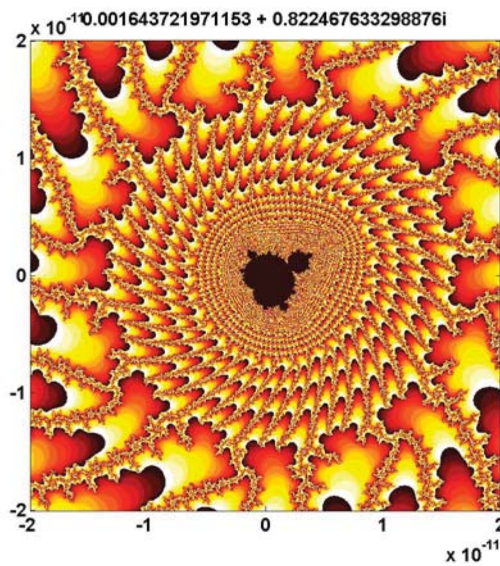



**Figure 13.11.** *Region #7. The "Buzzsaw" requires a magnification factor of* $10^{11}$ *to reveal a tiny copy of the Mandelbrot set.*

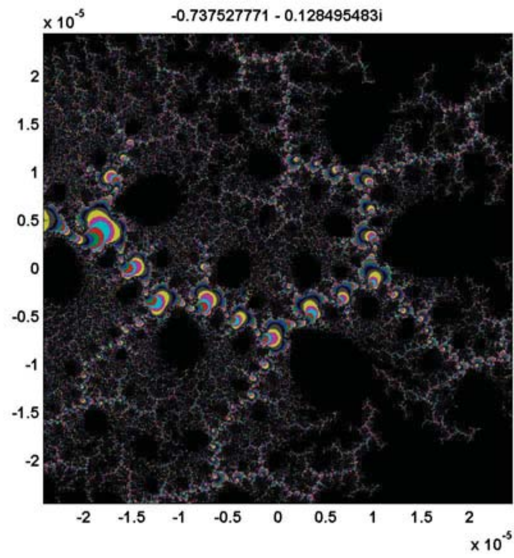



**Figure 13.12.** *Region #8. "Nebula". Interstellar dust.*

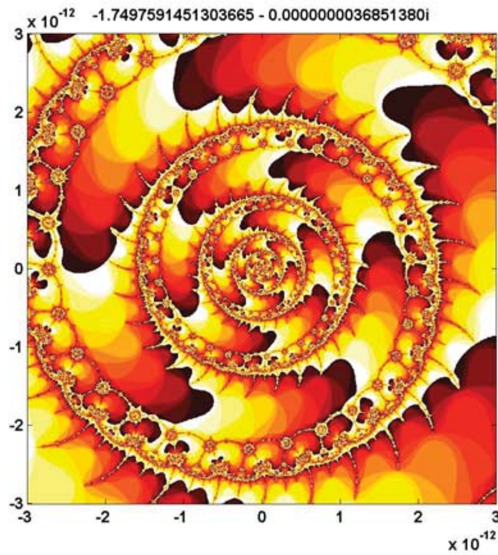



**Figure 13.13.** *Region #9. A vortex, not far from the West Wing. Zoom in on one of the circular "microbugs" near the left edge.*

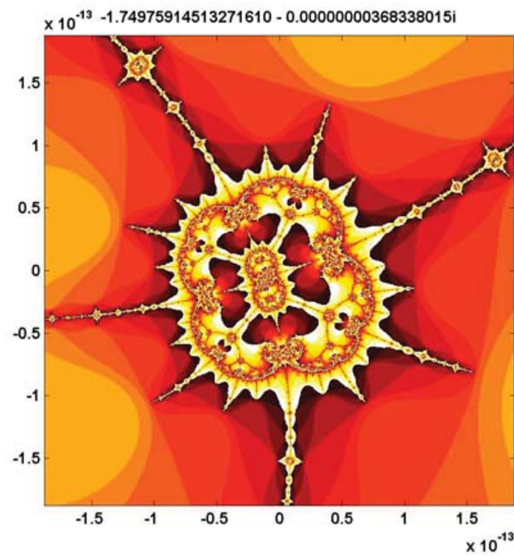



**Figure 13.14.** *Region #10. A* $10^{13}$ *magnification factor reveals a "microbug" within the vortex.*

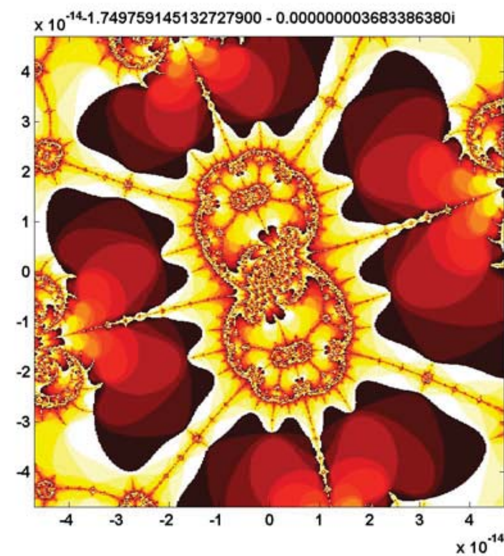



**Figure 13.15.** *Region #11. The nucleus of the microbug.*

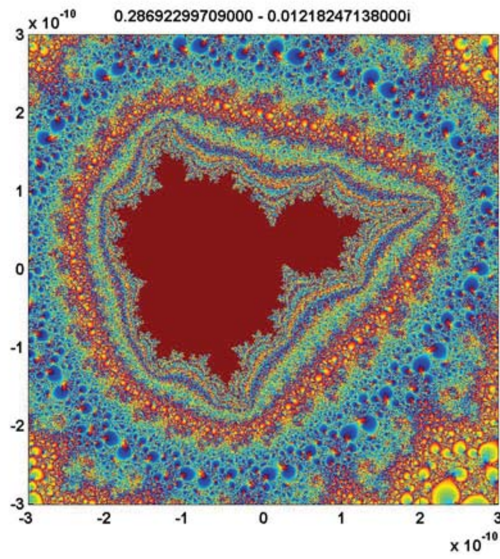



**Figure 13.16.** *Region #12. "Geode". This colorful image requires a 2048-by-2048 grid and depth = 8192.*

## Chapter 15

# Ordinary Differential Equations

*Mathematical models in many different fields.*

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

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

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

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

---

Copyright © 2011 Cleve Moler
MATLAB® is a registered trademark of MathWorks, Inc.™
October 4, 2011

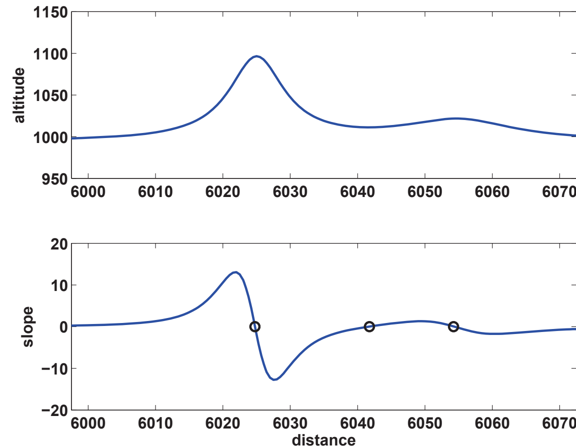



**Figure 15.1.** *Altitude along a mountain road, and derivative of that altitude. The derivative is zero at the local maxima and minima of the altitude.*

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

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

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

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

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

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

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

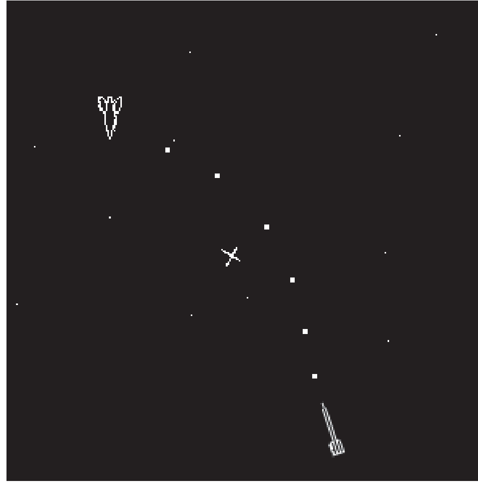


**Figure 15.2.** *Spacewar, the world's first video game. The gravitational pull of the central star causes the torpedo to move in an elliptical orbit.*

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

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

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

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

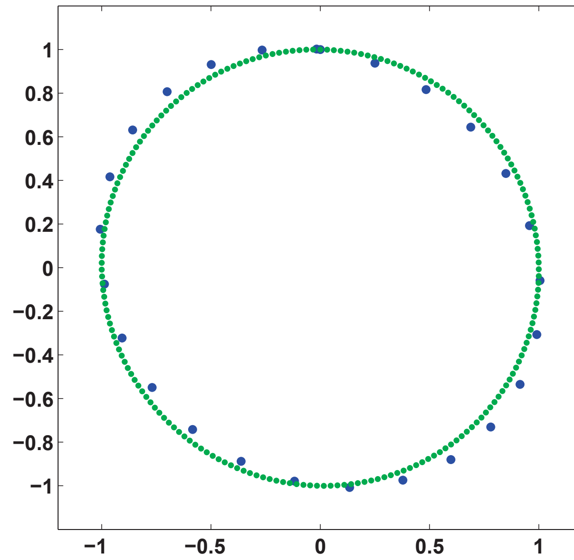



**Figure 15.3.** *The 25 blue points are generated by the Spacewar orbit generator with a step size of 1/4. The 201 green points are generated with a step size of 1/32.*

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

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

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

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

This can be rewritten as

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

You have probably noticed that the right hand side of this pair of equations involves

two different values of the time variable, $t$ and $t + h$. That fact turns out to be important, but let's ignore it for now.

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

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

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

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

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

The initial values of $x$ and $y$ provide the initial conditions

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

The exact solution to the system is

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

To see why, recall the trig identities

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

For small values of $h$,

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

Consequently

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

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

It turns out that the solution computed by the Spacewar orbit generator with a fixed step size $h$ is an ellipse, not an exact circle. Decreasing $h$ and taking more

steps generates a better approximation to a circle. Actually, the fact that $x(t+h)$ is used instead of $x(t)$ in the second half of the step means that the method is not quite as simple as it might seem. This subtle change is responsible for the fact that the method generates ellipses instead of spirals. One of the exercises asks you to verify this fact experimentally.

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

In MATLAB a system of odes takes the form

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

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

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

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

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

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

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

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

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

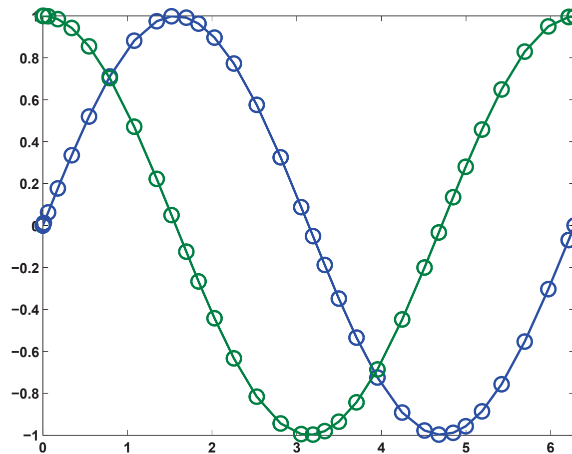



**Figure 15.4.** *Graphs of sine and cosine generated by ode23.*

- `y0`, the vector of initial conditions.

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

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

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

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

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

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

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

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

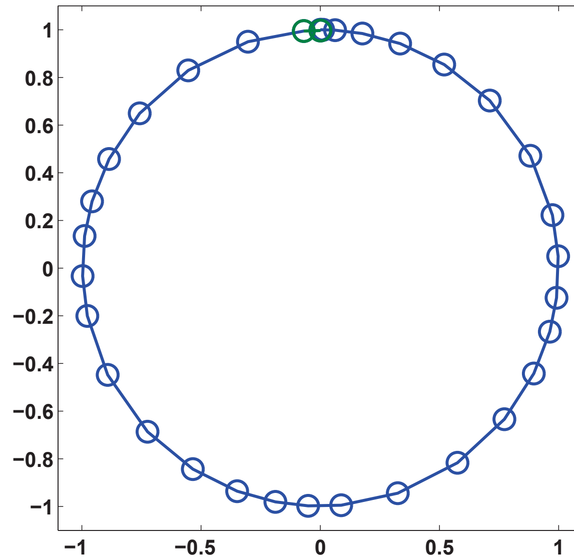



**Figure 15.5.** *Graph of a circle generated by ode23.*

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

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

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

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

Once `acircle` has been defined, the statement

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

automatically produces figure 15.4. And, the statement

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

captures the output so you can process it yourself.

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

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

will also produce figure 15.5.

Use the command

```
doc ode23
```

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

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

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

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

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

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

is

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

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

## Recap

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

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

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

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

%% Phase Plot.
    figure
    tspan = [0 2*pi];
    y0 = [0; 1];
    [t,y] = ode23(acircle,tspan,y0)
```

```
    plot(y(:,1),y(:,2),'-o')
    axis square
    axis([-1.1 1.1 -1.1 1.1])

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

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

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

## Exercises

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

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

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

and

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

to place two plots in one figure window. The statement

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

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

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

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

```
help diff
help subplot
```

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

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

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

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

15.5 *Linear system* Write the system of differential equations

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

in matrix-vector form,

$$\dot{y} = Ay$$

where $y$ is a vector-valued function of time,

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

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

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

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

with initial conditions

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

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

```
doc ode23
```

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

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

with initial conditions

$$y_1(0) = 0$$
$$y_2(0) = 1$$

on the interval

$$0 \leq t \leq 7.4163$$

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

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

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

with initial conditions

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

on an interval

$$0 \le t \le T$$

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

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

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

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

Why do solutions of

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

have such different behavior if $p$ is odd or even?

## Chapter 16

# Predator-Prey Model

*Models of population growth.*

The simplest model for the growth, or decay, of a population says that the growth rate, or the decay rate, is proportional to the size of the population itself. Increasing or decreasing the size of the population results in a proportional increase or decrease in the number of births and deaths. Mathematically, this is described by the differential equation

$$\dot{y} = ky$$

The proportionality constant $k$ relates the size of the population, $y(t)$, to its rate of growth, $\dot{y}(t)$. If $k$ is positive, the population increases; if $k$ is negative, the population decreases.

As we know, the solution to this equation is a function $y(t)$ that is proportional to the exponential function

$$y(t) = \eta e^{kt}$$

where $\eta = y(0)$.

This simple model is appropriate in the initial stages of growth when there are no restrictions or constraints on the population. A small sample of bacteria in a large Petri dish, for example. But in more realistic situations there are limits to growth, such as finite space or food supply. A more realistic model says that the population competes with itself. As the population increases, its growth rate decreases linearly. The differential equation is sometimes called the *logistic* equation.

$$\dot{y} = k(1 - \frac{y}{\mu})y$$

---

Copyright © 2011 Cleve Moler
MATLAB® is a registered trademark of MathWorks, Inc.™
October 4, 2011

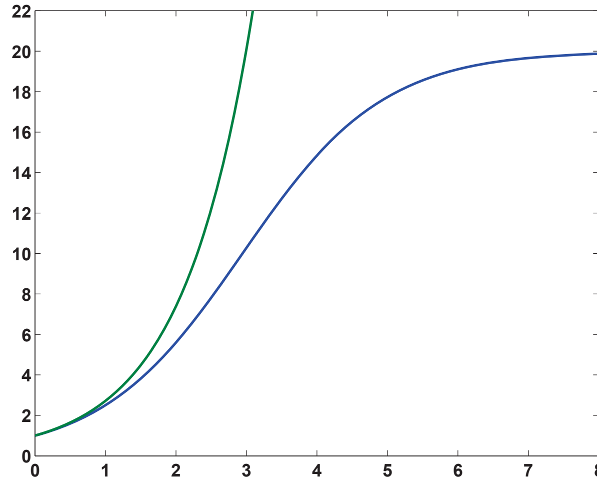



**Figure 16.1.** *Exponential growth and logistic growth.*

The new parameter $\mu$ is the *carrying capacity*. As $y(t)$ approaches $\mu$ the growth rate approaches zero and the growth ultimately stops. It turns out that the solution is

$$y(t) = \frac{\mu\eta e^{kt}}{\eta e^{kt} + \mu - \eta}$$

You can easily verify for yourself that as $t$ approaches zero, $y(t)$ approaches $\eta$ and that as $t$ approaches infinity, $y(t)$ approaches $\mu$. If you know calculus, then with quite a bit more effort, you can verify that $y(t)$ actually satisfies the logistic equation.

Figure 16.1 shows the two solutions when both $\eta$ and $k$ are equal to one. The exponential function

$$y(t) = e^t$$

gives the rapidly growing green curve. With carrying capacity $\mu = 20$, the logistic function

$$y(t) = \frac{20e^t}{e^t + 19}$$

gives the more slowly growing blue curve. Both curves have the same initial value and initial slope. The exponential function grows exponentially, while the logistic function approaches, but never exceeds, its carrying capacity.

Figure 16.1 was generated with the following code.

```
k = 1
eta = 1
mu = 20
t = 0:1/32:8;
```

```
y = mu*eta*exp(k*t)./(eta*exp(k*t) + mu - eta);
plot(t,[y; exp(t)])
axis([0 8 0 22])
```

If you don't have the formula for the solution to the logistic equation handy, you can compute a numerical solution with `ode45`, one of the MATLAB ordinary differential equation solvers. Try running the following code. It will automatically produce a plot something like the blue curve in figure 16.1.

```
k = 1
eta = 1
mu = 20
ydot = @(t,y) k*(1-y/mu)*y
ode45(ydot,[0 8],eta)
```

The `@` sign and `@(t,y)` specify that you are defining a function of `t` and `y`. The `t` is necessary even though it doesn't explicitly appear in this particular differential equation.

The logistic equation and its solution occur in many different fields. The logistic function is also known as the *sigmoid* function and its graph is known as the *S-curve*.

Populations do not live in isolation. Everybody has a few enemies here and there. The Lotka-Volterra predator-prey model is the simplest description of competition between two species. Think of rabbits and foxes, or zebras and lions, or little fish and big fish.

The idea is that, if left to themselves with an infinite food supply, the rabbits or zebras would live happily and experience exponential population growth. On the other hand, if the foxes or lions were left with no prey to eat, they would die faster than they could reproduce, and would experience exponential population decline.

The predator-prey model is a pair of differential equations involving a pair of competing populations, $y_1(t)$ and $y_2(t)$. The growth rate for $y_1$ is a linear function of $y_2$ and vice versa.

$$\begin{aligned}\dot{y}_1 &= (1 - \frac{y_2}{\mu_2})y_1 \\ \dot{y}_2 &= -(1 - \frac{y_1}{\mu_1})y_2\end{aligned}$$

We are using notation $y_1(t)$ and $y_2(t)$ instead of, say, $r(t)$ for rabbits and $f(t)$ for foxes, because our MATLAB program uses a two-component vector `y`.

The extra minus sign in the second equation distinguishes the predators from the prey. Note that if $y_1$ ever becomes zero, then

$$\dot{y}_2 = -y_2$$

and the predators are in trouble. But if $y_2$ ever becomes zero, then

$$\dot{y}_1 = y_1$$

and the prey population grows exponentially.

We have a formula for the solution of the single species logistic model. However it is not possible to express the solution to this predator-prey model in terms of exponential, trigonmetric, or any other elementary functions. It is necessary, but easy, to compute numerical solutions.

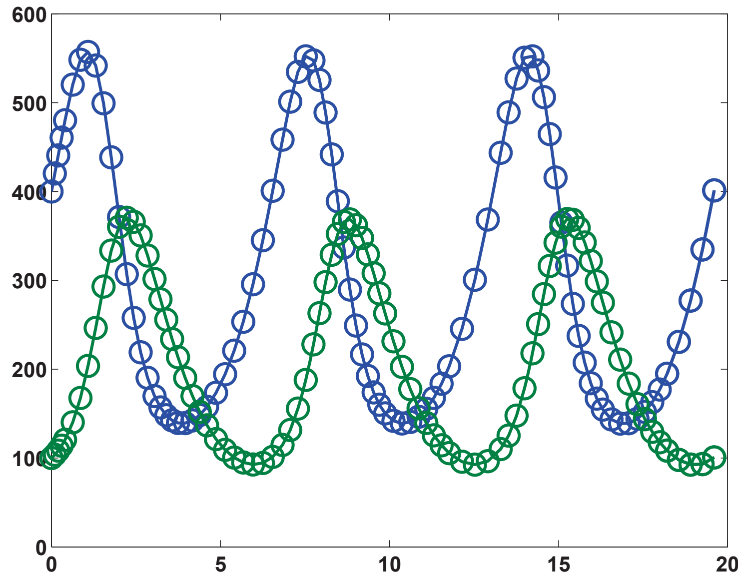



**Figure 16.2.** *A typical solution of the predator-prey equations.*

There are four parameters, the two constants $\mu_1$ and $\mu_2$, and the two initial conditions,

$$\eta_1 = y_1(0)$$
$$\eta_2 = y_2(0)$$

If we happen to start with $\eta_1 = \mu_1$ and $\eta_2 = \mu_2$, then both $\dot{y}_1$ and $\dot{y}_2$ are zero and the populations remain constant at their initial values. In other words, the point $(\mu_1, \mu_2)$ is an *equilibrium* point. The origin, $(0, 0)$ is another *equilibrium* point, but not a very interesting one.

The following code uses `ode45` to automatically plot the typical solution shown in figure 16.2.

```
mu = [300 200]'
eta = [400 100]'
signs = [1 -1]'
pred_prey_ode = @(t,y) signs.*(1-flipud(y./mu)).*y
period = 6.5357
ode45(pred_prey_ode,[0 3*period],eta)
```

There are two tricky parts of this code. MATLAB vector operations are used to define `pred_prey_ode`, the differential equations in one line. And, the calculation that generates figure 16.3 provides the value assigned to `period`. This value specifies a value of $t$ when the populations return to their initial values given by `eta`. The code integrates over three of these time intervals, and so at the end we get back to where we started.

The circles superimposed on the plots in figure 16.2 show the points where `ode45` computes the solution. The plots look something like trig functions, but they're not. Notice that the curves near the minima are broader, and require more steps to compute, then the curves near the maxima. The plot of $\sin t$ would look the same at the top as the bottom.

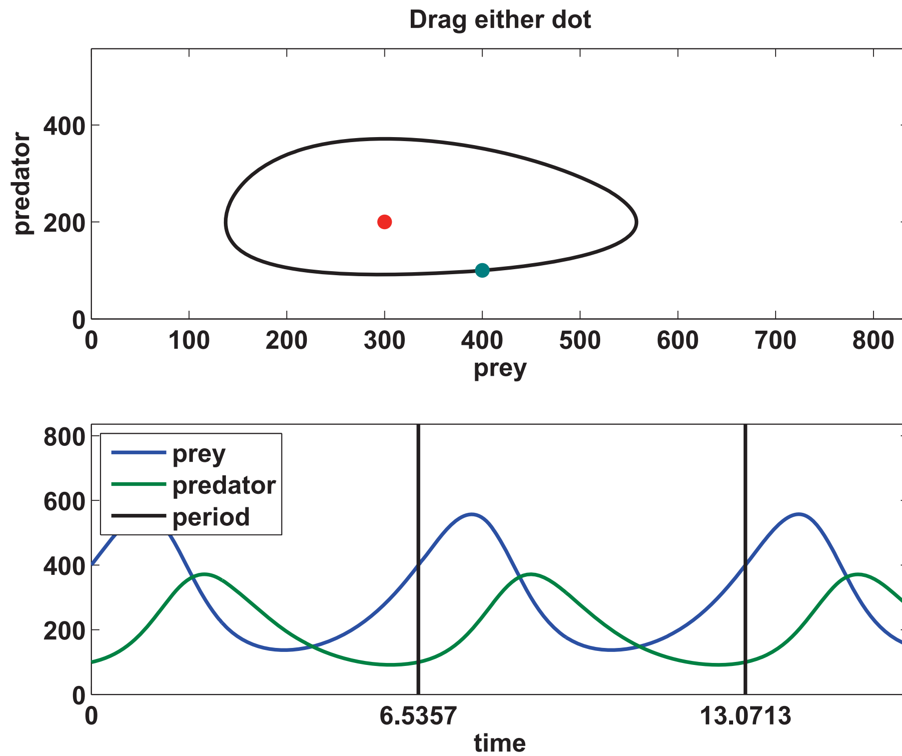



**Figure 16.3.** *The predprey experiment.*

Our MATLAB program `exm/predprey` shows a red dot at the equilibrium point, $(\mu_1, \mu_2)$, and a blue-green dot at the initial point, $(\eta_1, \eta_2)$. When you drag either dot with the mouse, the solution is recomputing by `ode45` and plotted. Figure 16.3 shows that two plots are produced — a *phase plane* plot of $y_2(t)$ versus $y_1(t)$ and a *time series* plot of $y_1(t)$ and $y_2(t)$ versus $t$. Figures 16.2 and 16.3 have the same parameters, and consequently show the same solution, but with different scaling of

the axes.

The remarkable property of the Lotka-Volterra model is that the solutions are always periodic. The populations always return to their initial values and repeat the cycle. This property is not obvious and not easy to prove. It is rare for nonlinear models to have periodic solutions.

The difficult aspect of computing the solution to the predator-prey equations is determining the length of the period. Our `predprey` program uses a feature of the Matlab ODE solvers called "event handling" to compute the length of a period.

If the initial values $(\eta_1, \eta_2)$ are close to the equilibrium point $(\mu_1, \mu_2)$, then the length of the period is close to a familar value. An exercise asks you to discover that value experimentally.

## Recap

```
%% Predator-Prey Chapter Recap
% This is an executable program that illustrates the statements
% introduced in the Preditor Prey Chapter of "Experiments in MATLAB".
% You can access it with
%
%    predprey_recap
%    edit predprey_recap
%    publish predprey_recap
%
%  Related EXM programs
%
%    predprey


%% Exponential and Logistic Growth.
   close all
   figure
   k = 1
   eta = 1
   mu = 20
   t = 0:1/32:8;
   y = mu*eta*exp(k*t)./(eta*exp(k*t) + mu - eta);
   plot(t,[y; exp(t)])
   axis([0 8 0 22])
   title('Exponential and logistic growth')
   xlabel('t')
   ylabel('y')

%% ODE45 for the Logistic Model.
   figure
   k = 1
```

```
   eta = 1
   mu = 20
   ydot = @(t,y) k*(1-y/mu)*y
   ode45(ydot,[0 8],eta)

%% ODE45 for the Predator-Prey Model.
   figure
   mu = [300 200]'
   eta = [400 100]'
   signs = [1 -1]'
   pred_prey_ode = @(t,y) signs.*(1-flipud(y./mu)).*y
   period = 6.5357
   ode45(pred_prey_ode,[0 3*period],eta)

%% Our predprey gui.
   figure
   predprey
```

## Exercises

16.1 *Plot.* Make a more few plots like figures 16.1 and 16.2, but with different values of the parameters $k$, $\eta$, and $\mu$.

16.2 *Decay.* Compare exponential and logistic decay. Make a plot like figure 16.1 with negative $k$.

16.3 *Differentiate.* Verify that our formula for $y(t)$ actually satisfies the logistic differential equations.

16.4 *Easy as pie.* In `predprey`, if the red and blue-green dots are close to each other, then the length of the period is close to a familar value. What is that value? Does that value depend upon the actual location of the dots, or just their relative closeness?

16.5 *Period.* In `predprey`, if the red and blue-green dots are far apart, does the length of the period get longer or shorter? Is it possible to make the period shorter than the value it has near equilibrium?

16.6 *Phase.* If the initial value is near the equilibrium point, the graphs of the predator and prey populations are nearly sinusoidal, with a phase shift. In other words, after the prey population reaches a maximum or minimum, the predator population reaches a maximum or minimum some fraction of the period later. What is that fraction?

16.7 *Pitstop.* The `predprey` subfunction `pitstop` is involved in the "event handling" that `ode45` uses to compute the period. `pitstop`, in turn, uses `atan2` to compute angles `theta0` and `theta1`. What is the difference between the two MATLAB functions `atan2`, which takes two arguments, and `atan`, which takes only one? What happens if `atan2(v,u)` is replaced by `atan(v/u)` in `predprey`?
Draw a sketch showing the angles `theta0` and `theta1`.

16.8 *tfinal.* The call to `ode45` in `predprey` specifies a time interval of [0 100]. What is the significance of the value 100? What happens if you change it?

16.9 *Limit growth.* Modify `predprey` to include a growth limiting term for the prey, similar to one in the logistic equation. Avoid another parameter by making the carrying capacity twice the initial value. The equations become

$$\begin{aligned}\dot{y}_1 &= (1 - \frac{y_1}{2\,\eta_1})(1 - \frac{y_2}{\mu_2})y_1 \\ \dot{y}_2 &= -(1 - \frac{y_1}{\mu_1})y_2\end{aligned}$$

What happens to the shape of the solution curves? Are the solutions still periodic? What happens to the length of the period?

## Chapter 17

# Orbits

*Dynamics of many-body systems.*

Many mathematical models involve the dynamics of objects under the influence of both their mutual interaction and the surrounding environment. The objects might be planets, molecules, vehicles, or people. The ultimate goal of this chapter is to investigate the *n-body problem* in celestial mechanics, which models the dynamics of a system of planets, such as our solar system. But first, we look at two simpler models and programs, a bouncing ball and Brownian motion.

The `exm` program `bouncer` is a model of a bouncing ball. The ball is tossed into the air and reacts to the pull of the earth's gravitation force. There is a corresponding pull of the ball on the earth, but the earth is so massive that we can neglect its motion.

Mathematically, we let $v(t)$ and $z(t)$ denote the velocity and the height of the ball. Both are functions of time. High school physics provides formulas for $v(t)$ and $z(t)$, but we choose not to use them because we are anticipating more complicated problems where such formulas are not available. Instead, we take small steps of size $\delta$ in time, computing the velocity and height at each step. After the initial toss, gravity causes the velocity to decrease at a constant rate, $g$. So each step updates $v(t)$ with

$$v(t+\delta) = v(t) - \delta\, g$$

The velocity is the rate of change of the height. So each step updates $z(t)$ with

$$z(t+\delta) = z(t) + \delta\, v(t)$$

Here is the core of `bouncer.m`.

Copyright © 2011 Cleve Moler
MATLAB® is a registered trademark of MathWorks, Inc.™
October 4, 2011

```
[z0,h] = initialize_bouncer;
g = 9.8;
c = 0.75;
delta = 0.005;
v0 = 21;
while v0 >= 1
   v = v0;
   z = z0;
   while all(z >= 0)
      set(h,'zdata',z)
      drawnow
      v = v - delta*g;
      z = z + delta*v;
   end
   v0 = c*v0;
end
finalize_bouncer
```

The first statement

```
[z0,h] = initialize_bouncer;
```

generates the plot of a sphere shown in figure 17.1 and returns `z0`, the $z$-coordinates of the sphere, and `h`, the Handle Graphics “handle” for the plot. One of the exercises has you investigate the details of `initialize_bouncer`. The figure shows the situation at both the start and the end of the simulation. The ball is at rest and so the picture is pretty boring. To see what happens during the simulation, you have to actually run `bouncer`.

The next four statements in `bouncer.m` are

```
g = 9.8;
c = 0.75;
delta = 0.005;
v0 = 21;
```

These statements set the values of the acceleration of gravity `g`, an elasticity coefficient `c`, the small time step `delta`, and the initial velocity for the ball, `v0`.

All the computation in `bouncer` is done within a doubly nested `while` loop. The outer loop involves the initial velocity `v0`.

```
while v0 >= 1
   ...
   v0 = c*v0;
end
```

To achieve the bouncing affect, the initial velocity is repeatedly multiplied by $c = 0.75$ until it is less than 1. Each bounce starts with a velocity equal to 3/4 of the previous one.

Within the outer loop, the statements

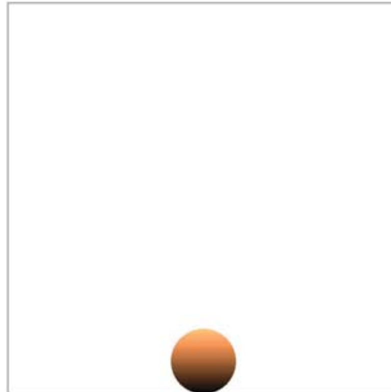


**Figure 17.1.** *Initial, and final, position of a bouncing ball. To see what happens in between, run bouncer.*

```
v = v0;
z = z0;
```

initialize the velocity `v` to `v0` and the height `z` to `z0`. Then the inner loop

```
while all(z >= 0)
   set(h,'zdata',z)
   drawnow
   v = v - delta*g;
   z = z + delta*v;
end
```

proceeds until the height goes negative. The plot is repeatedly updated to reflect the current height. At each step, the velocity `v` is decreased by a constant amount, `delta*g`, thereby affecting the gravitational deceleration. This velocity is then used to compute the change in the height `z`. As long as `v` is positive, the `z` increases with each step. When `v` reaches zero, the ball has reached its maximum height. Then `v` becomes negative and `z` decreases until the ball returns to height zero, terminating the inner loop.

After both loops are complete, the statement

```
finalize_bouncer
```

activates a pushbutton that offers you the possibility of repeating the simulation.

Brownian motion is not as obvious as gravity in our daily lives, but we do encounter it frequently. Albert Einstein's first important scientific paper was about Brownian motion. Think of particples of dust suspended in the air and illuminated

by a beam of sunlight. Or, diffusion of odors throughout a room. Or, a beach ball being tossed around a stadium by the spectators.

In Brownian motion an object – a dust particle, a molecule, or a ball – reacts to surrounding random forces. Our simulation of these forces uses the built-in MATLAB function `randn` to generate normally distributed random numbers. Each time the statement

```
randn
```

is executed a new, unpredictable, value is produced. The statement

```
randn(m,n)
```

produces an $m$-by-$n$ array of random values. Each time the statement

```
hist(randn(100000,1),60)
```

is executed a histogram plot like the one in figure 17.2 is produced. Try executing this statement several times. You will see that different histograms are produced each time, but they all have the same shape. You might recognize the "bell-shaped curve" that is known more formally as the Gaussian or normal distribution. The histogram shows that positive and negative random numbers are equally likely and that small values are more likely than large ones. This distribution is the mathematical heart of Brownian motion.

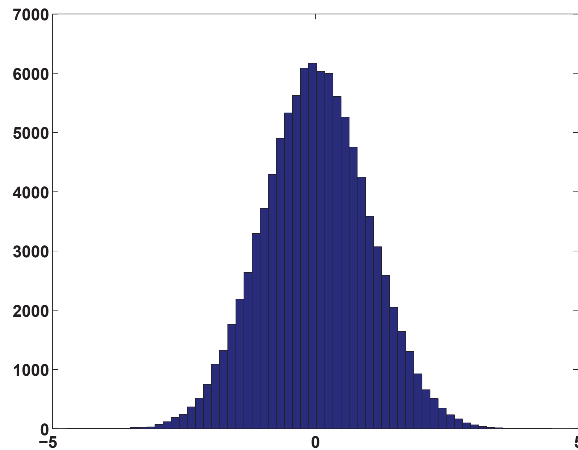



**Figure 17.2.** *Histogram of the normal random number generator.*

A simple example of Brownian motion known as a random walk is shown in figure 17.3. This is produced by the following code fragment.

```
m = 100;
x = cumsum(randn(m,1));
y = cumsum(randn(m,1));
```

```
plot(x,y,'.-')
s = 2*sqrt(m);
axis([-s s -s s]);
```

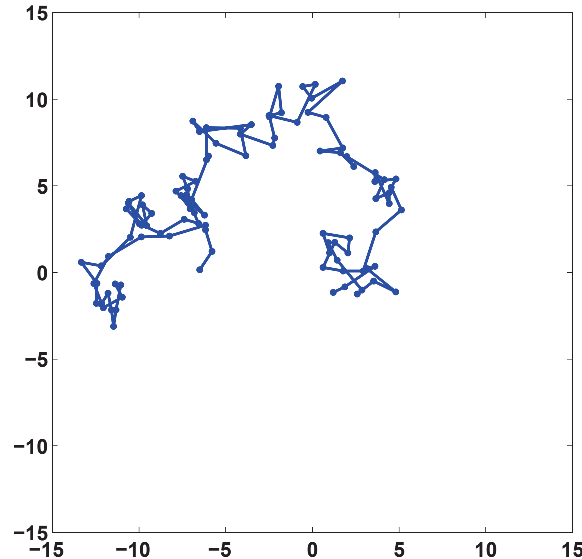



**Figure 17.3.** *A simple example of Brownian motion.*

The key statement is

```
x = cumsum(randn(m,1));
```

This statement generates the $x$-coordinates of the walk by forming the successive cumulative partial sums of the elements of the vector `r = randn(m,1)`.

$$
\begin{aligned}
x_1 &= r_1 \\
x_2 &= r_1 + r_2 \\
x_3 &= r_1 + r_2 + r_3 \\
&\ldots
\end{aligned}
$$

A similar statement generates the $y$-coordinates. Cut and paste the code fragment into the MATLAB command window. Execute it several times. Try different values of `m`. You will see different random walks going off in different random directions. Over many executions, the values of `x` and `y` are just as likely to be positive as negative. We want to compute an axis scale factor `s` so that most, but not all, of the walks stay within the plot boundaries. It turns out that as $m$, the length of the walk, increases, the proper scale factor increases like $\sqrt{m}$.

A fancier Brownian motion program, involving simultaneous random walks of many particles in three dimensions, is available in `brownian3.m`. A snapshot of the evolving motion is shown in figure 17.4. Here is the core of `brownian3.m`.

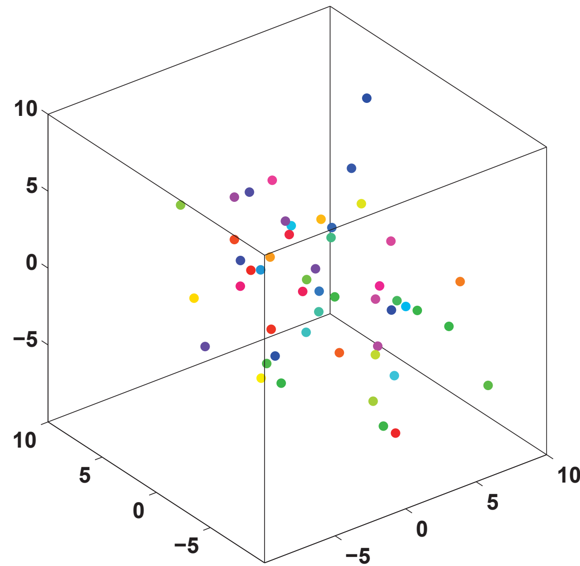



**Figure 17.4.** *A snapshot of the output from brownian3, showing simultaneous random walks of many particules in three dimensions.*

```
n = 50;  % Default number of particles
P = zeros(n,3);
H = initialize_graphics(P);

while ~get(H.stop,'value')

   % Obtain step size from slider.
   delta = get(H.speed,'value');

   % Normally distributed random velocities.
   V = randn(n,3);

   % Update positions.
   P = P + delta*V;

   update_plot(P,H);

end
```

The variable `n` is the number of particles. It is usually equal to 50, but some other number is possible with `brownian3(n)`. The array `P` contains the positions of `n` particles in three dimensions. Initially, all the particles are located at the origin, $(0, 0, 0)$. The variable `H` is a MATLAB structure containing handles for all the user

interface controls. In particular, `H.stop` refers to a toggle that terminates the `while` loop and `H.speed` refers to a slider that controls the speed through the value of the time step `delta`. The array `V` is an `n`-by-3 array of normally distributed random numbers that serve as the particle velocities in the random walks. Most of the complexity of `brownian3` is contained in the subfunction `initialize_graphics`. In addition to the speed slider and the stop button, the GUI has pushbuttons or toggles to turn on a trace, zoom in and out, and change the view point.

We are now ready to tackle the *n-body problem* in celestial mechanics. This is a model of a system of planets and their interaction described by Newton's laws of motion and gravitational attraction. Over five hundred years ago, Johannes Kepler realized that if there are only two planets in the model, the orbits are ellipses with a common focus at the center of mass of the system. This provides a fair description of the moon's orbit around the earth, or of the earth's orbit around the sun. But if you are planning a trip to the moon or a mission to Mars, you need more accuracy. You have to realize that the sun affects the moon's orbit around the earth and that Jupiter affects the orbits of both the earth and Mars. Furthermore, if you wish to model more than two planets, an analytic solution to the equations of motion is not possible. It is necessary to compute numerical approximations.

Our notation uses vectors and arrays. Let $n$ be the number of bodies and, for $i = 1, \ldots, n$, let $p_i$ be the vector denoting the position of the $i$-th body. For two-dimensional motion the $i$-th position vector has components $(x_i, y_i)$. For three-dimensional motion its components are $(x_i, y_i, z_i)$. The small system shown in figure 17.5 illustrates this notation. There are three bodies moving in two dimensions. The coordinate system and units are chosen so that initially the first body, which is gold if you have color, is at the origin,

$$p_1 = (0, 0)$$

The second body, which is blue, is one unit away from the first body in the $x$ direction, so

$$p_2 = (1, 0)$$

The third body, which is red, is one unit away from the first body in the $y$ direction, so

$$p_3 = (0, 1)$$

We wish to model how the position vectors $p_i$ vary with time, $t$. The *velocity* of a body is the rate of change of its position and the *acceleration* is the rate of change of its velocity. We use one and two dots over $p_i$ to denote the velocity and acceleration vectors, $\dot{p}_i$ and $\ddot{p}_i$. If you are familiar with calculus, you realize that the dot means differentiation with respect to $t$. For our three body example, the first body is initially heading away from the other two bodies, so its velocity vector has two negative components,

$$\dot{p}_1 = (-0.12, -0.36)$$

The initial velocity of the second body is all in the $y$ direction,

$$\dot{p}_2 = (0, 0.72)$$

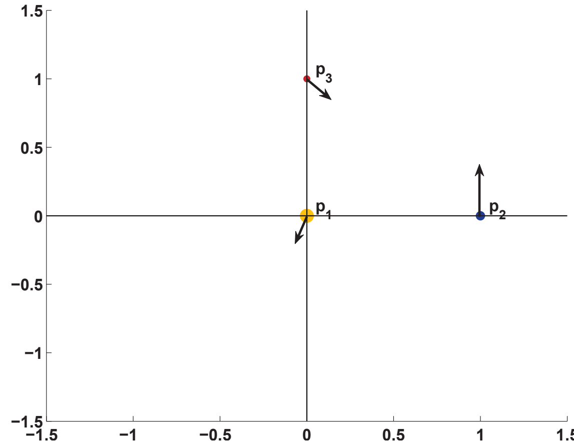



**Figure 17.5.** *Initial positions and velocities of a small system with three bodies in two-dimensional space.*

and the initial velocity of the third body is sending it towards the second body,

$$\dot{p}_3 = (0.36, -0.36)$$

Newton's law of motion, the famous $F = ma$, says that the mass of a body times its acceleration is proportional to the sum of the forces acting on it. Newton's law of gravitational says that the force between any two bodies is proportional to the product of their masses and inversely proportional to the square of the distance between them. So, the equations of motion are

$$m_i \ddot{p}_i = \gamma \sum_{j \neq i} m_i m_j \frac{p_j - p_i}{||p_j - p_i||^3}, \quad i = 1, \ldots, n$$

Here $\gamma$ is the gravitational constant, $m_i$ is the mass of the $i$-th body, $p_j - p_i$ is the vector from body $i$ to body $j$ and $||p_j - p_i||$ is the length or *norm* of that vector, which is the distance between the two bodies. The denominator of the fraction involves the cube of the distance because the numerator contains the distance itself and so the resulting quotient involves the inverse of the square of the distance.

Figure 17.6 shows our three body example again. The length of the vector $r_{23} = p_3 - p_2$ is the distance between $p_2$ and $p_3$. The gravitation forces between the bodies located at $p_2$ and $p_3$ are directed along $r_{23}$ and $-r_{23}$.

To summarize, the position of the $i$-th body is denoted by the vector $p_i$. The instantaneous change in position of this body is given by its velocity vector, denoted by $\dot{p}_i$. The instantaneous change in the velocity is given by its acceleration vector, denoted by $\ddot{p}_i$. The acceleration is determined from the position and masses of all the bodies by Newton's laws of motion and gravitation.

The following notation simplifies the discussion of numerical methods. Stack the position vectors on top of each other to produce an $n$-by-d array where $n$ is the

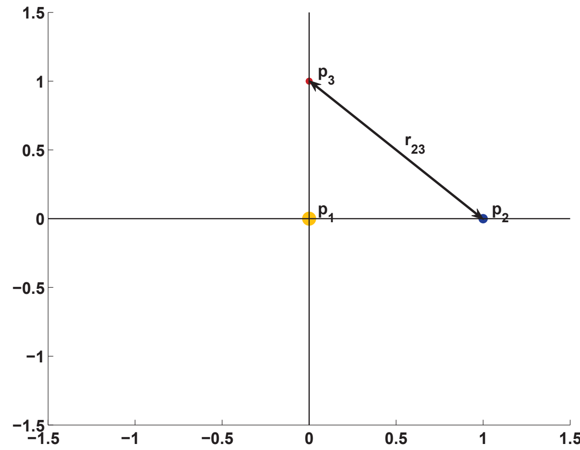



**Figure 17.6.** *The double arrow depicts the vectors* $r_{23} = p_3 - p_2$ *and* $-r_{32}$. *The length of this arrow is the distance between* $p_2$ *and* $p_3$.

number of bodies and $d = 2$ or 3 is the number of spatial dimensions..

$$P = \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{pmatrix}$$

Let $V$ denote a similar array of velocity vectors.

$$V = \begin{pmatrix} \dot{p}_1 \\ \dot{p}_2 \\ \vdots \\ \dot{p}_n \end{pmatrix}$$

And, let $G(P)$ denote the array of gravitation forces.

$$G(P) = \begin{pmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{pmatrix}$$

where

$$g_i = \gamma \sum_{j \neq i} m_j \frac{p_j - p_i}{||p_j - p_i||^3}$$

With this notation, the equations of motion can be written

$$\begin{aligned} \dot{P} &= V \\ \dot{V} &= G(P) \end{aligned}$$

For our three body example, the initial values of $P$ and $V$ are

$$P = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$$

and

$$V = \begin{pmatrix} -0.12 & -0.36 \\ 0 & 0.72 \\ 0.36 & -0.36 \end{pmatrix}$$

The masses in our three body example are

$$m_1 = 1/2, \quad m_2 = 1/3, \quad m_3 = 1/6$$

From these quantities, we can compute the initial value of the gravitation forces, $G(P)$.

We will illustrate our numerical methods by trying to generate a circle. The differential equations are

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

With initial conditions $x(0) = 0, y(0) = 1$, the exact solution is

$$x(t) = \sin t, \; y(t) = \cos t$$

The orbit is a perfect circle with a period equal to $2\pi$.

The most elementary numerical method, which we will not actually use, is known as the *forward* or *explicit Euler* method. The method uses a fixed time step $\delta$ and simultaneously advances both the positions and velocities from time $t_k$ to time $t_{k+1} = t_k + \delta$.

$$\begin{aligned} P_{k+1} &= P_k + \delta\, V_k \\ V_{k+1} &= V_k + \delta\, G(P_k) \end{aligned}$$

The forward Euler's method applied to the circle generator problem becomes

$$\begin{aligned} x_{k+1} &= x_k + \delta\, y_k \\ y_{k+1} &= y_k - \delta\, x_k \end{aligned}$$

The result for $\delta = 2\pi/30$ is shown in the first plot in figure 17.7. Instead of a circle we get a growing spiral. The method is unstable and consequently unsatisfactory, particularly for long time periods. Smaller time steps merely delay the inevitable. We would see more complicated, but similar, behavior with the n-body equations.

Another elementary numerical method is known as the *backward* or *implicit Euler* method. In general, it involves somehow solving a nonlinear system at each step.

$$\begin{aligned} P_{k+1} - \delta\, V_{k+1} &= P_k \\ V_{k+1} - \delta\, G(P_{k+1}) &= V_k \end{aligned}$$

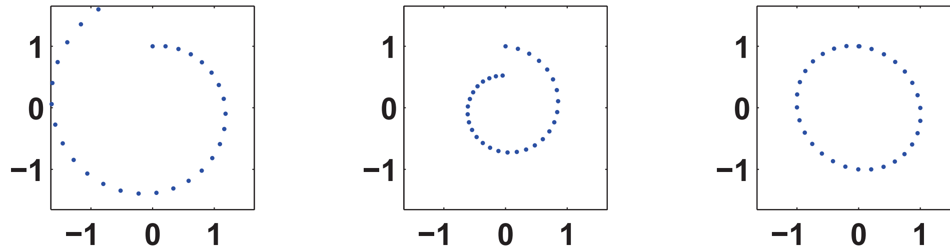



**Figure 17.7.** *Three versions of Euler's method for generating a circle. The first plot shows that the forward method is unstable. The second plot shows that the backward method has excessive damping. The third plot shows that symplectic method, which is a compromise between the first two methods, produces a nearly perfect circle.*

For our simple circle example the implicit system is linear, so $x_{k+1}$ and $y_{k+1}$ are easily computed by solving the 2-by-2 system

$$x_{k+1} - \delta\, y_{k+1} = x_k$$
$$y_{k+1} + \delta\, x_{k+1} = y_k$$

The result is shown in the second plot in figure 17.7. Instead of a circle we get a decaying spiral. The method is stable, but there is too much damping. Again, we would see similar behavior with the n-body equations.

The method that we actually use is a compromise between the explicit and implicit Euler methods. It is the most elementary instance of what are known as *symplectic* methods. The method involves two half-steps. In the first half-step, the positions at time $t_k$ are used in the gravitation equations to update of the velocities.

$$V_{k+1} = V_k + \delta\, G(P_k)$$

Then, in the second half-step, these "new" velocities are used to update the positions.

$$P_{k+1} = P_k + \delta\, V_{k+1}$$

The novel feature of this symplectic method is the subscript $k+1$ instead of $k$ on the $V$ term in the second half-step.

For the circle generator, the symplectic method is

$$x_{k+1} = x_k + \delta\, y_k$$
$$y_{k+1} = y_k - \delta\, x_{k+1}$$

The result is the third plot in figure 17.7. If you look carefully, you can see that the orbit in not quite a circle. It's actually a nearly circular ellipse. And the final value does not quite return to the initial value, so the period is not exactly $2\pi$ . But the important fact is that the orbit is neither a growing nor a decaying spiral.

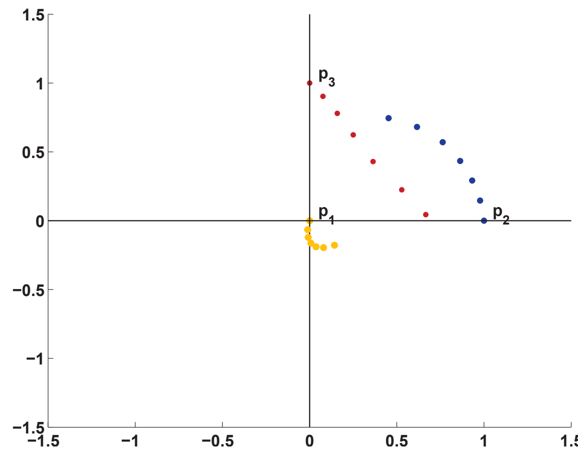



**Figure 17.8.** *The first few steps of our example system.*

There are more complicated symplectic algorithms that are much more accurate per step than this symplectic Euler. But the symplectic Euler is satisfactory for generating well behaved graphical displays. Most well-known numerical methods, including Runge-Kutta methods and traditional multistep methods, do not have this symplectic stability property and, as a result, are not as satisfactory for computing orbits over long time spans.

Figure 17.8 shows the first few steps for our example system. As we noted earlier, the initial position and velocity are

```
P =
         0         0
    1.0000         0
         0    1.0000

V =
   -0.1200   -0.3600
         0    0.7200
    0.3600   -0.3600
```

After one step with $\delta = 0.20$ we obtain the following values.

```
P =
   -0.0107   -0.0653
    0.9776    0.1464
    0.0767    0.9033

V =
   -0.0533   -0.3267
   -0.1118    0.7318
```

```
    0.3836   -0.4836
```

The three masses, 1/2, 1/3, and 1/6, are not equal, but are comparable, so all three bodies have significant affects on each other and all three move noticeable distances. We see that the initial velocity of the first body causes it to move away from the other two. In one step, its position changes from $(0, 0)$ to small negative values, $(-0.0107, -0.0653)$ The second body is initially at position $(1, 0)$ with velocity $(0, 1)$ in the positive $y$ direction. In one step, its position changes to $(0.9776, 0.1464)$. The $x$-coordinate has changed relatively little, while the $y$-coordinate has changed by roughly $0.72\,\delta$. The third body moves in the direction indicated by the velocity vector in figure 17.5.

After a second step we have the following values. As expected, all the trends noted in the first step continue.

```
P =
   -0.0079   -0.1209
    0.9325    0.2917
    0.1589    0.7793

V =
    0.0136   -0.2779
   -0.2259    0.7268
    0.4109   -0.6198
```

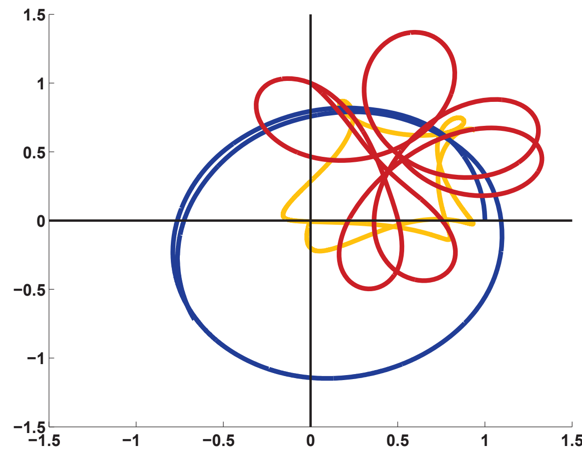



**Figure 17.9.** *The initial trajectories of our example system.*

Figure 17.9 shows an initial section of the trajectories. You should run our *Experiments* program `orbits(3)` to see the three bodies in motion. The small body and the large body orbit in a clockwise direction around each other while the medium-size body orbits in a counter-clockwise direction around the other two.

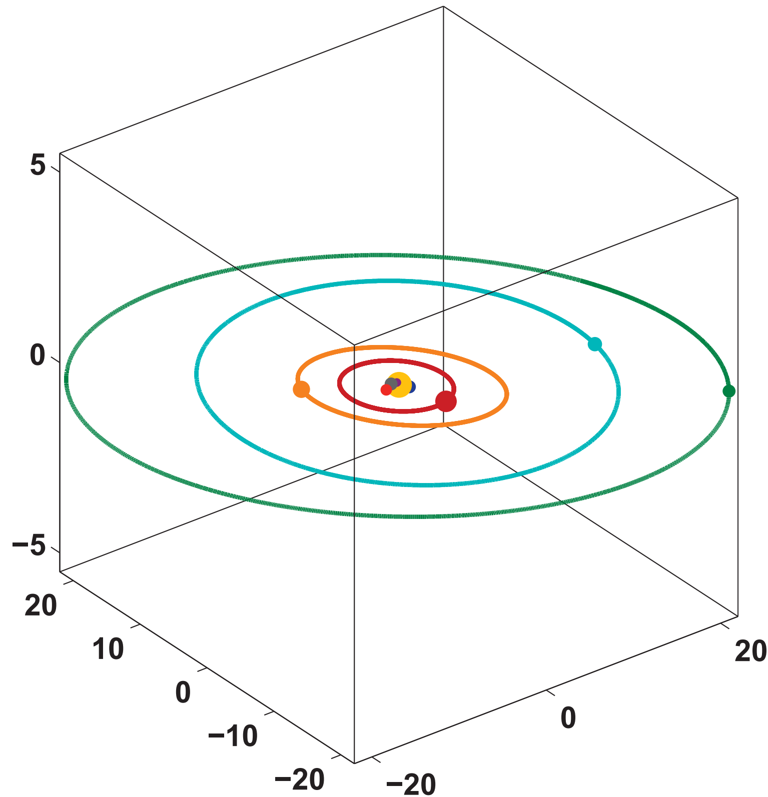



**Figure 17.10.** *The solar system, with the initial positions of all the planets and the orbits of the outer planets, Jupiter, Saturn, Uranus, and Neptune.*

Our *Experiments* program `orbits` models nine bodies in the solar system, namely the sun and eight planets. Figures 17.10 and 17.11 show snapshots of the output from `orbits` with two different zoom factors that are necessary to span the scale of the system. The orbits for all the planets are in the proper proportion. But, obviously, the symbols for the sun and the planets do not have the same scale. Web sources for information about the solar system are provided by the University Corporation for Atmospheric Research, the Jet Propulsion Laboratory, and the US National Air and Space Museum,

```
http://www.windows.ucar.edu
http://www.jpl.nasa.gov/solar_system
http://airandspace.si.edu:80/etp/ss/index.htm
```

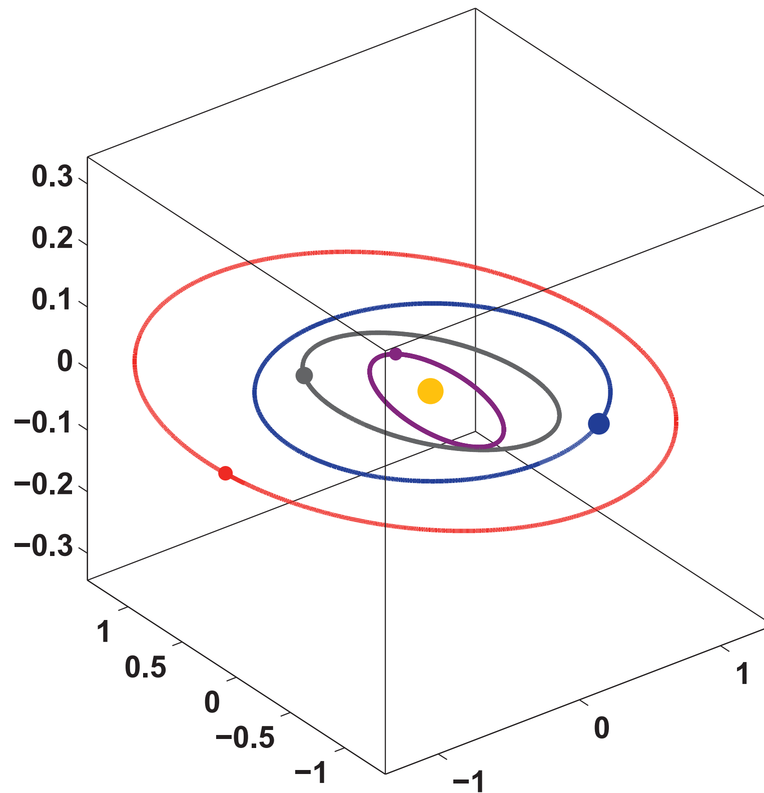



**Figure 17.11.** *Zooming in by a factor of 16 reveals the orbits of the inner planets, Mercury, Venus, Earth and Mars.*

## Recap

```
%% Orbits Chapter Recap
% This is an executable program that illustrates the statements
% introduced in the Orbits Chapter of "Experiments in MATLAB".
% You can access it with
%
%    orbits_recap
%    edit orbits_recap
%    publish orbits_recap
%
% Related EXM programs
%
%    bouncer
%    orbits
```

```
%% Core of bouncer, simple gravity. no gravity

   % Initialize

   z0 = eps;
   g = 9.8;
   c = 0.75;
   delta = 0.005;
   v0 = 21;
   y = [];

   % Bounce

   while v0 >= 1
      v = v0;
      z = z0;
      while z >= 0
         v = v - delta*g;
         z = z + delta*v;
         y = [y  z];
      end
      v0 = c*v0;
   end

   % Simplified graphics

   close all
   figure
   plot(y)

%% Normal random number generator.

   figure
   hist(randn(100000,1),60)


%% Snapshot of two dimensional Brownian motion.

   figure
   m = 100;
   x = cumsum(randn(m,1));
   y = cumsum(randn(m,1));
   plot(x,y,'.-')
   s = 2*sqrt(m);
   axis([-s s -s s]);
```

```
%% Snapshot of three dimensional Brownian motion, brownian3

   n = 50;
   delta = 0.125;
   P = zeros(n,3);

   for t = 0:10000
      % Normally distributed random velocities.
      V = randn(n,3);
      % Update positions.
      P = P + delta*V;
   end

   figure
   plot3(P(:,1),P(:,2),P(:,3),'.')
   box on

%% Orbits, the n-body problem.

%{
% ORBITS  n-body gravitational attraction for n = 2, 3 or 9.
%   ORBITS(2), two bodies, classical elliptic orbits.
%   ORBITS(3), three bodies, artificial planar orbits.
%   ORBITS(9), nine bodies, the solar system with one sun and 8 planets.
%
%   ORBITS(n,false) turns off the uicontrols and generates a static plot.
%   ORBITS with no arguments is the same as ORBITS(9,true).

   % n = number of bodies.
   % P = n-by-3 array of position coordinates.
   % V = n-by-3 array of velocities
   % M = n-by-1 array of masses
   % H = graphics and user interface handles

   if (nargin < 2)
      gui = true;
   end
   if (nargin < 1);
      n = 9;
   end

   [P,V,M] = initialize_orbits(n);
   H = initialize_graphics(P,gui);

   steps = 20;     % Number of steps between plots
```

```
   t = 0;          % time

   while get(H.stop,'value') == 0

      % Obtain step size from slider.
      delta = get(H.speed,'value')/(20*steps);

      for k = 1:steps

         % Compute current gravitational forces.
         G = zeros(size(P));
         for i = 1:n
            for j = [1:i-1 i+1:n];
               r = P(j,:) - P(i,:);
               G(i,:) = G(i,:) + M(j)*r/norm(r)^3;
            end
         end

         % Update velocities using current gravitational forces.
         V = V + delta*G;

         % Update positions using updated velocities.
         P = P + delta*V;

      end

      t = t + steps*delta;
      H = update_plot(P,H,t,gui);
   end

   finalize_graphics(H,gui)
end
%}

%% Run all three orbits, with 2, 3, and 9 bodies, and no gui.

   figure
   orbits(2,false)

   figure
   orbits(3,false)

   figure
   orbits(9,false)
```

## Exercises

17.1 *Bouncing ball.*
(a) What is the maximum height of the bouncing ball?
(b) How many times does the ball bounce?
(c) What is the effect of changing each of the four `bouncer` values `g`, `c`, `delta`, and `v0`.

17.2 *Pluto and Ceres.* Change `orbits` to `orbits11` by adding the erstwhile planet Pluto and the recently promoted dwarf planet Ceres. See Wikipedia:

```
http://en.wikipedia.org/wiki/Planet
http://en.wikipedia.org/wiki/Ceres_(dwarf_planet)
```

and

```
http://orbitsimulator.com/gravity/articles/ceres.html
```

17.3 *Comet.* Add a comet to `orbits`. Find initial conditions so that the comet has a stable, but highly elliptical orbit that extends well beyond the orbits of the planets.

17.4 *Twin suns.* Turn the sun in `orbits` into a twin star system, with two suns orbiting each other out of the plane of the planets. What eventually happens to the planetary orbits? For example, try

```
sun1.p = [1 0 0];
sun1.v = [0 0.25 0.25];
sun1.m = 0.5;
sun2.p = [-1 0 0];
sun2.v = [0 -0.25 -0.25];
sun2.m = 0.5;
```

Try other values as well.

## How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

*Symbolic Math Toolbox™ User's Guide*

© COPYRIGHT 1993–2015 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

#### Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

#### Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

1

# Getting Started


- "Symbolic Math Toolbox Product Description" on page 1-2
- "Access Symbolic Math Toolbox Functionality" on page 1-3
- "Create Symbolic Numbers, Variables, and Expressions" on page 1-4
- "Create Symbolic Functions" on page 1-8
- "Create Symbolic Matrices" on page 1-9
- "Perform Symbolic Computations" on page 1-12
- "Use Assumptions on Symbolic Variables" on page 1-27

# Symbolic Math Toolbox Product Description

**Perform symbolic math computations**

Symbolic Math Toolbox provides functions for solving and manipulating symbolic math expressions and performing variable-precision arithmetic. You can analytically perform differentiation, integration, simplification, transforms, and equation solving. You can also generate code for MATLAB, Simulink®, and Simscape™ from symbolic math expressions.

Symbolic Math Toolbox includes the MuPAD language, which is optimized for handling and operating on symbolic math expressions. It provides libraries of MuPAD functions in common mathematical areas, such as calculus and linear algebra, and in specialized areas, such as number theory and combinatorics. You can also write custom symbolic functions and libraries in the MuPAD language. The MuPAD Notebook app lets you document symbolic math derivations with embedded text, graphics, and typeset math. You can share the annotated derivations as HTML or as a PDF.

## Key Features

- Symbolic integration, differentiation, transforms, and linear algebra
- Algebraic and ordinary differential equation (ODE) solvers
- Simplification and manipulation of symbolic expressions
- Code generation from symbolic expressions for MATLAB, Simulink, Simscape, C, Fortran, MathML, and TeX
- Variable-precision arithmetic
- MuPAD Notebook for performing and documenting symbolic calculations
- MuPAD language and function libraries for combinatorics, number theory, and other mathematical areas

# Access Symbolic Math Toolbox Functionality

| In this section... |
| --- |
| "Work from MATLAB" on page 1-3 |
| "Work from MuPAD" on page 1-3 |

## Work from MATLAB

You can access the Symbolic Math Toolbox functionality directly from the MATLAB Command Window. This environment lets you call functions using familiar MATLAB syntax.

## Work from MuPAD

You can access the Symbolic Math Toolbox functionality from the MuPAD Notebook app using the MuPAD language. The MuPAD Notebook app includes a symbol palette for accessing common MuPAD functions. All results are displayed in typeset math. You also can convert the results into MathML and TeX. You can embed graphics, animations, and descriptive text within your notebook.

A debugger and other programming utilities provide tools for authoring custom symbolic functions and libraries in the MuPAD language. The MuPAD language supports multiple programming styles including imperative, functional, and object-oriented programming. The language treats variables as symbolic by default and is optimized for handling and operating on symbolic math expressions. You can call functions written in the MuPAD language from the MATLAB Command Window. For more information, see "Call Built-In MuPAD Functions from MATLAB" on page 3-31

If you are a new user of the MuPAD Notebook app, see Getting Started with MuPAD.

# Create Symbolic Numbers, Variables, and Expressions

This page shows how to create symbolic numbers, variables, and expressions. To learn how to work with symbolic math, see “Perform Symbolic Computations” on page 1-12.

## Create Symbolic Numbers

You can create symbolic numbers by using `sym`. Symbolic numbers are exact representations, unlike floating-point numbers.

Create a symbolic number by using `sym` and compare it to the same floating-point number.

```
sym(1/3)
1/3

ans =
1/3
ans =
    0.3333
```

The symbolic number is represented in exact rational form, while the floating-point number is a decimal approximation. The symbolic result is not indented, while the standard MATLAB result is indented.

Calculations on symbolic numbers are exact. Demonstrate this exactness by finding `sin(pi)` symbolically and numerically. The symbolic result is exact, while the numeric result is an approximation.

```
sin(sym(pi))
sin(pi)

ans =
0
ans =
   1.2246e-16
```

To learn more about symbolic representation of numbers, see “Numeric to Symbolic Conversion” on page 2-95.

## Create Symbolic Variables

You can use two ways to create symbolic variables, `syms` and `sym`. The `syms` syntax is a shorthand for `sym`.

Create symbolic variables `x` and `y` using `syms` and `sym` respectively.

```
syms x
y = sym('y')
```

The first command creates a symbolic variable `x` in the MATLAB workspace with the value `x` assigned to the variable `x`. The second command creates a symbolic variable `y` with value `y`. Therefore, the commands are equivalent.

With `syms`, you can create multiple variables in one command. Create the variables `a`, `b`, and `c`.

```
syms a b c
```

If you want to create many variables, the `syms` syntax is inconvenient. Instead of using `syms`, use `sym` to create many numbered variables.

Create the variables `a1, ..., a20`.

```
A = sym('a', [1 20])
```

```
A =
[ a1, a2, a3, a4, a5, a6, a7, a8, a9, a10,...
 a11, a12, a13, a14, a15, a16, a17, a18, a19, a20]
```

The `syms` command is a convenient shorthand for the `sym` syntax. Use the `sym` syntax when you create many variables, when the variable value differs from the variable name, or when you create a symbolic number, such as `sym(5)`.

## Create Symbolic Expressions

Suppose you want to use a symbolic variable to represent the golden ratio

$$\varphi = \frac{1+\sqrt{5}}{2}$$

The command

```
phi = (1 + sqrt(sym(5)))/2;
```

achieves this goal. Now you can perform various mathematical operations on `phi`. For example,

```
f = phi^2 - phi - 1
```

returns

```
f =
(5^(1/2)/2 + 1/2)^2 - 5^(1/2)/2 - 3/2
```

Now suppose you want to study the quadratic function $f = ax^2 + bx + c$. First, create the symbolic variables `a`, `b`, `c`, and `x`:

```
syms a b c x
```

Then, assign the expression to `f`:

```
f = a*x^2 + b*x + c;
```

---

**Tip** To create a symbolic number, use the `sym` command. Do not use the `syms` function to create a symbolic expression that is a constant. For example, to create the expression whose value is `5`, enter `f = sym(5)`. The command `f = 5` does *not* define `f` as a symbolic expression.

---

## Reuse Names of Symbolic Objects

If you set a variable equal to a symbolic expression, and then apply the `syms` command to the variable, MATLAB software removes the previously defined expression from the variable. For example,

```
syms a b
f = a + b
```

returns

```
f =
a + b
```

If later you enter

```
syms f
f
```

then MATLAB removes the value `a + b` from the expression `f`:

```
f =
f
```

You can use the `syms` command to clear variables of definitions that you previously assigned to them in your MATLAB session. However, `syms` does not clear the following assumptions of the variables: complex, real, integer, and positive. These assumptions are stored separately from the symbolic object. For more information, see "Delete Symbolic Objects and Their Assumptions" on page 1-28.

# Create Symbolic Functions

You also can use `sym` and `syms` to create symbolic functions. For example, you can create an arbitrary function `f(x, y)` where `x` and `y` are function variables. The simplest way to create an arbitrary symbolic function is to use `syms`:

```
syms f(x, y)
```

This syntax creates the symbolic function `f` and symbolic variables `x` and `y`. If instead of an arbitrary symbolic function you want to create a function defined by a particular mathematical expression, use this two-step approach. First, create symbolic variables representing the arguments of the function:

```
syms x y
```

Then assign a mathematical expression to the function. In this case, the assignment operation also creates the new symbolic function:

```
f(x, y) = x^3*y^3
```

```
f(x, y) =
x^3*y^3
```

Note that the body of the function must be a symbolic number, variable, or expression. Assigning a number, such as `f(x,y) = 1`, causes an error.

After creating a symbolic function, you can differentiate, integrate, or simplify it, substitute its arguments with values, and perform other mathematical operations. For example, find the second derivative on `f(x, y)` with respect to variable `y`. The result `d2fy` is also a symbolic function.

```
d2fy = diff(f, y, 2)
```

```
d2fy(x, y) =
6*x^3*y
```

Now evaluate `f(x, y)` for `x = y + 1`:

```
f(y + 1, y)
```

```
ans =
y^3*(y + 1)^3
```

# Create Symbolic Matrices

| In this section... |
|---|
| "Use Existing Symbolic Variables" on page 1-9 |
| "Generate Elements While Creating a Matrix" on page 1-10 |
| "Create Matrix of Symbolic Numbers" on page 1-10 |

## Use Existing Symbolic Variables

A circulant matrix has the property that each row is obtained from the previous one by cyclically permuting the entries one step forward. For example, create the symbolic circulant matrix whose elements are `a`, `b`, and `c`, using the commands:

```
syms a b c
A = [a b c; c a b; b c a]
```

```
A =
[ a, b, c]
[ c, a, b]
[ b, c, a]
```

Since matrix `A` is circulant, the sum of elements over each row and each column is the same. Find the sum of all the elements of the first row:

```
sum(A(1,:))
```

```
ans =
a + b + c
```

To check if the sum of the elements of the first row equals the sum of the elements of the second column, use the `isAlways` function:

```
isAlways(sum(A(1,:)) == sum(A(:,2)))
```

The sums are equal:

```
ans =
     1
```

From this example, you can see that using symbolic objects is very similar to using regular MATLAB numeric objects.

## Generate Elements While Creating a Matrix

The `sym` function also lets you define a symbolic matrix or vector without having to define its elements in advance. In this case, the `sym` function generates the elements of a symbolic matrix at the same time that it creates a matrix. The function presents all generated elements using the same form: the base (which must be a valid variable name), a row index, and a column index. Use the first argument of `sym` to specify the base for the names of generated elements. You can use any valid variable name as a base. To check whether the name is a valid variable name, use the `isvarname` function. By default, `sym` separates a row index and a column index by underscore. For example, create the 2-by-4 matrix `A` with the elements `A1_1, ..., A2_4`:

```
A = sym('A', [2 4])
```

```
A =
[ A1_1, A1_2, A1_3, A1_4]
[ A2_1, A2_2, A2_3, A2_4]
```

To control the format of the generated names of matrix elements, use `%d` in the first argument:

```
A = sym('A%d%d', [2 4])
```

```
A =
[ A11, A12, A13, A14]
[ A21, A22, A23, A24]
```

## Create Matrix of Symbolic Numbers

A particularly effective use of `sym` is to convert a matrix from numeric to symbolic form. The command

```
A = hilb(3)
```

generates the 3-by-3 Hilbert matrix:

```
A =
    1.0000    0.5000    0.3333
    0.5000    0.3333    0.2500
    0.3333    0.2500    0.2000
```

By applying `sym` to `A`

```
A = sym(A)
```

you can obtain the precise symbolic form of the 3-by-3 Hilbert matrix:

```
A =
[    1, 1/2, 1/3]
[ 1/2, 1/3, 1/4]
[ 1/3, 1/4, 1/5]
```

For more information on numeric to symbolic conversions, see “Numeric to Symbolic Conversion” on page 2-95.

# Perform Symbolic Computations


| In this section... |
| --- |
| "Differentiate Symbolic Expressions" on page 1-12 |
| "Integrate Symbolic Expressions" on page 1-13 |
| "Solve Equations" on page 1-15 |
| "Simplify Symbolic Expressions" on page 1-17 |
| "Substitutions in Symbolic Expressions" on page 1-18 |
| "Plot Symbolic Functions" on page 1-21 |


## Differentiate Symbolic Expressions

With the Symbolic Math Toolbox software, you can find

- Derivatives of single-variable expressions
- Partial derivatives
- Second and higher order derivatives
- Mixed derivatives

For in-depth information on taking symbolic derivatives see "Differentiation" on page 2-6.

### Expressions with One Variable

To differentiate a symbolic expression, use the `diff` command. The following example illustrates how to take a first derivative of a symbolic expression:

```
syms x
f = sin(x)^2;
diff(f)
```

```
ans =
2*cos(x)*sin(x)
```

### Partial Derivatives

For multivariable expressions, you can specify the differentiation variable. If you do not specify any variable, MATLAB chooses a default variable by its proximity to the letter `x`:

```
syms x y
f = sin(x)^2 + cos(y)^2;
diff(f)
```

```
ans =
2*cos(x)*sin(x)
```

For the complete set of rules MATLAB applies for choosing a default variable, see "Find a Default Symbolic Variable" on page 2-5.

To differentiate the symbolic expression `f` with respect to a variable `y`, enter:

```
syms x y
f = sin(x)^2 + cos(y)^2;
diff(f, y)
```

```
ans =
-2*cos(y)*sin(y)
```

### Second Partial and Mixed Derivatives

To take a second derivative of the symbolic expression `f` with respect to a variable `y`, enter:

```
syms x y
f = sin(x)^2 + cos(y)^2;
diff(f, y, 2)
```

```
ans =
2*sin(y)^2 - 2*cos(y)^2
```

You get the same result by taking derivative twice: `diff(diff(f, y))`. To take mixed derivatives, use two differentiation commands. For example:

```
syms x y
f = sin(x)^2 + cos(y)^2;
diff(diff(f, y), x)
```

```
ans =
0
```

## Integrate Symbolic Expressions

You can perform symbolic integration including:

- Indefinite and definite integration

- Integration of multivariable expressions

For in-depth information on the `int` command including integration with real and complex parameters, see “Integration” on page 2-23.

### Indefinite Integrals of One-Variable Expressions

Suppose you want to integrate a symbolic expression. The first step is to create the symbolic expression:

```
syms x
f = sin(x)^2;
```

To find the indefinite integral, enter

```
int(f)
```

```
ans =
x/2 - sin(2*x)/4
```

### Indefinite Integrals of Multivariable Expressions

If the expression depends on multiple symbolic variables, you can designate a variable of integration. If you do not specify any variable, MATLAB chooses a default variable by the proximity to the letter `x`:

```
syms x y n
f = x^n + y^n;
int(f)
```

```
ans =
x*y^n + (x*x^n)/(n + 1)
```

For the complete set of rules MATLAB applies for choosing a default variable, see “Find a Default Symbolic Variable” on page 2-5.

You also can integrate the expression `f = x^n + y^n` with respect to `y`

```
syms x y n
f = x^n + y^n;
int(f, y)
```

```
ans =
x^n*y + (y*y^n)/(n + 1)
```

If the integration variable is `n`, enter

```
syms x y n
f = x^n + y^n;
int(f, n)
```

```
ans =
x^n/log(x) + y^n/log(y)
```

#### Definite Integrals

To find a definite integral, pass the limits of integration as the final two arguments of the `int` function:

```
syms x y n
f = x^n + y^n;
int(f, 1, 10)
```

```
ans =
piecewise([n == -1, log(10) + 9/y],...
          [n ~= -1, (10*10^n - 1)/(n + 1) + 9*y^n])
```

#### If MATLAB Cannot Find a Closed Form of an Integral

If the `int` function cannot compute an integral, it returns an unresolved integral:

```
syms x
int(sin(sinh(x)))
```

```
ans =
int(sin(sinh(x)), x)
```

## Solve Equations

You can solve different types of symbolic equations including:

- Algebraic equations with one symbolic variable
- Algebraic equations with several symbolic variables
- Systems of algebraic equations

For in-depth information on solving symbolic equations including differential equations, see “Equation Solving”.

#### Solve Algebraic Equations with One Symbolic Variable

Use the double equal sign (==) to define an equation. Then you can `solve` the equation by calling the solve function. For example, solve this equation:

```
syms x
solve(x^3 - 6*x^2 == 6 - 11*x)

ans =
 1
 2
 3
```

If you do not specify the right side of the equation, `solve` assumes that it is zero:

```
syms x
solve(x^3 - 6*x^2 + 11*x - 6)

ans =
 1
 2
 3
```

### Solve Algebraic Equations with Several Symbolic Variables

If an equation contains several symbolic variables, you can specify a variable for which this equation should be solved. For example, solve this multivariable equation with respect to `y`:

```
syms x y
solve(6*x^2 - 6*x^2*y + x*y^2 - x*y + y^3 - y^2 == 0, y)

ans =
    1
  2*x
 -3*x
```

If you do not specify any variable, you get the solution of an equation for the alphabetically closest to `x` variable. For the complete set of rules MATLAB applies for choosing a default variable see “Find a Default Symbolic Variable” on page 2-5.

### Solve Systems of Algebraic Equations

You also can solve systems of equations. For example:

```
syms x y z
[x, y, z] = solve(z == 4*x, x == y, z == x^2 + y^2)

x =
 0
```

```
 2

y =
 0
 2

z =
 0
 8
```

## Simplify Symbolic Expressions

Symbolic Math Toolbox provides a set of simplification functions allowing you to manipulate the output of a symbolic expression. For example, the following polynomial of the golden ratio `phi`

```
phi = sym('(1 + sqrt(5))/2');
f = phi^2 - phi - 1
```

returns

```
f =
(5^(1/2)/2 + 1/2)^2 - 5^(1/2)/2 - 3/2
```

You can simplify this answer by entering

```
simplify(f)
```

and get a very short answer:

```
ans =
0
```

Symbolic simplification is not always so straightforward. There is no universal simplification function, because the meaning of a simplest representation of a symbolic expression cannot be defined clearly. Different problems require different forms of the same mathematical expression. Knowing what form is more effective for solving your particular problem, you can choose the appropriate simplification function.

For example, to show the order of a polynomial or symbolically differentiate or integrate a polynomial, use the standard polynomial form with all the parentheses multiplied out and all the similar terms summed up. To rewrite a polynomial in the standard form, use the `expand` function:

```
syms x
f = (x ^2- 1)*(x^4 + x^3 + x^2 + x + 1)*(x^4 - x^3 + x^2 - x + 1);
expand(f)
```

```
ans =
x^10 - 1
```

The `factor` simplification function shows the polynomial roots. If a polynomial cannot be factored over the rational numbers, the output of the `factor` function is the standard polynomial form. For example, to factor the third-order polynomial, enter:

```
syms x
g = x^3 + 6*x^2 + 11*x + 6;
factor(g)
```

```
ans =
[ x + 3, x + 2, x + 1]
```

The nested (Horner) representation of a polynomial is the most efficient for numerical evaluations:

```
syms x
h = x^5 + x^4 + x^3 + x^2 + x;
horner(h)
```

```
ans =
x*(x*(x*(x*(x + 1) + 1) + 1) + 1)
```

For a list of Symbolic Math Toolbox simplification functions, see "Choose Function to Rearrange Expression" on page 2-61.

## Substitutions in Symbolic Expressions

### Substitute Symbolic Variables with Numbers

You can substitute a symbolic variable with a numeric value by using the `subs` function. For example, evaluate the symbolic expression `f` at the point `x` = 1/3:

```
syms x
f = 2*x^2 - 3*x + 1;
subs(f, 1/3)
```

```
ans =
2/9
```

The subs function does not change the original expression f:

```
f

f =
2*x^2 - 3*x + 1
```

#### Substitute in Multivariate Expressions

When your expression contains more than one variable, you can specify the variable for which you want to make the substitution. For example, to substitute the value x = 3 in the symbolic expression

```
syms x y
f = x^2*y + 5*x*sqrt(y);
```

enter the command

```
subs(f, x, 3)

ans =
9*y + 15*y^(1/2)
```

#### Substitute One Symbolic Variable for Another

You also can substitute one symbolic variable for another symbolic variable. For example to replace the variable y with the variable x, enter

```
subs(f, y, x)

ans =
x^3 + 5*x^(3/2)
```

#### Substitute a Matrix into a Polynomial

You can also substitute a matrix into a symbolic polynomial with numeric coefficients. There are two ways to substitute a matrix into a polynomial: element by element and according to matrix multiplication rules.

##### Element-by-Element Substitution

To substitute a matrix at each element, use the subs command:

```
syms x
f = x^3 - 15*x^2 - 24*x + 350;
A = [1 2 3; 4 5 6];
```

```
subs(f,A)

ans =
[ 312, 250,  170]
[  78, -20, -118]
```

You can do element-by-element substitution for rectangular or square matrices.

#### Substitution in a Matrix Sense

If you want to substitute a matrix into a polynomial using standard matrix multiplication rules, a matrix must be square. For example, you can substitute the magic square A into a polynomial f:

1 Create the polynomial:

```
syms x
f = x^3 - 15*x^2 - 24*x + 350;
```

2 Create the magic square matrix:

```
A = magic(3)

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

3 Get a row vector containing the numeric coefficients of the polynomial f:

```
b = sym2poly(f)

b =
     1   -15   -24   350
```

4 Substitute the magic square matrix A into the polynomial f. Matrix A replaces all occurrences of x in the polynomial. The constant times the identity matrix eye(3) replaces the constant term of f:

```
A^3 - 15*A^2 - 24*A + 350*eye(3)

ans =
   -10     0     0
     0   -10     0
     0     0   -10
```

The polyvalm command provides an easy way to obtain the same result:

```
polyvalm(b,A)

ans =
   -10     0     0
     0   -10     0
     0     0   -10
```

### Substitute the Elements of a Symbolic Matrix

To substitute a set of elements in a symbolic matrix, also use the `subs` command. Suppose you want to replace some of the elements of a symbolic circulant matrix A

```
syms a b c
A = [a b c; c a b; b c a]

A =
[ a, b, c]
[ c, a, b]
[ b, c, a]
```

To replace the (2, 1) element of `A` with `beta` and the variable `b` throughout the matrix with variable `alpha`, enter

```
alpha = sym('alpha');
beta = sym('beta');
A(2,1) = beta;
A = subs(A,b,alpha)
```

The result is the matrix:

```
A =
[     a, alpha,     c]
[  beta,     a, alpha]
[ alpha,     c,     a]
```

For more information, see “Substitution”.

## Plot Symbolic Functions

You can create different types of graphs including:

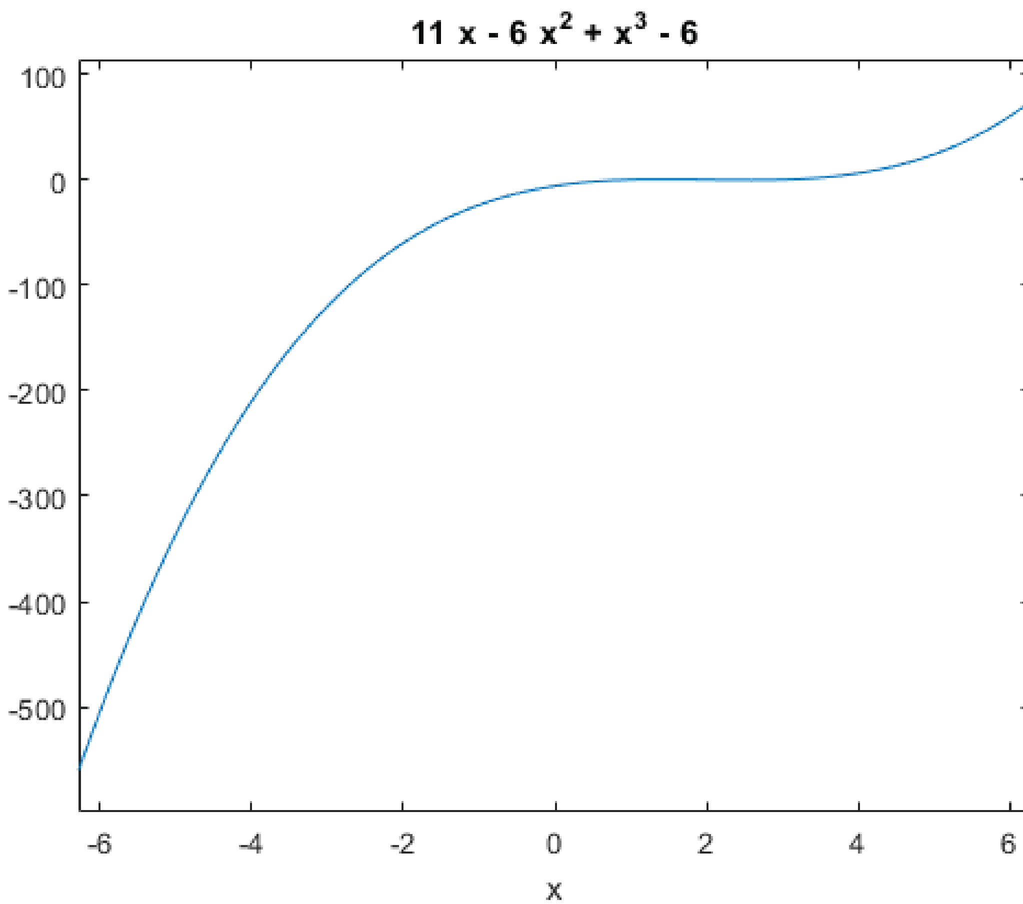
- Plots of explicit functions
- Plots of implicit functions

- 3-D parametric plots
- Surface plots

### Explicit Function Plot

The simplest way to create a plot is to use the `ezplot` command:

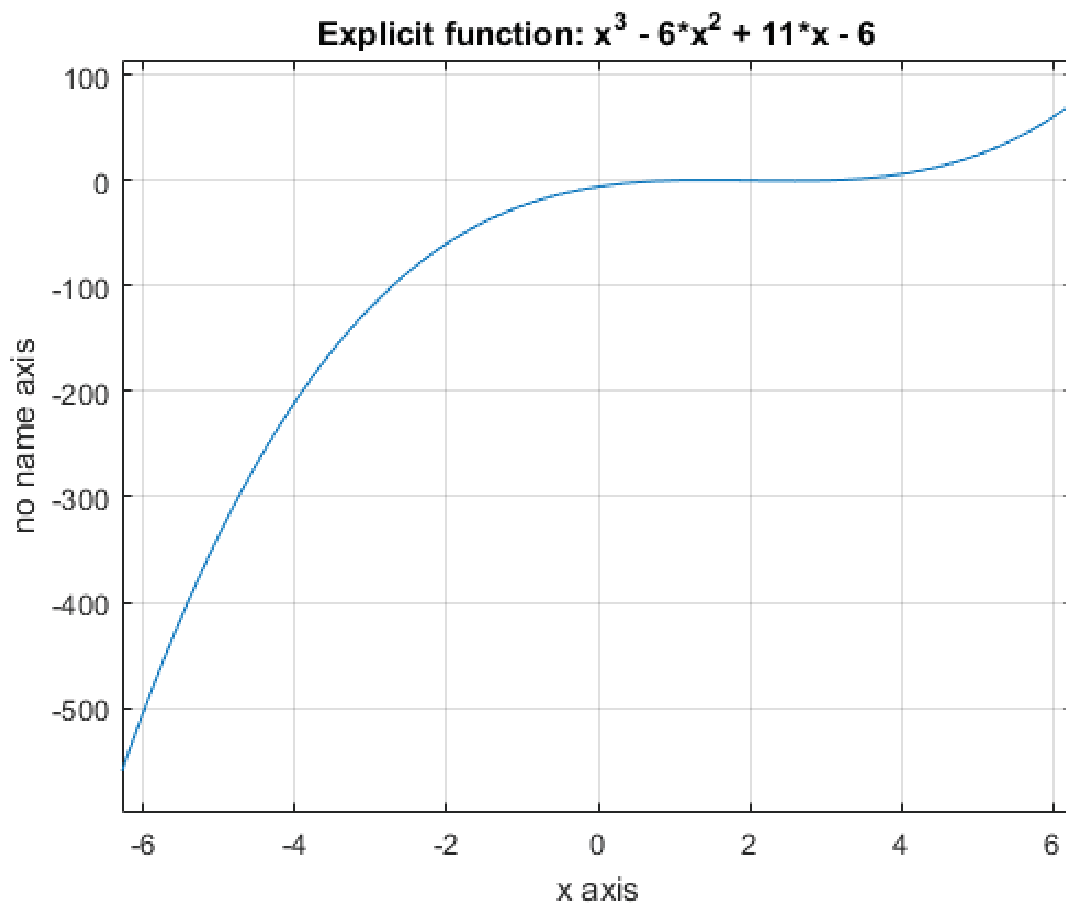
```
syms x
ezplot(x^3 - 6*x^2 + 11*x - 6)
hold on
```




The `hold on` command retains the existing plot allowing you to add new elements and change the appearance of the plot. For example, now you can change the names of the

axes and add a new title and grid lines. When you finish working with the current plot, enter the `hold off` command:

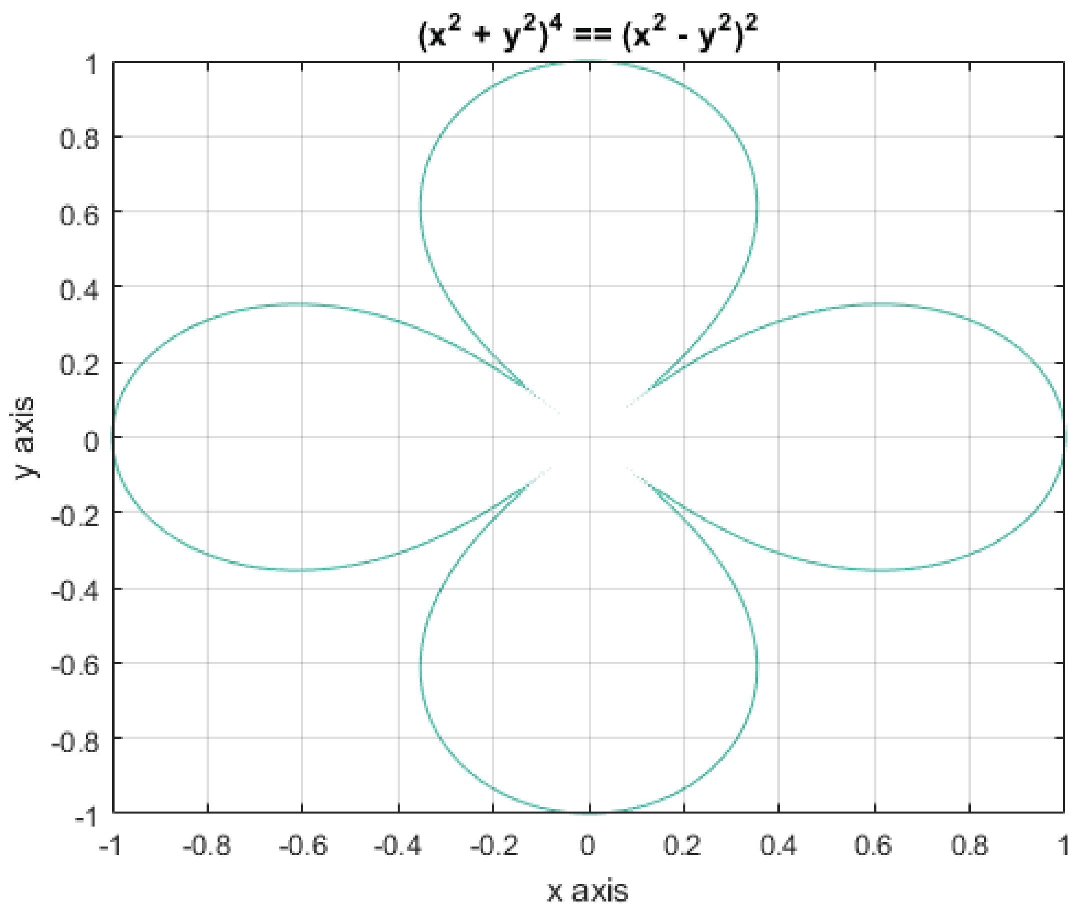
```
xlabel('x axis')
ylabel('no name axis')
title('Explicit function: x^3 - 6*x^2 + 11*x - 6')
grid on
hold off
```




### Implicit Function Plot

Using `ezplot`, you can also plot equations. For example, plot the following equation over $-1 < x < 1$:

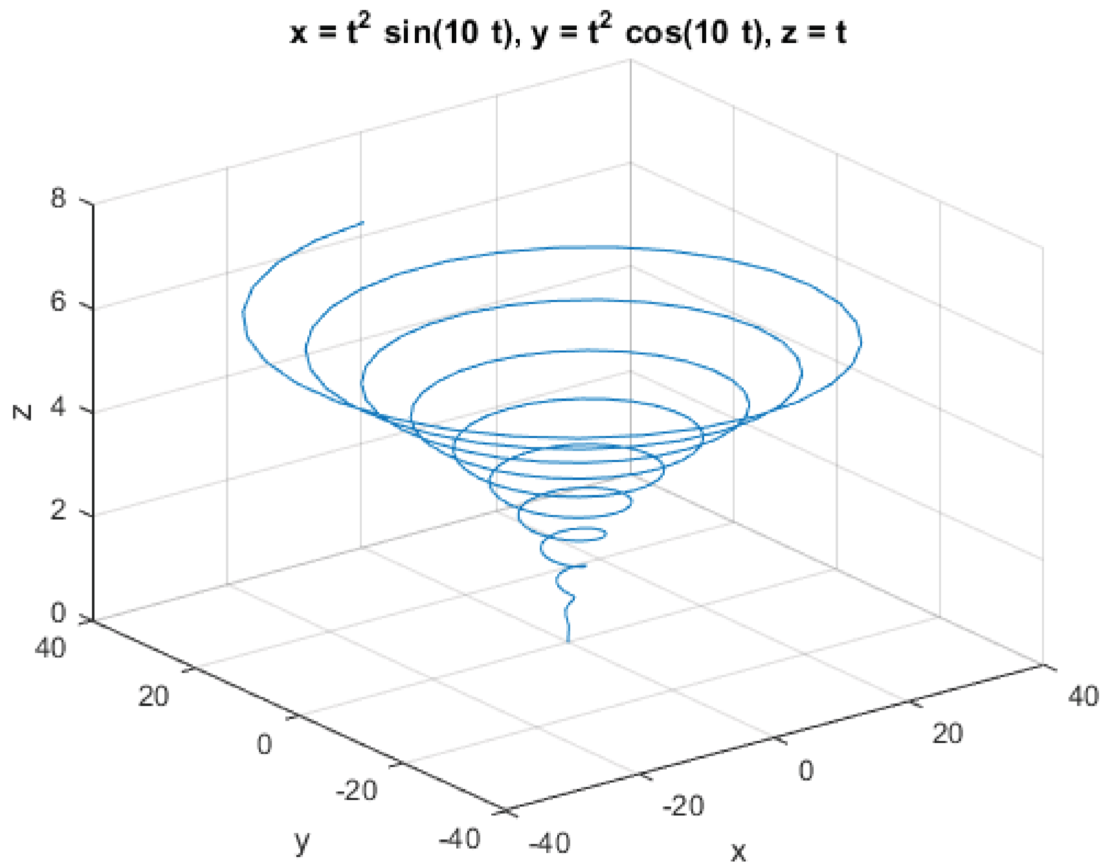
```
syms x y
ezplot((x^2 + y^2)^4 == (x^2 - y^2)^2, [-1 1])
hold on
xlabel('x axis')
ylabel('y axis')
grid on
hold off
```




### 3-D Plot

3-D graphics is also available in Symbolic Math Toolbox. To create a 3-D plot, use the `ezplot3` command. For example:

```
syms t
ezplot3(t^2*sin(10*t), t^2*cos(10*t), t)
```

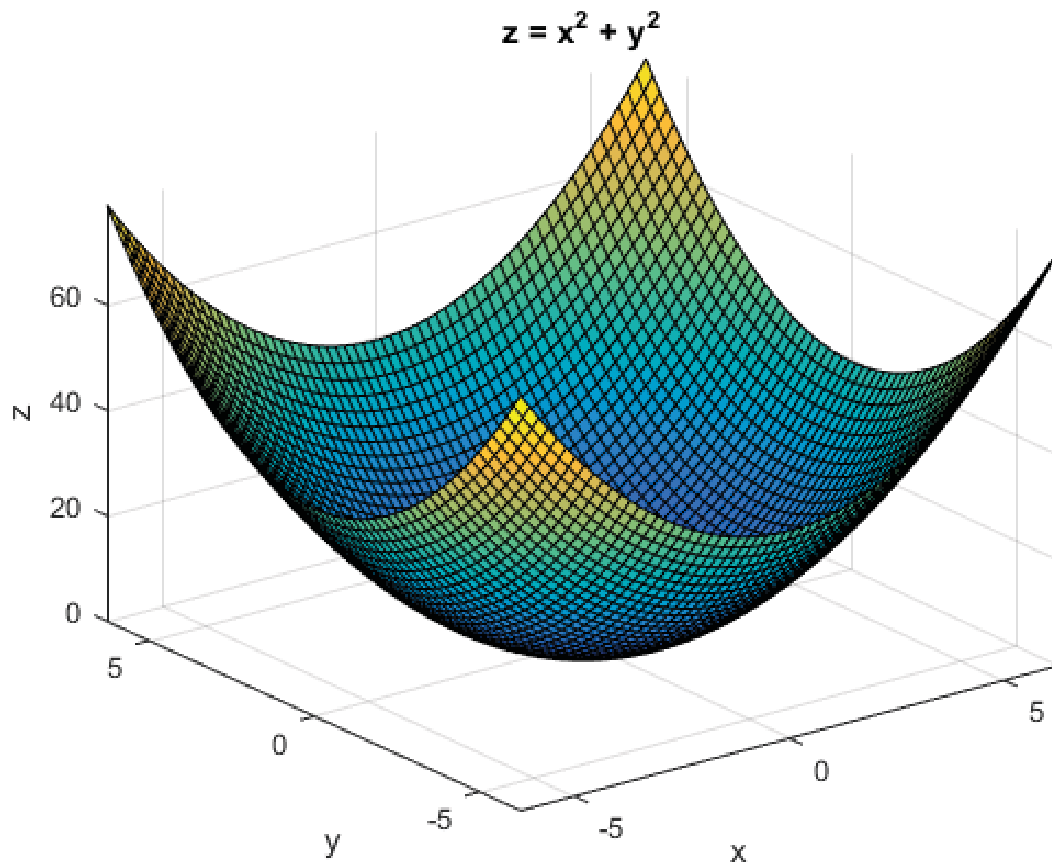



### Surface Plot

If you want to create a surface plot, use the `ezsurf` command. For example, to plot a paraboloid $z = x^2 + y^2$, enter:

```
syms x y
ezsurf(x^2 + y^2)
hold on
zlabel('z')
title('z = x^2 + y^2')
```

```
hold off
```



# Use Assumptions on Symbolic Variables

**In this section...**


"Default Assumption" on page 1-27
"Set Assumptions" on page 1-27
"Check Existing Assumptions" on page 1-28
"Delete Symbolic Objects and Their Assumptions" on page 1-28


## Default Assumption

In Symbolic Math Toolbox, symbolic variables are complex variables by default. For example, if you declare `z` as a symbolic variable using

```
syms z
```

then MATLAB assumes that `z` is a complex variable. You can always check if a symbolic variable is assumed to be complex or real by using `assumptions`. If `z` is complex, `assumptions(z)` returns an empty symbolic object:

```
assumptions(z)
```

```
ans =
Empty sym: 1-by-0
```

## Set Assumptions

To set an assumption on a symbolic variable, use the `assume` function. For example, assume that the variable `x` is nonnegative:

```
syms x
assume(x >= 0)
```

`assume` replaces all previous assumptions on the variable with the new assumption. If you want to add a new assumption to the existing assumptions, use `assumeAlso`. For example, add the assumption that `x` is also an integer. Now the variable `x` is a nonnegative integer:

```
assumeAlso(x,'integer')
```

`assume` and `assumeAlso` let you state that a variable or an expression belongs to one of these sets: integers, positive numbers, rational numbers, and real numbers.

Alternatively, you can set an assumption while declaring a symbolic variable using `sym` or `syms`. For example, create the real symbolic variables `a` and `b`, and the positive symbolic variable `c`:

```
a = sym('a', 'real');
b = sym('b', 'real');
c = sym('c', 'positive');
```

or more efficiently:

```
syms a b real
syms c positive
```

The assumptions that you can assign to a symbolic object with `sym` or `syms` are real, rational, integer and positive.

## Check Existing Assumptions

To see all assumptions set on a symbolic variable, use the `assumptions` function with the name of the variable as an input argument. For example, this command returns the assumptions currently used for the variable `x`:

```
assumptions(x)
```

To see all assumptions used for all symbolic variables in the MATLAB workspace, use `assumptions` without input arguments:

```
assumptions
```

For details, see “Check Assumptions Set On Variables” on page 3-44.

## Delete Symbolic Objects and Their Assumptions

Symbolic objects and their assumptions are stored separately. When you set an assumption that `x` is real using

```
syms x
assume(x,'real')
```

you actually create a symbolic object x and the assumption that the object is real. The object is stored in the MATLAB workspace, and the assumption is stored in the symbolic engine. When you delete a symbolic object from the MATLAB workspace using

```
clear x
```

the assumption that x is real stays in the symbolic engine. If you declare a new symbolic variable x later, it inherits the assumption that x is real instead of getting a default assumption. If later you solve an equation and simplify an expression with the symbolic variable x, you could get incomplete results. For example, the assumption that x is real causes the polynomial $x^2 + 1$ to have no roots:

```
syms x real
clear x
syms x
solve(x^2 + 1 == 0, x)
```

```
ans =
Empty sym: 0-by-1
```

The complex roots of this polynomial disappear because the symbolic variable x still has the assumption that x is real stored in the symbolic engine. To clear the assumption, enter

```
assume(x,'clear')
```

After you clear the assumption, the symbolic object stays in the MATLAB workspace. If you want to remove both the symbolic object and its assumption, use two subsequent commands:

1. To clear the assumption, enter

   ```
   assume(x,'clear')
   ```

2. To delete the symbolic object, enter

   ```
   clear x
   ```

For details on clearing symbolic variables, see “Clear Assumptions and Reset the Symbolic Engine” on page 3-43.